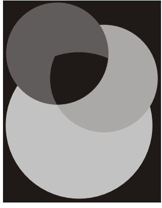


Business Objects:

Re-Engineering for Re-Use

Chris Partridge

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

First Edition published 1996 by Butterworth-Heinemann.

Second Edition published 2005 by The BORO Centre.
Second Edition printed and bound by Lodge Printers Ltd., Huntingdon.

© Copyright Chris Partridge 1996 & 2005
All rights reserved.

This document contains proprietary information of the author. It is his exclusive property. It may not be reproduced or transmitted, in whole or in part, without a written agreement from the author. No patent or other license is granted to this information.

Information in this document is subject to change without notice. Information concerning products is provided without warranty or representation of any kind, and the author will not be liable for any damage resulting from the use of such information.

The author, Chris Partridge, can be contacted at:
chris.partridge@BOROCentre.com

The right of Chris Partridge to be identified as author of this work has been asserted in accordance with the Copyright Designs and Patents Act 1988 (UK).

Trademarks/registered trademarks

REV-ENG™ and BORO Methodology™ are registered trademarks of The BORO Centre Ltd.

All other trademarks referred to are acknowledged to be the property of their respective owners.

Figures produced by Nathaniel Sombu

First Edition ISBN 0-75-06208-2 X
Second Edition ISBN x-xx-xxxxxx-x

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

Contents

Acknowledgements ix

Preface xiii

Prologue xxiii

Part One
Our Strategy for Re-Engineering Entities into Objects 1

Chapter 1 What and How We Re-Engineer 3

1 Introduction 4
2 What do we re-engineer? 4
3 How do we re-engineer? With thought experiments 14
4 The benefits re-engineering brings 17
5 Re-engineering entities into objects 22

Chapter 2 Focusing on the Things in the Business 23

1 Introduction 24
2 Focusing the re-engineering on things in the business 24
3 Problems identifying ‘things in the business’ 25
4 Ignoring ‘things in the business’ 27
5 What types of things (in the business) do we re-engineer? 31
6 Our starting point—the entity paradigm 34
7 Arriving at an object semantics for ‘things in the business’ 35
8 Re-engineering the ‘things in the business’ 36
9 The next part 36

© Copyright Chris Partridge
chris.partridge@BORCentre.com

Part Two
Our Starting Point—The Entity Paradigm 37

Chapter 3 What Is the Entity Paradigm? 39

1 Introduction 40
2 The entity paradigm's fundamental particles 40
3 The entity framework and its (re-)use of patterns 43
4 The entity paradigm and the file-record paradigm 45
5 Mapping entities and attributes onto files and records 46
6 The substance paradigm's secondary hierarchy 50
7 Simplifying the substance paradigm's treatment of relationships 56
8 Our current way of seeing stored information 61
9 The next chapter 62

Chapter 4 The Substance Paradigm's Semantics 63

1 Introduction 64
2 The semantics of the fundamental substance and attribute particles 64
3 Changes—a key type of thing 66
4 Generalising re-usable substance and attribute patterns 72
5 Our current way of seeing 80
6 The four key types of things 82
7 What's next 83

Part Three
Shifting Towards Objects—The Logical Paradigm 85

Chapter 5 The Emergence of the Logical Paradigm 87

1 Introduction 88
2 Origins of the logical paradigm 88
3 Shifting from substance to extension 91
4 Re-engineering primary substance 95
5 Re-engineering secondary substance 100
6 Re-engineering secondary attributes 101
7 Re-engineering relational attributes 102
8 Simplifying and generalising the information framework 107
9 Summary 109

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

Chapter 6 The Logical Paradigm's Framework 111
1 Introduction 112
2 A sense framework for logical objects 112
3 An environment that encourages compacting 122
4 The problem with logical changes 127
5 The four key types of things 130
6 A new, conceptually more accurate, logical way of seeing things 131
7 Summary 133

Part Four
Shifting to the Object Paradigm 135

Chapter 7 Physical Bodies as Four-Dimensional Objects 137
1 Introduction 138
2 The logical semantics for physical bodies 138
3 The shift to object semantics 143
4 Physical stuff objects 149
5 Classes of four-dimensional objects 151
6 Tuples of four-dimensional objects 153
7 A new way of seeing bodies—a key type of thing 153
8 Summary 154

Chapter 8 Changes as Three-Dimensional Objects 155
1 Introduction 156
2 States as physical body objects 156
3 Events – a new kind of physical object 169
4 The time-based 'consciousness' of information systems 179
5 A new way of seeing changes—a key type of thing 181
6 What's next 182

Part Five
Constructing Signs for Business Objects 183

Chapter 9 Constructing Signs for Business Objects 185
1 Introduction 186
2 Constructing signs for individual objects 187

© Copyright Chris Partridge chris.partridge@BOROCentre.com

3 Constructing signs for classes of objects 189
4 Constructing signs for tuples 206
5 Constructing signs for whole–part tuples 210
6 Constructing signs for dynamic objects 214
7 Signs as objects—modelling the model 216
8 What's next 217

Chapter 10 Constructing Signs for Business Objects' Patterns 219
1 Introduction 220
2 Patterns for the connections between extensions 220
3 State hierarchy patterns 236
4 Time ordered temporal patterns 239
5 Cardinality patterns for tuples classes 243
6 A pattern for compacting classes 248
7 Where we are 250

Part Six
Applying Business Objects 251

Chapter 11 The REV-ENG: An Approach to Applying Business Objects 253
1 Introduction 254
2 The REV-ENG approach 254
3 The worked examples 255
4 A systematic approach to re-engineering 256
5 A framework for the model 260
6 Generalisation and compacting 264
7 What's next 266

Chapter 12 Re-Engineering Country's Entity Format 267
1 Introduction 268
2 The systematic re-engineering process 268
3 The context for re-engineering 269
4 Re-engineering country entity type sign 271
5 Re-engineering attribute type signs 277
6 Basic elements of the re-engineering completed 291

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

Chapter 13 Generalising Country's Re-Used Patterns 293
1 Introduction 294
2 Generalising re-used patterns 294
3 First stage of the re-engineering completed 297

Chapter 14 Re-Engineering Our Conceptual Patterns for Country 299
1 Introduction 300
2 Finding conceptual patterns 301
3 Character strings patterns 303
4 Nested countries pattern 308
5 More accurate nested countries patterns 312
6 Current countries 316
7 Summary 317

Chapter 15 Re-Engineering Region 319
1 Introduction 320
2 Re-engineering the region entity formats of the existing system 320
3 Re-engineering our conceptual patterns for region 332
4 Summary 339

Chapter 16 Re-Engineering Bank Address 341
1 Introduction 342
2 Familiarising ourselves with bank address entity formats 342
3 Re-Engineering bank 343
4 Re-Engineering address 345
5 Nested address lines 349
6 Address joining events 354
7 Generalising name 354
8 An aspect of a pattern 356
9 Summary 357

Chapter 17 Re-Engineering Time 359
1 Introduction 360
2 Re-engineering an existing system's bank holiday entity format 360
3 Re-engineering day 361
4 Re-engineering bank holiday 365

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

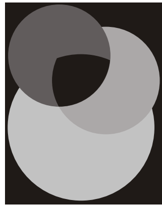
5 Re-engineering an existing system's weekend entity formats 367
6 Re-engineering our conceptual patterns for bank holiday and weekend 370
7 The object model for temporal patterns 373
8 Summary 373

Chapter 18 Starting a Re-Engineering Project 375
1 Introduction 376
2 Take a re-engineering approach 376
3 Establish priorities for the construction of fruitful, general, and so re-usable patterns 377
4 Taking care to manage large projects in a generalisation-friendly way 385
5 Produce a validated understanding of the business 392
6 Object model the migration of business patterns 393
7 Summary 396

Epilogue **397**
1 Introduction 398
2 The accounting paradigm's debit and credit pattern 399
3 Accounting's ledger hierarchy 409
4 Developing a new object-oriented accounting paradigm 410
5 Industrialising information 411
6 21st century information industries 413

Bibliography **415**

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



BORO

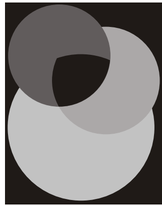
Acknowledgements

Second Edition

There are a number of people who have helped to both keep the interest alive in the approach described in the book and develop it further. First in temporal order is Nicola Guarino and the Laboratory for Applied Ontology at CNR in Padova - now situated in Trento (www.loa-cnr.it) where I spent a couple of years continuing my research into ontology. Barry Smith (Buffalo Center for Ontological Research at Buffalo University and Institute for Formal Ontology and Medical Information Science at Saarland University) has freely given advice and encouragement over the years. Then the team at Brunel University who worked on legacy system re-engineering at the Fluidity research centre and its SITE project - in particular Mark Lycett, but also Sergio de Cesare and Aseem Daga and Dirk Siebert. Finally the team at 42 Objects Ltd. (www.42Objects.com), John Allen (for his commitment to using the methodology commercially), Barie Brown, Rob Payne, Phil Lewis and Jeremy Wood.

I would also like to thank Kyoko Hayashi for her work in designing this second edition for a re-printing.

© Copyright Chris Partridge



BORO

Acknowledgements

First Edition

I owe an intellectual debt to Willard van Orman Quine and John Edwards; without their influence I would not have been able to ‘see’ the world of business objects described in this book. Quine has had a big influence on my way of thinking since I studied him at university, though I had no conception then that his ideas could be of any commercial use. I recently found this passage in Quine’s essay, *Applications of Modern Logic,* (1960) which indicates he appreciated their commercial potential;

> In the programming of problems for genuine machines [computers] there is bound to be vast scope for the application of logical techniques . . . For programming demands utter explicitness and formality in the analysis of concepts; furthermore it thrives on conceptual economy; and it rewards novel lines of analysis, with never a backward glance at traditional lines of thought. It is a remarkable fact that programming provides a strictly monetary motive for very much the sort of rigour, imagination, and conceptual economy that have hitherto been cultivated by theoretical logicians for purely philosophical or aethestic reasons. The extremes of abstract theory and practical application are seen converging here.

I first met John Edwards in 1987 (at the suggestion of James Odell). The presentations of his Ptech™ method and our many discussions afterwards acted as a catalyst for my thoughts on objects. He helped me realise that information engineers had not yet come up with a suitable paradigm for the way we see things, and that non-computing work on information, such as Quine’s, had a lot to contribute in this area. He gave me the intellectual push I needed to recognise that the ideas of Quine and other (apparently non-computing) thinkers could be applied fruitfully to the business modelling stage of building information systems.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

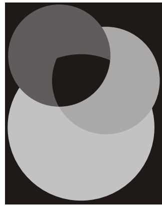
I am also indebted to a number of researchers whose work helped me to understand more about business objects. In particular, to Bill Kent and John F. Sowa for highlighting the important modelling issues. Descriptions of their work are contained in the books mentioned in the bibliography at the back of this book.

The approach described in the book has evolved over a number of years and contributions have been made by a number of people. I would like to mention the following people who contributed in various ways in the critical early stages; Zarna Bannerjee, Julian Bennett, Mike Briggs, Mark Bullock, Gary Constable, Cathy Hansford, Alec Hoffman, Cormac Kelly, Mei Liu, Ian Macleod, Victor Peters, Kay Preddy, Neil Prior, Tracy Riddle, Paul Sheldrick, Mark Slater, Kevin Slaughter, Dave Thomas, Sue Wilcock and Anne Williams. In addition a number of people gave indispensable support to the projects that helped to develop the approach. These included the late Trevor Ball, David Betts, Peter Gellatly and, particularly, Brian Finlayson.

Several people have helped in the writing of this book; it is much better as a result of their constructive comments and criticisms. I am particularly grateful to Julian Bennett, Sue Buzzacott, Peter Gellatly, Cecily Partridge and Mark Slater, who have reviewed a number of versions of it. I would also like to thank Steve Chambers, Richard Meakin and Francis Travis who looked at sections of the book. I am greatly indebted to Nathaniel Sombu for his comments on and patient, painstaking production of the figures in the book..

I also think it is worth mentioning that without my word-processing software (Microsoft's Word for Windows) and Nathaniel's graphics software (Corel's CorelDRAW) I would have found this book impossible to produce.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



BORO

Preface

Second Edition

The first edition of Business Objects went out of print in 1998, a year or so after it was published. Since then I have received a steady stream of requests for copies. I have met this to some extent through the provision of electronic copies - but this has not been wholly satisfactory. Hence my decision to produce a second print edition.

This decision raised The question of what needed to be revised. After some deliberation, I realised firstly that the content of the book still stands and secondly that to incorporate the new work would result in such a change in structure, that it would become effectively a new book. So I decided that, apart from typographical errors, there should be no changes.

When writing the book in 1994-5, I listened to advice that using the ‘O’ word - ontology - would put off many business readers, who were a key target audience. Accordingly instead of talking about ‘ontological paradigms’ or ‘ontologies’, I used the less contentious term ‘paradigm’. The world has moved on since then, and the term ‘ontology’ is not seen as inaccessible as it did then. One important reason for this is that Computer Science has started using the term - and the business community is becoming aware and interested in this. Hence, I considered inserting (in many cases, re-inserting) the ‘O’ word into the original text.

However, the implicit use of ontology in the book is based upon its long established use in philosophy (“the set of things whose existence is acknowledged by a particular theory or system of thought” - E. J. Lowe, The Oxford Companion to Philosophy). Computer Science’s use of the term (“the formal specification of a conceptualisation”) is, in important ways, different from this.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

To avoid confusion, I decided not to explicitly introduce the term in this edition. However, I believe that the philosophical sense is going to acquire more importance over the next decade. As well as the work I am doing, there are a number of other researchers doing work taking account of the philosophical tradition (for example, Barry Smith and Nicola Guarino). I believe that this work will help to make the philosophical notion of an ontology more generally known and understood.

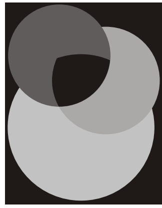
Substantial progress has been made since the publication of the book. I have set up two organisations to handle the two aspects of the work: the BORO Program (www.BORO-Program.org) to deal with research and the BORO Centre (www.BOROCentre.com) to handle consultancy and training.

Under the BORO Program banner, I spent two fruitful years (2000-2002) at CNR in Padova at the Laboratory for Applied Ontology (www.loa-cnr.it) continuing my research into ontology. On returning to England, I have been continuing my research into ontology and legacy system re-engineering with the Fluidity research centre at Brunel University, initially on the SITE project. Some of this BORO research has been published in papers - copies of which are available from the BORO Program website. I plan to publish the rest, initially as papers and then subsequently as a book.

Under the BORO Centre banner, the most important development is the work being done in collaboration with Forty Two Objects Ltd. (www.42Objects.com). They are a venture capital company (backed by 3i) that have based the development of their service oriented business applications upon an ontology developed using the methodology described in this book.

The methodology described in the book is referred to as the REV-ENG Methodology. In the work done with the BORO Centre, to mark its evolution into something more powerful, it has been re-named the BORO Methodology.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Preface

First Edition

© Copyright Chris Partridge
chrispartridge@BOROCentre.com

Introduction

The aim of this book is to show you how to use business objects to re-engineer your existing information systems into models—and so systems—that are not only functionally richer but also structurally much simpler. It is a practical guide to re-engineering your systems; when you finish reading it, you will be ready to start. (Business objects can also help you to re-engineer the underlying business that is processed by these systems; we discuss the implications of this in the ***Epilogue***.)

Background

The approach to business objects described in this book was sparked off by a need and desire to build systems more effectively. In many ways, it was a lucky accident. In 1985–86, after a number of years away from the sharp end of systems building, I became involved in the development of a management information system. This used the latest version of a traditional development methodology. I expected it to be a significant improvement upon the ones I had used some time before. However, the 'improvements' seemed to have made the methodology more cumbersome and bureaucratic, adding layers of complexity rather than understanding. I soon found that this was a general trend; other established methodologies were also becoming more unwieldy.

It seemed to me that there must be a better way. I could not believe that someone somewhere had not developed something better. So a group of us began looking. We soon discovered object-orientation (O-O) and started finding out all we could about it. Though most people were using it for programming, we quickly realised that it had a lot to offer at the analysis and, more interestingly, the business modelling stages.

Then, as luck would have it, in 1987 I became involved in a project to re-develop a large investment management system. After much discussion, the controlling committee decided that we would re-develop the system with a small team using these new and relatively untried O-O techniques. We believed—rightly so, as it turned out—that they would enable us to get high levels of re-use, reducing the effort (and therefore the cost) needed to re-develop the system.

There was one decision that shaped our whole approach. We decided to start by re-engineering the existing system into an O-O business model. This was a markedly different approach from almost every other O-O project at the time; they mostly used O-O to design and code systems. Two requirements motivated our approach:

- First, we needed to document the existing system.
- Second, we felt that we could salvage a substantial amount of the time and money invested in the existing system. We believed that we could capture the understanding of the business embedded in the system and re-use it in the new system.

The approach

Our approach, as with any new approach, evolved and changed. However, it was clear to us from early on that we had stumbled on something very different from traditional information modelling and that it had enormous potential.

The two stages

We soon found that modelling the existing system was not a straightforward case of building an O-O model, but that it fell into two stages:

- Reverse engineering, and
- Forward engineering.

In the first stage, we translated the existing system's business entities into business objects. In most cases, this involved translating the system's computer code straight into a business model. This reversal of the normal system building process is often called reverse engineering. (Strictly speaking, what we were doing was not 'reverse engineering' as we were not re-building the—implicit—entity model from which the system was built.) The 'reverse-engineering' approach met our two original requirements. It not only documented the existing system but salvaged some of the investment made in it for re-use in the new system.

In the second stage, the process of modelling with business objects naturally led us into re-engineering our more sophisticated conceptual patterns for the business. To contrast

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

this with reverse engineering, we called this stage forward engineering. We called the two stages together, re-engineering.

When forward engineering, we found that we naturally identified inaccuracies and artificial constraints in the way the existing system reflected the business. Typically, these were because of the simplified view of the business that entities force on modellers rather than any errors in modelling. Business objects' superior powers encouraged us to translate our unsimplified conceptual patterns—without the inaccuracies and constraints—into the model. The result was more general and so more re-usable business objects.

The benefits

This had two big benefits. First, we found that many of the business objects we constructed when re-engineering one part of the system were sufficiently general to be (re-)used many times in other parts. This meant that each business object was re-used to do jobs originally done by a number of entities (or attributes). This had a remarkable effect; as the scope of the re-engineering grew, the model became much simpler without losing any power. We called this process compacting. We monitored the compacting, using rough and ready measures, and found the business model used substantially fewer objects than the computer system had entities and attributes.

Second, it was plain that the business objects were general enough to be (re-)used, with no extra effort, to do things that the current system could not. They had become not only simpler, but functionally richer.

It is often claimed that O-O is able to reflect the business more directly. We discovered early on that not only is this true but, when we modelled the business more accurately, the objects we constructed tended to be more general and so more re-usable. This made the model even simpler and more compact.

The re-engineering also had a profound effect on the way we saw the business. As we became familiar with the object-view of the business, the old entity-view began to seem hopelessly inaccurate. In many cases, we wondered how we had managed to use the old entities for so long. This was just one indication among many that our business paradigms were changing radically and that we were developing a more accurate vision of the business.

The REV-ENG™ approach

The original modelling team nick-named the approach REV-ENG—a shortening of REVerse-ENGineering, our name for the first part of the process. This name was used for a number of years. When the BORO Centre was set up, the name was changed to the BORO Methodology.

Over a number of projects, we systematised the approach, streamlining the process of re-engineering. Over time, we have added further refinements and enhancements to make it even more effective. Under this new approach, a number of the old system

©Copyright Chris Partridge chris.partridge@BORO-Centre.com

building rules no longer seemed to apply. For instance, with a small team it would seem sensible to restrict the scope and, in particular, avoid difficult requirements. However, our small team found that seeking out and modelling difficult cases often made things easier. Modelling these produced business objects that both made the model simpler and had much more potential for re-use.

The budget-holders and users benefited from the approach as well. The budget holders were happy because systems were cheaper to build. This was mainly because the models were much simpler than their entity counterparts. But it also helped that the systems people using the models were less likely to misunderstand the business.

The users were happy because they got functionally much richer systems—ones that even included some of their most difficult requirements. This would not have been cost-effective with an entity oriented approach. They were particularly happy because the systems were not only functionally richer than their existing systems, but also all the others they had looked at.

Furthermore, as the users became more familiar with their systems, something remarkable begins to happen. The systems seem to have captured the essence of the business. We realised this when we found them being used to handle areas that had not been envisaged when we built the business model. For instance, on one project the users found that their re-engineered securities back-office system could already handle new financial instruments and situations that no-one had thought of when the system was built.

What are business objects?

By now you are probably asking yourself, what business objects are and how they are used to build models. Business objects are different, very different. Most of you will need to work your way through the first half of the book to get a genuine understanding. However, it helps to start off with some idea (even a vague one) of what they are, so I will try to give you that now.

Business objects are a new way of seeing things. Most people (although they often don't realise it) currently see things as entities and attributes. They see individual things, such as an individual warehouse or customer, as entities. They group these individual entities into types—for example, warehouses and customers in general—called entity types. Both entities and entity types have attributes—also sometimes called properties or qualities. For instance, warehouses in general have the attribute of size, and an individual warehouse may have the size attribute, small. In addition, things change—typically by changing attributes. For example, someone may build an extension on an individual small warehouse, making its size attribute change from small to large.

Currently, when we business model, we construct what we assume is a description of the business. What we are actually doing is describing the business entities we see. This is often called our business paradigm. We then embed these entities deep in our computer systems.

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

When we shift to an object way of seeing, everything becomes an object. All the ‘things’ we used to see as entities or entity types, along with their attributes, are now seen as objects. Even changes of attribute are objects. As you read through the book, you will begin to appreciate the implications of this. But for the moment, just think of business objects as a very general type of thing.

Business objects’ new way of seeing leads to a very different approach to modelling the business. Instead of just describing our business paradigm, we actively revise it. We transform its entities into business objects. This is a re*vision*ary rather than a descriptive approach—what is sometimes called business re-engineering. The re-engineering transforms the way we see things so that we end up with a fundamentally new, radically different and better vision of the business. This gives us a new, structurally simpler and more powerful business paradigm, and so computer systems.

Explaining business objects

As this brief outline makes clear, business objects need some explaining. We did not have to think too hard about this during our first few projects. The team wanted to tell other people what we were doing, but tight deadlines meant we could not spare the time. It was a number of years before we actually sat down and properly talked about business objects with people outside the team. Only then did we realise that people found it very difficult to get their heads around what they are.

We had, until then, been documenting the approach for ourselves as a method or procedure. We explained what we were doing, not why. We soon found that this was, unfortunately, not much use to the people trying to understand what business objects are. Someone needed to do something. So I decided to work out how to help people understand.

I started off with the assumption that this subject would be well covered; after all it is important. I soon found that it was not. For instance, I could not find any computing books that explained what business objects are. When I looked outside computing, I had more success. There were a number of useful books covering different relevant areas, but nothing that directly addressed re-engineering with business objects.

I set myself the task of co-ordinating what I had found into a single, reasonably coherent, framework. As I began doing this, I discovered that many aspects of business objects were really much more fundamental than we had suspected. I found that I needed to do a substantial amount of analysis of the nature and history of information, particularly computer information, to develop a reasonable understanding of what was going on. It is unfortunate that this topic is not well covered in works on business modelling. I hope that future work in this area will continue to develop the start made here in this book.

Once I had a reasonably clear picture in my mind, I realised that the best way for people to understand business objects was to focus on how they were re-engineered from business entities. From this perspective, people could see the business object para-

© Copyright Chris Partridge
chris.partridge@BOROcentre.com

digm (the conceptual framework surrounding business objects) as a natural response to the problems inherent in business entities.

Object-orientation is new and different, so mastering it involves acquiring new knowledge and new skills. IT managers have found that when their programmers move from traditional to O-O programming, they need a large amount of training and hand-holding. The move from traditional to business object modelling is much more fundamental. So it will be no surprise to hear that you need to go through some substantial re-learning to get to grips with the new ideas. That is the bad news, the good news is that once you understand them, the ideas, like all good ideas, are simple. This does not mean you will find them easy to pick up—after all the whole point is that they are new and radically different. But it does mean that, once you are familiar with them, you will find them easy to apply.

Understanding these fundamentally new ideas

Developing these new ideas to the stage that they are at now took me and the rest of the team many years of hard work. A large part of the task was demolishing our old ideas to clear the decks for the new ones. These enable us to take a radically different view of the business—to see it in terms of business entities rather than business objects.

Although you will have the benefit of following in our footsteps, seeing things as business objects may turn out to be a bigger task than you now envisage. When you start reading the book, you will realise that the only way for you to learn how to see such different things is to re-build the way you see from the foundations up.

Initially, I found it odd to be dealing with such fundamentally new ideas about information when working within information technology. All my training (in computing) had led me to expect the tasks to get more technical and detailed not less—more about computer systems not less. When I became used to dealing with new general ideas, I found it refreshing, and it made my job more, not less, enjoyable. I hope you all have the same experience.

Brief overview of the contents of the book

By now you will be getting an impression that business objects are very different from traditional business entities. Most of you will begin to realise that you need some help developing an authentic understanding of what business objects are. And that without this understanding, you have little or no chance of starting to acquire the skills needed to re-engineer business entities into objects.

This book gives you the help you need. Its first half helps you understand what business objects are and how to model them. Its second half shows you how to re-engineer business entities into business objects. When you finish these two halves, you will have acquired the skills needed to re-engineer the entity oriented business paradigms embedded in your existing systems.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

Understanding business objects

The first half of the book is divided into the following five parts.

- Part One - Our Strategy for Re-Engineering Entities into Objects
- Part Two - Our Starting Point—The Entity Paradigm
- Part Three - Shifting Towards Objects—The Logical Paradigm
- Part Four - Shifting to the Object Paradigm
- Part Five - Constructing Signs for Business Objects

Re-engineering the entity paradigm into the object paradigm is not a straightforward task. So, in Part One we define our strategy. We examine the approach we will take, looking at what and how we will re-engineer (in particular, the key types of things we need to re-engineer).

In Part Two we clarify where we are now. We look at the nature of the entity paradigm, seeing why it is a simplified version of another more sophisticated paradigm, called the substance paradigm. We also examine both paradigms potential for re-use.

We then re-engineer the more sophisticated substance paradigm into the object paradigm. Because this involves a considerable mental effort, it is easier to digest in two chunks. So, in Part Three, we re-engineer to the logical paradigm, a kind of halfway house. Then, in Part Four, we re-engineer to the object paradigm. At each stage, we examine the problems that prompt the re-engineering and how the new paradigm resolves them. We also examine how the re-engineering increases the potential for re-use.

The re-engineering enables us to see business objects and so see the business in a new way. Part Five shows us how to describe what we see and how to construct the signs in a model that map directly to these business objects.

As you read through the first half of the book, you will be re-engineering your current entity oriented way of seeing the business into a very different object-oriented way. When you have finished, you will understand what business objects are and how to model them.

Applying business objects

The second half of the book (Part Six) shows you how to re-engineer the business entities embedded in your current systems into re-usable business objects. It uses worked examples to give you a feel for how this works. These have been selected, not just to illustrate how to re-engineer, but also to provide you with objects that you can re-use in future re-engineerings. As the examples contain business entities that can be found in most business systems, you will be able to (re-)use their re-engineered business objects in your projects.

© Copyright Chris Partridge
chris.partridge@BORO.Centre.com

Reading the book

This book has been structured to help you build an understanding of the new object-oriented way of seeing things; the later chapters increase the understanding created by the earlier chapters. You should try and work your way from the beginning right through to the end. Those of you who are familiar with topics covered in a section may choose to skip it. But you should be careful. If you miss an earlier point, you may find later points difficult to understand.

Explaining concepts

I thought it sensible to assume that some of you would be unfamiliar with the general (apparently non-computing) ideas; so, I have been careful to explain them. Because the objective of the book is to help you understand business objects, I have kept, as far as possible, the explanation of the ideas simple.

I hope those of you who are familiar with the ideas will bear with me while I do this explaining. I also hope you will recognise that where I have simplified explanations of some complex subjects, this is appropriate given the overall objective.

Finding out more about the ideas

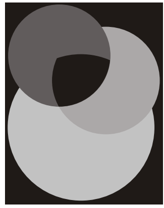
None of the individual ideas presented here are original. However the way in which they have been brought together and applied is sometimes novel. I found that one of the most enjoyable aspects of researching this book was finding that the new ideas not only explained to me my experience of the workings of business objects but fitted into a coherent whole.

I hope you will have a similar experience. This may inspire you to find out more about the topics covered in the book. I have compiled a bibliography (which is at the end of the book) that will tell you where to start looking.

Comments, suggestions and/or criticisms

This book is, in many ways, a report of work in progress. I know that there are some parts of it that, with even more work, I could improve. I am also sure that other people have useful ideas to contribute. For this reason, I would genuinely appreciate any comments, suggestions and/or criticisms. You can send them to me directly by electronic mail (mail@ChrisPartridge.com).

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Prologue

Using Objects to Reflect the Business Accurately

1 A core issue for business objects
2 Why we need business objects' revisionary approach
3 What do we re-engineer—paradigms
4 What are thebenefits of re-engineering business paradigms?
5 Summary

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 A core issue for business objects

Underlying the approach taken in this book is a core issue for objects and business modelling—how can we accurately reflect the business in a model and thus in a computer system as well? This “Prologue” provides a context to the way in which we aim to explore and resolve this issue. It outlines our approach and will assist you as you work your way through the book.

1.1 O-O’s original claim

In the late 1980s, object-oriented (O-O) system building became popular. At that time, one of the common claims of the experts was that the objects in their models directly reflected reality. For instance, Ivar Jacobson (in *Object Oriented Software Engineering*, Addison Wesley, 1994) wrote:

> A model which is designed using an object-oriented technology is often easier to understand, as it can be directly related to reality . . . Since objects from reality are directly mapped into objects in the model, the semantic gap is minimised.

Peter Coad and Ed Yourdin (in *Object-Oriented Analysis*, Yourdin Press, 1991) gave the same message:

> OOA directly maps problem domain and system responsibility directly into a model. Instead of an indirect mapping . . . the mapping is direct, from the problem domain to the model.

It was then generally accepted that the objects in O-O models map directly onto objects in the business. This appeared to neatly resolve the core reflection issue that direct mapping would ensure the business is reflected accurately.

1.2 Questioning the original claim

More recently, this direct mapping claim has been questioned, at least for the objects in object-oriented programming languages (OOPL). As people gain more experience with these languages, they realise that even though its objects may be better at reflecting reality, they do not do so directly. For example, Steve Cook and John Daniels (in an article entitled “Object-Oriented Methods and the Great Object Myth”) wrote:

> Many authors . . . propose, as though it were obviously the case, that the real world consists of encapsulated resources and predefined access procedures. So we find it stated that a real aircraft has take-off and fly operations, a real cup provides a drink operation, and so on.
>
> This view of the world—which we shall call the object myth—is nonsense. If you drink a cup of tea, you do not invoke the drink operation on the cup anymore than the cup invokes the drink operation on your lips, or, indeed, anything invokes an operation on anything else.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

Clearly, the OOPL (and the analysis and design methods based upon them) that lead to things such as cup objects that invoke drink operations do not directly reflect reality. However, this does not mean that the experts' original insight was misguided. We can still use objects to help us reflect reality directly. To do this, the whole approach to the issue must be changed. We need to examine what objects in the business are instead of dabbling with the 'objects' in models and programming languages.

1.3 Overturning a business object myth

A popular misconception has bedevilled the understanding of what objects in the business are. It has been assumed, even by the experts, that people tend to see the world in terms of objects. For example, Ivar Jacobson wrote:

> People regard their environment in terms of objects. Therefore it is simple to think in the same way when it comes to designing a model.

This is profoundly wrong. It is a myth that people see or think in terms of objects. This myth has seriously hindered the development of business objects.

People naturally see the world in terms of attributes belonging to entities (though they might not call them that). When most people see a red car, they think they see a car with the property (attribute) of redness. They are not seeing objects because neither the car nor its red attribute are objects. We examine what they do see in more detail in Part Two.

If we want to persuade people that objects are easy to use, then it might seem a good idea to suggest that O-O is based on the way people see the world. If we want our system building point to be successful, then it is a terrible idea. It leads us in the completely wrong direction. It stops us from recognising that we need to shift from our current entity (and attribute) way of seeing things to an object way.

1.4 The nature of business modelling

A fundamental reason for these misconceptions about what people see has not been clearly examined. In terms of the traditional stages of system building, we examine the business at the initial 'business modelling' stage (shown in ***Figure P.1***). This is the stage at which we should map business objects directly into a model. This is where we should shift to an object way of seeing the business.

The misconceptions persist because system builders typically think of the business modelling stage in the same terms as the other stages in system building rather than thinking of business modelling in its own terms.

Figure P.1 shows that this involves looking outward at the business rather than, as the other stages do, looking inwards at the final system. We also need to recognise that:

- Business models explain what the business does, and
- System models explain how the system will operate.

© Copyright Chris Partridge chris.partridge@BORO Centre.com

In other words, the business model works at an understanding level and the system model at an operational level. This is reflected in their objectives. Business modelling's objective is, or should be, to capture an understanding of the business. The later stages use this understanding, but their objectives are aligned with the successful operation of the implemented system. As we work through the book, we shall see how important making the distinction between understanding and operation is for business modelling.

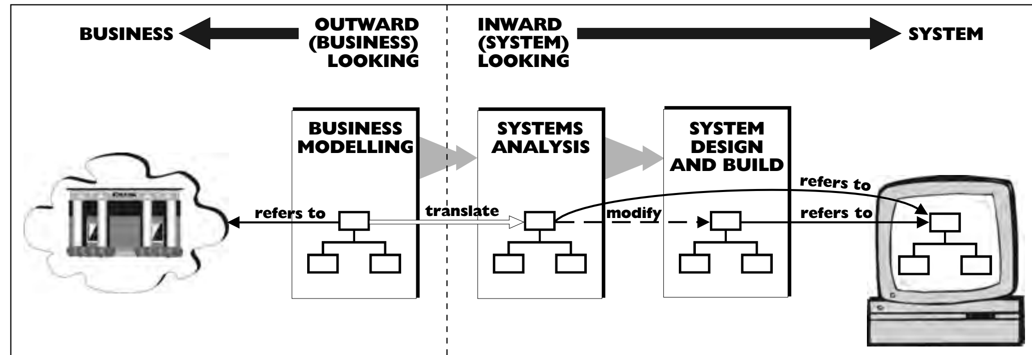


Figure P.1: Traditional stages of system development

2 Why we need business objects' revisionary approach

If business objects change the way we see things, then it seems reasonable to ask why we should go through with this change. Because it involves a substantial upheaval, there should be a pretty good reason to justify it.

2.1 Why entities and attributes are problematic

To appreciate the fundamental reason for a revisionary approach to business objects, we need to understand why entities and attributes are not suited to computer technology. This involves understanding why they developed in the first place.

The entity way of seeing the world (the entity paradigm) is practically prehistoric. It is based on the substance paradigm. This was first formalised by the Ancient Greek Aristotle in the 4th century BC (in other words, over two thousand years ago). We will look in more detail at these two paradigms in Part Two. What is of interest to us here is how pen and paper technology influenced the entity paradigm's development.

The entity way of seeing things is designed to make it xeasier to store information, using pen and paper technology. Information about a world seen as entities and attributes is much easier to divide into rows and columns. This division makes it much easier to store on two-dimensional paper. (The ease of use more than offsets the distortions that arise from imposing an entity-view—as we shall see in ***Chapter 3***.)

By contrast, computer technology is not constrained in the same way as two-dimensional paper. It is possible to store computer information in many more ways than just rows and columns. So, when using computer technology, imposing a view based on entities and attributes creates unnecessary constraints.

In these circumstances, we might expect computer information to have thrown off these constraints. However, a moment's consideration reveals that most computer information is still steeped in an entity-view based on paper and ink technology. This is not really surprising because we inherit most of our ways of thinking about information from an age dominated by this old technology.

That is why most computer information fits neatly onto paper forms, such as statements of account, sales invoices and deal slips; examples of which are shown in ***Figure P.2***. The paper-bound entity way these forms handle information has been imported wholesale into our computer systems.

Figure P.2: Forms—products of paper and ink technology

System builders recognise that it is a mistake to use the old manual paper-bound way of handling things when automating a process. Although they recognise this, it is ironic that they are still enchained to a paper-bound entity way of viewing the business.

2.2 Computing technology bringing radical changes

The technology leap from paper and ink to computers is enormous. Yet, the underlying entity paradigm with its rows and columns structure has not yet really changed. As we have just seen, forms such as sales invoices and deal slips have not been transformed into something radically different; i.e., something that looks as if it were based on computing—not paper and ink—technology.

Changes have occurred, but if we look closely, little change has taken place in the basic information structure. The big change is in the efficiency with which they are processed. Automated computer systems process more deal slips (and more sales invoices) faster and more accurately than the old manual paper systems.

Although this is a welcome improvement, i t is still disappointing that we have not yet had the kind of radical change (and the benefits it would bring) one might expect from computer technology. Business objects are now bringing this radical change—we will see the results in Part Six's worked examples.

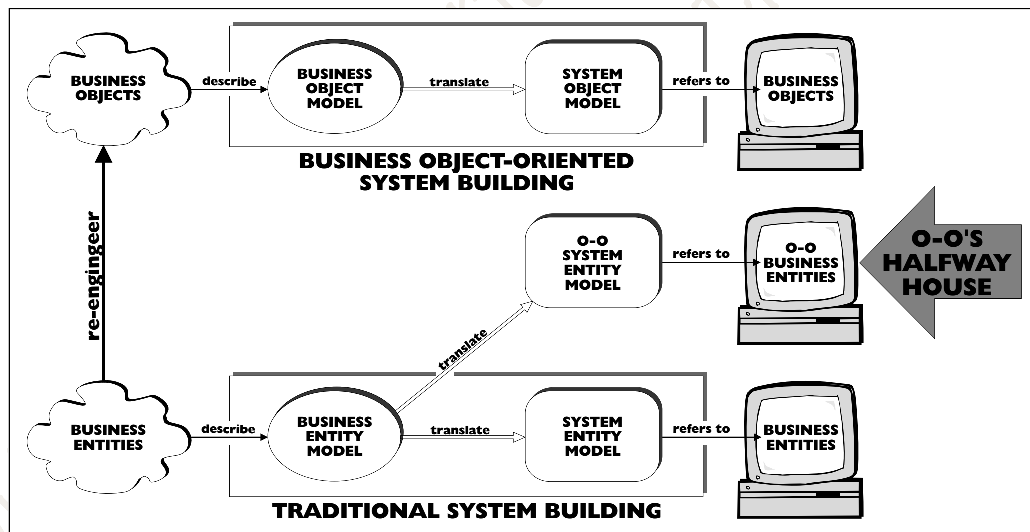
2.3 O-O programming's halfway house

Most O-O system developments are being carried out using an entity view of the business. Without an understanding of business objects, the developers have no choice. This situation is reflected in the many O-O textbooks that suggest using entity modelling for the early stages of system development.

This explains, to some extent, why O-O is currently having more success with objects whose task is to make systems work rather than reflect reality—such as the objects in screen interfaces. For example, O-O has been used to develop impressive graphical user interfaces (GUIs). However these systems have less impressive, more traditional, innards.

Figure P.3 explains why. Their innards contain objects that reflect business entities rather than business objects. The result is a kind of halfway house—built from O-O business entities rather than business objects. The figure also illustrates how we need to construct business objects—by re-engineering the business entities.

Figure P.3: O-O's halfway house



3 What do we re-engineer—paradigms

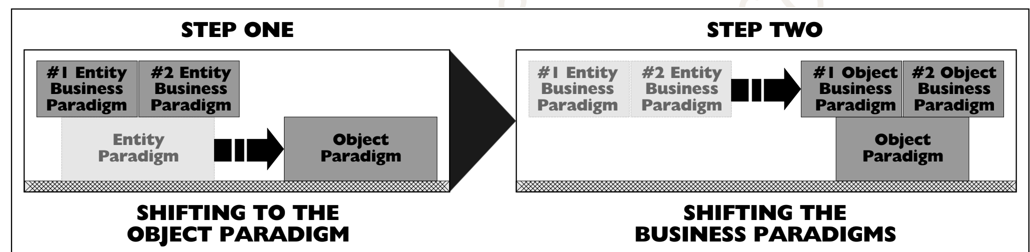
Figure P.3 implies that we need to re-engineer the business entities into business objects. However, this is only part of what must happen. The way we see the business is supported by a whole framework of ideas—a paradigm. It is this that we actually re-engineer.

For our purposes, it makes sense to divide this framework into two levels; a business paradigm and an information paradigm level. The specific objects we are interested in at the business paradigm level will vary from business to business. For example, the securities industry has one group of business level things, such as securities trades; while the oil industry has different business level things, such as oil barrels. Things at the

information paradigm level are more fundamental and do not vary from business to business. For example, within the entity paradigm, both securities trades and oil barrels are business entities. This division into two levels makes the re-engineering more straightforward. It falls neatly into two corresponding steps as shown in ***Figure P.4***.

First, we re-engineer the entity paradigm into the object paradigm (the first half of this book describes how this is done). It is worth noting that this first step does not, by itself, change the way we see the business. At the end of step one (shown in ***Figure P.4***), the entity business paradigms no longer have a foundation. So, in step two, we re-engineer them into object-oriented business paradigms. These are built upon the new object paradigm that we re-engineered in step one. It is at this stage that we start seeing the business in a radically different way. Part Six contains worked examples showing how this is done.

Figure P.4: Two re-engineering steps



The advantage of this division into two levels is that it separates the information foundations, which only need to be re-engineered once, from the business paradigms, which need to be re-engineered for each business (because they vary from business to business). This means we can re-engineer the information foundations once and for all in this book. With the new foundations in place, you only need to consider their business paradigm levels when you start to re-engineer your existing systems.

4 What are thebenefits of re-engineering business paradigms?

Once you understand what business objects are, it is relatively easy to re-engineer the business paradigm level of existing systems. But what are the benefits of doing this? When people start re-engineering their business paradigms, the superiority of the object foundations soon becomes apparent. The final business model is not only functionally richer than the original system, but it is significantly simpler and more compact.

4.1 Greater explicitness, increased accuracy and more re-usable

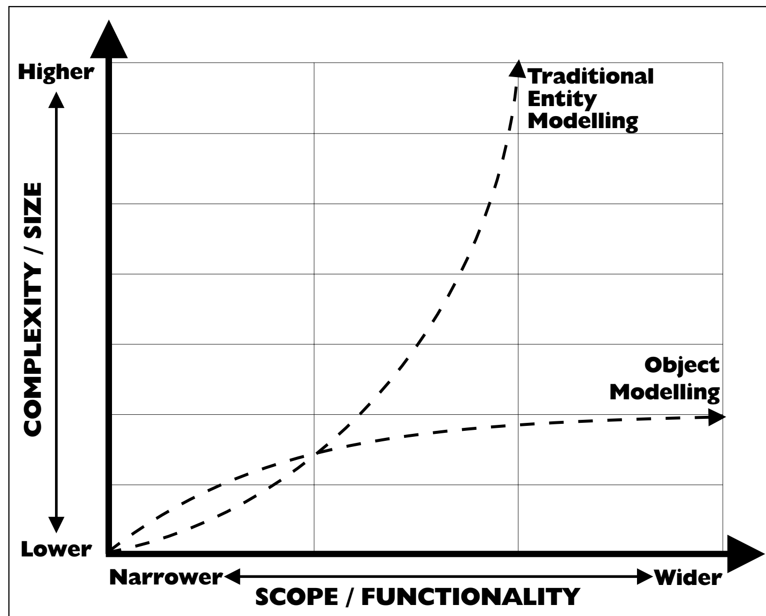
When we started to re-engineer business paradigms, we found that we were constructing a model whose business patterns were both more explicit and more accurate. One important result of capturing more of a pattern explicitly and capturing it more accurately is that it becomes more re-usable. This turns out to be part of a general trend towards greater accuracy in most engineering disciplines. In ***Chapter 1*** we look at how similar increases in accuracy in manufacturing engineering enabled the development of interchangeable (in other words, re-usable) parts.

© Copyright Chris Partridge chris.partridge@borocon.com

4.2 A substantially more compact business model

As we continued to re-engineer, we began to realise that not only does the object paradigm enable substantially more compact models, but the larger the scope of the re-engineering the greater the compacting. This is very different from traditional entity modelling (and computer system building). There, when the scope is increased, the overall complexity of the system increases. Each new pattern has to be harmonised with the existing patterns. Each time the scope increases, the task of harmonisation gets more onerous. The traditional rule of thumb is that the more patterns the model contains, the greater the cost of harmonising each new pattern (because there are more patterns to harmonise with).

Figure P.5:
Increases in scope



Business objects handle increases in scope in a very different way. Each new pattern, instead of adding to complexity, provides an opportunity for compacting a number of patterns into a single, more general, pattern and so creating a simpler model. Adding additional new patterns creates further opportunities to compact, generalise and simplify the model.

The more effective way in which the object paradigm deals with increases in scope is illustrated in ***Figure P.5***. Part Six provides examples.

5 Summary

To briefly summarise this Prologue:

- Underlying the approach taken in this book is a core issue—how can we accurately reflect the business in a model and so in a computer system? This book explains how business objects can tackle and resolve this issue

by changing the way we see things.

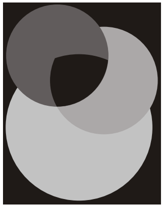
- We currently see things in terms of entities and attributes. These were developed for use with paper and ink technology and are unsuited for computing technology. Business objects, by contrast, can take full advantage of computing technology's potential.
- We need to re-engineer our business entities into objects. This is done in two steps; a re-engineering of the current entity information foundations to object foundations and then a re-engineering of the business paradigms from entity to object foundations.
- We undertake the first step here in this book. When you re-engineer your existing systems, you only need to undertake the second step—re-engineering the business paradigms.
- The re-engineering brings enormous benefits. It enables better business models—and so computer systems—to be built. These are simpler, more compact, more explicit, more accurate and more re-usable.

To build these models, you need to understand what business objects are and learn how to apply them. The second half of this book helps you acquire the skills in applying business objects by taking you through a number of worked examples.

We now move on to the first half of the book, which gives you the understanding of business objects you require by re-engineering the current entity paradigm into the object paradigm.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

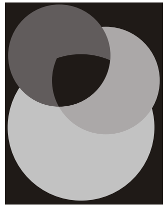


Part One

Our Strategy for Re-Engineering Entities into Objects

Chapter 1 What and How We Re-Engineer
Chapter 2 Focusing on the Things in the Business

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 1

What and How We Re-Engineer

1 Introduction
2 What do we re-engineer?
3 How do we re-engineer? With thought experiments
4 The benefits re-engineering brings
5 Re-engineering entities into objects

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction

In the Prologue, we divided the re-engineering of our business paradigm's entities into two steps— as illustrated in ***Figure 1.1***. Step one is a re-engineering of the entity foundations (in other words, the entity paradigm). Here, in Part One, we start by looking at our strategy for this re-engineering. Then, in Parts Two through Five, we implement it. Finally, in Part Six, we move onto step two and look at how we re-engineer our business paradigms.

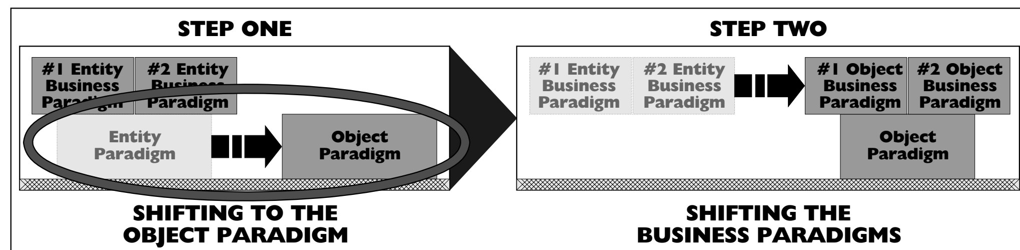


Figure 1.1: Re-engineering the entity paradigm

Part One covers the strategy for the re-engineering of the entity paradigm in two chapters. In this, the first, we look at our approach, addressing three questions:

- What do we re-engineer?
- How do we re-engineer?
- What benefits does it bring?

In the next chapter, we focus in on exactly what we will re-engineer—the things in the business.

2 What do we re-engineer?

What do we re-engineer? We ask the question at two levels, so, we get two answers—paradigms, and fundamental particles.

At the top level, we re-engineer paradigms. Seeing what this involves helps us to understand what is going on. At a lower level, we re-engineer the fundamental particles from which the paradigm is built. We look at these two levels in the following sections.

2.1 Paradigms

Seeing what happens when we shift from one paradigm to another is the best way to understand what a paradigm is. In the ***Prologue***, we made the point that this involves fundamental changes to the way we see things. But until you have actually been through the experience, it is difficult to appreciate that just seeing something differently can have a fundamental impact. However, we can use an analogy between ambiguous pictures and paradigm shifts to get a feel for what is going on.

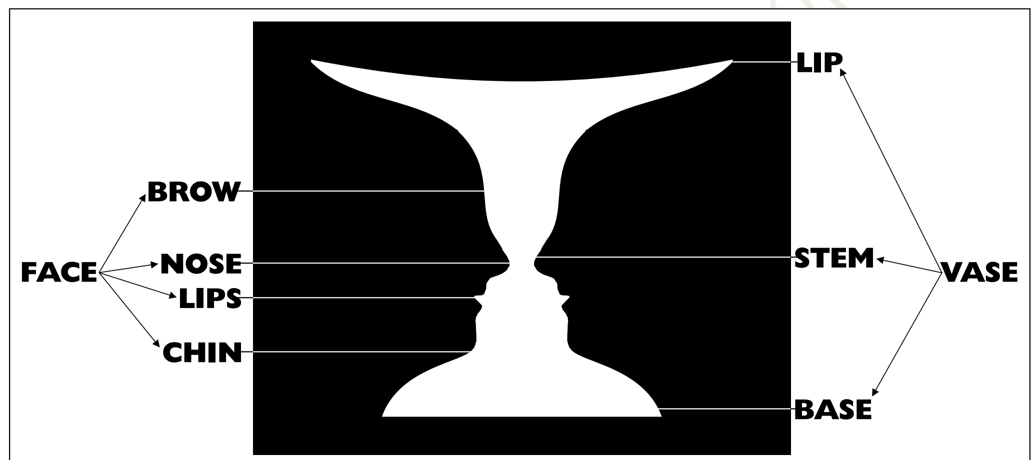
2.1.1 A re-engineering analogy

We use the well-known ambiguous picture in ***Figure 1.2***, not just to give us a feel for what is going on, but also to counter two common misconceptions. Most people tend to assume that:

1. Different views of the same thing must somehow be basically similar, and
2. Seeing something differently involves changing the thing.

Neither of these is true for the ambiguous picture. Its two views are not at all similar and, despite this, nothing in the underlying picture has changed. What changes when we shift from one view to another is how we see the underlying picture. The picture itself remains the same.

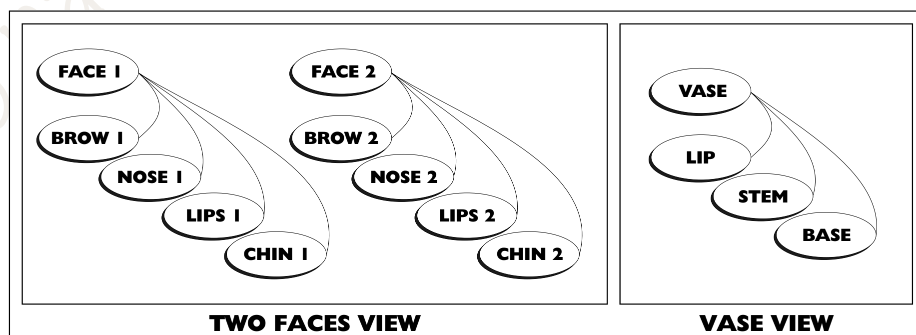
Figure 1.2: Two views of the same underlying picture



Let us look at what is going on in more detail. Assume I start by seeing two faces and then I switch to seeing a vase. When I switch, I have to dismantle my image of two faces and then construct an image of the vase. When I do this, the picture does not change. Nevertheless, I start to see the same picture in a radically different way.

We can get some idea of how different each perception is by looking at the way the two views classify the parts of the picture —at their semantic structure. This is mapped in ***Figure 1.3***. The two structures are so different that the only elements with similar names ('lip' and 'lips') refer to different parts of the picture.

Figure 1.3: Map of the semantic structure of the two views



©Copyright Chris Partridge christopher@...com

Fundamental paradigm shifts work in a way analogous to this ambiguous picture. Our current paradigm imposes one view on the world. The shift to a new paradigm leads to a radically different way of seeing exactly the same world. Like the picture, we can only see the world through one paradigm at a time. But unlike the picture, where we can shift back and forth between the two views at will, a paradigm shift is normally one way—from the old to the new. This is because when we see the new paradigm's world view, we recognise the faults of the old one. The new paradigm then appears obviously better: we are not tempted to shift back.

The analogy holds in another important way. Like the picture, a paradigm shift involves a substantial discontinuity. Intuitively we tend to assume that two views of the same underlying thing must be similar, but the opposite is true of both the picture and paradigm shifts—they are completely different. This explains why they have the potential for delivering enormous leaps in performance.

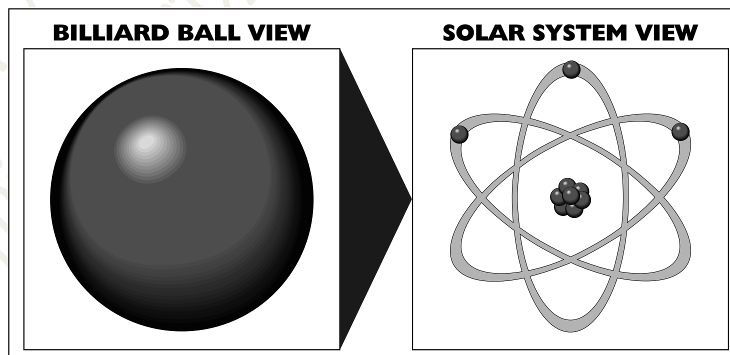
2.1.2 Radical changes lead to radically different questions—an example

When we start seeing something in a different way, we ask different questions about it. For example, we would naturally ask different questions about the vase and two faces in the ambiguous picture. Many historical examples of paradigm shifts significantly change the questions people ask and so the way they think and behave. The following example from chemistry illustrates how this happens.

In the 19th century, chemists assumed that things were made out of indivisible billiard ball-like atoms. They assumed that the atoms of each particular element were indistinguishable and that the atoms of different elements had different weights. In this scheme, it made sense for chemists to devote a lot of effort into trying to calculate precisely how much the standard billiard ball atom of a particular element weighed. For example, they calculated chlorine had an atomic weight of 35.453.

When the paradigm for atoms shifted in the 1920s, under the new scheme of things, atoms were seen as miniature solar systems—so they had divisible parts (see the two views illustrated in ***Figure 1.4***). Chemists then began to look at elements in a new light. Instead of indistinguishable atoms, they began to see that some elements had a number of different types of atoms; each with different weights that they called isotopes.

Figure 1.4: Two views of an atom



© Copyright Chris Partridge
chris.partridge@BOROCentre.com

This, they realised, meant that their cherished atomic weights were not a fundamental property of the element's atoms but a fortuitous mixing of different isotopes. Chlorine's, for instance, was the result of a natural mixture of two isotopes, 35 and 37, in the ratios 75.33 percent and 24.47 percent. It was not a real property of the chlorine atom at all. Chemists then lost all interest in atomic weights and stopped trying to calculate them. They regarded all their previous efforts as irrelevant. The new way of seeing atoms had changed the questions they asked and so the way they thought and behaved.

2.1.3 Paradigms as holistic frameworks

One of the reasons that paradigms have such an influence on our thinking and behaviour is because they provide holistic frameworks for our knowledge. In other words, they offer consistent and coherent systems for seeing the world.

2.1.3.1 Unambiguous views of the world

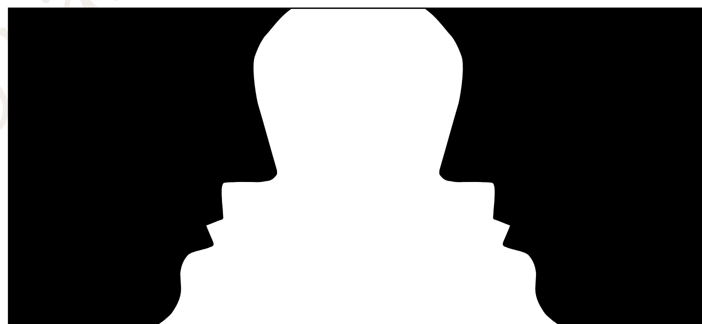
A key function of these holistic frameworks is to give us an unambiguous view of the world. The fact that we can view a picture in a number of ways (at different times) implies that the picture in itself cannot determine what we see. However, we can deal with a picture much more efficiently if we have an unambiguous view of what it is. That is why our brain naturally imposes such a view and why we only see one view at a time.

The same principle operates with paradigms. Our knowledge of the world is ambiguous. So our brain uses a paradigm to give us a particular unambiguous view. The paradigm makes us feel that this is the only natural view of a situation. Most of the time, it is so successful that we find it difficult to accept that our view is only one of many possible interpretations.

2.1.3.2 Needing the whole holistic picture—an analogy

In a holistic framework, the whole is more than the sum of its parts. For a paradigm, this means that we do not see its parts until we have seen the whole. In other words, we can only see the elements that make up a paradigm as its parts in the context of the whole paradigm. This sounds odd, but we can illustrate what it means with another picture analogy. Consider the picture of two faces in ***Figure 1.5***. This is unambiguous. It does not look like two vases, or indeed anything other than two faces.

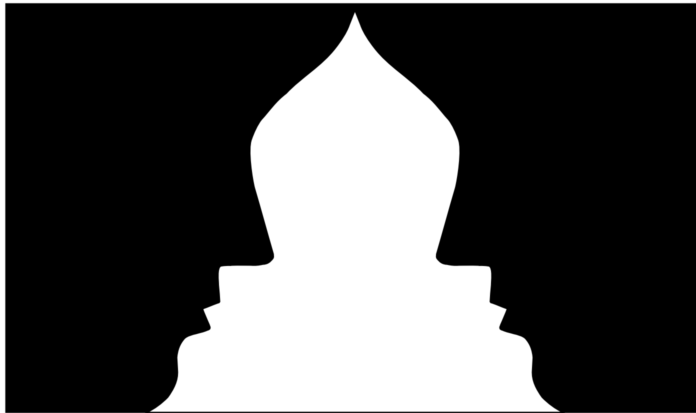
Figure 1.5: Another two faces



© Copyright Chris Partridge
chris.partridge@BorOCentre.com

Now look at the picture in ***Figure 1.6***. It is ambiguous; we can see it as either a mosque's minaret or two faces. You may have noticed that ***Figure 1.5*** is the same picture as Figure 1.6, with the top section removed. This means ***Figure 1.5*** must contain most of the elements that make up Figure 1.6's minaret view. When we first saw it, however, we did not see two-thirds of a minaret. This is because we need to see the whole minaret pattern before we can recognise a part of it. ***Figure 1.5*** does not have enough of the pattern to make up a whole minaret, so we do not see one. This means we also did not see the elements of ***Figure 1.5*** as parts of the minaret.

Figure 1.6: Two faces or a mosque's minaret



We need to see the whole pattern before we can see the parts. Now that we have seen the whole minaret pattern, we can look at ***Figure 1.5*** and see its elements as parts of the minaret.

2.1.4 Difficulties in seeing a new paradigm

The ease with which we can shift from one view of an ambiguous picture to another may seem to imply that shifting paradigms is just as easy. Unfortunately, this is not so. When we start re-engineering, we shall find all sorts of difficulties.

Paradigms, by their nature, do not encourage re-seeing, re-thinking and re-inventing. Their task is, as we said earlier, to make us see one unambiguous view of things. This makes them difficult to re-engineer. The features of a paradigm that are strengths when dealing with everyday tasks tend to become barriers to a successful re-engineering.

In everyday use, a paradigm's strength comes from enabling us to accommodate the new patterns we meet into its framework. This becomes a problem when we start re-engineering. Then we often need to recognise when a new pattern does not properly fit in with our current paradigm. This is what starts the re-engineering process rolling.

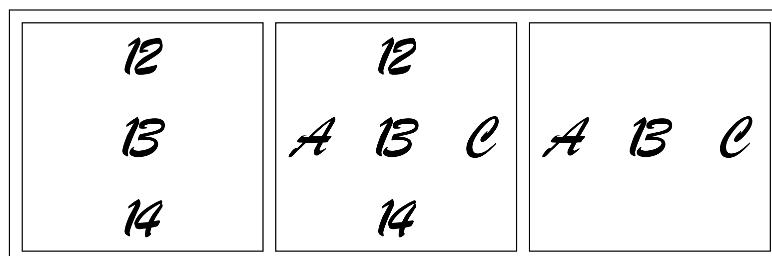
However, our current paradigm tends to make it difficult for us to recognise this. It trains us to see the world in a particular way. It also, by default, trains us not to see the world in the way we need to for the new paradigm. Instead, we see the new patterns that should provoke a re-engineering in the existing paradigm's terms.

© Copyright Chris Partridge chris.partridge@BORO Centre.com

“ƒSWe use another picture analogy to illustrate how this works. ***Figure 1.7*** has three boxes based upon those used in a common psychology experiment. These are shown to people one box at a time, and they are asked to say what they see.

The experiment reveals that what people see in the first box affects what they do-and do not see in the later boxes. When people are shown the box on the left first, they see the sequence of numbers 12–13–14. If they then look at the middle box, they still see the middle character as ‘13’. Surprisingly, a number of people, when looking at the box on the right, still see the middle character as ‘13’. Recognising the 12–13–14 pattern in the first box has stopped them seeing the A–B–C pattern in the second and, in some cases, the third box.

Figure 1.7:
How what we see first affects what we see later



This is not because people find it easier to see numbers than characters. This was proved by repeating the experiment in the reverse order, showing the box on the right first. People then start off seeing the sequence of letters A–B–C. This sets the pattern. So when they then look at the box in the middle, they see the middle character as ‘B’. And again when they look at the box on the left, some people still see the middle character as ‘B’, in other words, a sequence 12–B–14.

In both cases once people grasp the first pattern, they have some initial difficulty in seeing an alternative pattern even when the original one is incomplete (as in the last box). This gives us some idea of how difficult it is to see a new pattern that is ruled out by the current pattern. It also gives us an idea of how difficult it can be to re-engineer when the old paradigm trains us not to see the pattern we need to recognise for the new paradigm.

2.1.4.1 Germ paradigm—Pasteur example

This picture example is not just an academic psychological trick. In such practical disciplines as medicine, paradigms have trained doctors to see new patterns of disease as part of an old pattern, sometimes with deadly results. Consider, for example, what we shall call the germ paradigm.

In the 19th century, the French scientist Louis Pasteur (1822–1895) developed an understanding of germs (micro-organisms) and a recognition that these played an important role in disease. He used this knowledge to help the French beer, wine, and silk industries. He also used it to improve people’s health, developing vaccinations against anthrax and rabies. His and other scientists’ successes with the ‘germ paradigm’ led to a belief in the medical profession that, if a disease was not caused by a par-

asite, it must be caused by a germ. They assumed that the answer to the question—'What is causing this disease?'—involved either parasites or germs.

This acceptance of the germ paradigm eventually led to problems. We now know that some diseases are caused by a deficiency in diet (and not germs or parasites). One of these diseases is beriberi. In the early years of this century, there was an epidemic of beriberi in Asia that killed millions of Chinese and Indonesians. The germ paradigm was so deeply embedded in the medical establishment's thinking that they unconsciously and unthinkingly assumed that the beriberi epidemic was caused by germs and carried out their research accordingly.

Eventually, dietary experiments by the Japanese navy challenged this assumption. These helped to prove that it was not the presence of germs that caused the disease, but the absence of something in the rice people were eating. It was then discovered that the new processes of steam-polishing rice, imported from Europe to Asia, destroyed the vitamin B_1 in the hull of the rice. It was the lack of this vitamin B_1 that was causing the beriberi epidemic.

Intriguingly, a leading professor of tropical disease at that time, Patrick Manson, did not accept the new paradigm, despite all the evidence. He insisted on interpreting the Japanese navy findings in a way consistent with the old germ paradigm. He claimed that the germs that caused the disease can and do live in the polished rice but cannot live in the unpolished rice. His training in the old paradigm was so strong that he was seeing the new patterns in its terms. At that stage the 'new pattern' (in other words, the results of the Japanese navy's experiments) could be interpreted to support either theory. Only later on did it become clear that the germ paradigm was not a helpful way of looking at beriberi.

In a more modern medical context, some 'rogue' scientists are suggesting that AIDS researchers might be thinking and behaving in a similar way. They think that AIDS researchers might be stuck with a 'virus paradigm' that directs them to only look for a virus as the cause for AIDS. Their concern is that this may be making them ignore alternative patterns that might turn out to be more fruitful.

In a computing context, we can see something similar happening in O-O programming. When an O-O programming language (OOPL) is introduced into a traditional programming environment, programmers trained in traditional programming often still use the traditional patterns to program in the new language. They have been taught to see and ignore other patterns. These other patterns include those they need to see to make effective use of the new OOPL. As a result, they have some difficulty learning how to work with it.

2.2 What do we re-engineer? The fundamental particles of paradigms

We have seen that the answer to the question—'What do we re-engineer?'—at the top level is paradigms. We now ask this question at a lower level. The answer this time is fundamental particles. Paradigms are often built around one or more central patterns or particles. When this happens, a fundamental re-engineering usually involves changing those particles.

2.2.1 Re-engineering information's fundamental particles

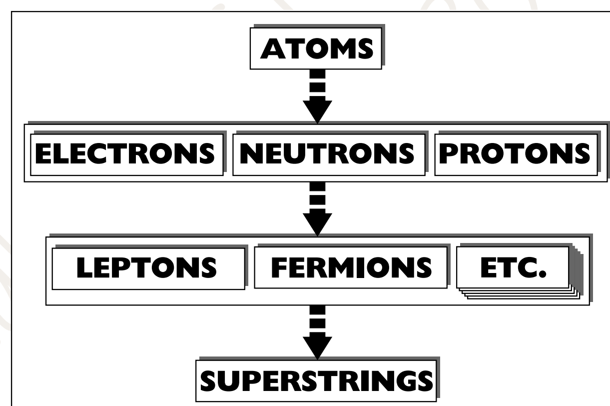
There is a close analogy between the way physical matter paradigms, such as the atom paradigm in the earlier chlorine example, work and the way information paradigms work. We now pursue that analogy.

2.2.1.1 The physical matter paradigm analogy

Physics explains the world in terms of its physical matter paradigm. The most fundamental patterns in this paradigm are physical particles. These are the building blocks from which physicists construct their world. They started the 20th century with a paradigm in which the atom was the fundamental physical particle. While they subscribed to this paradigm, they believed everything—from aardvarks to zebras—was made of indivisible atoms.

Since then, physicists have re-engineered the physical matter paradigm a number of times—each re-engineering is characterised by a complete change of fundamental particles (shown in ***Figure 1.8***). When physicists divided the atom, they introduced a whole new family of fundamental particles: electrons, neutrons, and protons. When they put these into their enormous particle accelerators, they found (and so shifted to) a profusion of new types of particle—things such as leptons and fermions. Their latest paradigm is less prolific; it has a single type of fundamental particle—superstrings.

Figure 1.8: Shifts of fundamental physical matter particles



2.2.1.2 The information paradigm's particles

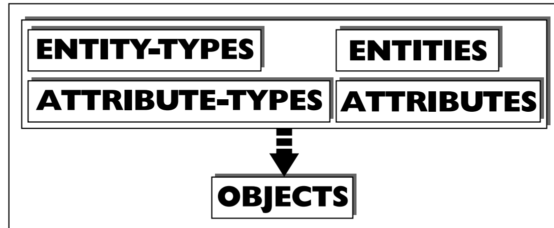
Information paradigms work in a similar way. Just as physical matter paradigms have fundamental physical particles, so they have fundamental information particles. When we use the paradigm, we use these particles to build up our picture of the world. For instance, the entity paradigm has four explicit fundamental particles: entity types, entities, attribute types, and attributes. When we use the entity paradigm, we build up our picture of the world using these four particles.

Re-engineering our entity paradigms, like re-engineering physical matter paradigms, involves a radical shift of fundamental particles. When, in Parts Three and Four, we re-engineer to the object paradigm, we shall see these fundamental particles change. We

© Copyright chrispartridge.net

will start with the entity paradigm's four particles and end up with the object paradigm's single particle (shown in ***Figure 1.9)***. This is a similar pattern of changing particles to the re-engineering of the physical matter paradigm illustrated in ***Figure 1.8***

Figure 1.9: Shifting fundamental information particles



2.2.2 *Recognising that business models have fundamental particles*

Some people initially find it difficult to think about fundamental particles and how we use them to see the world (or business). It gets too close to the foundations of how we see the world. Sometimes, people in the computer industry also succumb to the feeling that somehow the notion of fundamental particles does not apply to business models.

For example, most people working with computers would accept that one must be conceptually accurate when talking about computer code (computer software system's fundamental particles). However, some of them are less happy about being accurate when talking about the business's particles. They are not sure whether the things in the business are objects or entities; for example, is a car a car object or a car entity. They probably feel that this is not particularly relevant to them. They certainly do not think that their talk about business things commits them to any particular type of thing and certainly not any type of fundamental particle. For example, if they were to put a sign for a car entity in their business model, they would not feel that this commits them to classifying the car as an entity. Or, that it commits them to having entities as their fundamental particles.

This attitude may be appropriate for casual conversation, but is quite harmful when doing something formal, such as business modelling. If we unconsciously use an entity approach to business modelling then, whether we like it or not, we are seeing the business in terms of entities and attributes. Ontology, the branch of knowledge that studies fundamental information particles, calls this 'ontic commitment'. Until we realise how crucial this 'ontic commitment' is, we will not be able to start the re-engineering process.

We might not be conscious of making this ontic commitment when we build systems because we are focusing on technical problems. But it is still there, happening at a subconscious level. The problem with leaving these kinds of decisions to the whims of our subconscious mind is that our ontology (in other words, our scheme of fundamental particles) tends to end up as a confused hotchpotch. People may be able to muddle through system building with a confused ontology, but they are missing out on an enormous opportunity. To take advantage of it, they need to make accurate 'ontological' decisions about types of business things during business modelling.

Some people might think we can avoid this 'ontic commitment' by leaving out the business modelling stage altogether. But they are fooling themselves. As soon as we start

© Copyright Chris Partridge chris.partridge@BORO Centre.com

talking about business things—which we have to do at some stage—we have committed ourselves. So even if we start our system building by coding, we still make an ontic commitment. A system whose computer code refers to things in the business—even apparently innocuous things like ‘company’, ‘date’, or ‘amount’—is clearly committed to those things’ existence. And they are of a certain type: entities, objects, or something else. There is no way of avoiding this.

2.2.3 Fundamental particles versus complex business objects

We may now accept that, when we model, we commit ourselves to some kind of fundamental particle of business information. But people often succumb to another feeling—one that says thinking about these fundamental particles is a waste of time. They feel more benefit is to be gained from coming to grips with complex business objects. (In the financial sector a complex object would be something specialised, such as a ‘reverse repo’—a complex deal with a number of elements.)

What they (and we) need to recognise is that the only way to transform apparently complex business objects, such as reverse repos, into simple ones is to start with their fundamental particles. For most people, the problem is getting our ideas about complex objects into shape seems to have an obvious benefit. Whereas, the benefit of getting their fundamental particles right is not so obvious.

2.2.3.1 Building construction analogy

Another analogy, this time an engineering one, should help us see more clearly why starting with fundamental particles rather than complex business objects brings much bigger benefits. If we look at the history of building construction, we can see that, at each stage of its development, the nature of its fundamental particles placed a limit on what could be built. (These particles are relatively easy to spot because they are literally physical building blocks.) History shows that shifting to new and better particles has led to big improvements.

A long time ago when most buildings were made out of mud and straw, we could say the builders had a mud paradigm. While the buildings were attractive, and in hot dry countries practical, it was technically difficult, often impossible, to construct a building much higher than two stories. Mud (the fundamental particle) just did not have the strength for it.

Then builders discovered that once mud is baked in a kiln to produce a brick its strength increases substantially. Buildings with ten stories became feasible using this new, stronger, brick particle. But bricks have their limit. They cannot support the skyscrapers we see in most major city centres. These use a different, stronger, building block—re-enforced steel and concrete. With this new ‘particle’, buildings reaching up to the clouds can and have been built.

It is plain that the stronger the particle, the taller the building can be constructed. If someone had not worked at improving the fundamental particle, we would not be able to construct the tall structures we have today.

© Copyright Chris Partridge
chris.partridge@BORO.centre.com

A similar analogy can be made with the way in which we talk of early human civilisations. We talk of a Stone Age followed by the Bronze and then the Iron Age. These names refer to the material (the fundamental particles) used to make tools. The nature of the 'particles' clearly had an enormous influence on the overall nature of the civilisation. Advances in material (particles) led to substantial advances in technology.

The fundamental particles used in the information paradigm work in the same way. We can only build strong powerful computer systems if we use strong powerful particles. These are not physical, like building materials. The physical problems of building computer hardware are reasonably well understood. Engineers are having enormous success developing better hardware without a fundamentally new physical particle.

The fundamental particles of an information paradigm are more like ideas than physical building materials. Most system builders are now using entity and attribute particles (ideas). However, they are finding that these particles do not match up to the task of building very complex business systems—just as house builders found their mud 'particles' were not strong enough for tall houses. When they try to build complex business systems, they have to put in a substantial effort and, even then, often fail.

They need stronger and more powerful particles than entities and attributes. With a better information particle, such as business objects, they will have more success building these very complex systems. When we look at the problem in this way, spending time improving the fundamental particle, by re-engineering entities to objects, is not a waste of time. In fact, it is probably the only practical and sensible way to deal with the situation.

3 How do we re-engineer? With thought experiments

How do we re-see and re-think the entity paradigm's fundamental particles? What tools do we have to help us? If we were scientists trying to find new facts about the world, we could conduct physical experiments with test tubes or pulleys or whatever in our laboratory. But here we do not want to find new facts; we want to re-see and re-think existing ones. We do this using thought experiments—a kind of mental analogue of the physical experiments—scientists do.

3.1 How to do a thought experiment

Physical experiments involve carefully observing something happening, often in a laboratory. Typically, the experimenter predicts what he expects to happen and sees whether it actually does. Thought experiments are similar but they involve no physical observation whatsoever, merely thinking or mental observation. This means that they do not need a laboratory. These are the kind of experiments that can be performed in an easy chair.

© Copyright Chris Partridge chris.partridge@BORO.com

3.2 Principles of a thought experiment

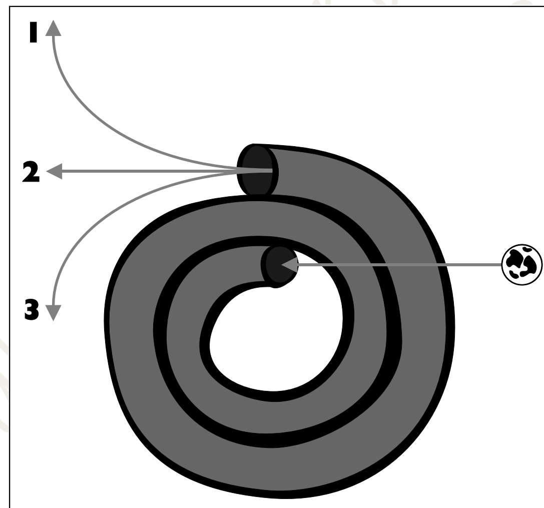
A thought experiment works by first making an inconsistency in a paradigm explicit. Highlighting the inconsistency engenders a distrust of the paradigm. Then, the thought experiment demonstrates how the new paradigm neatly gets around the inconsistency—clearly showing its superiority.

A typical experiment works like this. We are asked to think about what we would normally expect to happen in a situation. This is chosen to highlight the superior coherence of the new paradigm. We are often also shown how our current paradigm leads us to expect two contradictory things to happen—as in the example below. At no stage do we actually have to do anything.

3.3 An example of a thought experiment

Here is a simple thought experiment. It has been used by psychologists to show how misleading our intuitions can be. Consider ***Figure 1.10***, which shows the apparatus for the experiment—an imagined piece of coiled-up tube and a marble.

Figure 1.10: Shooting a marble into a coiled tube



Now imagine what would happen if the coiled tube was laid flat on a table and a marble was shot at great speed into the inner end of the tube. We all agree that it would speed around the coils of the tube and come out fast at the outer end of the tube. The question is:

What direction does it go in once it has left the tube?

Is it one of the directions marked in the figure or a different direction? Psychologists, who have done this test under controlled conditions, find most people (around 70 percent) chose direction 1. This includes physics graduates who have been taught the laws of motion and have a good understanding of what would happen.

© Copyright chrispartridge ... centre.com

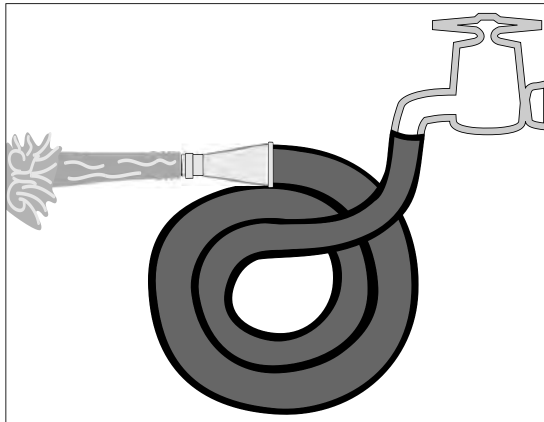


Figure 1.11: Water spurting out of a garden hose

To get our inconsistency, we do another similar thought experiment. This time, the experiment is conducted with a garden hose and water. We imagine water being pushed down a similarly coiled garden hose laid flat on the lawn. What direction do we see it emerging from the end? Everyone knows it gushes straight out—as shown in ***Figure 1.11***.

The two experiments have a similar pattern. They both involve something going through a coil and coming out the end at speed. Most people, when they recognise this, realise that there is a common general pattern and that direction 2 is the correct answer to the first thought experiment. When the thought experiment involves something as familiar as a garden hose then we can predict the results properly; the water goes straight out of the hose—not up or down. We then use the familiar pattern from the garden hose experiment to clear up our pattern for what happens in the marble experiment.

Interestingly, this example clearly shows how we still have some false ancient intuitions deeply rooted in our minds. Ancient and medieval physics predicted the marble in the first experiment would travel in direction 1, the common choice for 70 percent of people today. By coincidence, this physics is based on the work of the Ancient Greek Aristotle. (The substance paradigm, as we will see in ***Chapter 3***, is also based on his work.)

Yet, Aristotle's way of thinking has been scientifically out of date since the 17th century when Newton discovered his laws of motion. The hose and the marble experiments are actually both direct applications of his first law of motion:

A body continues in its state of rest, or uniform motion, unless acted upon by some external force.

When the water (or the marble) leaves the nozzle, it is moving straight forward. The other external forces acting upon it (gravity and friction) are too small to be relevant for the first few inches of movement and so can be ignored. Because there is effectively no external force, the water (and marble) should move with a uniform motion—in other words, in a straight line.

Since the 17th century, physics has predicted correctly that the marble would go in direction 2. It is just that these laws have not fully worked their way into everyone's minds. We shall see a similar situation in Part Two with Aristotle's ancient substance

© Copyright Chris Partridge chris.partridge@BORO-centre.com

paradigm. Most people still use it because later developments have not worked themselves into their minds.

3.4 Einstein's thought experiment

The thought experiment is a powerful tool for re-engineering paradigms. That's why scientists often use it to explain their major shifts. Even sophisticated modern paradigm shifts are often explained using simple thought experiments. Albert Einstein's theory of relativity is a good example. This is an extremely sophisticated theory (paradigm). It is so sophisticated that when Einstein published his results, most of his contemporaries had great difficulty in understanding them. Yet, he explained his theory of relativity using a simple thought experiment with such everyday objects as a moving train, bolts of lightning, and a couple of people to observe what was going on.

Thought experiments like these have been, and will continue to be, a natural and useful tool when re-engineering. They help us re-see, re-think and re-invent. You will come across a number of them in our re-engineering.

4 The benefits re-engineering brings

Re-engineering to objects creates a foundation for the re-engineering of business paradigms. I have found that together these bring two main benefits. They enable:

1. More accurate patterns, and so functionally richer systems, and
2. More compact patterns, and so simpler systems.

4.1 More accurate patterns, functionally richer systems

Re-engineering business paradigms enables us to construct more accurate, functionally richer business models. Working with business objects is like working with a powerful microscope. It enables us to see the real world more accurately. This, in turn, enables us to spot functionally richer, re-usable business objects.

In general, the more accurately a model reflects the world, the more powerful it is. This is true of most models, not just business models. Engineers testing a new car or aircraft design in a wind tunnel make the model accurate enough to reflect how the real car or aircraft would behave.

The less accurate a model, the less powerful it is. Imagine the model of a battle drawn up on a dinner table by a Colonel Blimp. The salt cellar is the advancing enemy army and the butter dish is a hill. This model has its uses, but these are limited by its inaccuracy. For example, we would not even think of saying that because the salt cellar cannot stand on top of the sloped butter dish, the enemy army would not be able to take the hill it represents. We know the model is not an accurate enough representation of the situation.

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

If we wanted to know what the enemy army could or could not do, we would need a more accurate model. The models built using our current entity paradigm are like Colonel Blimp's model in that they are not accurate enough for any heavy duty work. Whereas, business object models with their increased accuracy are.

4.1.1 The cost and benefits of accuracy

The traditional attitude in system building, based on the current entity paradigm, is that increasing accuracy leads to spiralling increases in costs. This is not the case with objects. I have found (and we shall see in the worked examples of Part Six) that the more accurately we model the business, the simpler, more general and so re-usable the objects are. As the accuracy of the model increases so does the potential for generalisation and re-use of its objects. These more accurate objects can then be compacted into less space than their less accurate predecessors.

This means that, within the object paradigm, the traditional rule that increased accuracy leads to increased cost is turned on its head. The new rule is increased accuracy leads to increases in re-use and so reductions in cost.

There are parallel situations of accuracy assisting re-use in a number of engineering disciplines. It may help us to appreciate the part accuracy plays in information engineering if we look outside computing at the broader picture. Information engineering for computers is a new discipline. It does not have enough of a history to give a feel for how accuracy works. If we look at accuracy in an older, more mature, engineering discipline, we can get a better idea. Manufacturing is a good example because it has a kind of physical analogue to information re-use—interchangeable parts.

4.1.2 Manufacturing accuracy and re-use

Physical accuracy played an important part in the industrial revolution of the 18th century. This is particularly clear in the introduction of interchangeable parts, a kind of re-use that revolutionised manufacturing. We are nowadays so used to interchangeable parts that we find it difficult to imagine what a world without them would be like. We expect a new wheel to fit onto a car; we expect a new plug to fit into a socket. This seems to us the natural order of things. Before the industrial revolution, things were very different. Parts were not interchangeable; they were individually hand crafted. An axle was made to size for the specific pair of wheels on a specific cart. It could not be re-used, without further work, in another cart.

With physical things, such as axles and wheels, it is clear that they are only interchangeable if they are made to a certain level of accuracy. This level just could not be systematically achieved in manufacturing until the 19th century. Before then, the levels of inaccuracy that were tolerated seem astonishing to us. For example, in James Watt's steam engine (built in the 18th century) a sixpenny coin could easily fit between the piston and the cylinder.

The American inventor Eli Whitney (1765–1825) developed the first working system for manufacturing interchangeable parts. He was motivated by the potential benefits of mass production. If he could make interchangeable parts then he could make the parts

© Copyright Chris Partridge chris.partridge@BORO Centre.com

en masse separately and assemble the whole product quickly and easily later on. He sold the American Congress on his idea that guns could be mass produced this way. He explained to them that he was going to machine his gun parts so accurately that his workers could assemble a gun from the first parts that came to hand. They would no longer have to tailor them to the individual gun. Congress gave him a government contract in 1798 to produce 10,000 army muskets, all with interchangeable parts. (This can be seen as an early example of military spending encouraging research and development.)

Whitney found the task more difficult than he had anticipated and took longer than planned; but, in the end, he was successful. He is said to have demonstrated his success dramatically. The story goes that he threw a box of the interchangeable parts at the feet of a government inspector and told him to make a musket from parts picked at random.

A colleague told me of a similar public demonstration arranged by his grandfather Frederick S. Bennett. Bennett was the British agent for the American car manufacturer Cadillac. In 1908, he arranged for Royal Automobile Club engineers to demonstrate that all the parts of a Cadillac car were interchangeable. They selected three new cars from their crates and took them completely apart—nut from bolt, piston from rings. The pieces were then put in a heap and thoroughly jumbled up. When the cars were reassembled, they started up the first time. Then, this was seen as a great feat.

This was an American achievement. Even as late as the Second World War, the parts for British Army vehicles and equipment, unlike their American counterparts, were not properly interchangeable. Soldiers had to adjust them with hacksaw and file to make them fit. Nowadays, when cars are routinely assembled from parts bought in from different factories all over the world, this seems remarkably primitive.

One interesting feature of Whitney's achievement is that it was accomplished without plans or sizes for the component parts. When he first introduced mass production, he relied on manual labourers using what were called filing jigs. These were used as templates to hand-file parts for his muskets to approximately matching dimensions. Both the filing jigs and the manufactured parts were the product of manual labour and depended for their accuracy upon the skill of the workers. Furthermore, not one person could measure the accuracy of a part using a standard unit of measurement. All they could do was look and feel whether the part matched the particular jig being used; measuring accuracy was limited to unaided human perception. As a result, manufacturing interchangeable parts was not easy.

Joseph Whitworth (1803–1887) helped to resolve this problem by establishing common standards for accuracy that enabled plans and sizes to be specified for components. He did this by developing precision instruments that measured accuracy far beyond the limits of the unaided human eye. He gave engineers not only a common standard for 'seeing' how accurate a part was, but a standard way of describing, in advance, its accuracy. This gave manufacturing the framework it needed to effectively and efficiently make interchangeable parts.

Whitworth developed measuring instruments that were far more accurate than any earlier instrument. Some were even accurate to a millionth of an inch. To some of his con-

temporaries, this level of accuracy seemed academic—only suitable for use in the laboratory. Yet, nowadays it is commonplace. Indeed, in some industries, such as silicon chip manufacturing, it is insufficient.

With these standards of accuracy in place, the practical benefits of Whitney's system of interchangeable parts became apparent. And his idea soon spread from the arms business to farm machinery and then to almost all mechanical production. It became known as the 'American system' of manufacturing. As time went by, the system was improved. More and more accurate machine tools and measuring devices were developed. This eventually led to the staggering success of 20th century mass production. A system that Ford used, during the Second World War, to deliver a B–17 bomber (the Flying Fortress) off their American production line every sixty-three minutes.

4.1.3 Accuracy's role in the shift to business objects

Business objects are leading to an industrialisation of information in which accuracy plays an important part. Just as physical accuracy was needed to make interchangeable parts, so referential accuracy is needed to construct general and so really re-usable objects. For example, general objects are constructed from the patterns of lower level objects. We need to be sure that we have captured the patterns for these lower level objects accurately. If we have not, then the generalisation magnifies the lower level inaccuracies and does not work.

Our current entity computing paradigm, like the old individually tailored methods of manufacturing, cannot deliver the required levels of accuracy. Business objects (like Whitney and Whitworth's approaches) can. As it brings greater and greater accuracy, it delivers an increasing potential for generalisation and re-use. This helps to drive the industrialisation of information.

4.2 More compact patterns, simpler systems

Most people find it counterintuitive that a system can be made both simpler and functionally richer—especially just by using more accurate patterns. When working within a paradigm (such as the entity paradigm), it is reasonable to assume that a piece of information has a natural complexity. If it is made simpler, it contains less information. When re-engineering to business objects, we cannot make this assumption. The purpose of the re-engineering is to transform complex patterns into simpler more compact ones.

4.2.1 A simple example of compacting

We can clarify how this counterintuitive purpose works with a simple example of how a complex pattern can be re-engineered into a simpler, functionally richer pattern. Consider the nodes and arcs in ***Figure 1.12***. We can describe the figure as follows:

A is a node.
B is a node.
C is a node.
D is a node.

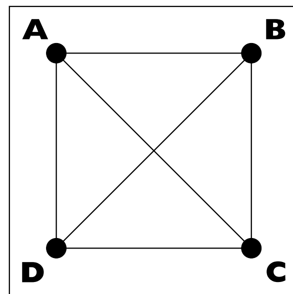
© Copyright Chris Partridge
chris.partridge@BoroCentre.com

Node A is connected by an arc to node B.
Node A is connected by an arc to node C.
Node A is connected by an arc to node D.
Node B is connected by an arc to node C.
Node B is connected by an arc to node D.
Node C is connected by an arc to node D.

This description has two basic patterns:
1. X is a node.
2. Node X is connected by an arc to node Y.

Pattern (1) occurs four times and pattern (2) occurs six times.

Figure 1.12: Nodes and arcs



Most of you, when you look at Figure 1.12, will 'discover' a regularity not highlighted by the description above. You will notice that arcs connect every node to every other node. We can capture this regularity in a pattern. We will call this the fully connected node pattern. A node is fully connected if it has arcs connecting it to all the other nodes in the figure.

If we shift to this new pattern, we can construct a more compact (more compressed) and structurally simpler description of the figure:

A is a fully connected node.
B is a fully connected node.
C is a fully connected node.
D is a fully connected node.

This description is much more compact; it has four lines instead of ten. It is also much simpler in that it only involves one basic pattern:
1. X is a fully connected node.

It is also richer than the first description. It explicitly recognises the fully connected nodes regularity.

4.2.2 A classic example of compacting

Because the previous example has been kept simple, it may seem contrived. But most paradigm shifts exhibit the same kind of compacting. Take, for instance, this classic example from the history of science. In the early 17th century, Johannes Kepler discovered that the planets moved in an elliptical pattern. Before Kepler, astronomers

© Copyright Chris Partridge chris.partridge@BoroCentre.com

assumed that they followed an epicyclical motion where one or more circles move on another. The elliptical pattern is structurally simpler than the epicyclic (shown in ***Figure 1.13***). It also gives a much simpler and more accurate overall theory of planetary movement. It has the same compacting characteristics as our simple example.

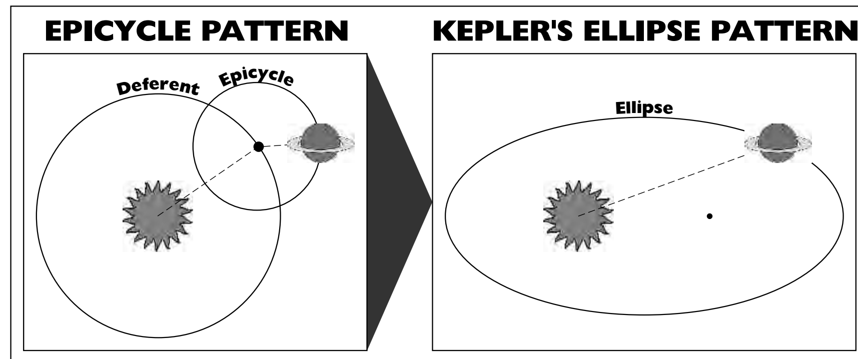


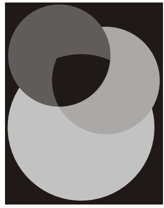
Figure 1.13: Epicyclical and elliptical patterns of planetary motion

5 Re-engineering entities into objects

The benefits of compacting and accuracy brought by the object paradigm make it a substantial improvement on its predecessor—the entity paradigm. The object paradigm is just beginning to change business modelling. As the change follows its course, business models will become substantially simpler, more compact, and more accurate.

In our journey from the entity to the object paradigm, we are going to follow the approach described here. We will use thought experiments to help us find the new patterns that undermine the old paradigm and start the re-engineering rolling. We will see how each re-engineering changes the paradigm's fundamental particles. We, no doubt, will find the entity paradigm hindering us from appreciating new patterns. We will also see how this new paradigm enables us to build simpler, more compact and more accurate models—and so, computer systems. In the next chapter, we sharpen the focus of our re-engineering.

© Copyright Chris Partridge
chris.partridge@BORO Centre.com



Chapter 2

Focusing on the Things in the Business

1 Introduction
2 Focusing the re-engineering on things in the business
3 Problems identifying ‘things in the business’
4 Ignoring ‘things in the business’
5 What types of things (in the business) do we re-engineer?
6 Our starting point—the entity paradigm
7 Arriving at an object semantics for ‘things in the business’
8 Re-engineering the ‘things in the business’
9 The next part

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction

In the previous chapter we looked at how we are going to re-engineer entities into objects. In this chapter we focus on the specific areas we need to re-engineer.

2 Focusing the re-engineering on things in the business

We focus the re-engineering on one area of the entity paradigm—the 'things in the business'.

2.1 The major elements of an information paradigm

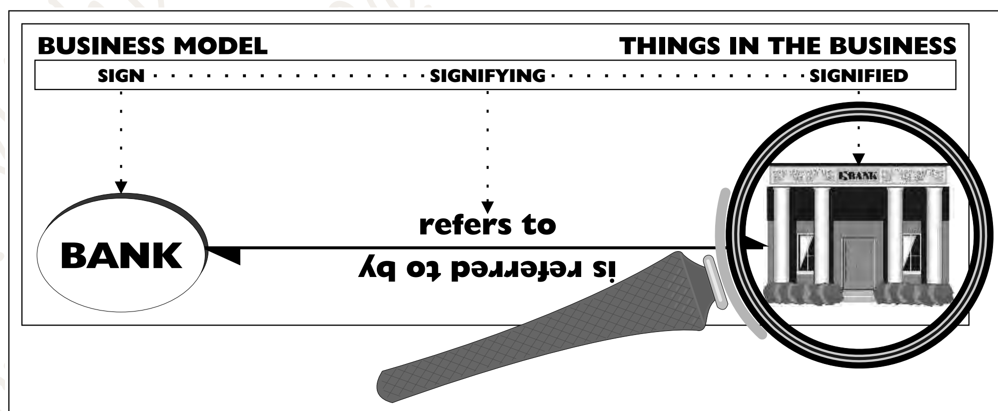
Information paradigms, such as the entity paradigm, are typically divided into the following three elements:

- Technology(or method and materials of construction),
- Syntax(or, structure), and
- Semantics(or, meaning).

Computer people naturally focus on the technology element. It seems indubitable that there is a fundamental change going on in information because of the new information technology—computing. And this technology is innovative and exciting. So it is not surprising that some people overlook the two non-technological elements.

2.2 Focusing on 'things in the business'

Figure 2.1: Focusing on 'things in the business'



However, it is in one of these non-technological elements, semantics, that we find the key to business objects—'things in the business'. Look at ***Figure 2.1***, which illustrates the semantic 'signifying relation' between the sign and the signified. It is an obvious truth that we cannot construct signs that reflect a business accurately unless we can

© Copyright Chris Partridge chris.partridge@BORO Centre.com

see the things in the business clearly. Our re-engineering focuses on developing a much clearer, more accurate, view of these.

3 Problems identifying 'things in the business'

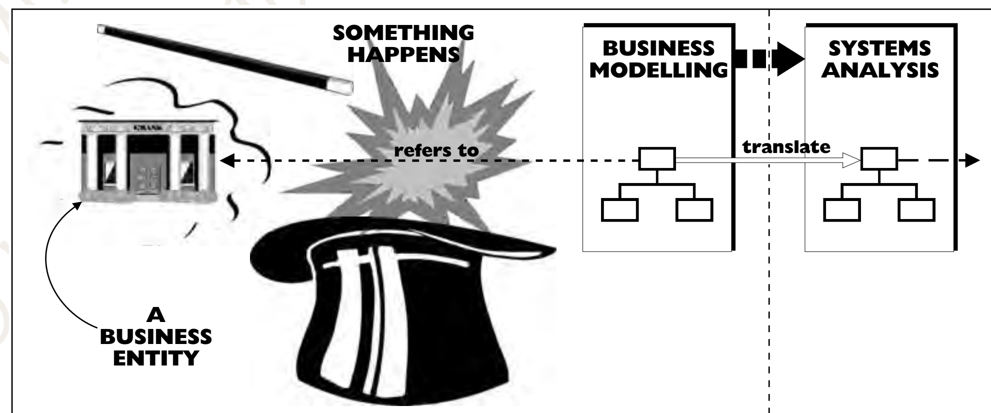
Most computer people currently assume it is easy to identify the 'things in the business' (the signified) and make sure that the model refers (maps directly) to them. However, if we actually try and identify the things in the business, we come across a problem. Astounding as it may seem, most people cannot clearly and explicitly articulate exactly what these entity 'things in the business' are.

3.1 The problem with our entity paradigm

Many people in the computer industry have lots of experience in constructing business entity models. They must know what a business entity is. One might think that if we asked them, they would tell us what one is. But when we actually ask them, they come up with examples not explanations. They say something like 'an entity is a thing like a car', or 'a company' or 'a foreign exchange deal'. They cannot provide an explanation because their understanding of entities is so deeply ingrained that it is unconscious. And their acceptance of it so complete that asking what an entity is seems to have no practical use.

Most people adopt a similar attitude. They instinctively assume that they and everyone else know what a business entity is. For instance, IT managers expect even the most lowly trainee programmers to come fully equipped with the knowledge of what a business entity is—though not necessarily what the particular entities are for their business. They expect this, despite the fact that the programmers are not consciously aware of what one is nor are they likely to be formally taught this.

Figure 2.2: Magical and mysterious semantics



Nevertheless, the managers' expectations are justified because it is plain from the trainee programmers' behaviour that they do know. It is as if something magical and mysterious is linking the model to the business entities—as shown in ***Figure 2.2***. It seems odd to me (and by the end of the book, it will seem odd to you) that companies

© Copyright Chris Partridge chris.partridge@BORO-centre.com

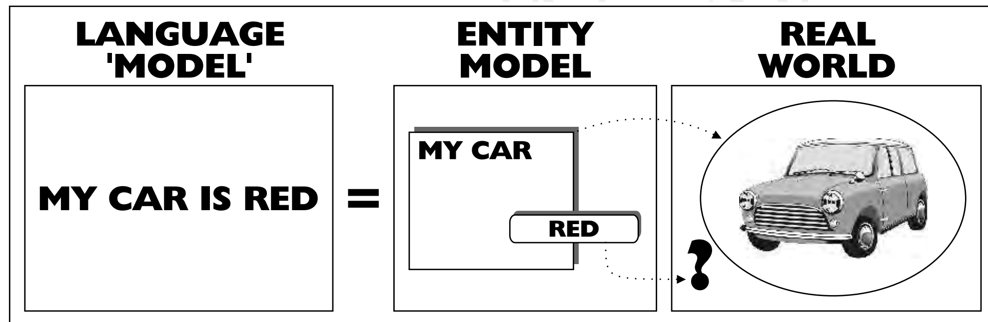
regularly spend millions of pounds building computer systems that depend on such mysterious semantics.

3.2 Problems finding 'things in the business' – a simple example

We can appreciate how mysterious our current entity semantics is by looking at this simple example. Consider what a simple entity model of the sentence, 'my car is red' would look like. It would have a 'my car' entity with a 'redness' attribute—as shown in ***Figure 2.3***. The entity 'my car' clearly refers to my car, as shown in the diagram.

My car's redness is more of a problem. It cannot point to my car; that is an entity not an attribute. Apparently, the sign for my car's redness does not refer to anything. If this is the case, then the model does not directly map onto things in the real world. In Parts Three and Four, we will see how resolving fundamental problems, such as these, leads us to the object paradigm.

Figure 2.3: 'My car is red'



3.3 Why this semantics problem exists

In everyday life, we quite often come across mysterious unexplained ideas. We find an unconscious awareness of something coupled with an inability to discuss it. In the case of business modelling, this is a sure sign of a paradigm. It is a sign of a way of seeing things that is, by its very nature, so deeply embedded in our minds that we are not conscious of it. And we believe so firmly in it that we cannot question it.

Unconscious control is normally an extremely sensible way of dealing with fundamental situations like this. Situations that we are so sure of that they rarely need conscious review. If all our actions had to be under our conscious control, it would take ages to make even the most simple decision. It would be as if the board of directors of a large company insisted on being involved in every decision, from appointing a new chairman to buying a box of rubber bands. The only practical way out is to delegate the control of those situations we are sure of into our unconscious.

The circumstances change when a paradigm needs shifting. The advantages of unconscious control now turn into disadvantages. We can see that here. If we are going to re-engineer the entity paradigm, we need to know what it is. However, whenever we start asking about entities, our unconscious kicks into operation and interrupts the question.

It tries to stop the question being consciously considered, often by insinuating that it is irrelevant or obvious. (The Chairman of the Board might use similar tactics to dismiss a question about buying a box of rubber bands.) This makes it difficult to start the re-engineering.

4 Ignoring 'things in the business'

This unconscious use of the entity paradigm has led to a tendency to ignore the 'things in the business' and focus on the signs that refer to them. For example, people working in the computing industry tend to focus on the world of computer information, especially when they are dealing with computers. As a result, they end up imposing the computer world's framework onto the 'things in the business'.

4.1 Imposing the data–process distinction onto 'things in the business'

A good example of this is the way they impose its data–process distinction. Computer technology, unlike paper technology, can both store and process information. So, in the computer world, the distinction between information that is stored—data—and processing information—process—is an important one. But it is only important in the computer system. It is irrelevant to the things in the business that the data and process refer to.

Nevertheless, business modellers impose the data–process distinction onto 'things in the business' with disastrous results. We can illustrate this with a simple example. Most accounting systems have an account movements file. They also have a program that processes the movement records on that file and posts them to the accounts file, updating the balance with the movement. In computing terminology, the account movements and accounts are both data and the accounts movements update program is process. A model of this part of the system would look something like ***Figure 2.4***.

Figure 2.4: Account movements system model

You will find that business models for accounting systems often have a similar shape to the model in Figure 2.4. Account movements are represented as data and the account movements update of the accounts as a process. This seems a natural way of modelling the business to anyone living in a computer world. It is also the wrong way.

To see this, we need to look at the data–process distinction again. It is a fundamental distinction in a computer system. Data and process are quite different. In an information system, data persists over time; whereas, processes do not. They happen. There is a similar distinction in the real, non-information, world. Things persist over time and changes do not. When people who live in a computer world model the real world, they represent it as data and process. They either ignore the real world's distinction between

© Copyright Chris Partridge

things and changes or assume that data maps directly onto things and process onto changes—as shown in ***Figure 2.5***.

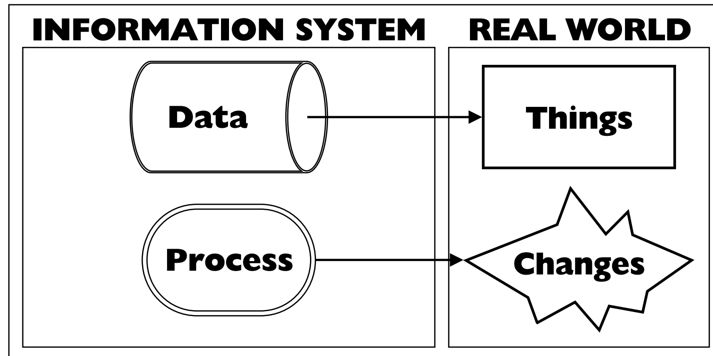


Figure 2.5: Data-process spuriously reflecting things—changes

This is a mistake. The business model, and not the information system, is meant to map onto the things in the business (the real world). The problem arises because data does not necessarily map onto things (or process, changes). To see how this causes a problem consider the account movements again. Ask yourself whether the individual account movement records represent a thing or a change in the business? The correct answer is they represent changes. For example, if I pay £100 into my bank account, my paying in is not a thing but a change. And the change is recorded (represented) by data in the form of an account movement record. Once we understand this, we no longer draw the account movement in our business models as data, but as a change. This is illustrated in ***Figure 2.6***.

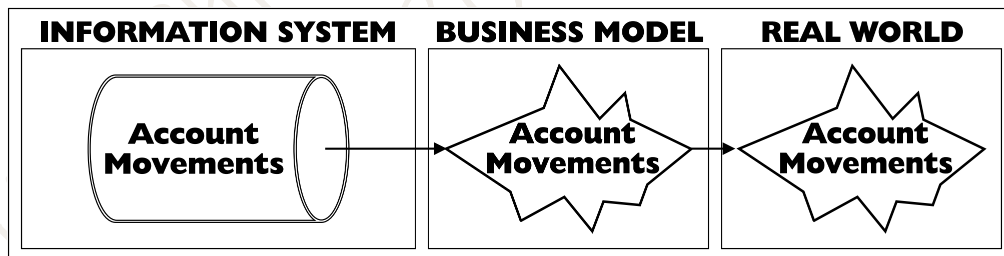


Figure 2.6: Account movements business model

This example clearly shows that the distinction between data and process in a computer system is not based on differences between things and changes in the real world. It is based on whether things in the computer system persist or not. The data-process distinction and the things—changes distinction have the same underlying basis. But this in no way implies that data represents things and process represents changes—as suggested incorrectly by ***Figure 2.5***. It turns out that the representing relationship is much more flexible. As illustrated in ***Figure 2.7***, both data and process can represent either things or changes.

This provides a simple test of whether a model is representing the business or an information system. If the model's notation classifies changes in the real world as data (as, for instance, the example in ***Figure 2.4*** classifies accounting movements as things or data), then it is describing the computer system and not the business. Therefore, it is

not a business model. Unfortunately, many so-called business models fall into this category.

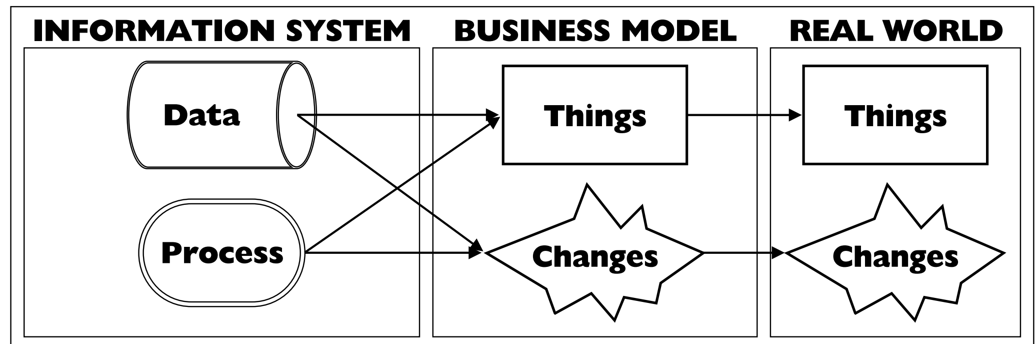


Figure 2.7: Data-process correctly representing things—changes

These distinctions also highlight one difference between system and business objects. System objects encapsulate data and process into one object. Business objects, on the other hand, deal with the things—changes distinction. As we shall see in Part Four, business objects equivalent of encapsulating data and process are patterns that generalise across the things—changes distinction.

4.2 Ignoring the difference between understanding and operation

This confusion about whether data–process represents things—changes is just part of a wider confusion between understanding things in the business and the operation of things in the computer system. In the ***Prologue*** we touched on the distinction—on how business modelling deals with understanding the things in the business; whereas, the other, later, stages of system building were more concerned with the operation of the final system.

This distinction between understanding and operation is important for business objects. We can get an idea of why, by looking at how it affects a notion dear to people working in O-O re-use. O-O creates an environment where there is more potential for re-use. That is why O-O systems can be simpler and more compact. This re-use works at both an operational and an understanding level. Unless we make a clear distinction between the two levels, we do not take full advantage of O-O's potential. The following example shows how we distinguish between the two.

4.3 Distinguishing between operational re-use and generalisation

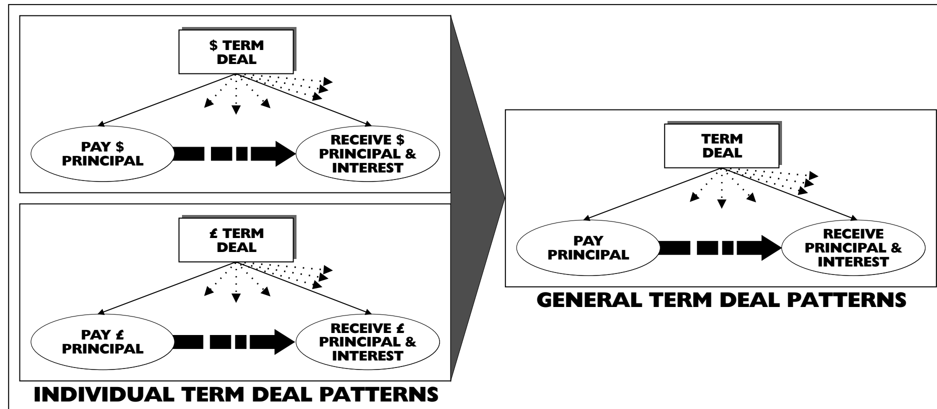
Everyone, not just O-O system builders, is familiar with operational re-use. We know what happens when we operationally re-use a pattern or component. We apply it in new situations. Re-use works in a different way at the understanding level, where it is closely tied in with generalisation. The following simple modelling example illustrates this.

Assume that we are building a model of a money market trading system and we are focusing our analysis on $ and £ term deposit placed deals. We notice that these two types of deals have a similar pattern of settlement. For instance, in both cases, the prin-

© Copyright Chris Partridge chrispartridge@BORO Centre.com

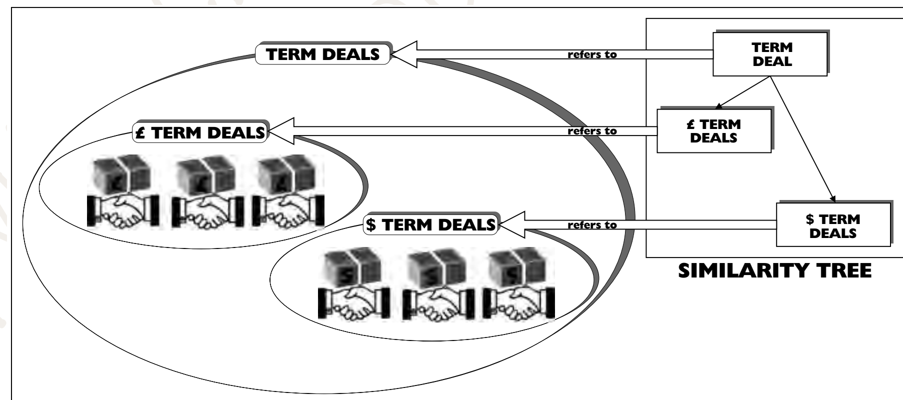
cipal is paid away on an agreed date and the principal plus interest is received back after an agreed term. It seems like a sensible idea to consider whether a copy of the program code for the $ term deal could be re-used to process £ term deals. This is an operational approach.

Figure 2.8: Constructing general patterns



On the other hand, we could think about re-use at an understanding level. Then our chief concern would be finding similarities in the patterns for the two types of things. Patterns that we could use to generalise. Thinking this way we would construct a more general term deal pattern that applies to both of the less general types of deal—as shown in ***Figure 2.8***. We are not really just finding a general pattern. What we have done is constructed a new more general type of thing in the business—a general term deal. We can think of this as building a similarity tree of things as illustrated in ***Figure 2.9***.

Figure 2.9: Identifying similar types of 'things in the business'



It is when we start designing the system that we should shift to an operational view of re-use. We can ignore the things in the business and talk of the general term deal pattern (or program code) being re-used for both $ and £ term deals.

5 What types of things (in the business) do we re-engineer?

We are already focused on the things in the business. I have found that the re-engineering to the object paradigm is greatly simplified if we restrict our focus further. All that we need for the re-engineering can be found in a small group of four types of things. When re-engineering these four, we re-engineer the whole paradigm. The four are:

- Particular things,
- General types of things,
- Relationships between things, and
- Changes happening to things.

Stretching a point, we call these the four key types of things (changes are not really things, more a pattern of relationship between things).

At first sight, these four appear simple and appear as if they would not be problematic. But, by the time we have re-engineered to the object paradigm, we shall realise how sophisticated our way of seeing has to be to handle them accurately. When we follow the re-engineering in Parts Three and Four, we will concentrate on how each paradigm deals with these areas.

5.1 Particular things

Particular things are individual things. For example, see a particular table or a particular chair—as illustrated in ***Figure 2.10***. These are often called physical bodies and are the simplest and most basic items in semantics. An information paradigm needs to explain what makes something particular—what particularity is. This involves more than just saying particular things are individual, concrete, and tangible. The paradigms we look at offer very different explanations of what they are.

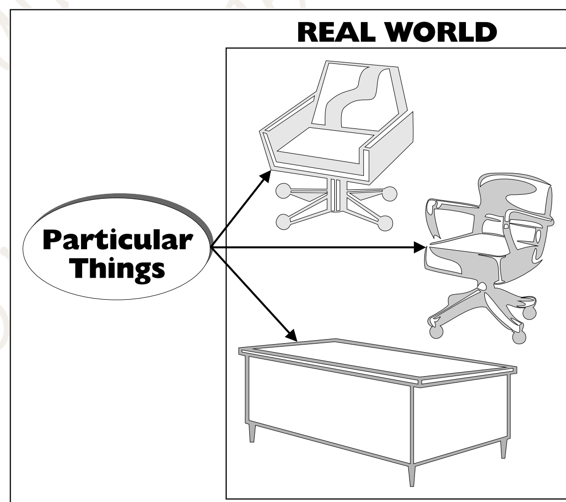


Figure 2.10: A particular table and two particular chairs

© Copyright Chris Partridge chris.partridge@BORO Centre.com

Particular things have one important pattern. We see them as having properties. This was raised in the earlier example of my car (a particular thing) illustrated in ***Figure 2.3***. As we said then, an information paradigm needs to explain what my car's redness is in the real world.

5.2 General types of things

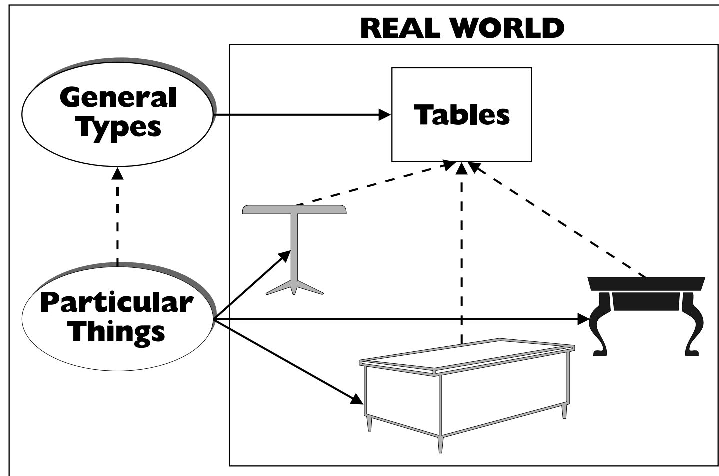


Figure 2.11: General types and particular things

We naturally group particular things into general types. For example, the 'particular' tables—illustrated in ***Figure 2.10***—may be grouped along with other 'particular' tables into the general type, tables. These two are quite different. Particular things are normally concrete and tangible; whereas types, such as table, are normally abstract and intangible. For example, it does not seem to make sense to ask what the type tables feels or looks like. Not surprisingly, we naturally distinguish the general from the particular. We also naturally relate them. For example, when we see a particular table, we naturally classify it as belonging to the general type, tables. This is illustrated in ***Figure 2.11***.

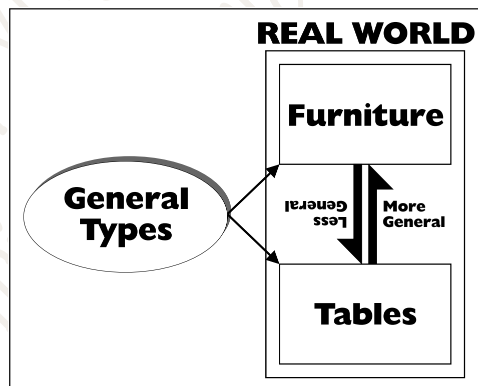


Figure 2.12: More and less general types

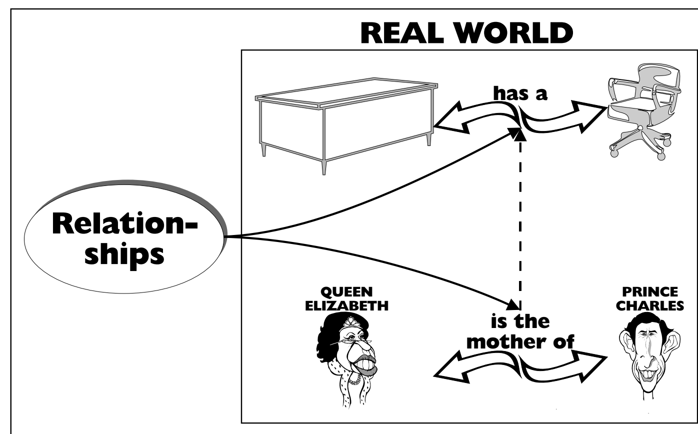
General types also have a common pattern relating one type to another, the 'more general' pattern. For example, the general type, furniture, is 'more general' than the general

type chairs. An information paradigm needs to explain what this more general pattern—illustrated in ***Figure 2.12***—is in the real world.

5.3 Relationships between things

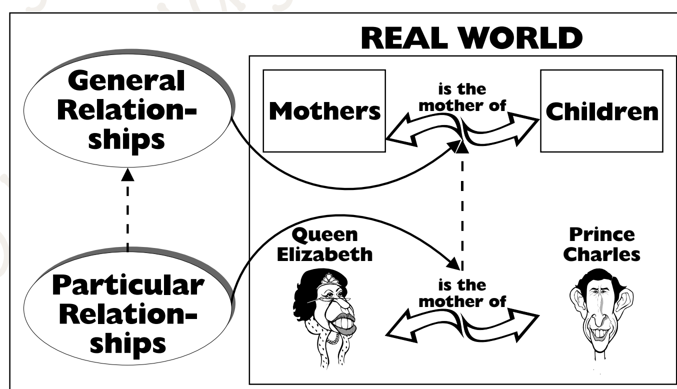
The next type of object is relationships. People have relationships—for instance the two ‘particular things’, Queen Elizabeth and Prince Charles, are related: Queen Elizabeth is the mother of Prince Charles. This is an example of a blood relationship. There are other non-blood relationships, for instance a particular chair may be at a particular desk.

Figure 2.13: Two examples of relationships



These two examples (illustrated in ***Figure 2.13***) are relationships between particular things—what might be called particular relationships. There are also relationships between general types. For example, we can generalise the ‘Queen Elizabeth is the mother of Prince Charles’ relationship to ‘(the type) mother can be a mother of (the type) children’. These are what might be called general relationships. Seen this way, the particular general pattern applies not only to things and types but also to relationships—as illustrated in ***Figure 2.14***. All this needs to be explained by an information paradigm.

Figure 2.14: General and particular relationships



5.4 Changes happening to things

At the beginning of the chapter, when looking at the data–process distinction, we introduced the things–changes distinction. This provides us with the last type of object: changes. As we said earlier, what distinguishes changes from things are that things persist through time; whereas, changes do not, they happen.

A strong pattern connects changes to things: changes happen to things. Consider for example, a change such as a green tomato turning red. The change (turning red) happens to the tomato (a particular thing). We readily appreciate that the things and changes are two very different types of objects. However, an information paradigm has to give a clear and consistent explanation of what both these objects are and why they are different.

Changes, like things, also have a general–particular pattern. For instance, the change in our example, a particular green tomato turning red, can be generalised to a general type of change, green tomatoes turning red. An information paradigm should explain why changes have the same pattern and what it is.

6 Our starting point—the entity paradigm

Our starting point for the re-engineering is the entity paradigm. In Part Two, we see how it and the substance paradigm on which it is based deal with these four key types of things. The substance paradigm was developed by the Ancient Greek Aristotle in the 4th century BC. Some people might find it odd that we currently use an entity paradigm based on something so ancient as the substance paradigm. But, on reflection, however, one should realise that, for practical everyday use, the age of a paradigm is not relevant. The real issue is its suitability for the job. This is why it makes sense for an engineering discipline to use a scientifically 'out-of-date' paradigm.

6.1 Engineers often use scientifically 'out-of-date' paradigms

Once we start looking, we can find many other examples of 'out-of-date' paradigms. We soon recollect that civil and mechanical engineers still use 'out-dated' Newtonian physics although its successor, quantum-mechanical physics, has been available for almost a hundred years. We may be less aware that ship's officers are taught to navigate using a millennia old paradigm of a fixed earth and moving star sphere—one that was superseded over four hundred years ago.

We use these old paradigms because, even though the scientist's latest paradigm may be the most accurate, it is not necessarily the most appropriate for everyday tasks. Unlike scientists, engineers faced with a task have to make a practical decision. That is why they try to choose the most suitable paradigm for the job, no matter how ancient or out-of-date. Information engineers picked the entity paradigm because it was the most appropriate for paper and ink technology. The issue we are facing is that they then imported it wholesale onto computer technology.

7 Arriving at an object semantics for ‘things in the business’

Given that the substance paradigm is so old, it should come as no surprise that it is scientifically ‘out-of-date. ‘Information scientists’ have re-engineered a number of new paradigms in the two millennia since it was first formalised. In fact, ‘information scientists’ were developing the shift to the object paradigm in the first half of this century, well before electronic computers were even invented.

7.1 The separate evolution of information semantics

The easiest way to think about these developments is in terms of the three elements of an information paradigm mentioned at the beginning of this chapter:

- Information technology,
- Syntax, and
- Semantics.

Until computers were developed, paper and ink were the leading information technology. And so the entity paradigm was rightly selected as the most practical option for working ‘information engineers’. However, at the same time, the world’s best thinkers from a variety of fields have been developing semantics. Over time, they have evolved increasingly sophisticated systems. Even though these systems often remained academic, without a practical application, this did not stop them from developing them further. With the development of computing, we have a technology that makes the ‘scientifically’ advanced systems of semantics a practical proposition.

It is not unusual for people to develop ideas ahead of their times’ technology. A well-known example is Leonardo da Vinci. In the 15th century, he drew designs for a helicopter—hundreds of years before the technology needed to build one was available. It was a good idea that could not be put into practice because of the state of technology. In the same way, when object semantics was developed, it was far too rich to work on the then current paper and ink technology.

7.2 Following the semantic re-engineering route

Thinkers have had a long time to develop their ‘scientifically’ advanced semantics. As you might expect, semantics has moved a long way in the two millennia since the substance paradigm was formalised—in paradigm-speak, there have been a number of shifts. We follow these in Parts Three and Four.

Following in these thinkers’ footsteps is a much easier way of re-engineering than starting from scratch. The shifts have already been worked out for us. All we have to do is understand them. This is just as well, because the computing industry has no real experience of re-engineering or developing semantics.

As our goal here is understanding the object paradigm, rather than the history of semantics, we take a drastically shortened version of the historical path. We just look at

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

the semantics of one intermediate paradigm in the evolution to object semantics—a kind of halfway house. This is the logical paradigm.

8 Re-engineering the ‘things in the business’

In this chapter we have focused in on the core areas we need to re-engineer. We first focused on the ‘things in the business’.

We recognised that system builders currently tend to ignore these ‘things in the business’. We saw the undue importance they attached to the data–process distinction, imposing it on the things in the business—confusing it with the things–changes distinction.

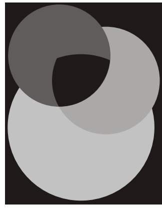
We saw that this was part of a wider confusion between understanding the ‘things in the business’ and the operation of things in the computer system. As an example, we looked at how re-use—a notion dear to O-O people—worked at the operational and understanding levels.

We then refined our focus, identifying the types of things we need to re-engineer to arrive at the object paradigm. We identified four key types of things. We then determined how we were going to arrive at an object semantics. We learned that business entities had already been re-engineered into objects—outside computing. We do not need to re-engineer it from scratch, but can follow in the footsteps of the original re-engineers. We also briefly touched on the origin of the entity paradigm in the Ancient Greek substance paradigm. This is the starting point for our journey to business objects. We stop off at one intermediate paradigm on the way, the logical paradigm.

9 The next part

In the next part of the book (Part Two), we clarify the starting point of our journey to the object paradigm. There, we unearth the entity paradigm’s semantics—finding it in its origins in the substance paradigm.

© Copyright Chris Partridge, chris.partridge@BORO-centre.com

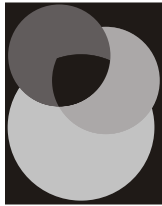


Part Two

Our Starting Point—The Entity Paradigm

Chapter 3 What Is the Entity Paradigm?
Chapter 4 The Substance Paradigm's Semantics

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 3

What Is the Entity Paradigm?

1 Introduction
2 The entity paradigm's fundamental particles
3 The entity framework and its (re-)use of patterns
4 The entity paradigm and the file-record paradigm
5 Mapping entities and attributes onto files and records
6 The substance paradigm's secondary hierarchy
7 Simplifying the substance paradigm's treatment of relationships
8 Our current way of seeing stored information
9 The next chapter

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction

In Part One, we looked at our strategy for re-engineering the entity foundations of information. Here in Part Two, we examine our starting point—the entity paradigm—concentrating on the four key types of things we identified in the last chapter. Then in Parts Three and Four we will follow the entity paradigm's evolution into the object paradigm.

The entity paradigm is what most computer systems' information is based on. Because our aim is to re-engineer the business paradigms embedded in our existing computer systems, it is a natural starting point. Most people who work with it, instinctively recognise that the entity paradigm is a tool for doing things rather than understanding them. To use a distinction raised in the ***Prologue***, it works at an operational rather than an understanding level.

The entity paradigm is a simplified version of a powerful paradigm developed by the Ancient Greek Aristotle, which we call the substance paradigm. It is a good approximation to the way most of us see the world. It was simplified into the entity paradigm to work more effectively with paper and ink technology.

In the last chapter, we noted that most of us cannot explain what an entity is. What we do here is drag the entity paradigm up to the surface, making ourselves conscious of it. Sometimes people are surprised when they first see it in the cold light of day—some even find it difficult to accept that it is what they have been working with.

We start this chapter by looking at the entity paradigm. We look at its fundamental particles, the patterns of re-use it enables, and how it relates to paper and computer information. Then we look at the elements of the substance paradigm that were eliminated when the entity paradigm was simplified, and why they were removed. This gives us an insight into the nature of entity-oriented systems. In the next chapter, we turn our full attention onto the substance paradigm, particularly its semantics.

2 The entity paradigm's fundamental particles

We start by looking at the entity paradigm's fundamental particles:

- The entity, and
- The attribute.

The entity particle is more fundamental, so we start with it.

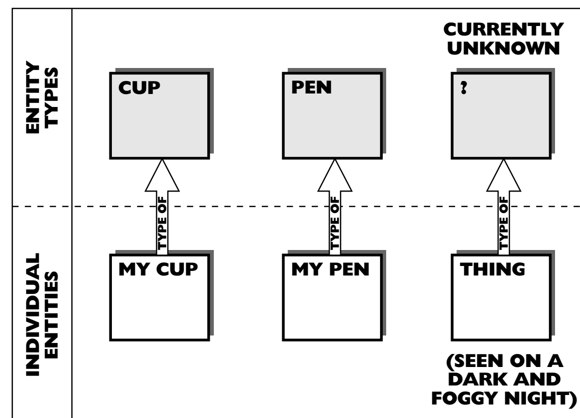
2.1 Individual entities

We naturally divide the world into concrete particular things. When we look around a room, our eyes receive a continuous stream of data. We unconsciously analyse this stream and consciously see chairs, tables and so on. These are individual entities.

2.2 Entity types

Individual entities are not the only type of entity. We also naturally group together individual entities that are, in some sense, the same. We then say that the entities in the group all belong to the same entity type. Individual entities naturally belong to entity types. They also always belong to one. For instance, we might catch a fleeting glimpse of something (it may be a fox or a dog) on a dark and foggy night. Even though we cannot say what entity type the thing belongs to, it still has one—as shown in ***Figure 3.1***.

Figure 3.1: Individual entities naturally belong to entity types



2.3 Attributes belonging to entities

Attributes have the same two-tier pattern as entities. Just as there are individual entities and entity types, so there are:

- Individual attributes, and
- Attribute types.

2.3.1 Individual attributes

We naturally see that individual entities have attributes (also called properties or qualities). Someone looking at my car naturally notices it is red (in other words, it has a red attribute/property/quality). We always see individual attributes belonging to individual entities; this is one of their inherent features. Notice how difficult it is to see the particular red property existing on its own (out in the real world) with no entity attached to it—as illustrated by ***Figure 3.1***. This seems impossible. It is important to remember that we are talking about individual attributes out in the real world. We can, of course, imagine the general idea of red on its own; but that is in our head.

If we think about it, we can see that the individual attributes determine how an entity appears. All of my car's appearance is based on its properties, its individual attributes. If

© Copyright Chris Partridge
chris.partridge@BORO-centre.com

my car looks red, then it has the property of being red—it has a red attribute. The entity itself is not red, being red is the function of the individual red attribute.

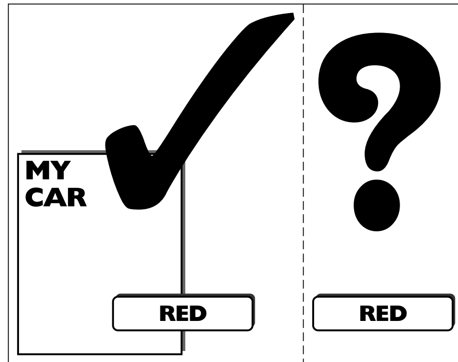


Figure 3.1: Individual attributes always belong to an individual entity

2.3.2 *Attribute types*

The relationship between individual attributes and attribute types is similar to that between individual entities and entity types. Every individual attribute belongs to an attribute type and an attribute type can have a number of individual attributes. Also the relationship between an attribute type and its entity type is similar to the relationship between an individual entity and its individual attributes. This is shown schematically in ***Figure 3.2***. An example using my red car is given in ***Figure 3.3***.

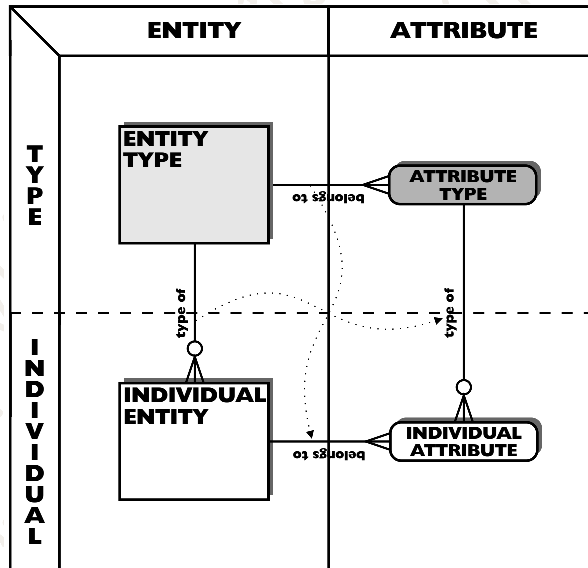


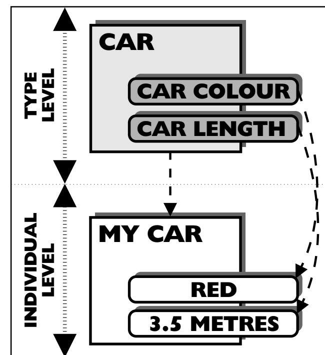
Figure 3.2: Entities and attributes

© Copyright christiscambridge ... centre.com

3 The entity framework and its (re-)use of patterns

The entity paradigm is a good example of how a framework enables the (re-)use of general patterns.

Figure 3.3:
My red car



3.1 Re-use working down the entity framework's hierarchy

Re-use works its way down the entity framework. The general entity and attribute patterns are used to construct all the patterns at the type level. These are then used (and so their embedded general patterns re-used) to construct the individual level patterns.

We can see this happening in ***Figure 3.4***. There, the general entity-attribute pattern provides a framework for the staff member entity type patterns. This in turn provides a framework for the individual staff member patterns.

Figure 3.4:
The entity paradigm's general structure

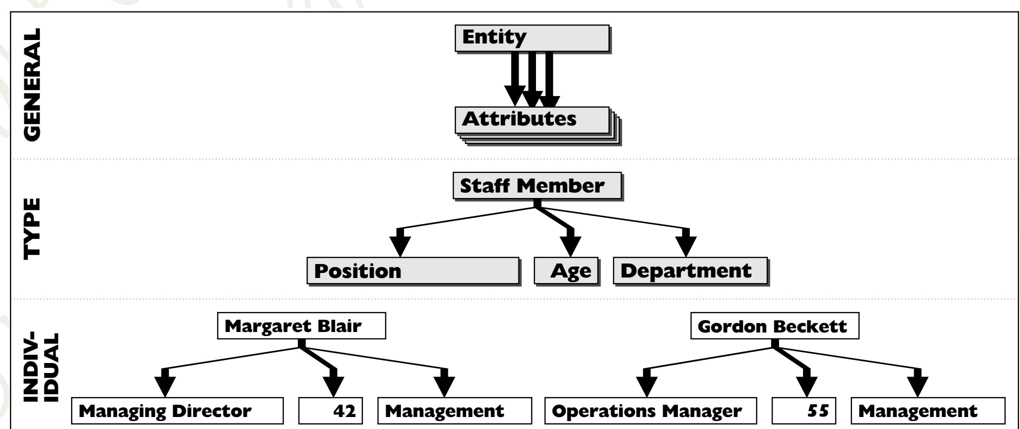


Figure 3.5 illustrates how re-use operates at the general level. We can see that the entity paradigm does not determine what the actual entity and attribute types are, just the framework in which they live. The various entity and attribute types work in a similar way. They do not determine what the actual entities and attributes are, just their framework.

© Copyright - Chris Partridge
chris.partridge@BORO Centre.com

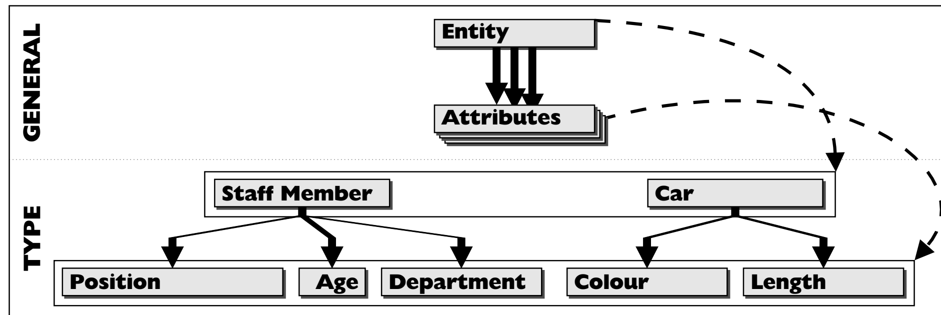


Figure 3.5: Re-using the general entity-attribute pattern

This type of framework derives its power from the repeated use (and re-use) of a pattern from a higher level at a lower level. For example, the general entity-attribute pattern is repeatedly used in the construction of entity types and their attribute types. There is a similar pattern of re-use between the type and individual levels. An individual entity and its attributes are constructed using the patterns from its entity type and associated attribute types. This usually means that the patterns for the entity type and its associated attribute types are re-used many times. The general entity-attribute pattern is embedded in the entity type and attribute type patterns. So, when the individual entities and their attributes are constructed using type level patterns, the general pattern is also implicitly used. In this way it pervades all levels of the framework.

The entity–attribute pattern is generative. As well as applying to the existing framework, the general pattern can be used to 'generate', when required, new type and individual level patterns. Type level patterns are generative as well; they can be used to 'generate' new individual level patterns.

It might be easier to see this with a simple example. Assume that I have constructed a car entity type pattern and used it to construct an individual entity pattern for my car. Also assume that I have constructed a car colour attribute type for the entity type and, based on this pattern, a red attribute at the individual level for the 'my car entity'. If I now see John's car, I can generate its pattern from the existing framework. I can construct an individual entity for it, using the car entity type and car colour attribute type patterns—as shown in ***Figure 3.6***. When I use the car entity pattern to construct the entity John's car, I also automatically commit myself to constructing an attribute using the car colour attribute type. Notice also that 'John's car' entity inherits the general entity–attribute pattern through the car entity type pattern.

We will appreciate the generative nature of the type level better if we consider what would have happened if there had been no middle type level in the framework, just the top general entity-attribute level. Without the type level, whenever we saw a car for the first time we would have to regard it as a unique entity with its own variety of individual attributes. This means we would have to go through the kind of analysis that we do for completely new and unknown types of things. So, when we see John's car for the first time, we would have to construct its entity and attributes without the guidance of a car entity type. We would not know that it was a car nor that it has a colour attribute. This would not only make life complicated, but very time consuming.

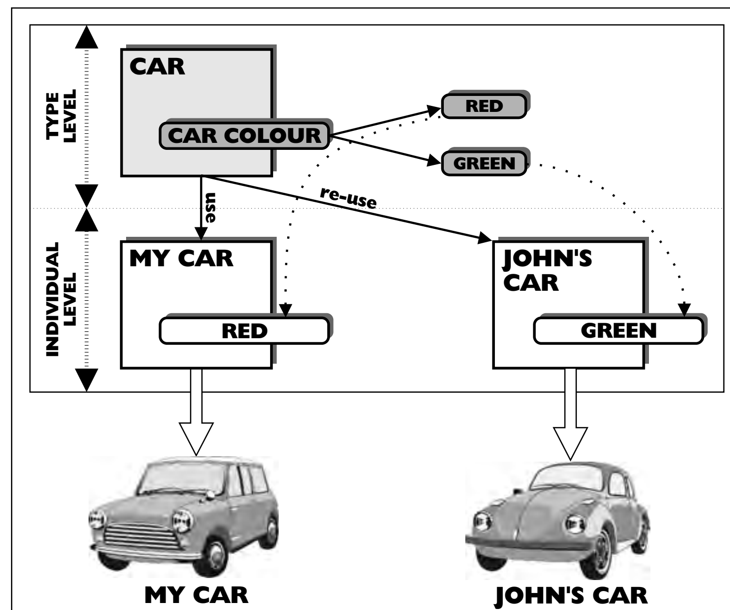


Figure 3.6: The generative type level

4 The entity paradigm and the file-record paradigm

The entity paradigm is closely related to what we will call the file-record paradigm.

4.1 Where does the file-record paradigm come from?

Computer users often talk about information being stored in files and records, rather than entities and attributes. However, the terms, 'file' and 'record', are rooted in paper and ink technology. Well before the invention of computer technology, records were made on pieces of paper and kept in files.

Tables paradigm	File–record paradigm	Entity paradigm	Example
Rows	Computer record	Individual entity	My car
Element	Computer field	Individual attribute	Red
Table	Computer file	Entity type	Car
Column	Computer field type	Attribute type	Car colour

Table 3.1: Closely linked fundamental particles

These paper records and files were based on the way in which information is naturally stored on paper in rows and columns. We distinguish paper's framework of rows and columns from computing's files and records by calling it the tables paradigm. These par-

© Copyright Chris Partridge chris.partridge@BOROCentre.com

adigms are all closely linked. For example, their fundamental particles all map directly onto one another—as shown by ***Table 3.1***.

4.2 Reinforcing the entity paradigm

Computer-literate people's training in file and record patterns reinforces the entity paradigm, and vice versa. For example, it appears to make no sense to talk about an individual field existing apart from its record—as illustrated in ***Figure 3.7***. Just as it makes no sense to talk about individual attributes existing without their individual entities (shown in ***Figure 3.1***). What could such a field or attribute be?

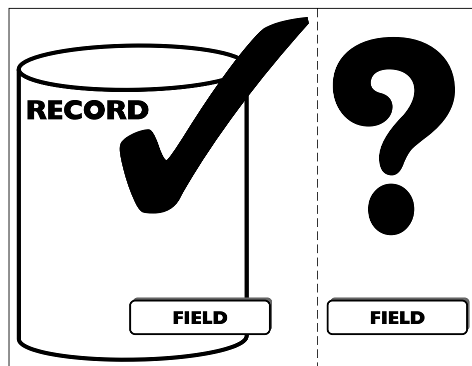


Figure 3.7: An individual computer field without its computer record

5 Mapping entities and attributes onto files and records

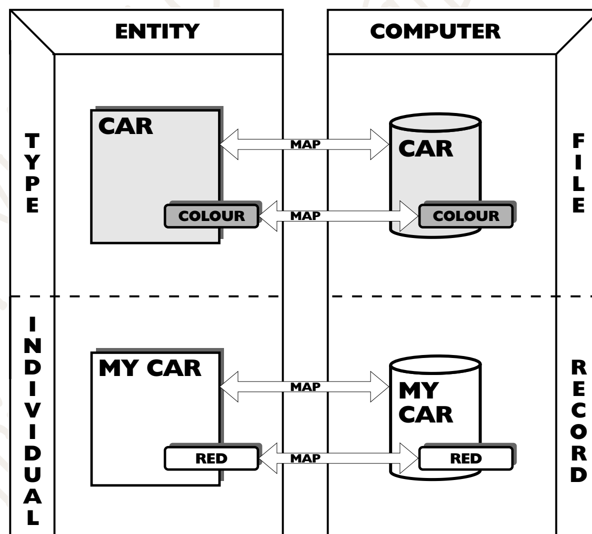


Figure 3.8: Physically implementing the entity paradigm

These intimate links between the paradigms (shown in ***Table 3.1***) lead many system builders to see computing's file-record paradigm as a physical implementation of the entity paradigm—as illustrated in ***Figure 3.8***.

5.1 Staff example

This apparently close mapping leads many people to the false assumption that files and records directly reflect the real world's entities and attributes. This simple staff example shows how wrong this assumption can be.

5.1.1 Two incompatible entity formats

Consider the two lists in ***Table 3.2***, one of salespersons and the other of account managers. They could equally well be combined into one staff list – as shown in ***Table 3.3***. Let's assume that these two lists are from computer systems whose records and fields directly reflect entities and attributes. We can then work out what the systems' entity formats are and, by implication, the entities and attributes they reflect. The formats are shown in ***Figure 3.9***

Salespersons			Account Managers		
Surname	First name	Middle initials	Surname	First name	Middle initials
Bottomley	Margaret	V.G.	Beckett	Gordon	O.B.
Clarke	Michael	L.	Blair	Margaret	J.
Heseltine	Kenneth	J.	Brown	Claire	F.L.
Howard	Cecil	S.F.A.	Short	Tony	L.
Thatcher	Virginia	J.K.	Thatcher	Virginia	J.K.

Table 3.2: Salespersons and Account Managers list

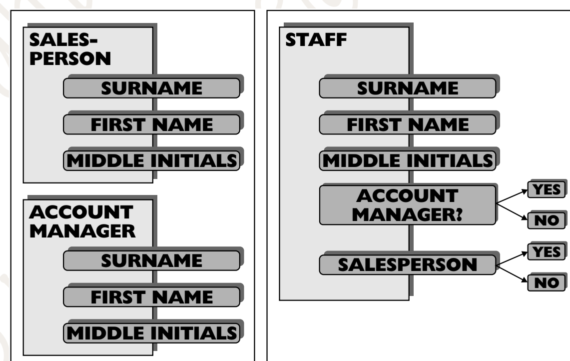


Figure 3.9: The two assumed entity formats

As the business only has one entity structure, it cannot be reflected by both the entity formats in ***Figure 3.9***. Only one of them can be 'right'. If we compare the formats of the two systems, then we immediately see a major difference. In the first system, there are signs for two entity types, Salesperson and Account Manager. Whereas, in the second format, there is only one entity type sign, Staff. Which of these entity type signs actually refer to real entity types in the business? If the business has a Staff entity type, then the

first format is ‘wrong’; its two entity type signs do not reflect the business—as shown in ***Figure 3.10***.

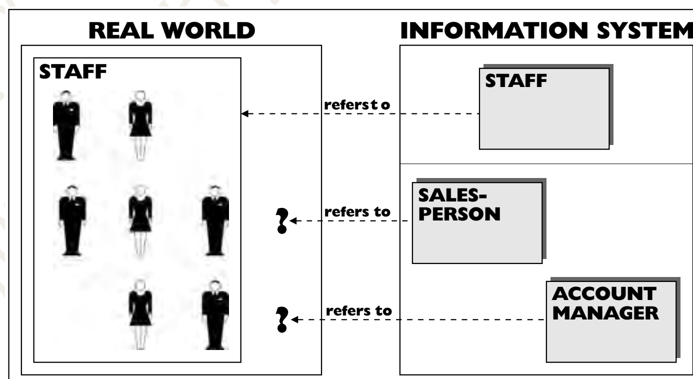
Surname	First name	Middle initials	Salesperson indicator	Account Manager indicator
Beckett	Gordon	O.B.		Yes
Blair	Margaret	J.		Yes
Bottomley	Margaret	V.G.	Yes	
Brown	Claire	F.L.		Yes
Clarke	Michael	L.	Yes	
Heseltine	Kenneth	J.	Yes	
Howard	Cecil	S.F.A.	Yes	
Short	Tony	L.		Yes
Thatcher	Virginia	J.K.	Yes	Yes

Table 3.3: Staff List

If Salesperson and Account Manager are entity types then the Staff format—shown in ***Figure 3.11***—is ‘wrong’.

There is another contradiction in ***Figure 3.9***. Look at it again, and compare the two systems’ entity formats. You may notice that, in the first system, Salesperson and Account Manager are signed as entity types; but, in the second system, they are signed as attribute types (the Salesperson and Account Manager indicators). If the entity paradigm reflects the structure of the world, then the world divides irrevocably into entities and attributes. Something cannot be both an entity and an attribute. However, here we have two systems; one with Salesperson signed as an entity type and the other with it signed as an attribute type. Which type is it—entity or attribute?

Figure 3.10:
Staff world view



© Copyright Chris Partridge
chris.partridge@BOROCentre.com

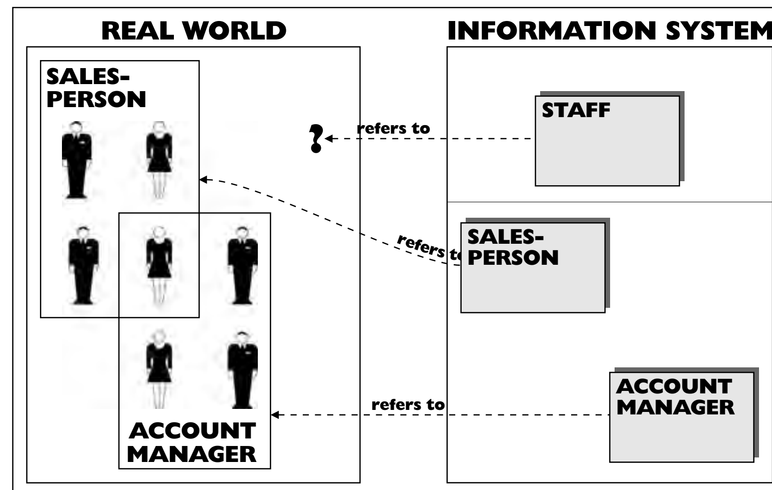


Figure 3.11: Salesperson and Account Manager's world view

5.1.2 Not 'wrong' but different objectives

It turns out that neither format is 'wrong'. What is wrong is our assumption that a computer system's files and records necessarily reflect the world's entities and attributes. What dictates the structure of these files and records is not the world they describe but what we want to do with them in our system. The structures of the two systems are different because they serve different purposes. For example, if we wanted to send the same letter to all staff, it would be easier to use the list from the staff system. If we were sending different letters to Salespersons and Account Managers, it would be easier to use the two lists from the other system.

This example shows that the decision on whether to use a record or a field in a computer system does not necessarily involve distinguishing between entities and attributes in the outside world. It is more about different ways of handling the data inside the information system.

6 The substance paradigm's secondary hierarchy

When the substance paradigm was simplified into the entity paradigm, the structurally key secondary hierarchy was eliminated. This is part of the reason why the staff example's files and records do not map directly onto the outside world; and why the entity paradigm focuses on the data inside information systems rather than the things in the business. We now look at the substance paradigm, particularly its secondary hierarchy. We see how it was simplified and how the simplification changes what we see.

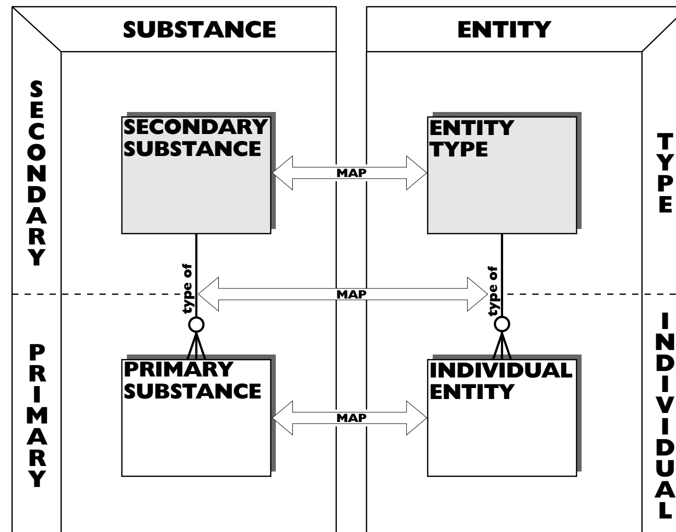
6.1 The substance paradigm

The substance paradigm is ancient. It was constructed by the Ancient Greek Aristotle in the 4th century BC and subsequently developed by his followers. It is, like much of Aris-

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

totle's work, a rationalisation of people's intuitive ideas. These ideas are still with us today. Most people see the world through the eyes of the substance paradigm.

Figure 3.12: The substance particle corresponds to the entity particle



The substance and entity paradigms share similar fundamental particles. The substance paradigm's particles are substances and attributes, where substance corresponds to entity (as shown in ***Figure 3.12***) and attribute, not surprisingly, to attribute. (Students of Aristotle use the words 'substance' and 'entity' almost interchangeably; however, we use 'substance' for Aristotle's paradigm and 'entity' for the entity paradigm.) As we shall see in the next section, the main difference is that, at the secondary level (the substance paradigm's name for the type level), the substance paradigm is more powerful.

6.2 The secondary level hierarchy

We now look at the part of the substance paradigm that was 'simplified' out of the entity paradigm—its secondary level hierarchy. This can be divided into two interlinked hierarchies:

- The secondary substance hierarchy, and
- The secondary attribute hierarchy.

6.2.1 Secondary substance hierarchy

We use my car to illustrate what the secondary substance hierarchy is. My car is a car—in substance-speak, my car's primary substance has a car secondary substance. Cars are a type of vehicle—vans are another type. Similarly, vehicles are a type of transport—aircraft and ships are other types. From a substance paradigm viewpoint these types are all secondary substances and the paradigm recognises that they form a hierarchy (shown in ***Figure 3.13***). In the entity paradigm, there is no hierarchy and entity types are restricted to a single level—also indicated in ***Figure 3.13***.

© Copyright Chris Partridge chris.partridge@BORO Centre.com

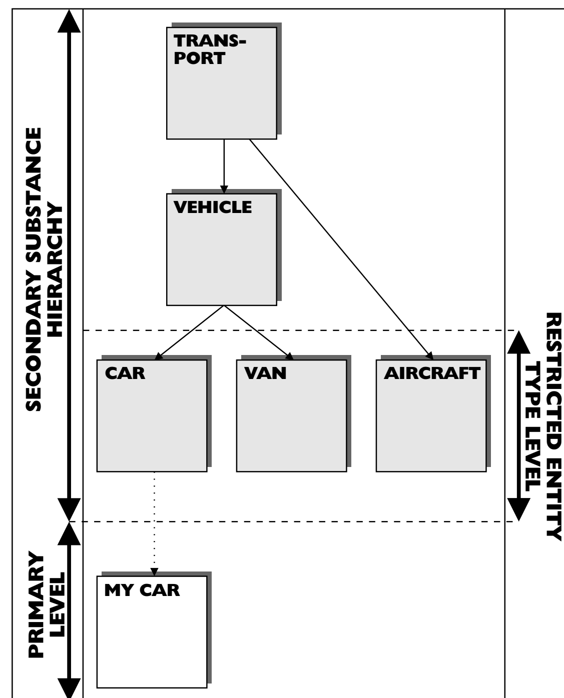


Figure 3.13: A substance hierarchy

6.2.2 *Secondary attribute hierarchy*

It is not only secondary substances that have a hierarchy, so do secondary attributes. My car is red and cars are coloured. In substance-speak, my car's primary substance has a red primary attribute and secondary car substance has a colour attribute. In this respect, the substance and entity paradigm are similar. But other substances are red (apples, fire engines, etc.), so there are other red primary attributes and these red attributes have something in common, they are all red. In the substance paradigm this is explained by having an independent red secondary attribute.

Similarly, other secondary substances (as well as cars) are coloured; so there is an independent colour attribute that has the independent red attribute (and the other individual colour attributes) as part of it. This results in the framework shown in ***Figure 3.14***. My car's red attribute can be called its colour attribute—it is inherited from the secondary level car's colour attribute. We then say the value of my car's colour attribute is red. The red-colour relationship, independent of my car, is reflected by red 'belonging to' colour in the secondary attribute hierarchy.

© Copyright Chris Partridge
chris.partridge@BORO-centre.com

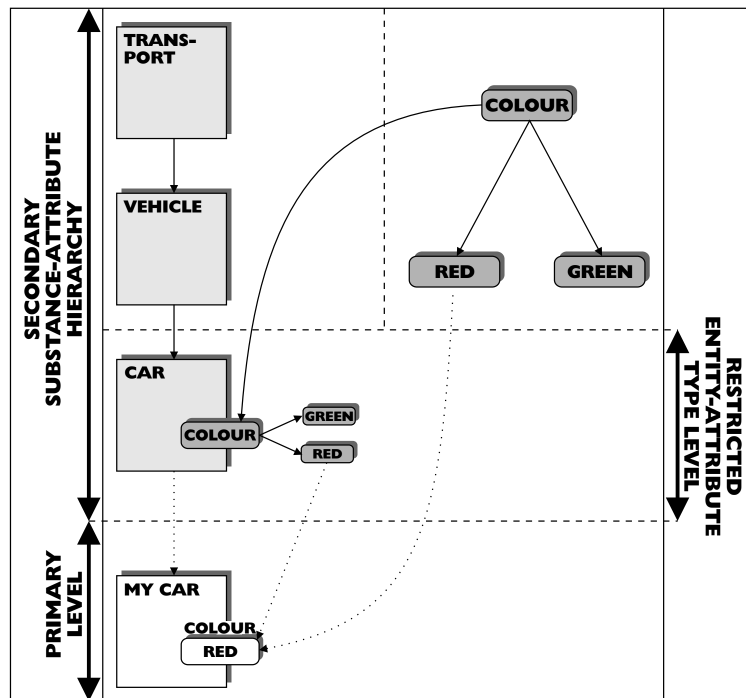


Figure 3.14: Substance–attribute framework

The schema of the relationships between the substance paradigm's particles in ***Figure 3.15*** shows these secondary level hierarchies. Compare this with ***Figure 3.2*** and you can clearly see that the substance paradigm has more structure at the secondary level than the entity paradigm. This structure is what was removed by the entity paradigm's simplification.

6.3 The substance paradigm's solution to the staff example

We can use the earlier staff example to illustrate the substance paradigm's superior semantics. We can show how its secondary hierarchy enables it to reflect the real world more accurately.

When we look at the staff example through substance spectacles, we discover a more consistent system. In this, individual staff (such as Margaret Bottomley) are primary substances and Staff, Salesperson and Account Manager are secondary substances. There is a substance hierarchy in which the Salesperson and Account Manager substances both 'belong to' the Staff substance as shown in ***Figure 3.16***. When the entity paradigm was simplified, hierarchies such as these were flattened, making it difficult to reflect the real world accurately and consistently.

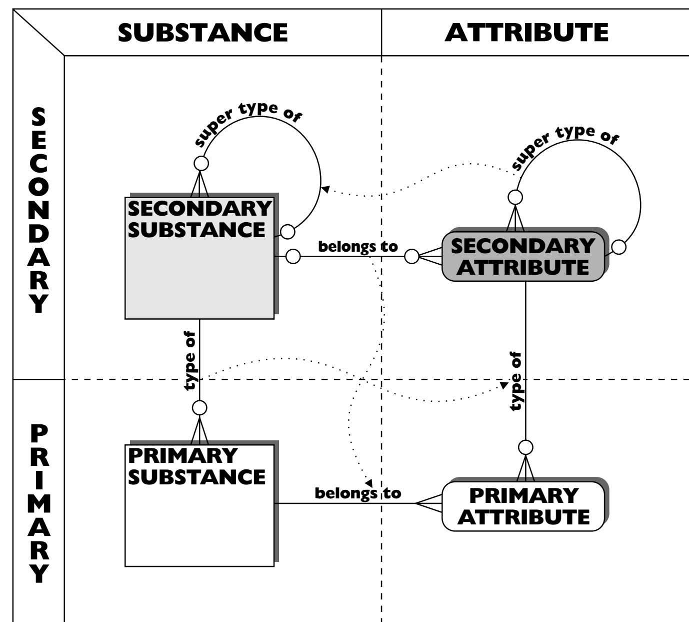


Figure 3.15: Schema of substance paradigm's particles and levels

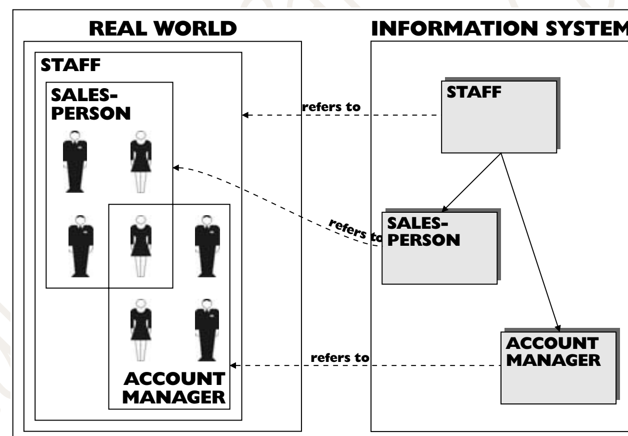


Figure 3.16: Aristotelian staff hierarchy

6.4 Why the substance paradigm was simplified

Why did the secondary hierarchy have to be simplified? When the entity paradigm was being developed, paper and ink were the prevailing information technology. Its two-dimensional structure could not handle the more sophisticated structure of the substance paradigm. This had to be simplified, so that it could operate within paper and ink technology's rows and columns.

The entity paradigm developed, within the constraints of paper and ink technology, in response to operational needs. It did not develop in the same rational way as the substance paradigm. There was, as far as we know, no-one working out what the entity paradigm's fundamental particles were and what they meant. However, as most people see the world through the substance paradigm, it was a natural starting point for the many

© Copyright chrispartridge.com

minds that contributed to the entity paradigm's development. The substance paradigm was too sophisticated for paper and ink technology and so the question was—how to simplify it?

6.5 How the substance paradigm's particles were simplified

The entity paradigm has a basic structure of three levels as shown in ***Figure 3.4***. This results in a framework restricted to a flat entity type level as shown on the right-hand side of ***Figure 3.17***. Compare this with secondary substance's unconstrained multiple-level hierarchy on the left-hand side of the figure.

Figure 3.17 makes it clear that it is really only the substance paradigm's secondary level that needed simplifying. Its primary level can be directly translated into the individual entity level. Whereas, its secondary level hierarchy cannot be directly translated into the flat entity type level. To fit the substance paradigm into the entity paradigm's framework shape, its secondary level—both substance and attribute—has to be flattened down to a single level.

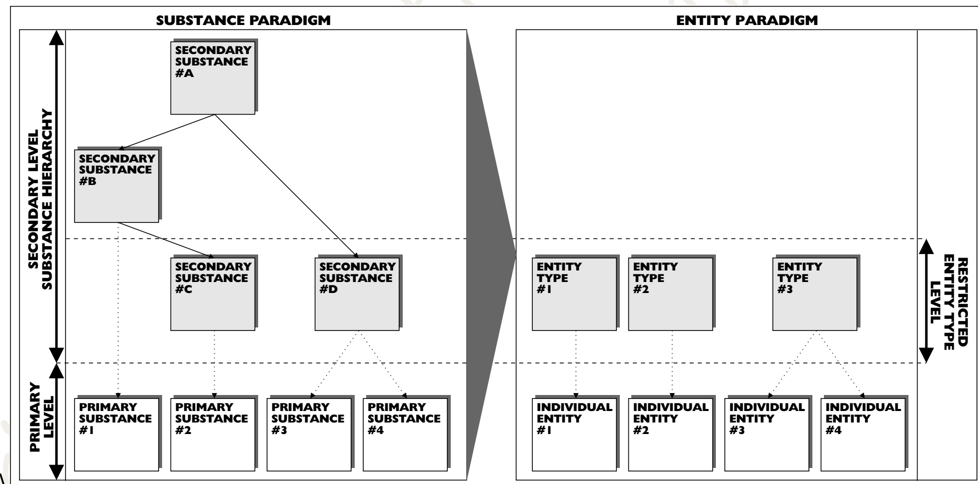


Figure 3.17: The paradigms' different structures

6.5.1 *Selecting the natural type level entity*

For secondary substances, this means that we have to select a single layer, slicing through the hierarchy, to stand as entity types. We can illustrate this with an example. Consider the secondary hierarchy in ***Figure 3.18***. It has a band across it indicating the selected layer of substances. ***Figure 3.19*** shows this layer transformed into entity types. The substances above the selected layer in the hierarchy disappear, while the substances below the selected layer in the hierarchy are transmuted into attributes. A similar transmutation happened in the staff example. The Account Manager and Salesperson substance in ***Figure 3.16*** are flattened into attributes in the staff entity system in ***Figure 3.9***.

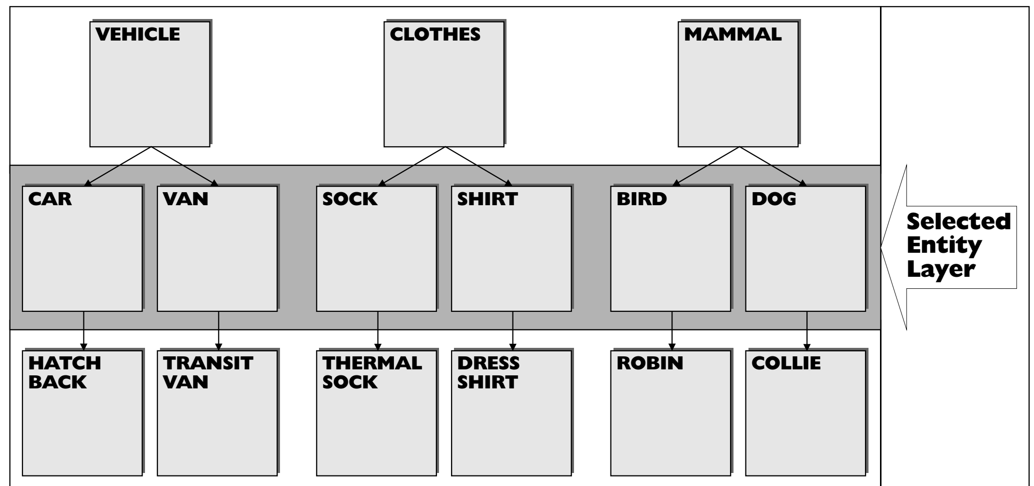


Figure 3.18: Selecting a secondary substance layer

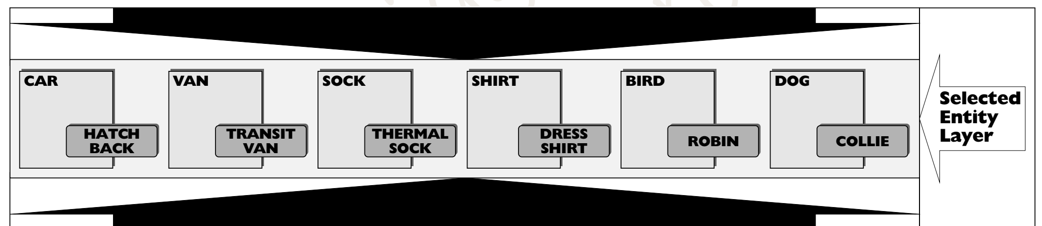


Figure 3.19: The transformed entities

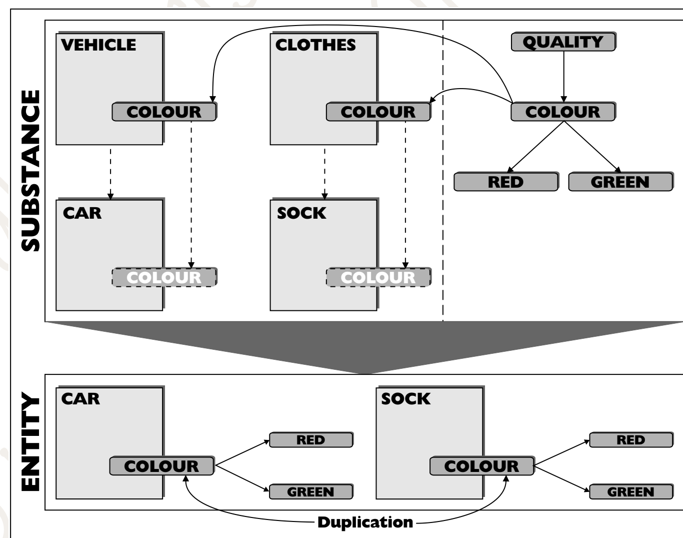


Figure 3.20: Flattening the secondary attribute hierarchy

Selecting a layer in the hierarchy is not as unnatural as it may look. Psychologists have shown that we immediately allocate a thing to a natural level of 'substance'. For instance, most people immediately classify a robin in their garden as a bird rather than the lower level robin or higher level animal. However, this 'natural' level is not a feature of the world. It can vary from person to person. For instance, a bird watcher, unlike other

people, is more likely to immediately classify a robin in his or her garden as a robin (maybe even the variety of robin) rather than a bird.

6.5.2 Making attributes dependent

The natural level of classification helps us to decide how to flatten the secondary substance hierarchy into an entity type layer. It is also a vital factor in the design of good user interfaces. Items presented to users on a screen need to be at their 'natural' level.

It is not just the secondary substance hierarchy we need to flatten; we also need to flatten the secondary attribute hierarchy to fit it into the entity structure. Not only do we have to flatten it, but we also have to make it completely dependent on its corresponding entity type. We can see how this happens in the colour hierarchy example shown in ***Figure 3.20***.

You may have noticed that the dependent attributes belonging to substances above the selected layer, such as vehicle's colour in the example, disappear along with their substances. Notice also that the independent colour attribute hierarchy disappears completely. This means that it can no longer be re-used across the car and sock entity types. In this example, the simplification creates a need for two dependent instances of each colour attribute—one for each entity. This is an example of a general constraint in the entity paradigm. When an attribute has a fixed range of values—as the colour attribute does here—the substance paradigm's independent attribute hierarchy has to be re-constructed anew as dependent attributes for each of the entity types. This significantly reduces the re-use potential.

7 Simplifying the substance paradigm's treatment of relationships

It is not just the substance paradigm's secondary hierarchy that was simplified away. So was part of its treatment of relationships—one of the four key types of things we identified in the last chapter. We now look at how the substance paradigm handles relationships and how this was simplified for the entity paradigm.

7.1 Aristotle's relations and co-relations

In the substance paradigm there are two particles—substances and attributes. Everything has to be one or another of these (or some combination). So we have to use one or another particle to describe relationships such as:

Queen Elizabeth is the mother of Prince Charles

The only practical solution, within the paradigm, is to treat relationships as relational attributes. As 'is the mother of Prince Charles' is the predicate of the sentence, it is an attribute. As 'Queen Elizabeth' is the subject, it is the substance that the attribute belongs to.

However this relationship can also be described as:

Prince Charles is the son of Queen Elizabeth

This and the earlier sentence describe the same relationship. But, a subject–predicate analysis of this sentence gives a different result. In this case, as 'is the son of Queen Elizabeth' is the predicate of the sentence, it is an attribute. As 'Prince Charles' is the subject, it is the substance that the attribute belongs to. We appear to have a problem—two different relational attributes for the same relationship.

Aristotle was well aware of this problem. In Categories, he wrote:

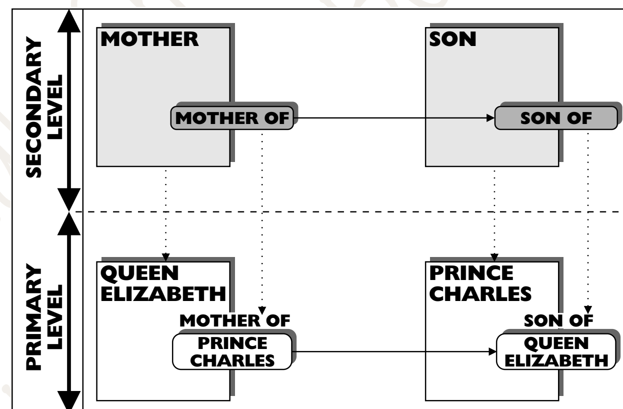
> Let us now turn to Relation. We call a thing relative, when it is said to be such as it is from being of some other thing or, if not, from its being related to something in some other way. Thus 'the greater' is said to be greater by reference to something outside it. For, indeed, when we call a thing 'greater' we mean by that greater than something. 'The double' is called what it is from its being the double of something. For 'double' means double of something. And so with all terms of that kind.

His 'solution' to this problem was facile. He suggested that we give each relational attribute a co-relational or correlational attribute (also called a correlative):

> All relatives have their correlatives. 'Slave' means the slave of a master, and 'master' in turn implies slave. 'Double' means double of its half, just as 'half' means half of its double. By 'greater', again, we mean greater than this or that thing which is less, by 'less' less than that which is greater. So it is with all relative terms.

In our 'Queen Elizabeth is the mother of Prince Charles' example, the correlation of the 'mother of' attribute is the 'son of' attribute. This is illustrated in ***Figure 3.21***.

Figure 3.21: Relational and correlational attributes



We will briefly look at correlational attributes again in ***Chapter 5***, when we re-engineer the relational attribute pattern into its logical counterpart.

7.2 The entity paradigm's simplified treatment of relations

Aristotle was concerned with seeing the real world clearly and accurately. The entity paradigm is more concerned with effectively implementing information in paper and ink technology. This gave it a problem with relational attributes having correlational

© Copyright chris partridge chris.partridge@BORO Centre.com

attributes. The additional correlational attributes are—as far as it is concerned – duplicates. The relational attributes contain all the information it needs. Its solution is to drop all correlational attributes. The entity paradigm's treatment of the 'Queen Elizabeth is the mother of Prince Charles' example looks like ***Figure 3.22***.

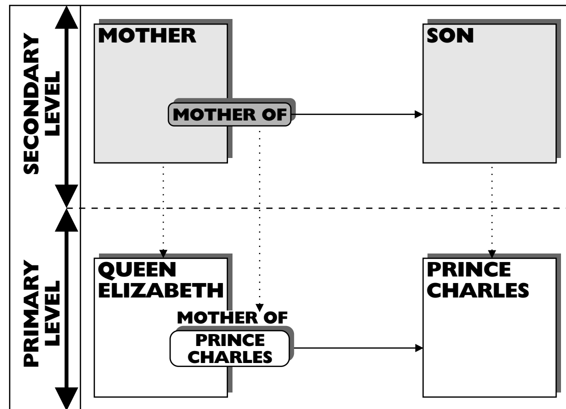


Figure 3.22: Relational attributes without correlational attributes

7.3 The problem with relations as attributes

As in the earlier staff example, here we have a weakness in the paradigm's particles leading to a distorted view of the world. The major distortion caused by the relational attribute particle is clearly visible in ***Figures 3.21*** and ***3.22***. In both figures, the most important part of the relationship, the link between the two substances is drawn as a line. But this link is only implicit in the substance and entity paradigms. Attributes, by their nature, belong to only one substance. Neither paradigm has a particle that can capture explicitly the vital linking element. Because neither paradigm is powerful enough to explicitly reveal this, there is no chance of it being reflected accurately in a business entity model.

There are other distortions. We can follow the substance paradigm and assume that each relational attribute has a correlational attribute. But then we end up with two fundamental particles to handle a single relationship pattern in the business. The entity paradigm avoids this problem by dropping the correlational attribute. But this has its own problems. We can see this in our example. The 'mother of/son of' relationship really involves two entity's, Queen Elizabeth and Prince Charles, and we can only choose one of these entities as the home for the attribute. In ***Figure 3.22***, we chose Queen Elizabeth as its home; but there is no reason why we should not have chosen Prince Charles. The real world only has a single pattern; so this choice is an indication that we are not capturing its patterns accurately.

7.4 Entity business modelling's problem with many-to-many relations

As we have just seen, relational attributes—even without correlations—are an awkward pattern. This has had an unhealthy impact on entity business modelling. One good example is the way in which modellers typically resolve many-to-many relations in entity-oriented models. Consider this example.

© Copyright Chris Partridge chris.partridge@BORO Centre.com

Assume a system records the employees of a company and the projects that they are working on. Assume also that an employee can work on several projects and that a project typically has many employees working on it. Then the situation, from an entity-oriented point of view, is shown in the Venn diagram in ***Figure 3.23***.

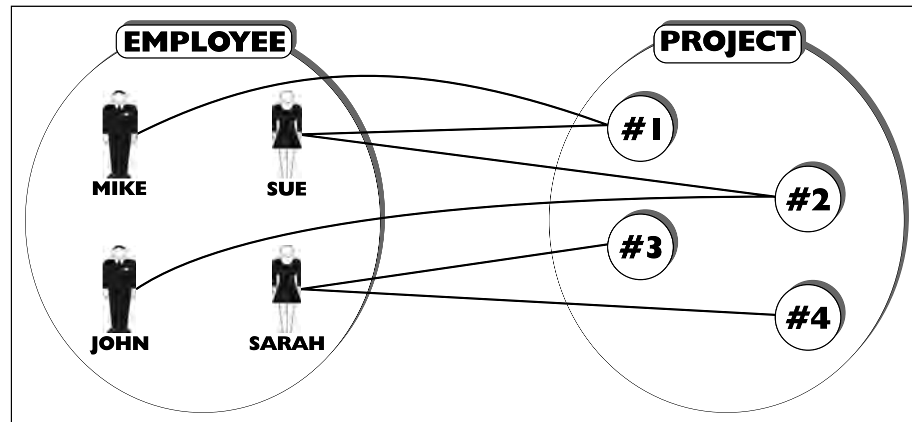


Figure 3.23: Venn diagram for employee/project system

A problem now arises. The attribute pattern is not strong enough to reflect a many-to-many relation such as 'employee works on project'. The traditional solution is to build the model as if there was a new 'relational' pseudo entity, employee–project, with employee and project attributes as shown in ***Figure 3.24***.

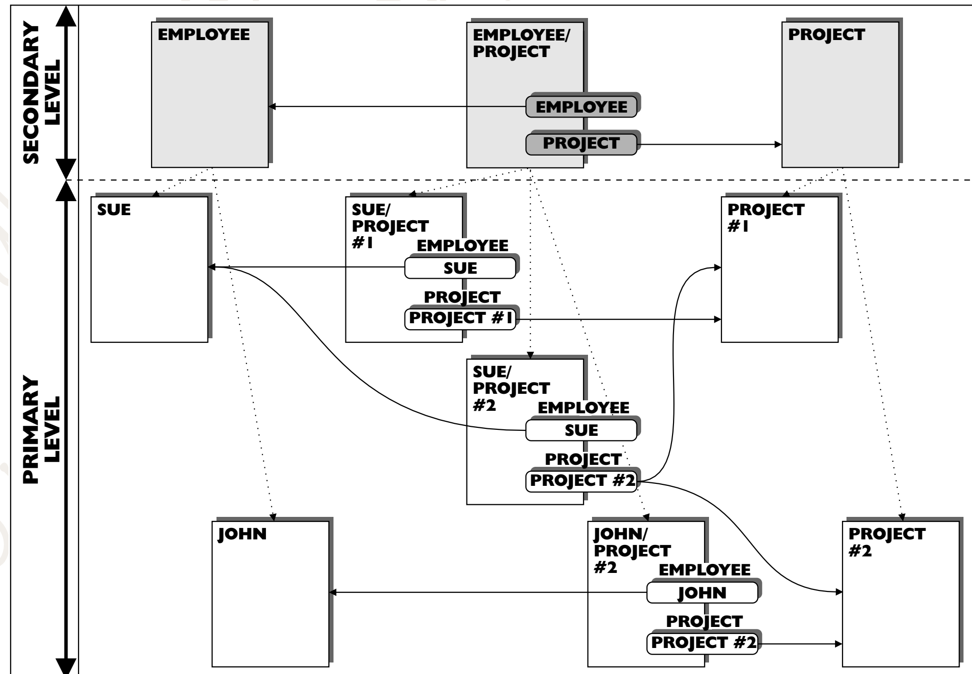


Figure 3.24: Entity model for employee/project relation

From a practical point of view, this model can be used to build a working system. But because we have assumed the new entity exists rather than mapped it, we are faced with awkward questions:

- What is an employee–project entity?
- Is an employee-project as much of an entity as an employee and a project?

The answer to the second question is clearly no. The many-to-many relation has forced us to build the model as if there were employee–project entities, even though there is no external evidence for them. It is the weakness of the attribute pattern when it comes to relationships that forces us into this position. This weakness means that not only can we not reflect the structure of the world directly but we have to construct false 'pseudo' entities.

7.5 A partial solution: entity–attribute–relation modelling

Modellers have recognised this problem of fitting relations into the attribute pattern for some time. They have found a way around the problem that keeps most of the entity paradigm intact. It involves having a new particle to handle relations—a relation particle—with its own new patterns. This extended paradigm is known as the entity-attribute–relation (E–A–R) paradigm. We can see how it works in our example. We now model the problem employee-project relation as a relation not a new entity as shown in ***Figure 3.25***.

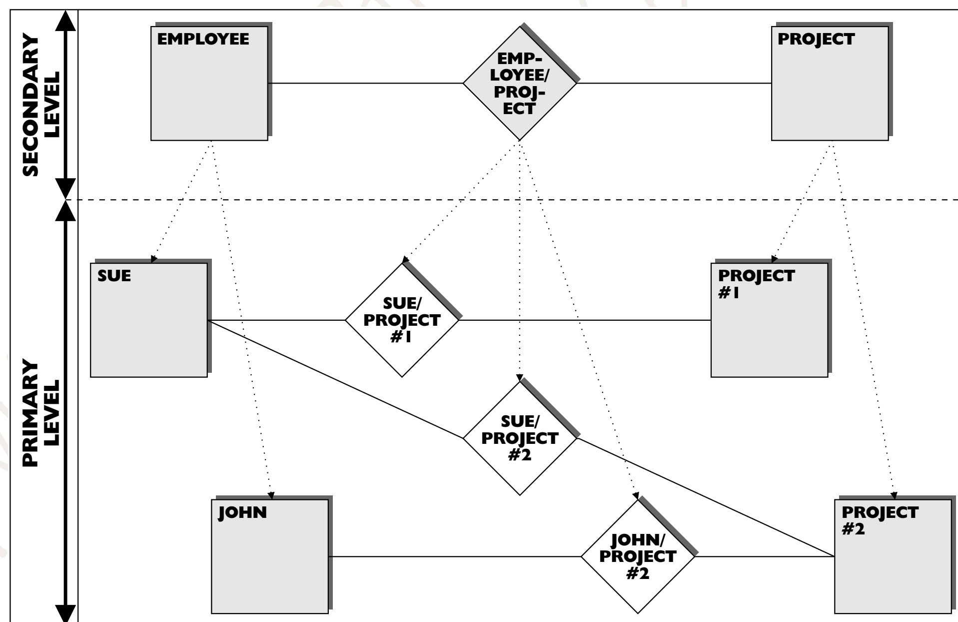


Figure 3.25: Entity–attribute–relation model for employee-project system

Introducing this new particle takes two steps in the right direction. It makes the structure of relations more explicit in the model and it recognises that relations are not entities. But its overall semantics is still solidly based on entities. We shall see that this is shaky ground when we investigate the logical paradigm in ***Chapters 5*** and ***6***.

7.6 Another partial solution: O-O programming languages' group attributes

O-O programming languages adopt a different and apparently simpler solution to the many-to-many relation problem. They allow an object's 'attribute' to have a group of values and so point to many programming 'objects'. This eliminates the need to create extra 'objects' to resolve many-to-many connections. However, while it works from an operational system point of view, from a semantic, modelling, point of view, it does not. This becomes clear when we ask for a semantic explanation of what an attribute with a group of values refers to. There isn't one.

8 Our current way of seeing stored information

The entity paradigm is the natural way for literate people to see information stored on paper or computer systems. It has its foundations in the substance paradigm; our current way of seeing things. And it evolved out of our culture's thousands of years of experience in storing information on paper in lists and tables. Just think how easy most of us find it to draw up lists and tables. We naturally move from the substance paradigm for information in our minds to the entity paradigm for information stored outside them.

8.1 The four key types of things

However, if we assess the entity paradigm's semantic power in terms of the four key types of things identified in ***Chapter 2***, we can see the deleterious effect its simplification has had. It mainly affects generality, where there are two changes and both are for the worse. First, simplifying the secondary hierarchy away has removed the apparatus for handling more and less general types (shown in ***Figure 2.12***). Secondly, it is not practical to accurately reflect the distinction between entity types and attribute types in information systems. The staff example showed how attribute type signs can refer to entity types. ***Figures 3.18*** and ***3.19*** illustrated how this inevitably happens as the secondary hierarchy is simplified.

8.2 Learning to ignore the semantic problems

However, the simplified entity paradigm was, and is, a very successful means of managing business information. It may have problems reflecting the structure of the world directly. But, while the paradigm is successful, it does not seem to make sense to pursue these problems. In fact, the best policy seems to be—learn to ignore them.

This is how most people work with the entity paradigm. To see this just ask yourself whether the staff example above proves to you that the entity paradigm is inherently wrong. I suspect many of you will say that it does not; that all it highlights is an academic issue about semantics. In this way, we legitimise ignoring the problem.

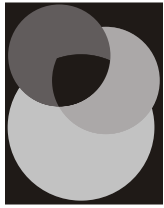
However, as we go through the re-engineering to the object paradigm and we get a better understanding of the semantic issues, we will develop a different perspective on this.

It will become clearer and clearer that if we ignore these semantic issues, then we will miss a big opportunity for improving business modelling and so computer systems.

9 The next chapter

In the next chapter we look in more detail at how the substance paradigm addresses these semantic issues.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 4

The Substance Paradigm's Semantics

1 Introduction
2 The semantics of the fundamental substance and attribute particles
3 Changes—a key type of thing
4 Generalising re-usable substance and attribute patterns
5 Our current way of seeing
6 The four key types of things
7 What's next

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction

In the last chapter, we saw how the entity paradigm's simplification confused its semantics. In this chapter, we look at the substance paradigm's clearer semantics. We shall see that it is not only consistent but extremely powerful. In particular, we shall see how its secondary hierarchy significantly increases the potential for re-use.

Developing a firm grasp of substance semantics is important because, in our journey to the object paradigm, we will use it as a benchmark, checking whether we are making progress. This is not as easy as it sounds. Unlike technology, which clearly improves over time, conceptual systems (such as semantics) do not progress quite so clearly. Old semantics can be just as good as, if not better than, new semantics. We will see, for example, in Part Three how logical semantics' attempts to improve on the semantics for change only work partially.

2 The semantics of the fundamental substance and attribute particles

In the last chapter, we looked at the framework of the substance paradigm in terms of what was taken out during the construction of the entity paradigm. This is useful and important. But now we go to the heart of the matter; we look at the semantics of the fundamental substance and attribute particles. This explains why the substance paradigm has the framework it does, and so gives us an insight into why the simplified entity and attribute particles are the way they are.

2.1 Primary particles

The primary level is where our ideas of substance and attribute particles make closest contact with actual particles in the real world. There is a simple, direct, one-to-one relationship between the two. Substance is, in some ways, more fundamental than attributes so we start with it.

2.1.1 Underlying primary substance

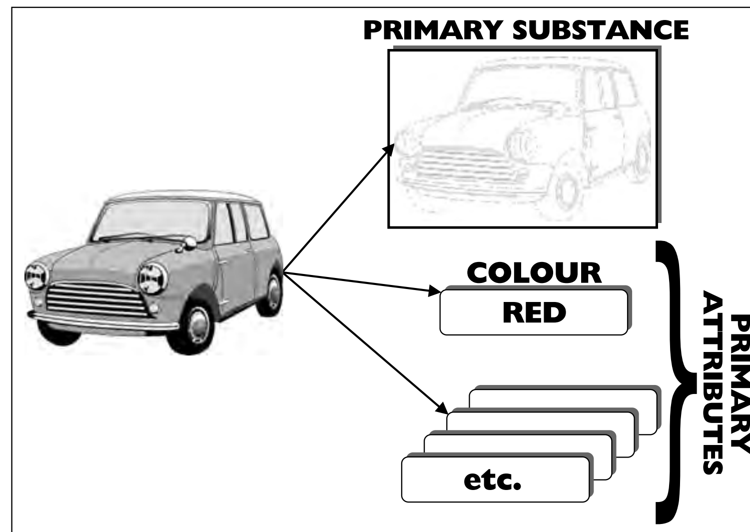
Some people find the semantics of Aristotle's primary substance difficult to grasp. They find it easy to manipulate the primary substance signs (such as the words 'my car') used for information, but find it difficult to see what these signs refer to. It is not so much that they find it difficult in itself. It is more that, when looked at directly, primary substances appear odd.

We can get an understanding of what primary substance is from this simple thought experiment. Imagine my car. Imagine each of its attributes in turn and then imagine the

© Copyright Chris Partridge chris.partridge@BOROCentre.com

car without that attribute. Eventually we are left with a ghostly hulk that has no attributes as shown in ***Figure 4.1***. This is my car's substance.

Figure 4.1: Underlying primary substance



For Aristotle, substance was a neutral foundation for things. Each thing was a single inert hunk of matter impregnated by a number of attributes, rather like water soaking into a sponge. This is why, when we take my car and strip away all it's attributes (mentally, we cannot do this physically), all that we are left with is its substance. This ghost of the original car is a single inert hunk of matter that is no impregnated by attributes.

Most people unconsciously use Aristotle's substance paradigm when seeing attributes. They are happy seeing attributes that belong to something. The problems arise when they start asking themselves what the attributes belong to. The logical (and historically correct) answer is a ghostly substance. But this seems, to modern eyes, unbelievable.

2.1.2 Modern science's view of matter

Part of the reason people now find the notion of substance unbelievable is that its view of things containing a neutral hunk of inert matter is completely foreign to modern science. Since the 17th century, scientists have regarded matter as the small particles of which things are composed. They believe that the way these small particles of matter behave determines the thing's attributes. So a piece of lead is heavy because its particles are heavy; a piece of cloth is red because its particles emit light rays of the right wavelength.

Modern scientists' particle matter is very different from Aristotelian substance. This raises a mental barrier to us accepting that, in most of our everyday life, we see things in terms of substances and their attributes. However, we tend to see a body's attributes as belonging to something, and that something is substance. There is no way for us to escape this fact because attributes are logically dependent for their existence upon substances.

© Copyright Chris Partridge BORO Centre .com

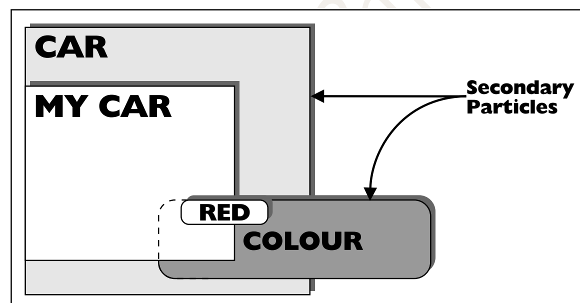
2.2 Secondary particles

The semantics of the secondary particles in the substance paradigm is not well defined. Aristotle expresses the difference between primary and secondary substance in his Categories as follows:

Substance in the most literal and primary and common sense of the term is that which is neither predicated of a subject nor exists in a subject, as for example, the individual man or horse. Those things are called secondary substances to which, as species, belong the things called substances in the primary sense and also the genera of these species. For example, the individual man belongs to the species man, and the genus of the species is animal. These, then, are called secondary substances, as for example, both man and animal.

The notion of how a primary substance 'belongs' to a secondary substance is unclear. I find that the easiest way to think about it is to consider a secondary substance as an amalgamation of primary substances. Similarly, I think of a secondary attribute as an amalgamation of primary attributes. So, for example, my car's primary substance is part of car secondary substance and my car's red attribute is part of the secondary colour attribute. This is shown schematically in ***Figure 4.2***.

Figure 4.2: Secondary particles



3 Changes—a key type of thing

As noted in ***Chapter 2***, changes are one of the key types of things our re-engineering focuses on. Working out what changes are, eventually leads us to the object paradigm. The substance paradigm lays the groundwork with a basic set of patterns. When we see how Aristotle used these patterns to describe changes, we will better appreciate the sophistication of his paradigm.

We start by looking at what changes are in the substance paradigm. We then look at a problem with changes and at how Aristotle's substance particle neatly avoids it. We then look at how Aristotle used his solution as a general pattern for changes.

© Copyright chris.partridge@borocentre.com

3.1 Changes in the substance paradigm

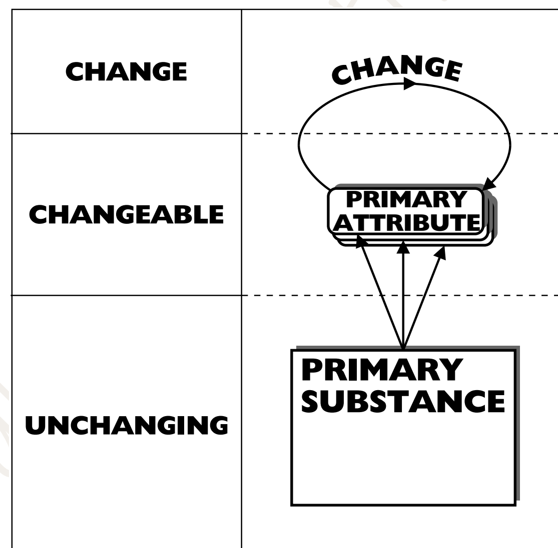
Once we recognise that primary substance is the hunk of neutral inert matter underlying things, it becomes clear that attributes play a vital role in explaining what changes are.

3.1.1 Defining change

If substance is inert and does not change, then when something changes it must be an attribute that changes. It is like a chameleon changing colour—the chameleon itself does not change, only its colour attribute. This division into unchanging substance and changing attributes dictates the substance paradigm's definition of what change is; it is one attribute changing into another.

This leads to the three-tier structure for dealing with change shown in ***Figure 4.3***. At the bottom is unchanging substance, in the middle is potentially changing attributes and at the top is the actual change process, the changing of attributes. This makes change, one of our key types of things, an implicit third non-thing kind of particle.

Figure 4.3: Primary levels of change



3.1.2 Accidental (changing) and essential (unchanging) attributes

There is one small addition to the substance paradigm's framework for change. One of the first things Aristotle and his followers noticed is that not all attributes are capable of change—some are essential to the substance's existence. These were called essential attributes (from the Latin *esse*—to be). When we talk about the 'essential nature' of something, we are harking back to this Aristotelian distinction. Attributes that could potentially change were called accidental (from the Latin *accidere*—to happen). This distinction is shown schematically in ***Figure 4.4***.

© Copyright chrispartridge.com

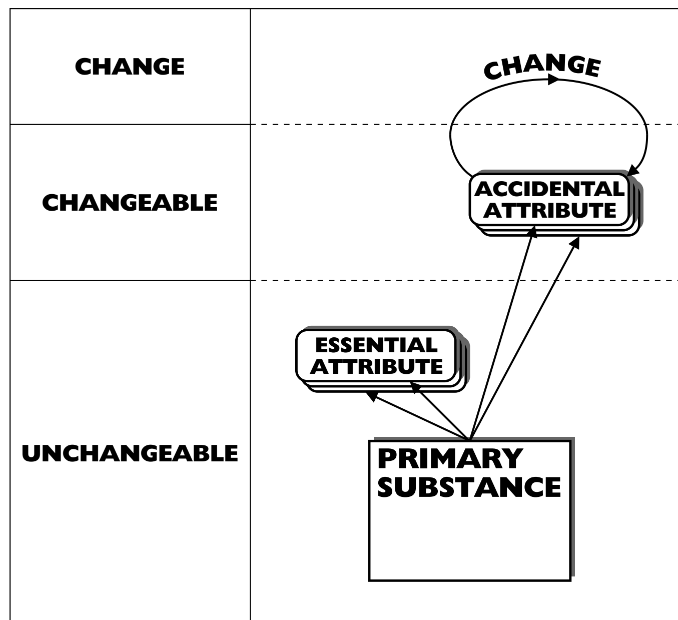


Figure 4.4: Essential and accidental attributes

This is an easy distinction for people used to working with computer systems to comprehend. They are familiar with files (secondary substances) in computer systems that have fields (secondary attributes). They are familiar with programs for amending the file's records (primary substances), which enable users to change some fields (primary attributes). These fields are the computing equivalent of accidental attributes. Fields that the users are not allowed to change are typically the computing equivalent of essential attributes.

3.2 A problem with changes

We can see the benefits of an unchanging substance if we look at how it handles a problem with changes well known to the Ancient Greeks. One of them, Heraclitus of Ephesus, was referring to this problem when he asked his famous question:

> Can we bathe in the same river twice?

The answer is obviously both yes and no. Yes, it is the same river—no, it is not the same water.

3.2.1 The problem—what makes something the same?

But why is this the answer? Why do we call the river the same at different times? It stays roughly the same size and shape and stays in roughly the same position. But this cannot be what makes it the same river. To see this, consider a lepidopter. It starts out as a caterpillar then metamorphoses into a pupa then metamorphoses again into a butterfly (shown in ***Figure 4.5***). It does not stay even roughly the same size and shape, it certainly does not stay in roughly the same position. Yet we have no problems with saying

© Copyright Chris Partridge chris.partridge@BorOCentre.com

it is the same thing through all its changes. There must be something other than similar size and shape making it the same.

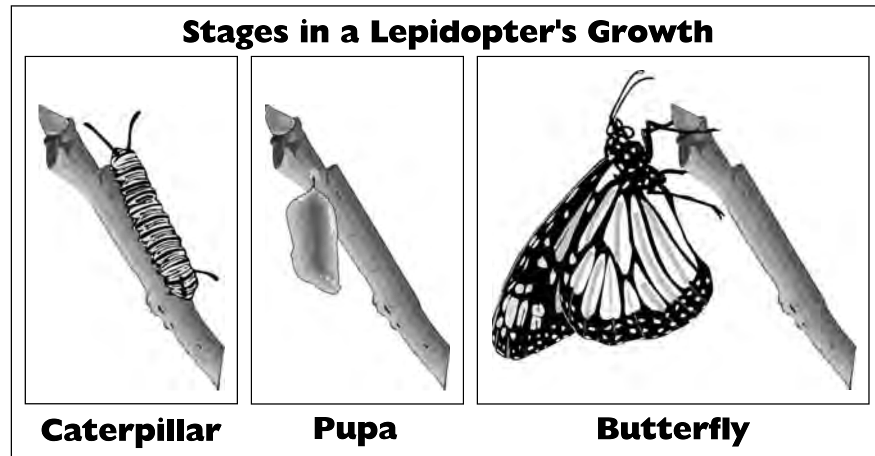


Figure 4.5: A changing lepidopter

3.2.2 The answer—unchanging substance

The substance paradigm has a simple solution to the problem. It suggests that it is the lepidopter and river's unchanging substance that makes them the same. This is illustrated schematically for lepidopters in ***Figure 4.6***.

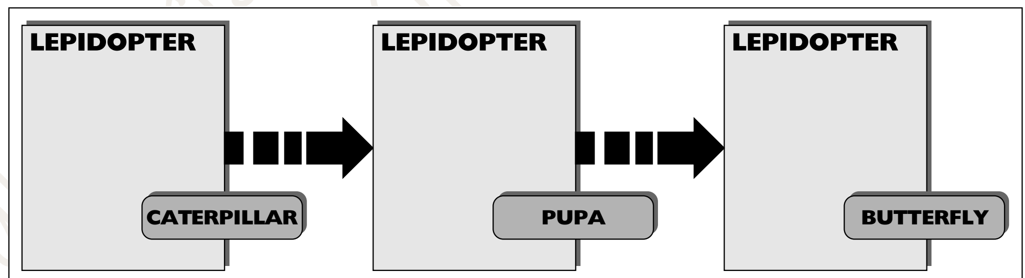


Figure 4.6: Unchanging substance

© Copyright chris.partridge@BORO Centre

3.3 Aristotle's general pattern for change

Once Aristotle established that change was the process of one attribute changing into another, he used this as the general pattern for change.

3.3.1 Change as change of attributes

He claimed that this general pattern covered a wide variety of, to us now, unrelated changes. Examples include:

- Growth (the transformation of an acorn to an oak or the growth of a child into adulthood),
- Alterations of intensity (the heating up of a cold iron bar), as well as

- Change of position (the falling of a stone).

These patterns are illustrated in ***Figure 4.7***.

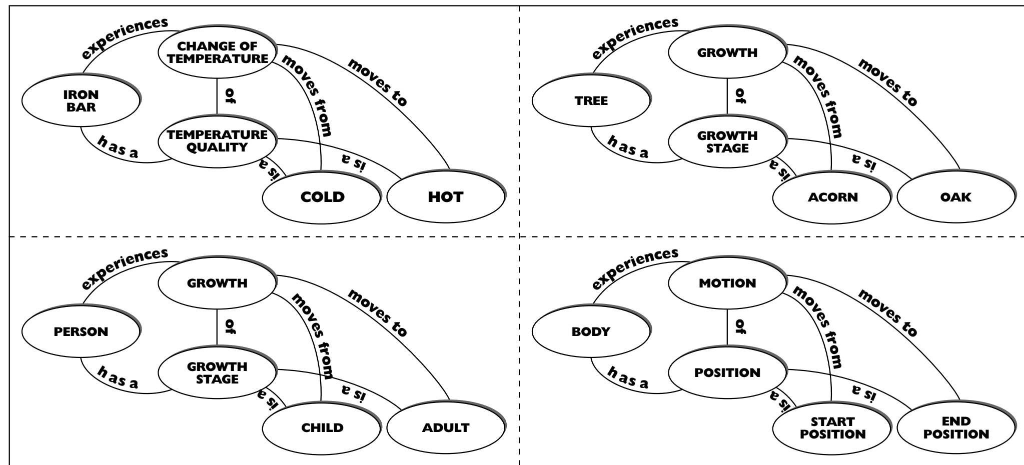


Figure 4.7: Examples of patterns of change

For Aristotle, all these various types of change were similar; he saw them as members of a single natural family, each exhibiting the same general pattern. Because they all shared the same pattern, he could, and did, generalise them into the single comprehensive pattern illustrated in ***Figure 4.8***.

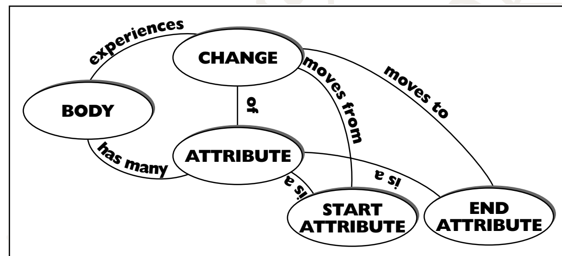


Figure 4.8: General pattern of Aristotelian change

This general pattern bound the particular patterns of change closer together. It also had a big effect on how Aristotle and his followers thought about change. For them, analysing change primarily involved identifying the basic characteristics that apply to all the members of the family of change patterns. Analysing the individual characteristics of the various sub-types of change was much less important. In this way, the general pattern influenced the way in which people thought of the lower level patterns.

3.4 Aristotle's pattern for motion

One good example of how the general pattern shaped the lower level patterns is motion—change of position. Within the Aristotelian paradigm, the shape of motion's pattern was largely dictated by the shape of the general change pattern. (Compare the pattern for motion in the bottom right corner of ***Figure 4.7*** with the general pattern in ***Figure 4.8***.)

We can see this in the substance paradigm's resolution of an ancient paradox. The Ancient Greek thinker, Zeno of Elea (who lived around the early 5th century BC), raised a number of paradoxes. The one we are interested in is based on a problem with change and relates to the problem of motion. It appears to prove that moving arrows could not be moving.

We are happy to accept that the phrase 'is red' refers to a red attribute. So it is easy to assume that because the phrase 'is moving' in the sentence 'the arrow is moving' looks similar, it must also refer to something. Within the substance paradigm, we would assume it referred to a moving attribute. The similar linguistic shape turns out to be misleading. Zeno proposed a simple thought experiment that showed the arrow cannot be moving.

Consider a situation where an arrow has been shot. Think of it the instant after it leaves the bow. It would have a particular position, say two inches in front of the bowstring. Is it moving; does it have a moving attribute? One's initial reaction is to say yes. However, on reflection, if the arrow is at a particular point at a particular instant in time, it cannot be moving—it must be at rest.

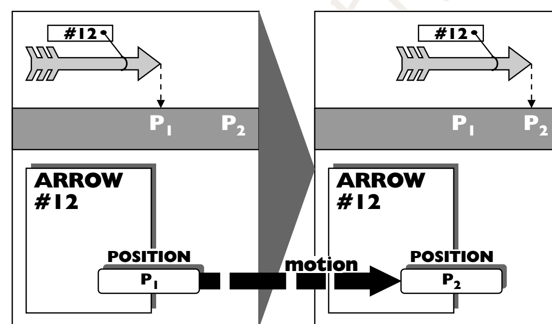


Figure 4.9: Aristotelian motion of an arrow

Consider the arrow a second later. It is again at rest in a particular position and again, not moving. In fact, if we consider the arrow at any point in its trajectory, it will be at rest, not moving. If it is not moving, how can it have a moving attribute. This led Zeno and others to say motion is, in one sense, an illusion. We can still explain motion within the substance paradigm. It is changing one position attribute for another—as shown in ***Figure 4.9***. This falls neatly under the general change pattern.

The general pattern for change inherent in the substance paradigm elegantly explains what motion is in a way that avoids Zeno's paradox. It is a change in the position attribute. The 20th century thinker, Bertrand Russell, called this an 'at-at' approach — the arrow is 'at' one position 'at' one time and 'at' another position 'at' another time.

Essentially what the thought experiment brings to our attention is that even though 'is moving' looks like an attribute it cannot be one. Within the substance paradigm, 'is moving' refers to a process of changing position attributes. In terms of the three levels of change (shown in ***Figure 4.3***), it belongs to the top level and so is neither a substance nor an attribute.

Zeno's paradox is a useful way of assessing how well a paradigm deals with change. In ***Chapter 6***, the paradox will reveal the logical paradigm's similar implicit change parti-

cle—dynamic classifications. It is only in the object paradigm that the paradox is resolved with a type of object particle that explicitly captures the pattern for change. We shall see how this new type of object is re-engineered in ***Chapter 8***.

4 Generalising re-usable substance and attribute patterns

The substance paradigm has significantly more potential for generalisation, and so re-use, than the entity paradigm. Because it is a more sophisticated version of the entity paradigm, it retains all the entity paradigm's potential for re-use, and supplements it with its own. This increased potential comes from the generalisation inherent in the secondary level hierarchies. We now look at this and also at how Aristotle tried to harness its power into a general hierarchy of types of things—the categories.

4.1 Inheritance down the secondary level hierarchy

We saw in the last chapter, how entity paradigm re-use operates at the individual level. How it can fix a pattern of attribute types for an entity type, which is then (re-)used for its individual entities (shown in ***Figure 3.6***). In the substance paradigm, generalisation and re-use operate in the secondary level hierarchies. This involves a higher level secondary substance fixing patterns of secondary attributes for lower level secondary substances.

4.2 Inheriting secondary attributes

For example, if the secondary substance vehicle has a colour attribute then this attribute is inherited by all the secondary substances below it in the hierarchy. As shown in ***Figure 4.10***, this includes the car and van secondary substances. The figure shows three colour attributes. However only one of these, the vehicle's colour attribute, actually exists. The other two, the car and van substances' inherited colour attributes are there to illustrate where vehicle's colour attribute is being inherited.

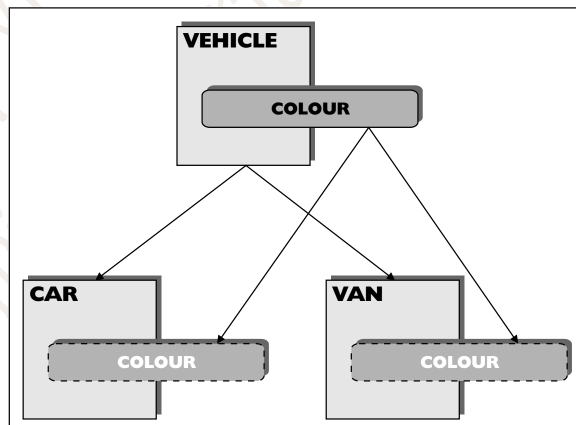
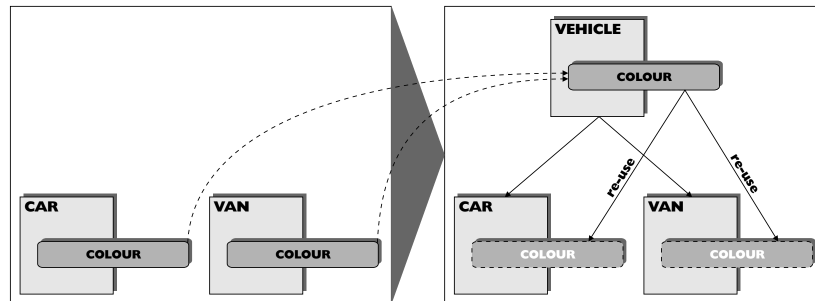


Figure 4.10: Inheriting secondary attributes down the secondary substance hierarchy

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

This secondary substance hierarchy can be used to compact more information in less space. ***Figure 4.11*** illustrates how this works. On its left-hand side is a model with no secondary substance hierarchy, where the car and van substances both have a colour attribute. On its right hand side is the same model with a secondary hierarchy. In this model, there is only one colour attribute. This belongs to the vehicle substance and is, as shown, inherited by the car and van substances. In this very simple example, two attributes are compacted into one.

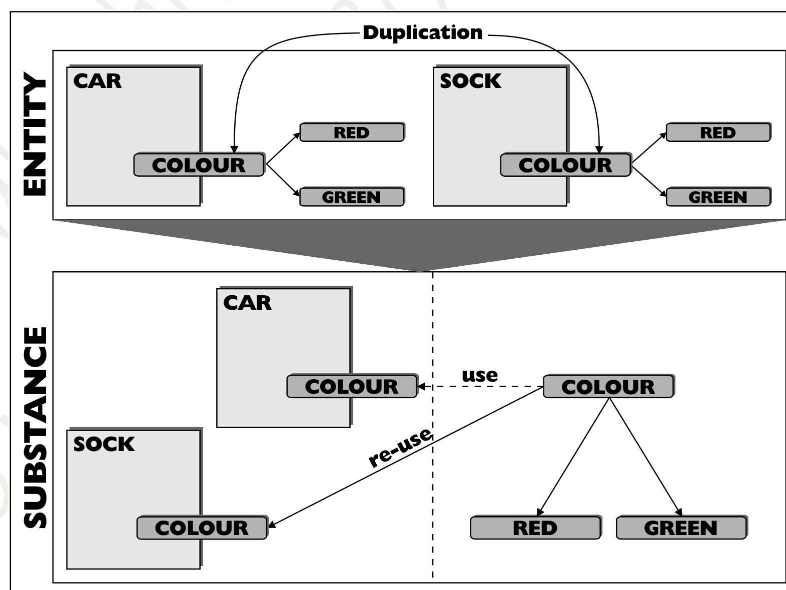
Figure 4.11: Re-use compacting information



4.2.1 Re-use across secondary substances

There is another form of re-use that occurs in the substance paradigm. Because the secondary attribute hierarchies are independent of substance, it is possible for them to be re-used across secondary substances. Like attribute inheritance, this operates at the secondary level and through re-use leads to compacting.

Figure 4.12: Re-using the colour attribute hierarchy



It is easiest to see how this works with an example. We use cars and colours again. There is a colour attribute hierarchy that is linked to the car substance hierarchy. It is, however, independent of the hierarchy, which means it can be linked to other substances. It could, for example, be linked to socks. These are coloured; some socks are

red, some green. When we start analysing the sock secondary substance, we need to recognise that it is linked to the colour attribute hierarchy—as shown in the substance section of ***Figure 4.12***. If we did not have a secondary hierarchy we would have to construct the colour attribute anew—as shown in the entity section of ***Figure 4.12***.

The higher up the secondary substance hierarchy an independent attribute hierarchy is connected, the more fruitful re-use and thus compacting we get. Consider what happens when we generalise the independent colour attribute connections in the substance section of ***Figure 4.12*** up a level. We take the connections from car up to vehicle and from sock up to clothes—as shown in ***Figure 4.13***. Even in this simple example the scale of compacting is significant. The colour attribute hierarchy does not have to be rebuilt for each type of vehicle or clothes. This kind of compacting cannot be done in the entity paradigm—without a secondary level hierarchy in its framework, it just is not powerful enough.

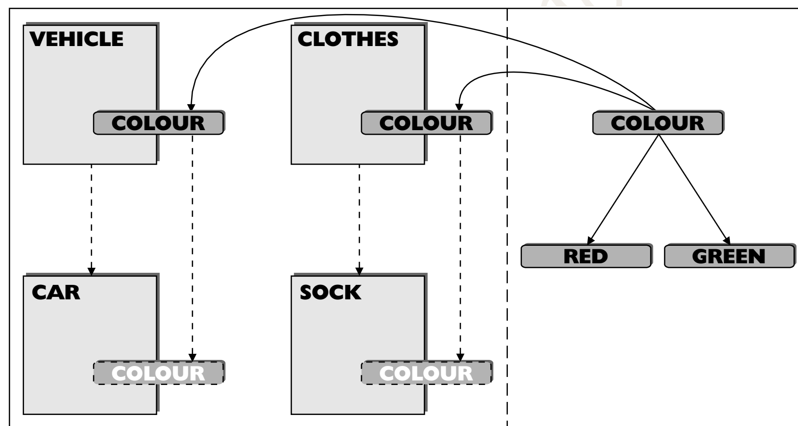


Figure 4.13: Generalising the re-use of the colour attribute hierarchy

4.3 Extending the framework for re-use—the Aristotelian categories

Aristotle also made another very important step for generalisation and so for re-use. He suggested that there was a general framework below the level of fundamental particles. This would mean that all information systems, computer or otherwise, could share a common, high level, framework.

Today it is normal for people in different corporations to use computer systems with different frameworks. It is even common for people within a large organisation to use systems with different frameworks. If there was one common, high level, framework across all these systems, this would greatly simplify integrating information.

All those centuries ago, Aristotle saw the need for a wider general framework. He outlined his proposal for a system of ten categories that identified specific types of secondary attributes. This was then developed and enhanced by his followers. In ***Chapter 11***, we will see how the system of categories develops into the object-oriented notion of a general lexicon.

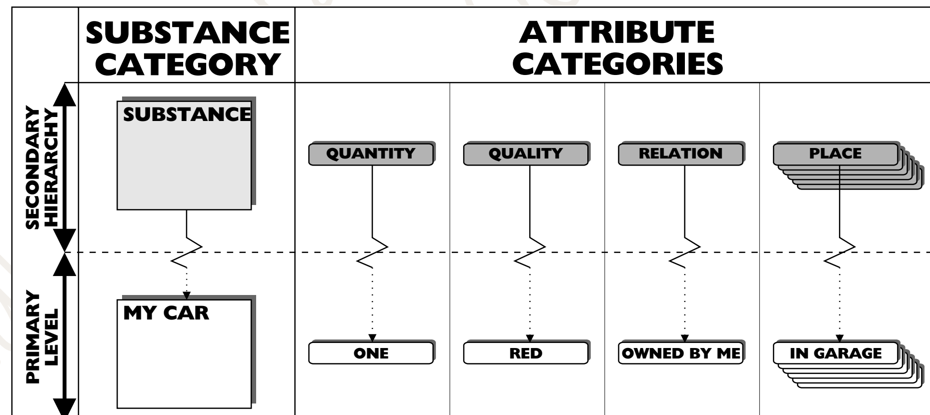
4.3.1 Types of categories

Aristotle worked from an analysis of language. He found, or thought he found, that words or phrases, and so the things they referred to, fell into one or more of ten categories. These were:

- Substance,
- Quantity,
- Quality,
- Relation,
- Place,
- Time,
- Posture,
- State,
- Action, and
- Passion.

In this framework, the nouns 'plant' and 'animal' signify kinds of (secondary) substance and so are in the category substance. The noun 'colour' signifies a quality and so is in the category quality. The first category is substance, the other nine categories are kinds of attributes. So the category structure ends up looking like ***Figure 4.14***.

Figure 4.14: Category structure



4.3.1.1 Tree structure of categories

The ten categories were only the top level of the structure. Below each of them there were divisions and sub-divisions. Aristotle took a relaxed view on whether the list of categories was exhaustive and whether categories could overlap (in other words, have a lattice structure).

However, his followers, like all followers, moulded Aristotle's relaxed view into a stricter orthodoxy. For them, there were ten mutually exclusive categories whose divisions and sub-divisions had also to be mutually exclusive. They developed the traditional system of definition by genus and differentia—the 'method of division'. This starts with a very general classification (the genus) and divides it into smaller mutually exclusive types

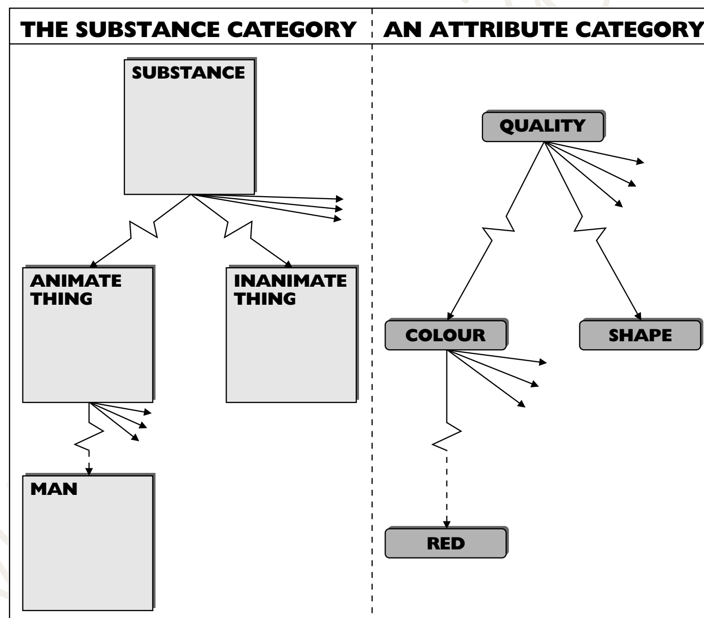
© Copyright chris partridge BORO Centre.com

(species). This is done by means of some property (the differentia), which every member of the genus either does or does not have. The result is a tree structure.

The simplest and best known system of categories was developed by Porphyry, a 3rd century AD commentator on Aristotle's categories. His 'Tree of Porphyry' started by dividing things into material (bodies) and the immaterial; bodies into the animate (living things) and the inanimate; living things into those that had sensation (animals) and those that did not (vegetables); and the animals into rational (man) and non-rational (brutes). This served as a model for most subsequent systems of taxonomy. For example, the modern classification of the animal kingdom based on work done by the English naturalist John Ray (1627–1705), and the botanical classification devised by the Swedish taxonomist, Linnaeus (Carl von Linné, 1707–78).

Similar divisions are made in the attribute categories. For example, colour in the attribute category quality is divided into red, blue, green, etc. and then further divided and sub-divided. The shape of the resulting structure is shown in ***Figure 4.15***.

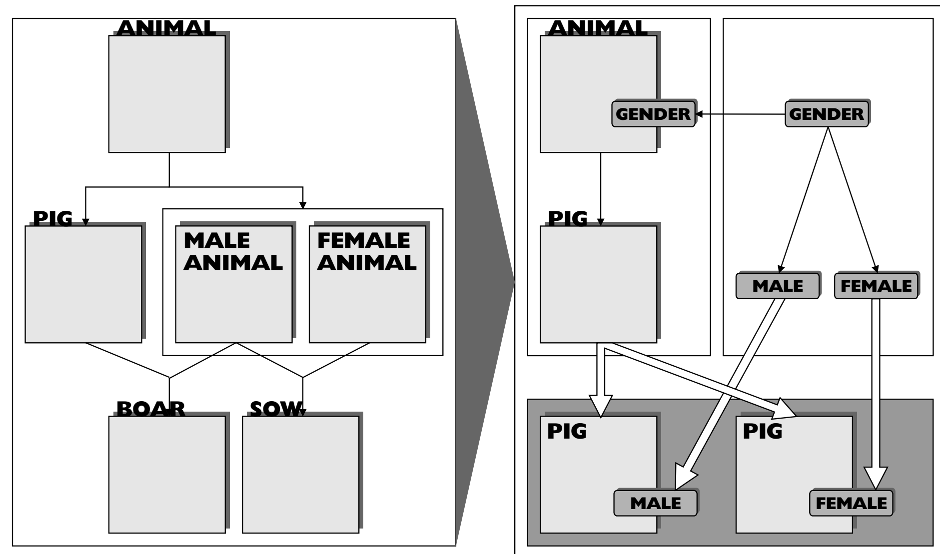
Figure 4.15: Category tree structure



4.3.1.2 Categories as a rudimentary lattice

A tree structure is too constraining to reflect the world adequately. However, it would be incorrect to describe Aristotle's system of categories as a pure tree structure. It is really a type of rudimentary lattice structure built using parallel tree structures, one that is less constraining than a simple tree structure. For example, the lattice shown on the left-hand side of ***Figure 4.16*** would be translated into a tree substance hierarchy and a parallel gender attribute hierarchy. However, these parallel tree structures are not as powerful as a full lattice structure. In other words, they are not really powerful enough to describe the type of structures that exist in the real world.

Figure 4.16: Parallel tree structures – rudimentary lattice structure



What is interesting is that this tree constraint is not necessary to the substance paradigm. Aristotle's followers imposed it in the (mistaken) belief that they were making the structure more organised. To them it somehow seemed better if each category was divided into *mutually exclusive* sub-categories. This shows how deeply the tree way of seeing was embedded in people's minds then—as it still is now.

4.3.1.3 Single inheritance and OOPs

Aristotelian categories have been enormously influential. They are still a powerful influence on the way we see and 'categorise' the world. We can see this influence in O-O programming languages. Early versions had what was called a single inheritance structure—what we have been calling here a tree structure.

Now these languages have developed multiple inheritance structures, but programmers still have difficulty in breaking away from the tree category way of seeing things. For example, at a recent O-O conference, most speakers who talked about multiple inheritance said they had found it was of limited use. Which it is, if you are still working within a tree category pattern.

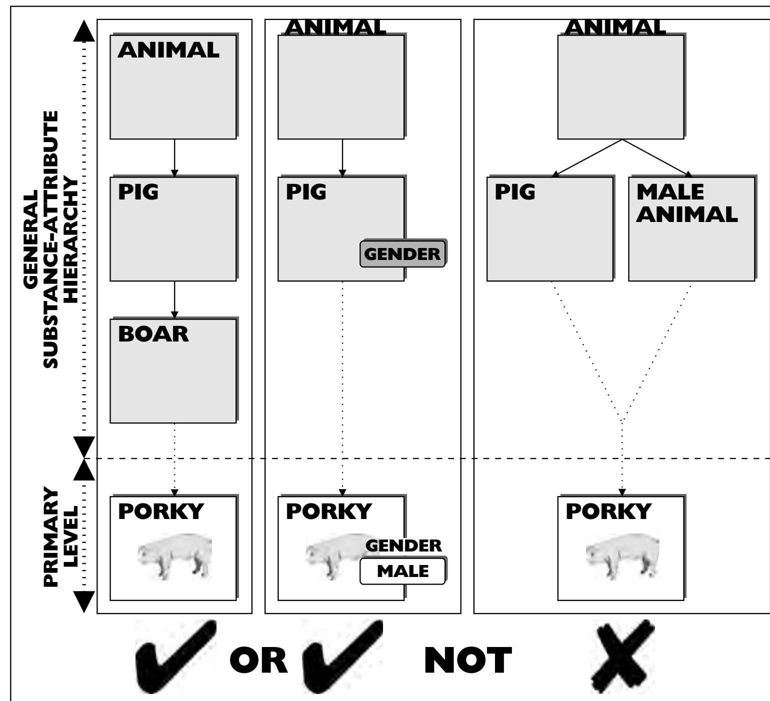
4.4 Relation between primary and secondary levels

In the substance and entity paradigm, the relationship between the primary and secondary levels is refreshingly simple. Primary level particles belong to one and only one secondary level particle. This is known as single classification. A more flexible relationship is possible—at least from a structural point of view—where primary level particles can belong to more than one secondary level particle. This is known as multiple classification.

4.4.1 *Single and multiple classification*

Part of the reason that both paradigms have a single classification framework is in the nature of substance. This fosters a feeling that primary substances are of particular type and only of that type (in other words, belong to one particular secondary substance). This means that the possibility of multiple classification is not naturally considered by people working within the paradigm.

Figure 4.17: Tendency towards single classification



However, if they were to consider it then there would be semantic problems. If the hunk of inert matter was composed of two substances, would the two substances be thoroughly mixed? How would it inherit the patterns of attributes from both substances? If the substances were mixed, would the attributes belong to the whole mixture or only those bits of substance that inherited them?

In ***Figure 4.17*** we can see the structural differences between these two types of classification illustrated. If multiple classification were allowed, we could classify Porky as a pig and male animal. Because we are restricted to single classification, we have two options. We can identify a new substance, boar (male pig), instead of male animal. Because this new substance only belongs to the pig substance, no multiple classification is involved. Or we can treat male as an attribute—again this does not involve multiple classification. These three options are shown in ***Figure 4.17***.

Interestingly this tendency towards single classification means that the parallel tree structure format of the categories (shown in ***Figure 4.16***) is preserved across the primary–secondary level divide. As ***Figure 4.17*** shows, multiple classification allows a lattice across the divide; whereas, single classification restricts the link to a tree structure.

4.4.2 *Static and dynamic classification*

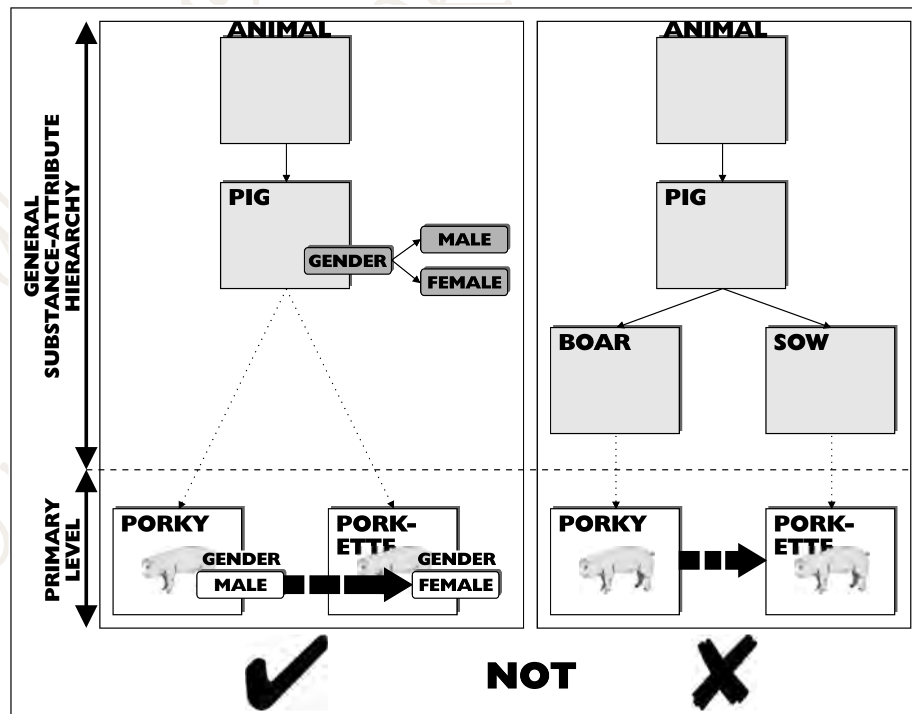
Seeing primary substance as a homogenous hunk of matter underlying things has another effect on the relationship between the primary and secondary levels. We tend to assume that a primary substance always belongs to the same secondary substance and that the relationship between the two never changes. This is called static classification. If a primary substance could change its secondary substance, then the link between the two would be called a dynamic classification.

The reason for the substance paradigm restricting itself to static classification is, like single classification, rooted in the nature of substance. Substance gives a body its identity over time. So the idea of a substance changing its type appears contradictory. If something's substance changed its type, how could it remain the same thing? If something changes, then—within the substance paradigm —it must be an attribute.

We can see how this works in an example. Assume that Porky the pig has a sex change—he/she starts off as a boar and ends up a sow. If dynamic classification were allowed, we would classify Porky initially as boar secondary substance. Then, during the sex change operation, dynamically change the classification to sow substance.

However, the substance paradigm does not allow dynamic classification. As Porky's sex changes, this means that, by definition, it is an attribute. So what is happening is a gender attribute changing, not a boar substance being re-classified. The two alternatives are shown in ***Figure 4.18***.

Figure 4.18: Tendency towards static classification



4.4.3 O-O programming languages

This static classification aspect of the substance paradigm has, like single classification, influenced the development of O-O programming languages. These typically follow the substance paradigm in having a static classification framework, where objects cannot dynamically change type. This means that, as in the substance paradigm, attributes have to handle change.

5 Our current way of seeing

Static classification in O-O programming languages is just one of a myriad of ways in which the substance paradigm has influenced the way we now see things. Over the centuries, Aristotle's paradigm has embedded itself deeper and deeper in our consciousness, until it now seems a natural and normal way to see. As with most paradigms, this works at an unconscious level.

What this chapter has done is make it conscious—revealing the semantics at the heart of the substance paradigm and so also the entity paradigm. We now have a clear and consistent idea of our current way of seeing's semantics. We consciously appreciate what a substance and an attribute are; and also, what the corresponding entity and attribute signs in entity-oriented models refer to. This conscious appreciation of our current way of seeing is an essential precursor to consciously working our way forward to the object paradigm.

5.1 The development of finer, more accurate, distinctions

The development of the substance paradigm in 4th century BC Greece was part of a general improvement in semantics enabled by the development of writing technology and the invention of the alphabet. One aspect of this that is relevant to our re-engineering is an overall development of finer, more accurate, distinctions. (We shall see, in later chapters, how the re-engineering from entities to objects continues this development.)

5.1.1 Distinguishing between the literal and the metaphorical

One good example of the development of more accurate distinctions is recognising the difference between literal and metaphorical descriptions. Before Aristotle's time, people did not make this distinction. Aristotle's teacher Plato had not quite arrived at it. In the *Sophist* he condemns 'likenesses' (in other words, metaphors) as 'a most slippery tribe' even though he is himself using one.

Aristotle, however, made the distinction both forcefully and explicitly. He condemned metaphors outright, insisting that they should not be used in definitions and criticising them in his predecessors work. For example, Aristotle criticises Empedocles for describing salt water as the sweat of the Earth (and so, by implication, that sweat and sea-water are the same).

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

Aristotle comments:

> Perhaps to say that is to speak adequately for poetic purposes—for metaphor is poetic—but it is not adequate for understanding the nature [of a thing].

With hindsight we can see that it was not so much that Empedocles was mistaken, but that he had not developed a sufficiently accurate framework to distinguish between the literal and metaphoric.

5.1.2 Comparing oral and literate cultures

If we compare the way oral and literate cultures see signs and sameness, we can clearly see the development of the finer and more accurate distinctions that came with writing. The notion of a sign is quite broad. We say things like:

A rapid pulse is a sign of a fever.
This footprint is a sign someone passed here recently.
Pottery fragments are a sign of human civilisation.

We also talk of things as signs representing other things:

The elephant represents the (US) Republican party.
The (UK) Member of Parliament represents his constituents.

We use both of these ways of talking to explain the meaning of words and symbols:

The word ‘dog’ is a sign for a dog.
The symbol ‘$’ represents dollars.

At a general level, all these types of signs have something in common. But modern western culture, with the resources of writing, has developed a sophisticated understanding of their differences. It recognises that the link between the word ‘dog’ and a dog is not the same as the link between a rapid pulse and fever.

However, these distinctions took a long time to develop. Often oral cultures not only do not make these distinctions but also do not make a sharp distinction between the thing itself and something that represents it. A well-researched example is the Huichol Indians of Mexico, who sacrifice deer to their gods. When no deer is available, they offer corn in its place. They explain this by saying that the corn is the same as deer.

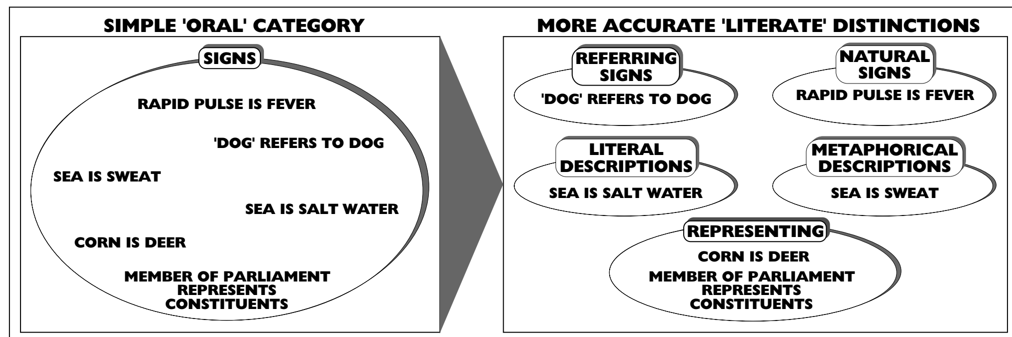
This sounds amazing to our literate ears. We can see it is in their interest to say that corn is deer; it means they have something to sacrifice to the gods. The connections go deeper than that. For example, their mythology claims that corn was once a deer. However this does not explain why, in discussions, they adamantly claim that corn and deer are the same. A claim that, to our literate minds, is unintelligible.

The Huichol are not unique: other oral cultures make similar claims. The Nuer of Sudan claim that twins are birds. The Zafimaniry claim that the centre post of the clan’s chief hut is an ancestor. The Puluwat Islander navigators claim that east is a big bird. We have similar claims in our history. Statements such as ‘this is my body’ in the Christian bible, puzzled the scholars developing a literate (and literal) understanding of text in the Middle Ages and still puzzles some people today.

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

We can understand what is going on once we realise that the problem is not with the oral cultures' ideas of corn and deer, but with their ideas of signs and sameness. Without the resources of writing, they have not yet developed our modern, more accurate, distinctions between different types of signs and sameness. ***Figure 4.19*** illustrates this development schematically. The Huichol Indians claim that corn is deer because, for them, saying that 'corn is deer' is the same type of thing as saying that 'corn represents deer'. 'Is the same as' and represents' belong to the same conceptual category.

Figure 4.19: Shift to finer, more accurate, distinctions



This is why the Huichol form of identity claim is common in oral cultures, but unintelligible to literate ones. Oral cultures do not need as sharp a distinction as literate ones; for them the potential ambiguity it is not a problem. We shall re-visit the Huichol Indians when we look at how the shift to the logical and object paradigms leads to similar developments of finer distinctions. There the boot will be on the other foot. Most of us will be in the position of the Huichol Indians. We will (initially, at least) find it difficult to see what these new distinctions are and why they need to be made.

6 The four key types of things

We have seen that the substance paradigm, despite its extreme age, was and is sophisticated. We have seen how it addresses all four of the key types of things we identified in ***Chapter 2***:

- How it uses the notion of primary substance and attributes to handle things' particularity.
- How it uses the notion of secondary substance and attributes to handle types. How it uses the secondary hierarchies to handle levels of generality.
- How it uses relational attributes to handle relationships—although, as we have seen, this is not really a satisfactory solution.
- How it uses shifts to new accidental attributes to handle changes—although these shifts are, in a sense, a new implicit type of particle.

For our purposes, it offers a comprehensive semantics for the four key types of things. This is why it makes such a good benchmark and starting point for our re-engineering to object semantics.

7 What’s next

In the last chapter, we saw why the substance paradigm was simplified into the entity paradigm and the semantic confusion this caused. We also recognised that this simplification taught us to down play, even ignore, the semantic aspects of business entity modelling. We discussed the root cause of this, the two-dimensionality of paper and ink technology. With the invention of computing technology, this constraint disappears and we have an opportunity to re-introduce semantics into business modelling.

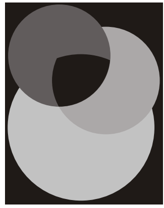
Some people may be tempted to do this within the substance paradigm’s semantics described in this chapter. It is, in many ways, an improvement on the entity paradigm. It has the benefits of both secondary substance and attribute hierarchies and, through the use of independent attribute hierarchies, the potential for re-use across secondary substances. But the substance paradigm is not just old; it is ancient—over two thousand years old. There have been several generations of developments in semantics since then. It only makes sense to take advantage of the improvements they offer.

Furthermore, the substance paradigm seems to have a couple of potential problems in reflecting the real world accurately. We have seen how its treatment of relationships is unsatisfactory. We have also seen that its secondary hierarchies cannot handle multiple inheritance and classification. If we want to take full advantage of computing technology’s flexibility, we need to rise above these constraints.

Nevertheless, the substance paradigm plays an important part in our re-engineering of the information paradigm. It acts as a benchmark against which we can measure the progress of the re-engineering. Each step forward should offer better solutions to the issues, and, in the end, more potential for re-use. In Part Three, we re-engineer into the next paradigm on our route to objects—the logical paradigm.

© Copyright
chris.partridge@

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

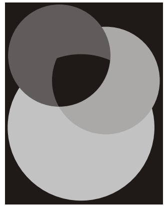


Part Three

Shifting Towards Objects—The Logical Paradigm

Chapter 5 The Emergence of the Logical Paradigm
Chapter 6 The Logical Paradigm's Framework

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 5

The Emergence of the Logical Paradigm

1 Introduction
2 Origins of the logical paradigm
3 Shifting from substance to extension
4 Re-engineering primary substance
5 Re-engineering secondary substance
6 Re-engineering secondary attributes
7 Re-engineering relational attributes
8 Simplifying and generalising the information framework
9 Summary

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction

In Part Two, we clarified what the entity and substance paradigms were. Here, in Part Three, we re-engineer the substance paradigm into the logical paradigm. This is an important step because it establishes a number of patterns that are inherited by the object paradigm. In this chapter we construct the logical paradigm's framework out of its fundamental particles. In the following chapter, we fill in this framework.

1.1 Re-engineering the substance paradigm

We start off this chapter, as we did with the entity paradigm, by looking at the origins of the logical paradigm. Then we build its framework by re-engineering the substance paradigm. We take the substance paradigm's four fundamental particles:

- Primary substance,
- Primary attributes,
- Secondary substance, and
- Secondary attributes.

And re-engineer them into the logical paradigm's three fundamental particles:

- Logical classes,
- Logical tuples, and
- Individual logical objects.

2 Origins of the logical paradigm

Mathematics provided the inspiration for the logical paradigm. We shall see the mathematical nature of the paradigm when we look at its fundamental particles. The paradigm's two core patterns of class and tuple (its new fundamental particles) come from set theory. This is a branch of mathematics that has its origins in work done in the 19th century. Foundational work was done by a number of mathematicians; including George Boole (1815–1864), John Venn (1834–1923) and Georg Cantor (who developed modern set theory between 1874 and 1897). The paradigm's framework was developed by the German, Gottlob Frege (1848–1925), and the American, Charles Sanders Peirce (1839–1914).

2.1 Meaning and understanding

To support his mathematical work, Frege developed a new notion of meaning that is central to both the logical and object paradigms. In the ***Prologue***, we noted that the objective of business modelling is capturing understanding. It may help us to appreciate the importance of Frege's work if we re-phrase this as 'capturing meaning'.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

2.1.1 Frege's analysis of meaning as sense and reference

Frege suggests that the meaning of a concept (or sign) has two components:

- Sense, and
- Reference.

Reference is the relationship between the concept and the thing it refers to (as illustrated earlier in ***Figure 2.1***). Sense is the relationships a thing has to other things (reflected in a concept's relationships to other concepts). Together these two components are a concept's meaning.

Frege developed his analysis to resolve a problem found with earlier theories that based their analysis solely on reference. He used the following example to illustrate how his analysis worked and the problem it solved.

The planet Venus is visible in the morning sky, where it is called the morning star. It is also visible in the evening sky, where it is called the evening star. If the simplistic notion of the concept's meaning only being what it refers to is correct then, as the three concepts:

- The morning star,
- The evening star, and
- Venus

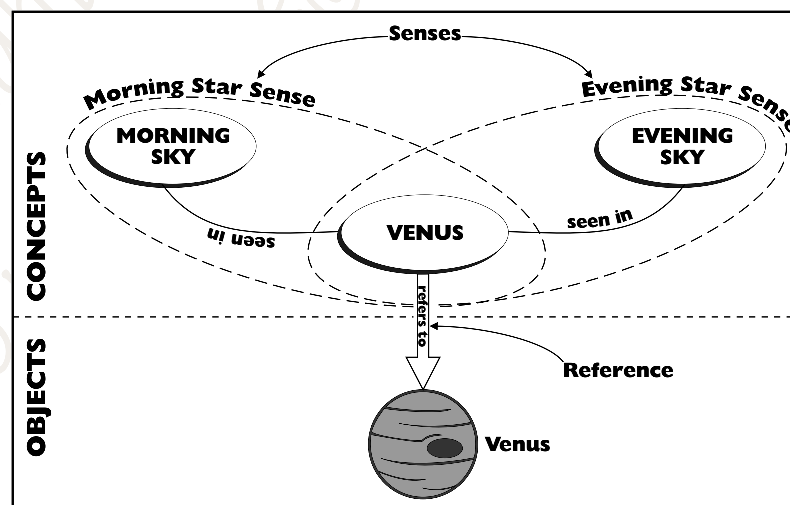
All refer to the same object; they should be interchangeable. However, when we interchange them in the sentence;

Venus is Venus.

We get the sentence:

The morning star is the evening star.

Figure 5.1: Venus's meaning map



© Copyright Chris Partridge chris.partridge@BOROCentre.com

The first sentence is obviously and tautologically true. The second sentence tells some people a new fact (because they do not know that both the morning star and the

evening star are Venus). In our everyday sense of meaning, the two sentences certainly have different meanings. This, Frege explains, is because of the different sense elements in their overall meanings. We can see how different these are in the map of Venus's meaning in ***Figure 5.1***.

Scholars recognise Frege's explanation as a big step forward. The respected philosopher and Fregean scholar, Michael Dummett, calls it (with characteristic academic caution) 'the beginnings of a plausible account . . . of understanding':

> Frege arrived, for the first time in the history of philosophical enquiry, at what was at least the beginnings of a plausible account of sense, and thus of understanding.

2.1.2 Mapping sense explicitly and accurately

Frege's analysis enables us to neatly explain what a business model is. It is a map of the (Fregean) sense of the business. It is an explicit map of the objects in the business and their patterns of relationship. This explains what reflecting the business directly—a key feature of O-O—involves.

A direct reflection faithfully and accurately maps sense—the business objects and their patterns of connections. An accurate map enables people to see the meaning of the business clearly and directly, and so helps them to develop a better understanding of it.

An indirect reflection distorts the patterns so that they can fit into the framework of the model, making it difficult to see the underlying objects clearly. The worked examples in Part Six provide examples of this. The examples show how patterns are distorted to fit into the entity framework of existing computer systems.

2.2 The need to determine reference

Most business modellers intuitively recognise the need to capture sense. They instinctively understand how sense supports reference, making a cohesive whole. They can see that sense helps to fix reference and so they naturally model it. They see, for example, that saying 'deal #123' does not fix reference anything like as well as saying 'the deal you did yesterday with NatLand Bank for £10m'—which has more Fregean sense.

However, very few people, even business modellers, intuitively recognise the need to determine the reference of the concepts they are using. If indeed the concepts they are using actually refer to anything (we raised a similar point in ***Chapter 2***—using the data–process distinction as an example). Business modellers, like most people nowadays, tend to work with concepts and ignore the objects in the world that they are meant to refer to. They treat the concepts as if they were the objects themselves. This tends to mean that the needs of the computer system (in the shape of concepts) drive the system building process, rather than the business itself (in the shape of objects).

Frege recognised that this was a serious problem. He took as a central principle (in *The Foundations of Arithmetic*):

> . . . never to lose sight of the distinction between concept and object.

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

If we adopt this principle then we will never assume that we are modelling business objects when we are focusing on concepts.

2.2.1 The reference principle

Frege's analysis of meaning assumes a basic principle of the logical and object paradigms—the strong reference principle. It is not difficult to grasp; indeed it seems simple and obvious. It is that a sign (concept) in the business model explicitly refers to one and only one object in the business. And conversely a business object is explicitly referred to by one and only one sign in the model. A simple example is given in ***Figure 5.2***. This principle, if strictly adhered to, ensures that a business model is a direct, and so explicit, reflection of the world.

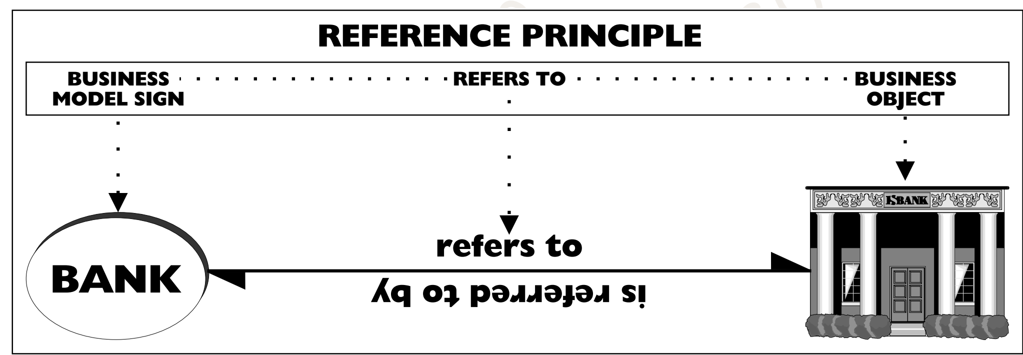


Figure 5.2: Example of the strong reference principle

One reason this principle is so central is that it implies another central principle—that everything is an object. If every sign in the model refers to an object; then, as far the model is concerned, everything is an object. (This includes the 'sense' connections between objects.) To keep things tidy and avoid confusion, it also suggests that there is no more than one sign for each object. The result is a simple, clean semantics.

For individual physical things, this principle appears obvious; for other types of things, it is more difficult to uphold. In fact, as we shall see, it has taken a substantial re-engineering of the way we see things to make it work. The principle has been a major driving force behind many of the innovations in both the logical and object paradigm.

3 Shifting from substance to extension

A shift from substance to extension is at the core of the re-engineering to the logical paradigm. As we shall see, both the logical and object paradigms put extension in the central place that substance occupies in the substance paradigm. Here we take a historical look at how the substance pattern was undermined and the benefits of extension recognised.

3.1 Appreciating the difference between appearance and reality

After the development of the microscope in the 17th century, people started seriously questioning the substance paradigm. Before its development, most people (and the substance paradigm) assumed that the world was composed of natural everyday people-sized things, such as tables and chairs. And that these things had attributes that people saw and touched directly.

The microscope that changed all this. People looking through it could see that everyday people-sized things were really composed of particles smaller than the unaided eye could see. It soon became clear that attributes, such as taste and smell, were ideas created by the mind from the sensations of these small particles rather than inherent in either the people-sized things or the particles.

3.2 The problem with substance

This created a climate of doubt that naturally led to questions about substance. The ghostly hulk shown in ***Figure 4.1***, which is left behind when we take away all of a thing's attributes, poses a problem. Not only has no one ever seen or touched such a thing; in principle, no one can ever examine it directly. We can only see or touch its properties. When these are gone, nothing is left for us to perceive. So we can never directly prove its existence.

In the 17th and 18th centuries, thinkers began to voice their concerns. We now look at those raised by the Englishman John Locke (1632–1704) and the Scotsman David Hume (1711–1776). We first see how John Locke expressed qualms about substance. Then, we see how David Hume (born seven years after Locke died) went much further, calling substance an 'unintelligible chimera'.

3.2.1 John Locke's qualms

John Locke most famously expressed his dissatisfaction with unknown substance in this passage from his book *An Essay Concerning Human Understanding*:

> §1. Ideas of substances how made.
>
> THE Mind being, as I have declared, furnished with a great number of the simple *Ideas,* conveyed in by the *Senses,* as they are found in exteriour things, or by *Reflection* on its own Operations, takes notice also, that a certain number of these simple *Ideas* go constantly together; which being presumed to belong to one thing, and Words being suited to common apprehensions, and made use of for quick dispatch, are called so united in one subject, by one name; which by inadvertency we are apt afterward to talk of and consider as one simple *idea,* which indeed is a complication of many *Ideas* together; Because, as I have said, not imagining how these simple *Ideas* can subsist by themselves, we accustom our selves, to suppose *some Substratum,* wherein they do subsist, and from which they do result, which therefore we call *Substance.*
>
> §2. Our Idea of Substance in general.

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

So that if any one will examine himself concerning his *Notion of pure Substance in general,* he will find he has no other *Idea* of it at all, but only a Supposition of he knows not what support of such Qualities, which are capable of producing simple *Ideas* in us; which Qualities are commonly called Accidents. If any one should be asked, what is the subject wherein Colour or Weight inheres, he would have nothing to say, but the solid extended parts: And if he were demanded, what is it, that that Solidity and Extension inhere in, he would not be in much better case, than the *Indian* before mentioned; who, saying that the World was supported by a great Elephant, was asked, what the Elephant rested on; to which his answer was, a great Tortoise: But being again pressed to know what gave support to the broad-back'd Tortoise, replied, something, he knew not what. And thus here, as in all other cases, where we use Words without having clear and distinct Ideas, we talk like Children; who, being questioned, what such a thing is, which they know not, readily give this satisfactory answer, That it is something; which in truth signifies no more, when so used, either by Children or Men, but that they know not what; and that the thing they pretend to know, and talk of, is what they have no distinct Idea of at all, and so are perfectly ignorant of it, and in the dark. The *idea* then we have, to which we give the general name substance, being nothing, but the supposed, but unknown support of those Qualities, we find existing, which we imagine cannot subsist, *sine re substante*, without something to support them, we call that Support *Substantia*; which, according to the true import of the Word, is in plain *English, standing under,* or *upholding.*

3.2.2 David Hume's scepticism about substance

David Hume went much further. He wrote about 'the fictions of ancient philosophy concerning *substances*'. His attack on substance was based, like Locke's qualms, on our inability to ever know anything about it.

Hume quite rightly recognised that substance played two roles. It was the home for attributes and the means for things to preserve their identity over time (we examined both when we looked at the substance paradigm in ***Chapter 4***). Here is how Hume describes the first role in his *A Treatise of Human Nature*:

Hence the colour, taste, figure, solidity, and other qualities combin'd in a peach . . . are conceiv'd to form *one thing*. . . . But the mind rests not here. Whenever it views the object in another light, it finds that all these qualities are different, and distinguishable, and separable from each other; which . . . obliges the imagination to feign an unknown something, or *original* substance and matter, as a principle of union and cohesion among these qualities, and as what may give the compound object a title to be call'd one thing, notwithstanding its diversity and composition.

As Hume pointed out, without substance to support it, our notion of attributes needs to change:

The notion of accidents is an unavoidable consequence of this method of thinking with regard to substances . . . nor can we forbear looking upon colours, sounds, tastes, figures, and other properties of bodies, as existences, which cannot subsist apart, but require a subject of inhesion to sustain and support them. For having never discover'd any of these sensible qualities, where, . . . we did not likewise fancy a substance to exist; . . . [we] infer a dependence of every quality on the unknown substance. The custom of imagining a

© Copyright Chris Partridge chris.partridge@BORO Centre.com

> dependence has the same effect as the custom of observing it wou'd have. This conceit, however, is no more reasonable than any of the foregoing. Every quality being a distinct thing from another, may be conceiv'd to exist apart, not only from every other quality, but from the unintelligible chimera of a substance.

Hume is right about the close link between substance and attributes. As we shall see, when the logical paradigm shifts from substance to extension, it has also to shift from attributes to logical classes and tuples.

Hume also describes substance's second role, preserving a thing's identity over time:

> When we gradually follow an object in its successive changes, the smooth progress of the thought makes us ascribe an identity to the succession; . . . When we compare its situation after a considerable change the progress of the thought is broke; and consequently we are presented with the idea of diversity: In order to reconcile which contradiction the imagination is apt to feign something unknown and invisible, which it supposes to continue the same under all these variations; and this unintelligible something it calls a *substance*.

Hume explained how we preserved identity in our minds. But, he could not explain what was going on with the physical objects outside our minds. However, he was clear that the conclusions our minds jumped to were not justified by what we sensed:

> Objects have a certain coherence even as they appear to our sense; but this coherence is much greater and more uniform, if we suppose the objects to have a continu'd existence; and as the mind is once in the train of observing an uniformity among objects, it naturally continues, till it renders the uniformity as compleat as possible. The simple supposition of their continu'd existence suffices for this purpose, and gives us a notion of a much greater regularity among objects, than what they have when we look no farther than our senses.

This problem of explaining how a thing's identity is preserved over time is particularly intractable. At the end of the following chapter, we look at the difficulties that the logical paradigm has dealing with it. It is only when we come to the object paradigm that we arrive at a satisfactory explanation.

3.3 The benefits of extension

It was doubts about what we know that led the French thinker, René Descartes (1596–1650), to extol the benefits of extension. He was looking for ideas that could not be doubted—and developed what has been called the 'method of doubt'. When he applied this method to physical bodies, he found that the one property that he could not doubt was extension (in other words, its length, breadth and height). There was no question in his mind that that extension was real.

Descartes still clung to the substance pattern, so he suggested that extension was the primary essential attribute of substance. Extension is in many ways an updated version of the Aristotelian category place—the fourth attribute category in ***Figure 4.19***. This fourth category is still deeply embedded in our way of speaking, such as when we say 'someone is sitting in my place'.

© Copyright Chris Partridge chris.partridge@BOROCentre.com

Descartes also suggested that shape, size and motion were the only 'modes' (his name for Aristotelian accidental attributes), and that all other attributes were mental phenomena:

> I observed that nothing at all belonged to the nature of essence of body except that it was a thing with length and breadth and depth, admitting of various shapes and various motions. I found also that its shapes and motions were only modes, which no power could make to exist apart from it; and on the other hand that colours, odours, savours and the rest of such things, were merely sensations existing in my thought, and differing no less from bodies than pain differs from the shape and motion of the instrument which inflicts it.

Descartes was suggesting that extension is the one property of bodies that cannot be doubted. It is a very attractive suggestion because extension is such a basic aspect of our perception of the world. We see and touch extension when we see and touch individual objects, so (he argues) we cannot doubt it exists outside our mind. This gives extension a striking advantage over attributes such as taste and smell, which appear to exist in the mind. Descartes suggested a thought experiment to confirm the fundamental nature of extension:

> We have only to attend to our idea of some body, e.g. a stone, and remove from it whatever we know is not entailed by the very nature of body. We first reject hardness; for if the stone is melted, or divided into a very fine powder, it will lose this quality without ceasing to be a body. Again, we reject colour; we have often seen stones so transparent as to be colourless. We reject heaviness; fire is extremely light, but none the less conceived as a body. Finally, we reject coldness and heat and all other such qualities; either they are not what we are considering in thinking of the stone, or at least their changing does not mean that the stone is regarded as having lost the nature of a body. We may now observe that absolutely no element of our idea remains, except extension in length, breadth, and depth.

In the next section, we see how the logical paradigm re-uses Descartes' extension pattern.

4 Re-engineering primary substance

We start our re-engineering of the substance paradigm with its primary substance for particular bodies.

4.1 Bodies as extension

In the substance paradigm, the foundation of a particular body is its underlying primary substance (to which its primary attributes belonged). The logical paradigm uses a different foundation upon which to build its pattern for bodies. It follows through on Descartes' lead and uses extension (length, breadth and height) as its foundation. (It uses extension as a replacement for substance, not—as Descartes did—an essential attribute.) So when I shift from the substance to the logical paradigm, I shift from seeing my car as a primary substance to seeing it as an extension (shown in ***Figure 5.3***).

© Copyright Chris Partridge
chris.partridge@BORO.com

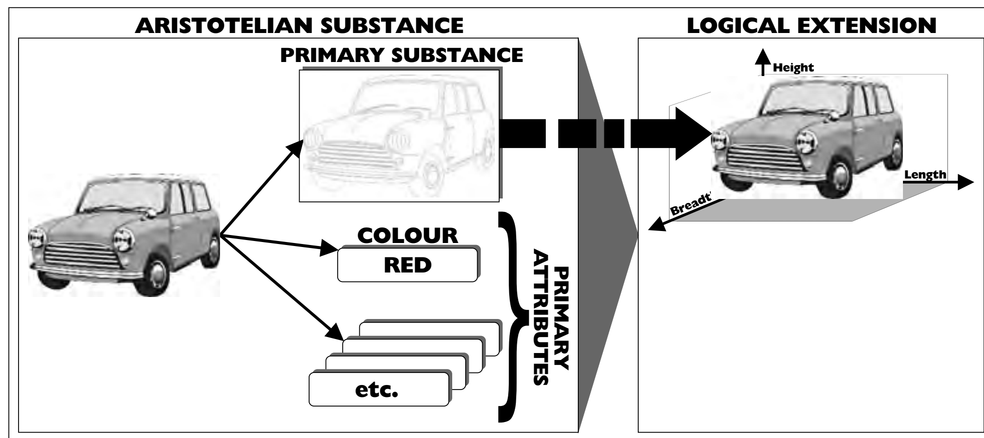


Figure 5.3: My car

The two patterns of particular extension and reference complement one another perfectly here. Without extension we would not have anything tangible to refer to. It gives us a strong sense for reference. Because we can point to what we are referring to, we can specify it accurately. Individual physical objects, such as my car, provide a good role model for strong reference. The task we face is making all the other types of objects so well behaved.

4.2 What happened to primary substance's attributes?

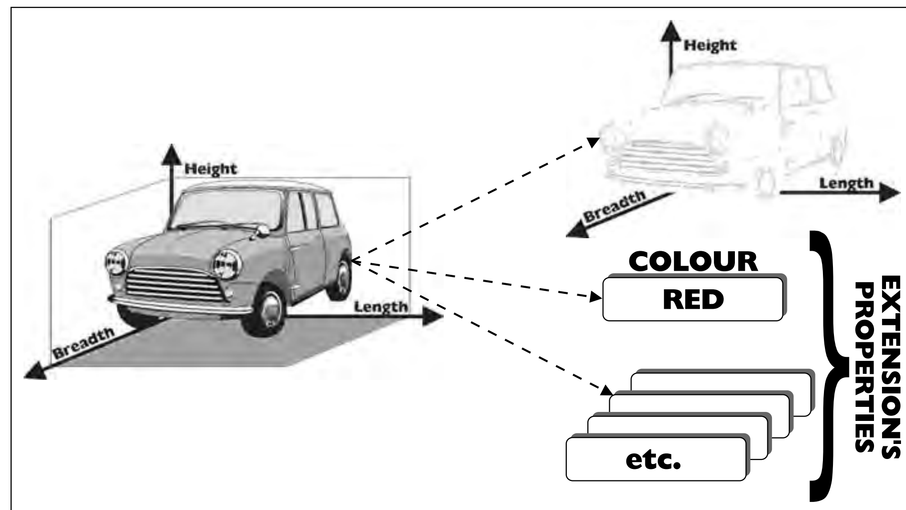
As ***Figure 5.3*** suggests, we have only covered half the distance. In the Aristotelian paradigm, particular bodies, such as my car, are composed of primary substance *and* primary attributes. While extension can replace substance, we also have to find something to replace attributes. It turns out that classes fit the bill, but first we look at why primary attributes need re-engineering.

4.2.1 The problem with primary attributes

The problem with primary attributes is that it is difficult, under a scheme of strong reference, to see what we might be referring to when we talk of them. In the substance paradigm, they are mysterious things that attach themselves to even more mysterious substance.

Once the logical paradigm had re-engineered substance into extension, it might have seemed sensible to stop there and let extension take over substance's role of having attributes belong to it. My car would then be composed of its extension and various properties that belong to the extension. As ***Figure 5.4*** illustrates, this scheme of things has a similar pattern to the substance paradigm.

Figure 5.4:
My car's extension with attributes



However, this would have caused problems. Under the logical paradigm's strong reference principle, all things have to have an extension—one that we can point to. So, if extensions had primary attributes, these attributes (as things) would have to have extension. This would have been the source of all sorts of problems.

For a start, we would not have been able to work out where we could find a primary attribute (as extension). Consider, for example, my car's primary attribute of redness. It could not exist everywhere the car does—in other words, have the same extension as the car. If it did, it would be the same object as the car. It could not be a part of the car, such as the front half of the car, because other parts, such as the back half, are red. It could not exist somewhere else apart from the car. That would not make sense. There is just nowhere for the extension to exist.

We can push this further. Assume that we had somehow found a satisfactory extension for my car's primary attribute of redness. Still, there is a problem. As the red attribute has extension, it is an object and can have properties. Assume we examine the extension, taste it and weigh it. We find it tastes metallic and weighs ten pounds. The notion of a primary red attribute having a taste is odd. What is even odder is that we can, in principle, look and see what colour attribute it has. Let's play safe and assume it is red. This means my car's primary attribute of redness has a primary attribute of redness. This second level red attribute is still an attribute and so is an extension. Let's assume we can found the extension (remember it cannot be the same extension as the first level attribute or else they would be the same thing). We can ask what colour the new extension is and so on ad infinitum.

Making attributes extensions something they were not intended to be leads to odd and contradictory results. It should be becoming clear that this is a pointless exercise because primary attributes were never designed to be extensions. Extensions, the basis for strong reference, and primary attributes are parts of two completely frameworks. Primary attributes are meant to be attached to substance not extensions. Attributes need to be re-engineered into something compatible with extension.

4.2.2 The solution—transform primary attributes into classes

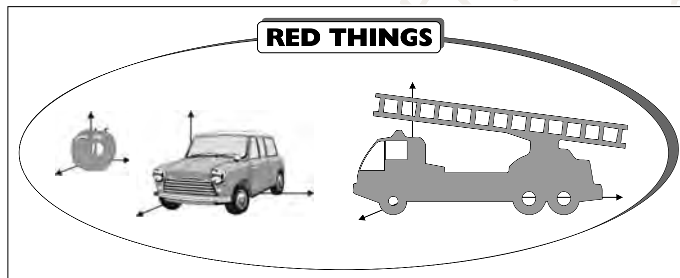
In the late 19th century, Georg Cantor provided the tool needed for this re-engineering. He invented the notion of a class and defined it as:

> . . . the result of collecting together certain well-determined objects of our perception or thinking into a single whole; these objects are called the elements of the set.

The key phrase is 'a single whole'. This transforms a collection of objects into a single object.

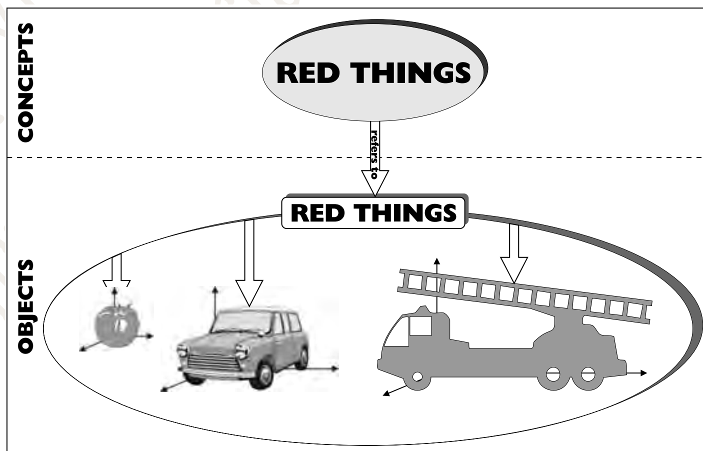
Gottlob Frege was one of the few mathematicians to recognise the importance of Cantor's work when it first appeared. One of the uses to which he put it was re-engineering the attribute pattern into a class pattern. We can see how this works in our car example. My car's red attribute is transformed into my car being a member of the class of red things (shown in ***Figure 5.5***). From the logical paradigm's viewpoint, when I say 'my car is red', I mean that 'my car belongs to the class of red things' and not 'my car has the mysterious primary attribute of redness'.

Figure 5.5: My car belongs to the class of red things



After Frege's transformation, there was no need for the extensionless attribute pattern. Classes, the transformed attributes, have an extension soundly rooted in the extensions of the objects belonging to the class. They can be referred to in a strong sense. As my car is an extension, so the class of red things is a collection of extensions (shown in ***Figure 5.6***). This gave the logical paradigm a simpler and stronger semantic framework.

Figure 5.6: Referring to the class of red things



4.2.3 Referring to classes

When dealing with the extension of classes, the strong reference principle forces us to see them more accurately than usual. In everyday life, we do not always clearly distinguish between a collection and a fusion of extensions. When using classes to business model, however, we need to. This simple example shows what happens if we do not make the distinction. (It is taken from Willard Van Orman Quine's book *From a Logical Point of View*.)

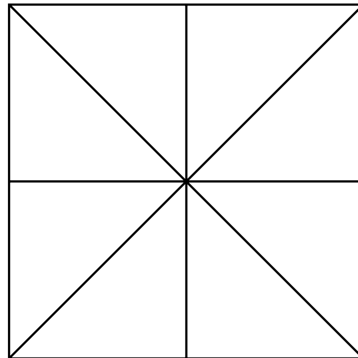


Figure 5.7: Our world

Assume that ***Figure 5.7*** is our world and that its 33 regions are our individual objects. We now classify the regions into these shapes:

- An isosceles right triangle,
- A square,
- A two-to-one rectangle, and
- Two types of trapezoid.

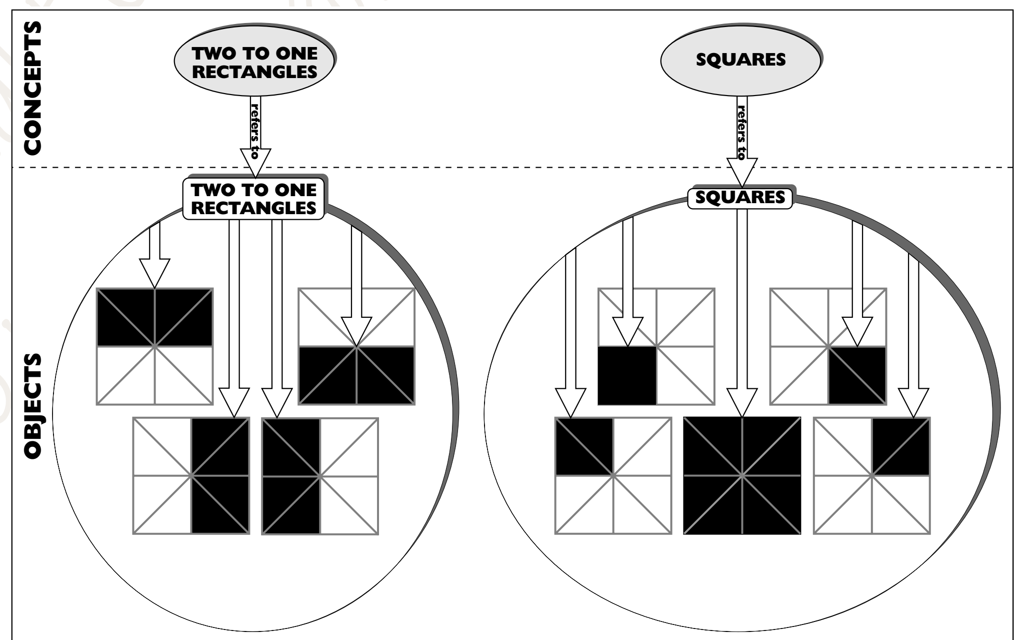


Figure 5.8: Shapes' collected extensions

© Copyright - Chris Partridge
chris@BOROCentre.com

From a logical paradigm viewpoint, these shapes are classes of individual objects. Now, we assume (incorrectly) that the extension of a class is the fusion of the extensions of its members. Then the extension of the class squares is the complete figure, as, between them, the five individual square shapes' extensions cover the complete figure. The extension of the four two-to-one rectangles is also the complete figure. In fact, all the five shape classes have the complete figure as their extension. Because objects with the same extension are identical, we are faced with the inescapable conclusion that all the shape classes are the same object.

This is clearly wrong. The error is assuming that the extension of a class is the fusion of its members' extensions. It is not their fusion but their collection (shown in ***Figure 5.8***). This shows that what distinguishes the two-to-one rectangle and square shape classes is that they have different collections of extensions.

Once we become familiar with this more accurate way of seeing classes, we begin to see examples of it everywhere. Quine offers us another, less theoretical, example. He asks us to consider the states of the USA and the counties of the USA. If we see either the counties or the states classes as a fusion of their individual extensions, the result is a single extension—the USA. However, if we see them as collections of their member's extensions, we get the correct result—two separate collections of extensions.

5 Re-engineering secondary substance

The logical paradigm's transformation does not end there. We saw earlier how the class pattern was used to re-engineer the primary attribute particle. Now, we see how it is used to transform the secondary substance particle. In the substance paradigm, general words referred to mysterious secondary substances. For example, the general word 'pig' referred to a mysterious secondary pig substance. Because no one has ever seen or touched any secondary substance, this reference was weak. The logical paradigm offered an alternative explanation of what general words referred to, one that used the class pattern and so had strong reference.

From a logical paradigm viewpoint, general words, such as 'pig', refer to the class of pigs. This is the collection of individual pigs. So when I say 'Porky is a pig', I mean that 'Porky belongs to the class of pigs' (shown in ***Figure 5.9***) and not 'Porky's primary substance belongs to a (mysterious) secondary pig substance'.

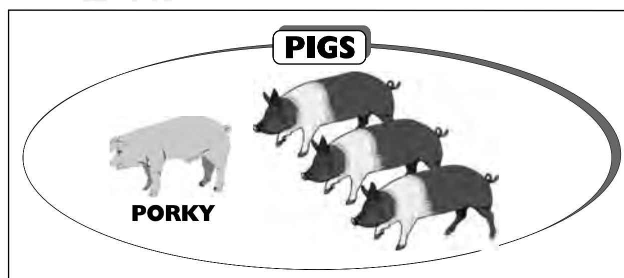
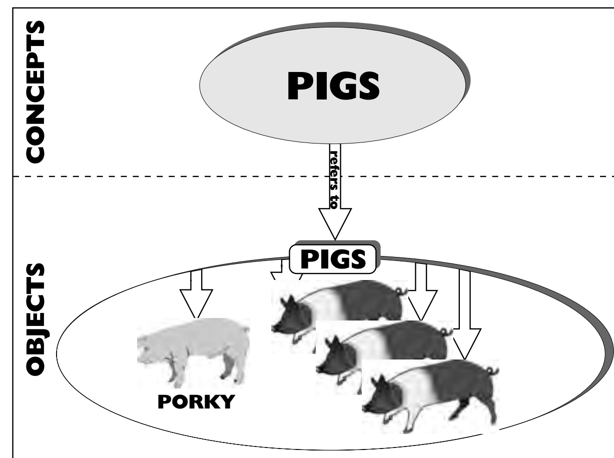


Figure 5.9: Porky belongs to the class of pigs

©Copyright Chris Partridge chris.partridge@BORO Centre.com

Like the class of red things (shown in ***Figure 5.6***), the class of pigs has an extension. This is the collection of individual pig extensions illustrated in ***Figure 5.10***.

Figure 5.10: Referring to the class of pigs

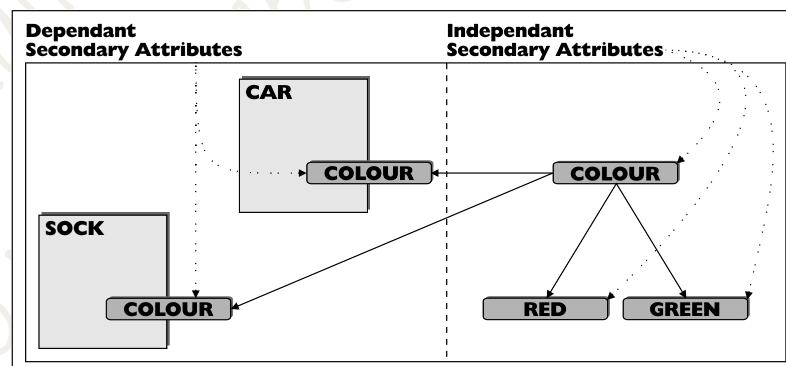


6 Re-engineering secondary attributes

Re-engineering the secondary substance particle still does not exhaust the power of the class pattern. It was used to transform the substance paradigm's last fundamental particle—the secondary attribute particle.

These secondary attributes could either belong to a secondary substance or live in an independent hierarchy. For example, the secondary substances car and sock both have a dependent secondary colour attribute, which are both related to an independent secondary colour attribute (shown in ***Figure 5.11***).

Figure 5.11: Two types of secondary colour attributes



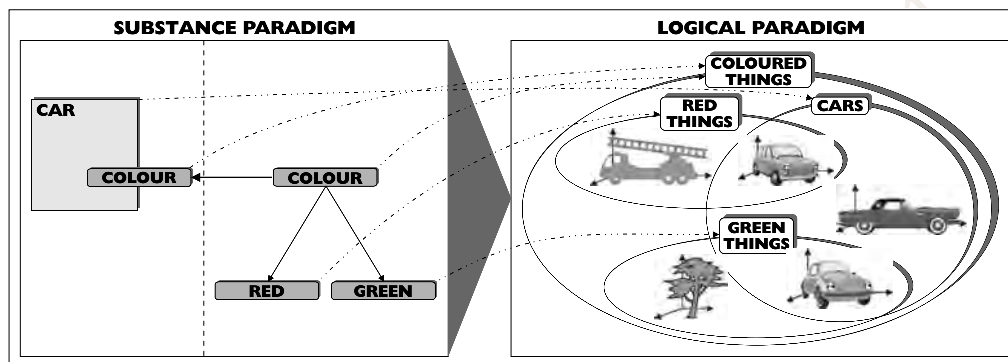
In the substance paradigm, the two types of secondary attribute served two distinct purposes. The independent attributes were used to describe the hierarchy of attributes and the dependent attributes to show their connections to substances. The logical paradigm uses the class pattern to re-engineer both types of secondary attribute, 'killing two birds with one stone'.

© Copyright Chris Partridge chris@BORO Centre.com

In some cases it re-uses the class re-engineered from a primary attribute. For example, the class pattern for red things in ***Figure 5.5*** re-engineered from my car's primary attribute of redness is re-used to capture the independent secondary red attribute pattern in ***Figure 5.11***, as shown in ***Figure 5.12***.

Where there are corresponding dependent and independent secondary attributes, these are re-engineered into a single class. For example, the dependent and independent secondary colour attributes in ***Figure 5.11*** are re-engineered into the single class, coloured things (shown in ***Figure 5.12***). This is not the same as the colours class, with red and green as members, which we re-engineer in the next chapter.

Figure 5.12: Re-engineering the two secondary colour attributes



We are beginning to see incontrovertible evidence that the logical paradigm's class pattern works at a more general level than the substance paradigm's patterns. It has single-handedly replaced both substance and attribute patterns, primary and secondary.

7 Re-engineering relational attributes

In ***Chapter 3***, we looked at the substance paradigm's semantics for relational attributes and saw that its treatment was awkward. We examined some of the problems that attributes had in capturing the relationship pattern. This awkwardness comes out of the woodwork in our re-engineering of relational attributes. The pattern we used earlier to transform primary attributes into classes cannot be applied to primary relational attributes. They are a different kind of thing and need a different approach. We now look at this, but first we remind ourselves of the substance paradigm's relational attribute pattern and its problems.

7.1 Relational and correlational attributes

We saw in ***Chapter 3*** how Aristotle fitted primary relational attributes into his substance–attribute structure using a relational attribute pattern. He took a sentence such as:

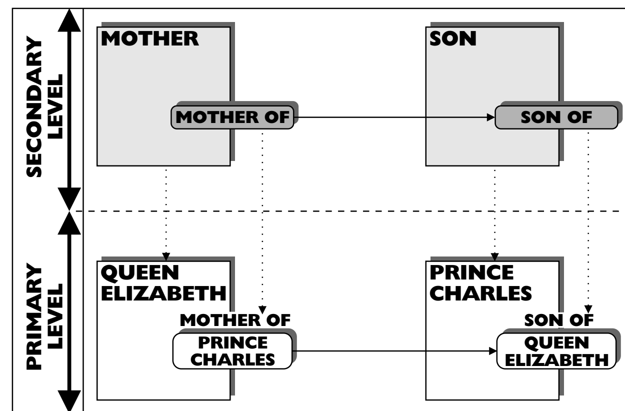
> Queen Elizabeth is the mother of Prince Charles.

and analysed it as follows. The subject of the sentence, 'Queen Elizabeth' is a substance. The predicate 'is the mother of Prince Charles' is a relational attribute belonging

to the Queen Elizabeth substance. He was well aware that there was another sentence that described the same relationship:

Prince Charles is the son of Queen Elizabeth.

Figure 5.13: Relational and correlational attributes



He analysed this as the corresponding correlational 'is the son of Queen Elizabeth' attribute of the Prince Charles substance. The relational and correlational attributes are illustrated in ***Figure 5.13*** (reproduced from ***Figure 3.21***).

7.1.1 The problem with relational attributes

We saw in ***Chapter 3*** two semantic problems with relational (and correlational) attributes. First, the connection between the relational and correlational attribute, shown by a line in ***Figure 5.13***, is not explicitly captured in the substance paradigm. There relational attributes—like all attributes—belong to only one substance.

Secondly, the substance paradigm uses two attributes to capture a single relationship. It is perhaps easier to think of this as two signs (two relational attribute signs belonging to different substance signs) referring to one relationship in the world. This is clearly a breach of the strong reference principle (shown in ***Figure 5.2***) that expects each object to be referred to by one and only one sign. Dropping one of these attributes, as the entity paradigm does, resolves this problem but leads (as we saw in ***Chapter 3***) to its own semantic problems.

7.2 The solution—connection objects

The logical paradigm resolved these problems by introducing a new type of object to help capture the relationship pattern, the tuple.

7.2.1 Solving the reference problem

This new object, like the class object, comes from mathematics. There, a connection between two things is treated as having two components, a tuple and a tuples class (where tuple is the general name for couple, triple and so on). It is easier to see how

©Copyright Chris Partridge chris.partridge@BORO Centre.com

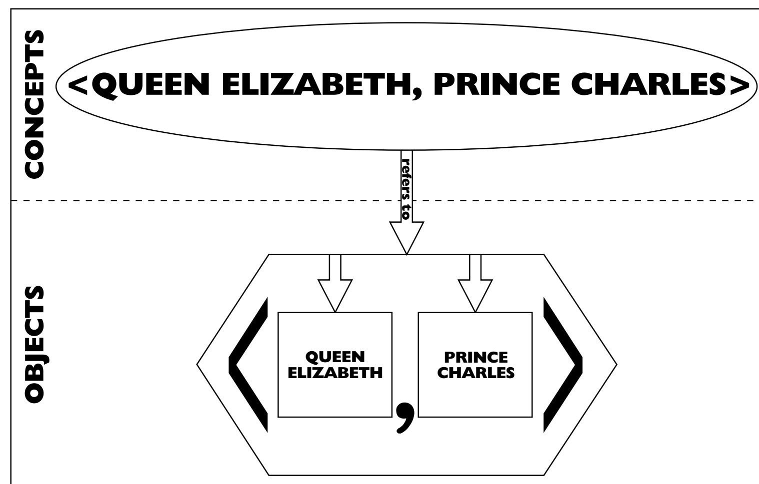
these components work in an example. So let us re-consider the connection described by the sentence:

Queen Elizabeth is the mother of Prince Charles.

A mathematical version of this has two components:

- A couple (a tuple of two objects), and
- A tuples class.

Figure 5.14: Referring to the <Queen Elizabeth, Prince Charles> couple

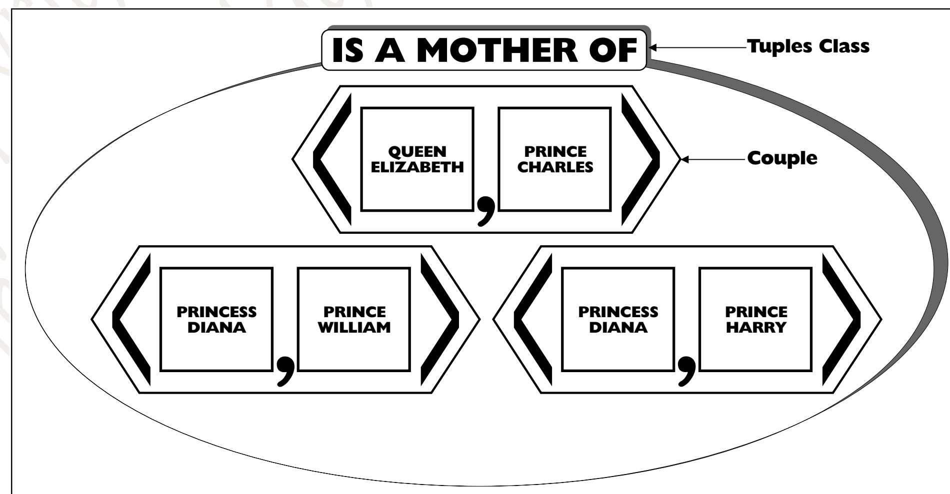


The couple (also known as an ordered pair) is constructed from the Queen Elizabeth and Prince Charles objects. It is normally written as:

<Queen Elizabeth, Prince Charles>.

Because this is constructed out of individual objects with extensions, the tuple, like a class, is rooted in their extensions (shown in ***Figure 5.14***). This means we can refer to it directly.

Figure 5.15: Collected together 'is a mother of' tuples class



© Copyright Chris chris.partridge@BORO-ce.com

We now have a new type of fundamental particle, a tuple. Because it is an object, we can apply the class pattern to it. As Cantor explained, any object can be collected into a class. So tuples can be collected into a class. For example, we can construct the ‘is a mother of’ tuples class by collecting together the couples that would belong to it—including <Queen Elizabeth, Prince Charles>. ***Figure 5.15*** shows the result.

So we can now translate the sentences ‘Queen Elizabeth is the mother of Prince Charles’ and ‘Prince Charles is the son of Queen Elizabeth’ into logical-speak. They both become:

> <Queen Elizabeth, Prince Charles> is a member of the ‘is a mother of’ tuples class.

7.2.2 Connections between more than two objects

The substance paradigm’s relational attributes work for connections between two substances. The logical paradigm’s tuples are more powerful. They can handle connections between any number of objects. Consider this sentence, which describes a three–way connection:

> Prince William is the son of Prince Charles and the brother of Prince Harry.

This translates into the triple:

> <Prince William, Prince Charles, Prince Harry>,

which is a member of the tuples class,

> X is the son of Y and the brother of Z.

The following triples are also members of the same tuples class.

> <Prince Harry, Prince Charles, Prince William>, and
>
> <Prince Charles, Prince Philip, Prince Andrew>.

7.2.3 Tuple and relational attribute identity

This ability to handle multiple connections shows that the tuple pattern is fundamentally different from the relational attribute pattern. We can illustrate quite neatly just how different by comparing how identity works for relational attributes and tuples. In other words, comparing when two relational attributes or tuples are the same or different. Consider the connections described by the following sentences:

> Prince Charles is the father of Prince William.
>
> Prince Charles is taller than Prince William.

From a substance paradigm viewpoint, the Prince Charles substance has two relational attributes (we ignore the correlational attributes in this example). From a logical paradigm viewpoint, there are not two attributes but a single couple belonging to two classes (shown in ***Figure 5.16***).

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

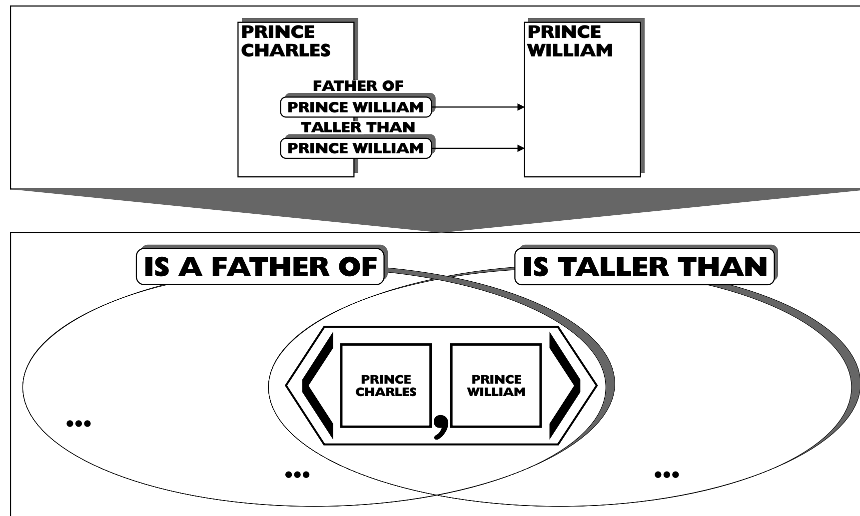


Figure 5.16: Tuple identity

Where the substance paradigm uses father of and taller than relational attributes (and their correlational attributes) the logical paradigm uses the single <Prince Charles, Prince William> couple. The substance paradigm implies that there are two connections out there in the world. The logical paradigm assumes that only one connection belongs to two classes. This is quite a different way of seeing things. I have found that people adept at seeing in the old relational attribute way take quite a while to get proficient in this new way.

You may have noticed that ***Figure 5.16*** also shows an important implication of tuples being objects and that they can be collected into more than one class.

7.2.4 Modelling many-to-many connections with tuples

In ***Chapter 3***, we looked at the problem the substance paradigm had capturing a 'many-to-many' connection pattern. We also looked at two solutions within the paradigm—creating 'pseudo' entities and creating a new relationship particle. These were illustrated in ***Figures 3.24*** and ***3.25***. We saw that both of these 'solutions' have semantic problems; their 'pseudo' entities and relationship particles have difficulty referring to the world directly.

The logical paradigm's semantics for many-to-many connections has no such problems. We can see this if we re-consider the employee/project example in ***Chapter 3***. ***Figure 3.23*** shows a number of employees working on a number of projects, including:

Employee Sue working on project #1.

Employee Sue working on project #2.

Employee John working on project #2.

From a logical paradigm viewpoint, we see this as these tuples,

<Sue, project #1>,

<Sue, project #2>, and

© Copyright Chris Partridge chris.partridge@BORO-centre.com

<John, project #2>

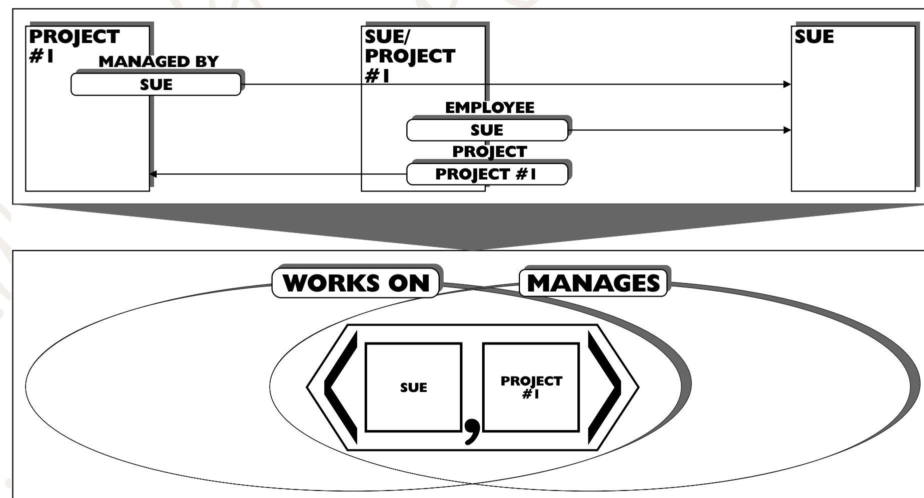
belonging to an ‘employee works on project’ tuples class. There is no semantic problem as the tuples (and tuples class) are objects with extensions.

7.2.5 Compacting with tuples

We can use this employee/project example to illustrate how tuples can compact information. In the example, assume that Sue manages project #1. From a substance paradigm viewpoint, we see this as a new ‘managed by Sue’ relational attribute of the project #1 substance (ignoring the correlational attribute). From a logical paradigm viewpoint, there is no new object. Instead, we see the <Sue, project #1> couple, which we used in the ‘employee working on project’ tuples class, belonging to a managed by tuples class. The two views are illustrated in ***Figure 5.17***. See how the three relational attributes are re-engineered into a single tuple.

It is interesting to compare this with the way relational databases work. These make explicit use of the mathematical tuples pattern. But because they do not treat the tuple as an object that can belong to a number of classes, they cannot use a single tuple to describe the two ‘relationships’. So they cannot compact information in the way shown in this example. In many ways, relational databases are more interested in operationally managing the rows and columns of stored computer information than in constructing a consistent semantics.

Figure 5.17: Tuples compacting information



8 Simplifying and generalising the information framework

The logical paradigm’s framework is not only more general than the substance paradigm’s, it is simpler. This is the sign of a good re-engineering. Look at ***Figures 5.18*** and ***5.19***. These illustrate how the logical paradigm uses less to do more. In ***Figure 5.18***, the dependent and independent secondary colour attributes both map

onto one class, coloured things. Furthermore, the primary red attribute and the independent secondary red attribute map onto the single class, red things. In ***Figure 5.19***, the primary relational attribute maps onto the <my car, me> couple and the secondary relational attribute maps onto the owned by tuples class.

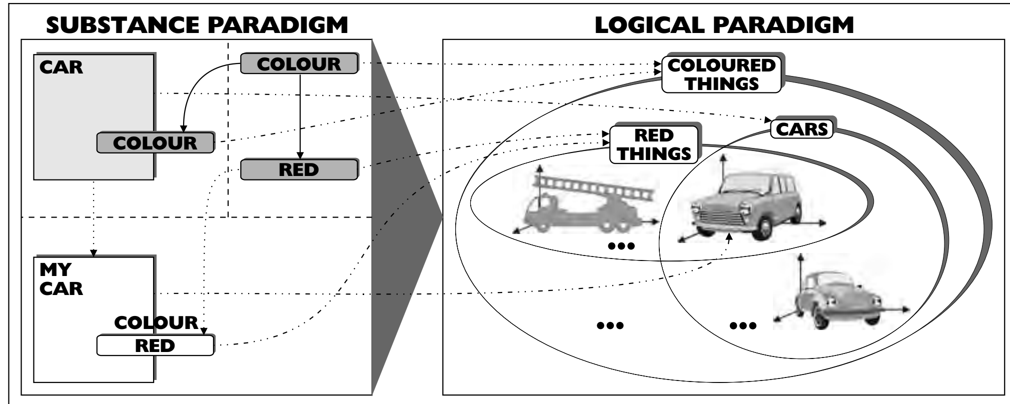


Figure 5.18: Comparing the paradigms—classes and members

There are also fewer, more general, types of thing—the logical paradigm only needs three sub-types of the general type – logical objects;:

- Logical classes,
- Logical tuples, and
- Individual logical objects.

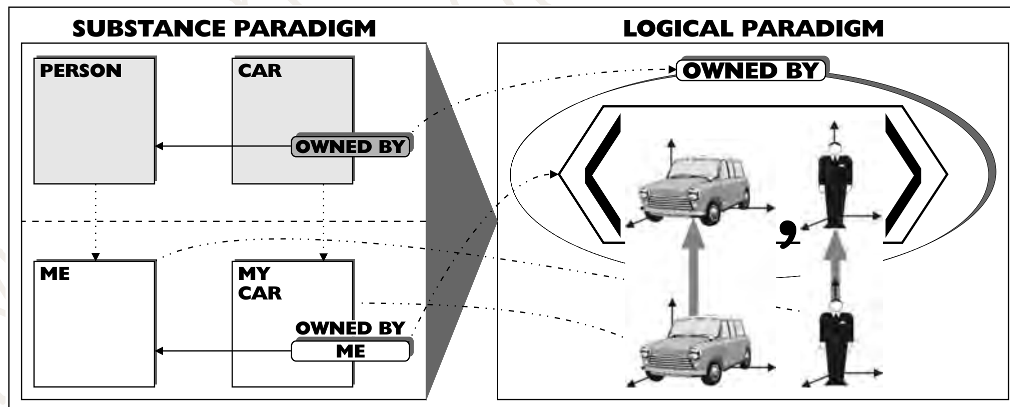


Figure 5.19: Comparing the paradigms—a tuples class and its member

This can be seen clearly in the more general comparison of frameworks in ***Figure 5.20***. This shows how, in the re-engineering, the distinction between substance and attributes disappears completely. For instance, both secondary substance and secondary attributes are transformed into classes.

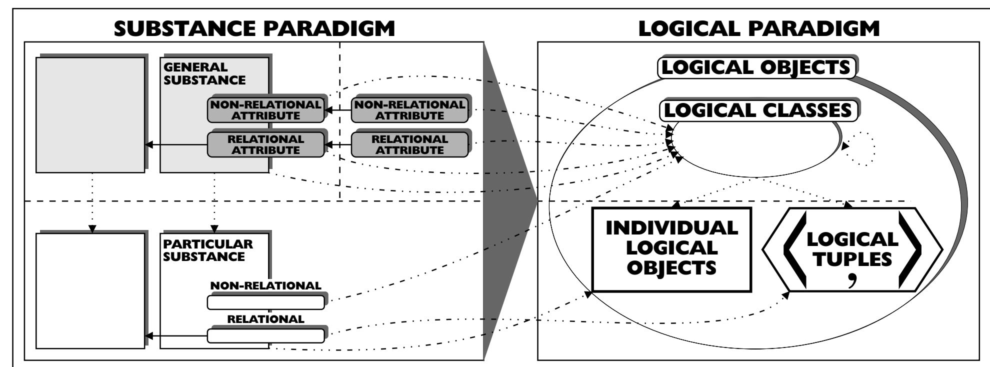


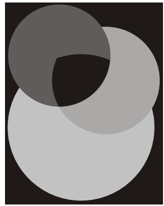
Figure 5.20: Comparing the frameworks

9 Summary

In this chapter we have re-engineered the substance paradigm's fundamental particles into their logical equivalents. We have just seen how this gives us a simpler and more general framework. However, this is just the bare bones of the framework. In the following chapter, we put the flesh on these bare bones.

© Copyright Chris Partridge chris.partridge@BOROCentre.com

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 6

The Logical Paradigm's Framework

1 Introduction
2 A sense framework for logical objects
3 An environment that encourages compacting
4 The problem with logical changes
5 The four key types of things
6 A new, conceptually more accurate, logical way of seeing things
7 Summary

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction

In the previous chapter we re-engineered the fundamental particles of the logical paradigm. This gave us a skeleton framework, which we flesh out in this chapter. As we do this, we will see how the new paradigm leads to a conceptually more accurate way of seeing.

2 A sense framework for logical objects

In the previous chapter, we focused on the extension–reference element in meaning's sense–reference combination. We now look at the sense element—at the types of structures or patterns that logical objects form. This helps to fill out the meaning (and so our understanding) of the paradigm.

When logical classes and tuples are used to describe things, several useful general patterns regularly appear. These give the paradigm its sense framework. We look at two of its key patterns, the super–sub-class pattern and the class–member pattern. We also spend some time looking at an important class pattern—classes of classes—that the substance paradigm is not powerful enough to capture.

2.1 The super–sub-class pattern

The substance paradigm recognised a restricted form of the super–sub-class pattern in its secondary level hierarchies (which we looked at in ***Chapter 3***). However, there was no semantic explanation of why the hierarchies existed, or what they were. The logical paradigm provides one and, at the same time, enhances the power of the pattern.

In the substance paradigm, secondary substances were arranged in a hierarchy of generality. For instance, animal substance was regarded as more general than pig substance and so had a higher position in the hierarchy. However, no real reason for this was given; it was taken as obvious. While it is obviously true that animal is more general than pig, it is not so clear why this is so within the substance paradigm. However, given that the notion of substance is so mysterious, this additional mystery is probably of little consequence.

Things are very different with classes. It is easy to explain why the class animals is more general than the class pigs and so what the hierarchy of generality is. The class animals is regarded as a super-class of the class pigs, which means that every member of the class pigs is also a member of the class animals. The class animals includes the class pigs and so it is put above it in the super–sub-class hierarchy (shown in ***Figure 6.1***). There is a perfectly straightforward physical explanation for the super–sub-class pattern; it is no semantic mystery.

2.1.1 Logical class 'inheritance'

One useful aspect of the secondary hierarchies in the substance paradigm was the inheritance pattern. In ***Chapter 4***, we looked at how the colour attribute was inherited by the car and van substances from vehicle substance (shown in ***Figure 4.15***). However, within the substance paradigm, we cannot explain what this inheritance pattern actually is.

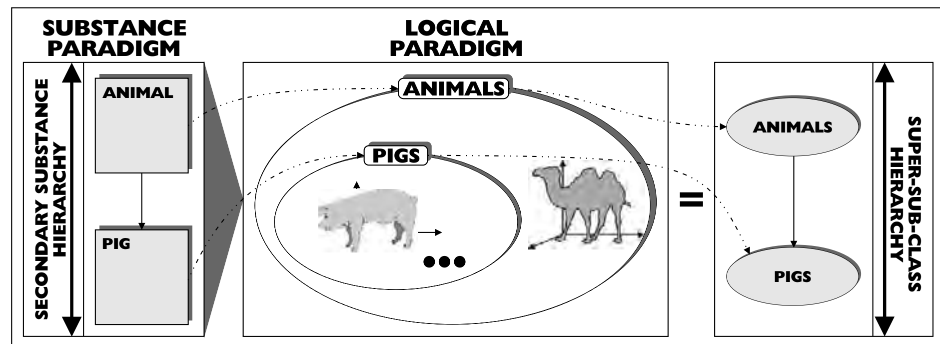


Figure 6.1: The super–sub-class pattern

The inheritance pattern is not lost in the shift to the logical paradigm, but it has a different basis—classes. You can see this in ***Figure 6.2***, which shows the pattern shifting. In the logical paradigm, the 'inheritance' of the colour attribute is explained by the super–sub-class hierarchy. The classes cars and vans are coloured because they are included in the class vehicles, which is included in the class coloured. The super–sub-class pattern provides a reassuringly tangible explanation for inheritance.

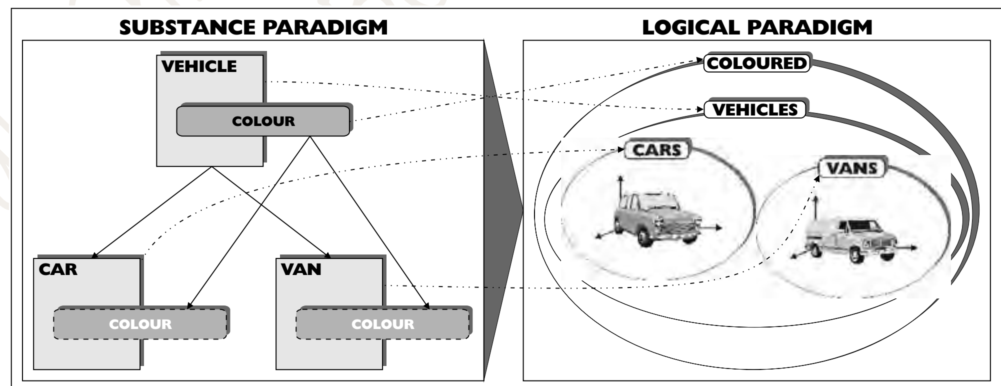


Figure 6.2: The shift to logical class 'inheritance'

2.1.2 Logical class 'multiple inheritance'

In ***Chapter 4***, we also identified a constraint on the secondary substance hierarchy down which attributes were inherited. It had a tendency towards what O-O calls single inheritance (see ***Section 4.3.1.3, "Single inheritance and OOPs"***). In the logical paradigm, with its super–sub-class pattern, this constraint does not exist. Let's look at this less constrained structure.

The single inheritance constraint meant a less general secondary substance could only belong to a single more general substance. In structural terms, it restricted the hierarchy to a tree ordering. In the logical paradigm, this constraint does not exist—a class can have as many super-classes as is needed. This results in a lattice ordering hierarchy, which O-O calls multiple inheritance.

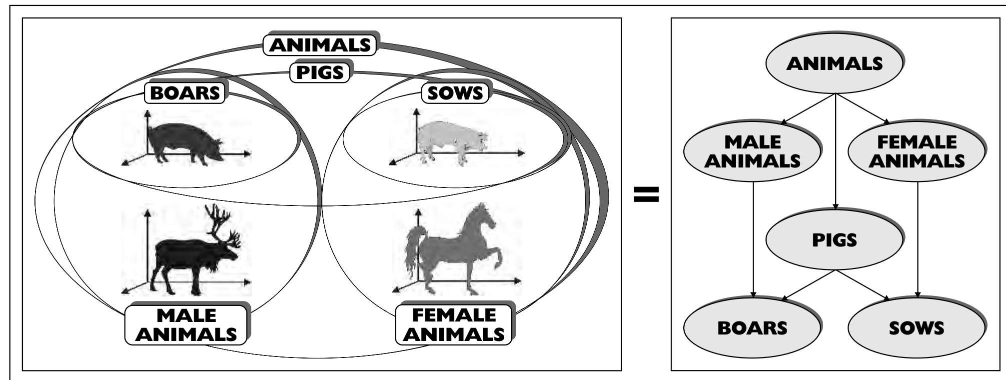


Figure 6.3: Logical class 'multiple inheritance'

The example in ***Figure 6.3*** illustrates the differences between the two hierarchies. In the substance paradigm, single inheritance stops us from seeing the secondary boar substance as a combination of the more general pig and male animal substances. Whereas, in the logical paradigm, this is the natural way to see it. The class boars is clearly the result of intersecting the classes pigs and male animals. It is easy to see this intersection's lattice structure in the class hierarchy diagram on the right-hand side of ***Figure 6.3***, where the classes boars and sows both have two super-classes.

2.2 The class–member pattern

The logical paradigm has a new hierarchy pattern, one that did not (and cannot) exist in the substance paradigm—the class–member pattern. And with it comes a new type of class object, a class whose members are also classes. When re-engineering entity systems, I am constantly surprised at the number of implicit classes of classes they contain. We will come across a good example of an implicit class of classes in the re-engineering of country full names attributes in ***Chapter 12*** (shown in ***Figure 12.30***.)

2.2.1 A weak pattern for classes

As noted in the last chapter, when Cantor defined a class, he defined it as an object that could be collected into a class. However, people steeped in a pre-class way of seeing things find this notion difficult to grasp. They often only take on board a weak pattern for classes. For example, this has happened in most object-oriented programming languages (OOPLs).

The weak pattern for classes sees a class as a collection, but does not see that a class is an object—only particular objects are objects. In this scheme of things, the idea of collecting together classes into a class cannot arise because weak classes are not objects and so cannot be collected together into a class.

This is a major structural constraint. Classes collect together things that are similar—things that have similar patterns. Classes themselves can have similar patterns. How do we capture this insight if we cannot collect them together into a class? We cannot.

2.2.2 A class of classes

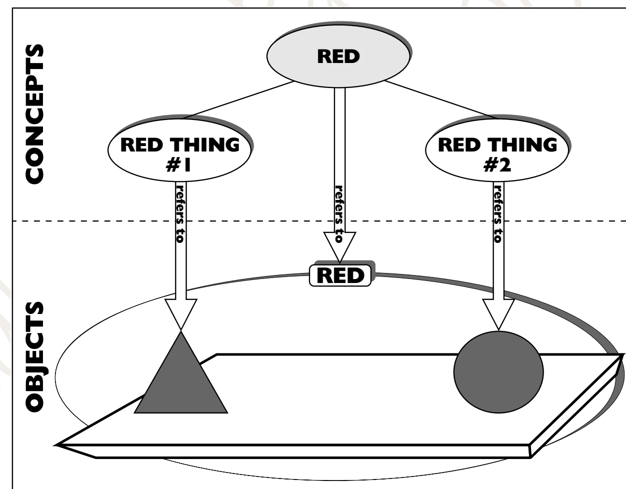
To help us get a good grasp of the strong notion of a class, we start by looking at a couple of simple examples, colours and car types.

2.2.2.1 A common example of a class of classes—colours

The colours class (which has individual colours, such as red and green, as members) is a class of classes. This idea can take some getting used to. Most people think of the colours red and green as abstract, intangible individual objects, which makes the class of these colours abstract and intangible as well.

The logical paradigm takes a very different view, which we work our way up to in this thought experiment. Assume I have two red shapes on a table. Under the logical paradigm, they belong to the class red. We now have three objects in the real world, two individual objects and one class object (shown in ***Figure 6.4***).

Figure 6.4:
Two red shapes

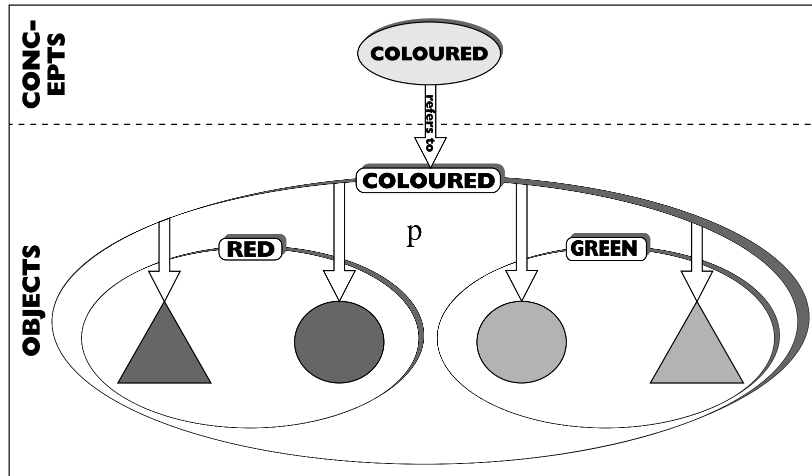


2.2.2.2 Expanded to colours

Now, assume that two more shapes, this time green shapes, are put on the table. Under the logical paradigm, they belong to the class green. I can now generalise across the classes red and green to their super-class coloured (shown in ***Figure 6.5***). I could do something similar in the substance paradigm, generalising the independent secondary red and green attributes into an independent secondary colour attribute. This was shown in ***Figures 4.12*** and ***4.13***.

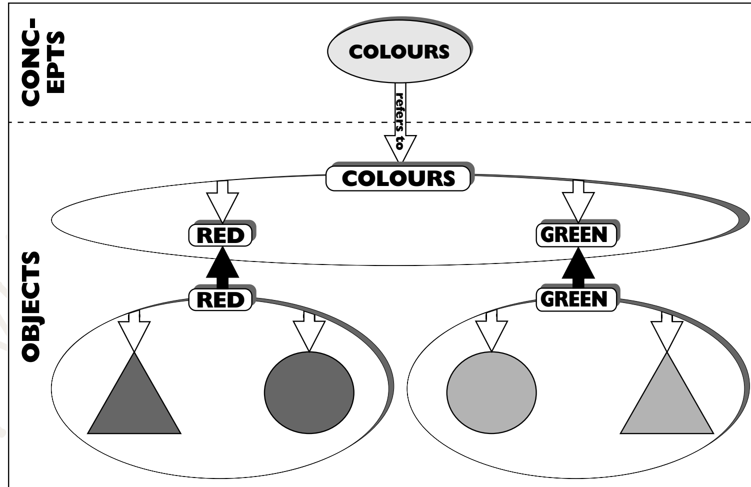
©Copyright chris.partridge@brore.com

Figure 6.5:
The class coloured



Then, I do something new and different. Something I cannot do in the substance paradigm. I consider the red and green classes as objects and collect them together into a class, colours (shown in ***Figure 6.6***). I can do this because classes are objects and all objects, even class objects, can be collected into a class. The class colours is very different from the class coloured. ***Figure 6.5*** shows that the class coloured has the red and green shapes as its members. Whereas ***Figure 6.6*** that shows the class colours has the class red and the class green as its members.

Figure 6.6:
The class colours



The form of diagram used in ***Figure 6.6*** is not wholly satisfactory. A class object (such as red) that is also a member of another class has to be signed twice. In this case, once as a lozenge on top of the red class oval sign containing the red member signs and again as just a lozenge inside the colours class oval sign. We will look at a better form of diagramming when we examine the object notation in Part Five.

2.2.2.3 *Another common example of a class of classes—user-defined car types*

Classes of classes are not just of academic interest. Almost every system of a reasonable size has a number of them. However references to them tend to be implicit. System parameters and user-defined types are often classes of classes. Account types, invoice types, and deal types are all classes of classes.

System builders are now so familiar with these type objects that often they can—with little or no analysis—intuitively work out how to implement them. The ease and familiarity of their implementation means that the obvious, but awkward, semantic questions that should arise during business modelling are not asked. Questions such as:

What are these objects? and

What do they model in the business?

We now ask these question about one such user-defined type—car types. Consider a simple computer system with user-defined car types constructed using a weak pattern for classes (in other words, classes of classes are not allowed). Most current systems are like this. To keep it simple, we only consider information about cars and car types in the system.

We assume that the users keep records of car types on a file. On each record they set a small car type indicator field to yes or no. We interpret this as implying that each car types record refers to an individual object belonging to the car types class. If a record's small car type indicator is set to yes, then we infer that the individual car type also belongs to the small car types class. This makes the small car types class a sub-class of the car types class.

We assume that the users also keep records of cars on a file, which can be linked to an appropriate car types record. We interpret this as saying that each car record refers to an individual object belonging to the cars class. The individual object is also linked to an object belonging to the car types class by a tuple belonging to the types-of tuples class.

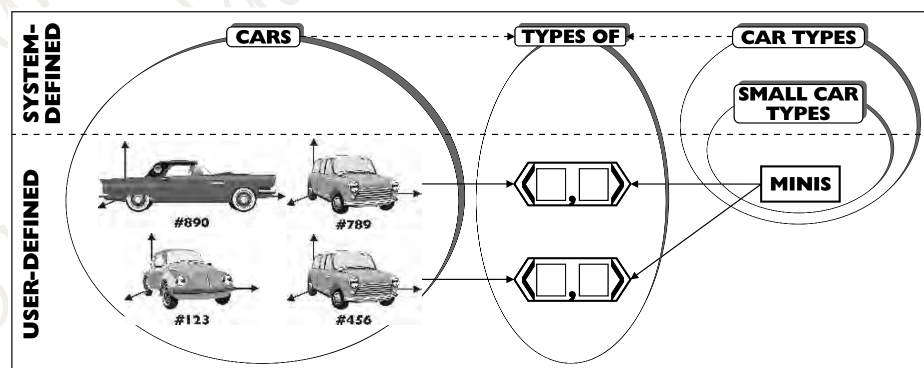


Figure 6.7: Original system

Assume that the users have already set up a record of one type of car—Minis—and set its small car type indicator 'field' to yes. They have also set up records of four cars—Car # 123, Car # 456 Car # 789 and Car # 890. Where Car #s 456 and 789 are Minis and so the users link their records to the individual car type record for Minis. The situation rep-

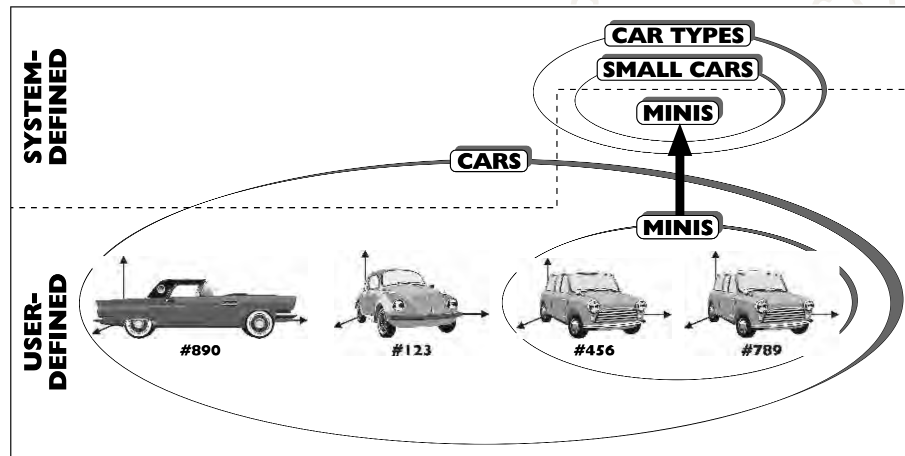
© Copyright Chris Partridge chris@BORO Centre.com

resented by the system is diagrammed in ***Figure 6.7***. This pattern of cars and their types will be familiar to most system builders.

Now, we ask the obvious question raised by the strong reference principle. What is an individual car types object, such as Minis, in the outside world? This will tell us what the cars types and small car types classes are. To answer the question, we need the strong notion of class, which allows classes to be members of classes.

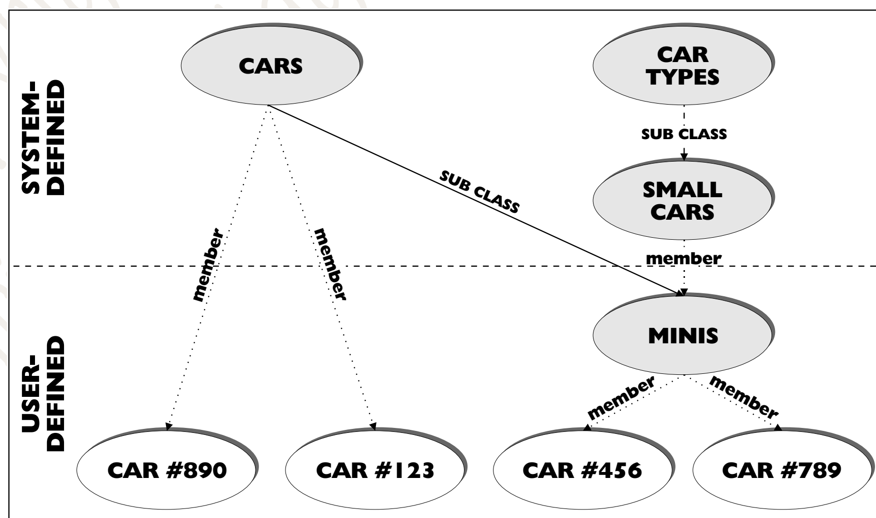
From a logical paradigm viewpoint the answer is clear. Minis is not an individual object, but a class —the class of Minis. Cars #456 and #789 are members of it. This makes the car types class a class of classes. The small car types class is also a class of classes. It is the class of all the car types that are small, so Minis belongs to it. This gives us the structure in ***Figure 6.8***.

Figure 6.8: Car types as a class of classes



It is easier to see the class–member pattern if we show it as a hierarchy (shown in ***Figure 6.9***). Compare this hierarchy with that in ***Figure 6.7***. Just by looking at the two, we can see that the class–member hierarchy has a simpler more coherent structure.

Figure 6.9: Car types class–member hierarchy



The weak class pattern forced us to make Minis an individual object because we wanted it to be a member of the car types class. This meant that it had to be linked back to the individual minis by a contrived types-of tuples class.

By contrast, the strong class pattern allows us to recognise that Minis is a class and also a member of car types. The resulting class–member hierarchy is more expressive, making clear the nature of minis and car types. The class of classes structure enables us to see in a simpler, clearer (and more accurate) way, and so gives us a semantically richer understanding of what the files and records really refer to.

2.2.3 Unconstrained class–member tuples

In ***Chapter 4*** (see ***Figures 4.17*** and ***4.18***), we identified that the way we saw substance tended to place two constraints on how primary substance could belong to secondary substance:

- single classification, and
- static classification.

The logical paradigm's replacement for secondary substance, classes, does not force these constraints on us. In fact, in its class–member pattern, they do not make sense. We shall see later on in this chapter how this greatly enhances the conceptual power of the paradigm. Here, we see how the class–member pattern enables:

- multiple classification, and
- dynamic classification.

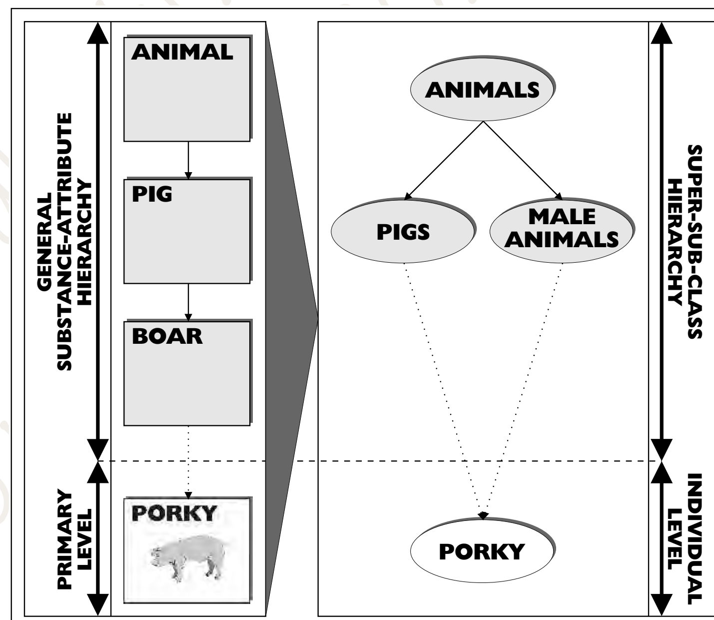


Figure 6.10: Logical class 'multiple classification'

© Copyright chris partridge BORO centre.com

2.2.3.1 Logical class–member 'multiple classification'

In the substance paradigm, a primary substance could only belong to one secondary substance. For instance, if Porky the boar belonged to boar secondary substance, he could not belong to any other secondary substance. In the logical paradigm, an individual object can belong to as many classes as is needed. So Porky could belong to the pigs class and the male animals class (shown in ***Figure 6.10***). This is useful, because it eliminates the need for a boar class. We examine how this enables us to compact more information into less structure later on in this chapter.

As we saw in ***Chapter 4*** (see ***Figure 4.16***), the substance paradigm can, to an extent, escape from its single classification structure, using attributes. It does this by mimicking a limited version of multiple classification. A primary substance cannot belong to more than one secondary substance. But, through its primary attributes, it can have more than one link to the secondary level. ***Figure 6.11*** has an example of this. There, both Porky and Porkette have two links to the secondary level; one to pig secondary substance, the other to a secondary gender attribute. This gives us a similar lattice structure to the multiple classification in ***Figure 6.10***.

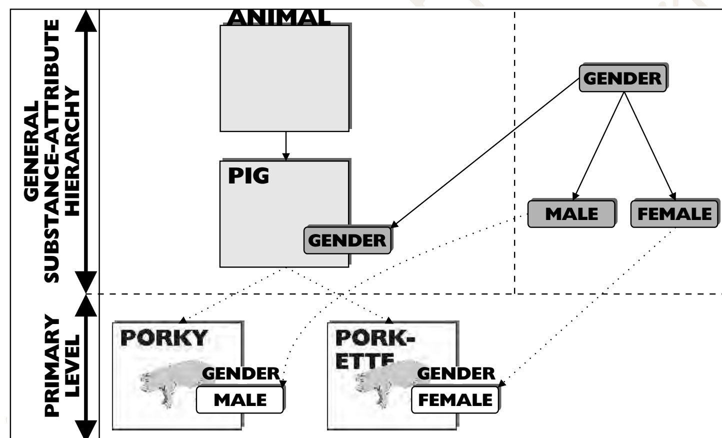


Figure 6.11: Multiple secondary links

2.2.3.2 Logical class–member 'dynamic classification'

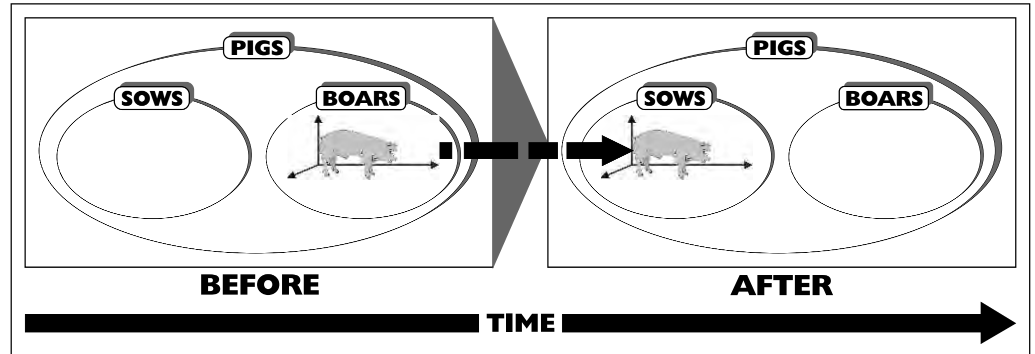
Primary substance was also constrained by static classification. If a primary substance belonged to a secondary substance, it always belonged to it; that could not change. For instance, if Porky the boar belonged to boar secondary substance, he always belonged to it.

The logical paradigm's class–member pattern enables dynamic classification —a less constrained structure. This means an individual object can change classes if that is what is needed. For example, if they bring in sex-change operations for pigs and Porky chooses to become a Porkette, then we can see this as he/she dynamically transferring from the class boars to the class sows (diagrammed in ***Figure 6.12***). The logical paradigm needs to explain this commonplace business pattern; one where bank accounts become overdrawn and deals settlements become overdue. Dynamic classification is

© Copyright chris.partridge@bors-centre.com

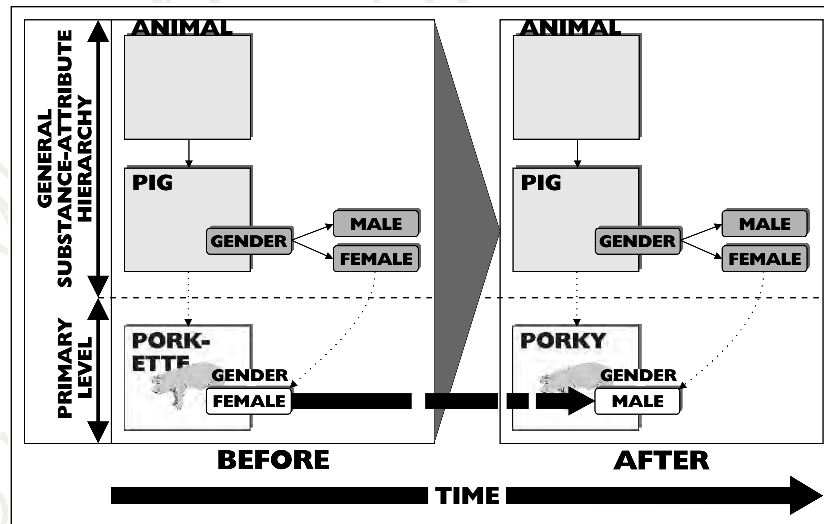
its solution. (We will re-engineer a very different view of this pattern when we look at the object paradigm in ***Chapter 8***.)

Figure 6.12: Logical class 'dynamic classification'



Again, through the use of attributes, the substance paradigm can mimic a constrained version of dynamic classification. A primary substance cannot change its secondary substance, but its primary attributes can change the secondary attributes they belong to. We can illustrate how this works with Porky's sex change. As shown in ***Figure 6.13***, Porky's primary gender attribute changes its secondary gender attribute. It starts off belonging to the secondary male gender attribute and ends up belonging to the secondary female gender attribute. The range within which an attribute can change is restricted. For example, a male gender attribute cannot change into a red colour attribute.

Figure 6.13: Dynamic secondary attributes



The way attributes mimic logical multiple and dynamic classification can be used to translate an object model onto an entity-oriented database. This is a useful feature because it means that, with some manipulation, an O-O business model can be fully implemented on a traditional entity database—although this gives the database an unusual structure.

I normally make use of this feature to build a validation system for the business model on an easy-to-use PC-based entity-oriented database (such as Microsoft's ACCESS). This means I can test the conceptual correctness of the model before it is translated into the system specification. I find that it saves a lot of time and effort if conceptual errors are found and fixed during business modelling. Without the validation system, they would be embedded into the system specification and only unearthed during acceptance testing.

2.3 The sense framework

Our look at the logical paradigm's sense framework should have convinced us that it is superior to the substance paradigm. There are the structural enhancements:

- The super–sub-class pattern can support a lattice hierarchy; it is no longer constrained to 'single inheritance'.
- The class–member pattern can support multiple and dynamic classification; it is no longer constrained to single and static classification.

These benefits proceed from a superior semantics, one based on extension and strong reference. The shapes of these patterns come from a more accurate understanding of the connections between tangible objects in the real world. This accuracy is not bought at the expense of complexity, quite the opposite in fact. The logical paradigm's framework is simpler than the substance paradigm's. And most of the framework that we have examined here, in particular the notions of class and tuple, is re-used by the object paradigm.

3 An environment that encourages compacting

The object paradigm provides a friendly environment for generalising patterns and so compacting the model. We can see the beginnings of this here in the logical paradigm. The removal of simple structural constraints, such as single classification, has encouraged generalisation and so increased the potential for compacting.

It is easy to dismiss the increased flexibility and sophistication of the structure of the logical paradigm as only marginally useful to modelling. This is only true if the new semantics is not understood and so the potential of the changes not exploited. When people understand the semantics (when they master the new way of seeing), they start using its enormous potential. We now look at a simple example that illustrates one part of this enormous potential. We see how the change from single classification to multiple classification offers the potential for literally astronomic increases in conceptual economy, packing more information into a smaller space.

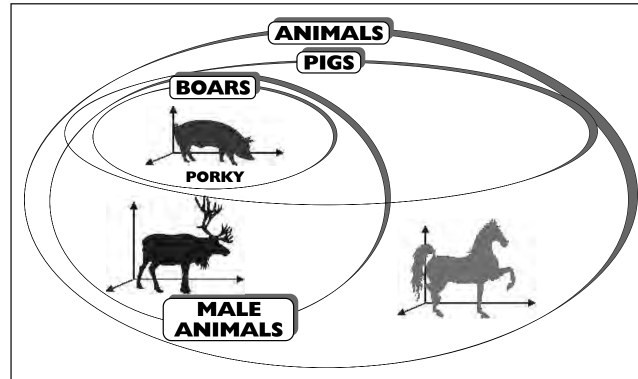
3.1 Example of the increased potential for conceptual economy

Consider Porky a boar, a member of the class boars, shown in ***Figure 6.14***. What is a boar? If we look it up in a dictionary, one definition (the one we are using here) is a male pig. If we interpret this in class terms, we are saying that the class boars is the intersec-

© Copyright Chris Partridge

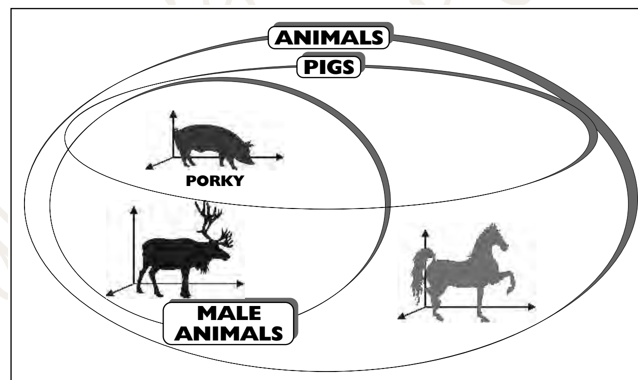
tion of the class pigs and the class male animals. In other words, as shown in ***Figure 6.14***, members of the class boars are just those members of the class pigs that are also members of the class male animals.

Figure 6.14:
Porky the Boar



Now that we have defined the class boars with its two super-classes, it is clearly superfluous. We could eliminate it from the final system without losing any functionality. The boars class would no longer exist and Porky would be directly a member of the pigs and male animals classes (see ***Figure 6.15)***. (However, we would typically keep a record of the generalisation, and so the redundant boars class. This would be used to help people understand the generalised model.)

Figure 6.15:
Generalised male pigs



We can see this gives us a multiple classification structure when the classes are drawn in a hierarchy, as in ***Figure 6.16***. This type of structure is not available in the substance paradigm, and so definitely not available to entity-oriented business modellers. It only becomes available with the logical paradigm.

This is a very tangible example of conceptual economy. We have eliminated a class with no loss of information. It is also an example of generalisation using super-classes, because the two super-classes, pigs and male animals, make the lower level class, boars, superfluous, enabling us to eliminate it from the system.

© Copyright chrispartridge.com BORO Centre.com

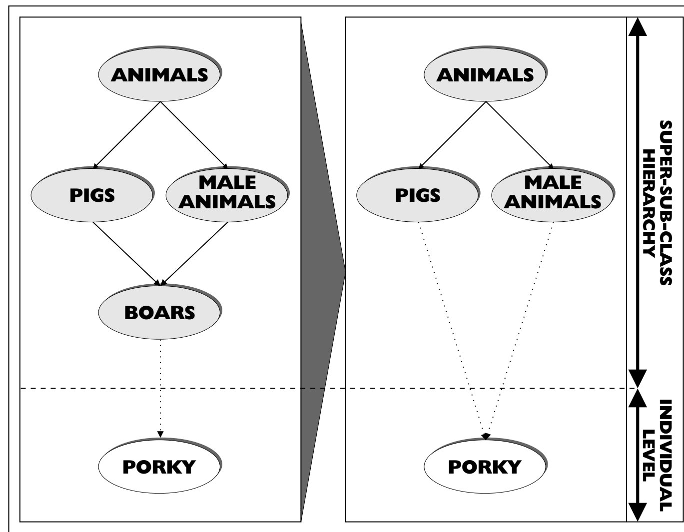


Figure 6.16: Shift to multiple classification

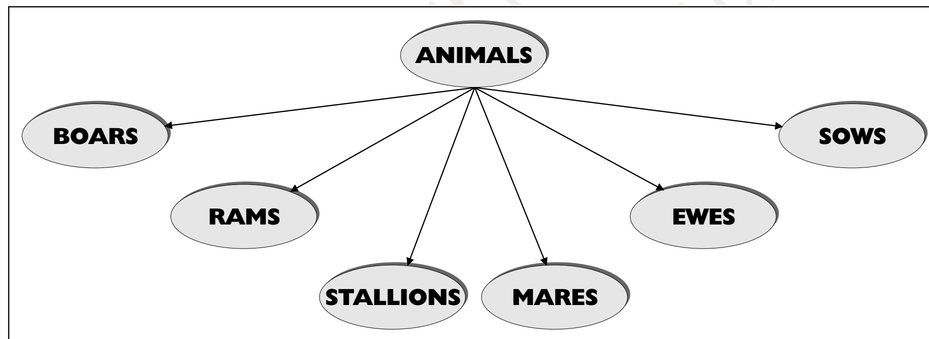


Figure 6.17: T Expanded framework of classes

his generalisation has led to the elimination of one class (boars). It also involves the construction of two new classes (pigs and male animals). So overall, this is hardly conceptual economy. This kind of generalisation only begins to deliver conceptual economy when there are more than a few objects in the system. If we expand the scope of the example to include sows, rams, ewes, stallions and mares, we get something like ***Figure 6.17***.

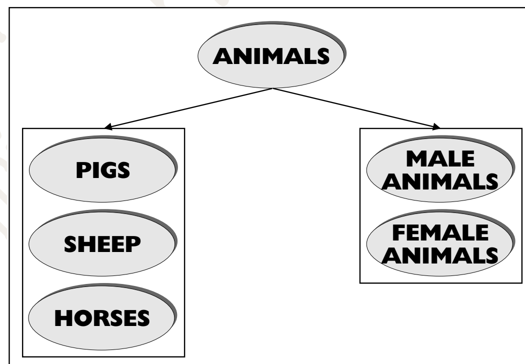


Figure 6.18: The generalised classes

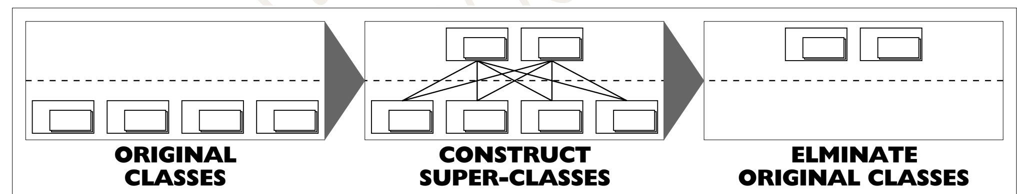
We now generalise these classes and construct five super-classes: pigs, sheep, horses, male animals, and female animals. We can now eliminate all the six original classes—such as stallion (male horse) and sow (female pig)—leaving the five super-classes. ***Figure 6.18*** shows the result. This is a similar structure to our original generalisation (shown in ***Figures 6.15*** and ***6.16***), the only change is the addition of the extra classes. This shows that the pattern in the original generalisation is fertile. It is applicable across a range of classes, something the lower level classes are not. We have found a re-usable pattern.

Things are also looking up from a conceptual economy viewpoint. We started with six classes and ended up with five more general classes—one class less. We have made a saving, but not on a grand scale. For this to happen, we need to increase the number of classes we start with.

3.2 Measuring multiple classification's potential for generalisation

If we want to measure the potential for generalisation that multiple classification offers, we take a different tack. In a generalisation, we construct a new top level of super-classes that renders all the lower levels superfluous. Their information is compressed and compacted into the more general super-classes (shown in ***Figure 6.19***).

Figure 6.19: The stages of generalisation



To measure multiple classification's potential for generalisation, we start at the final stage with the generalised classes and ask how many lower level classes they could have generalised. If we assume that the final classes are totally generalised, then working out the maximum potential number of lower level classes that can be constructed from them is a relatively simple mathematical calculation. The results are shown in ***Table 6.1***.

t is important to remember that the potential number of lower level classes is only a measure of multiple classification's *theoretical* power of generalisation—the actual number is likely to be much less. However, the table gives us a feel for the actual power. As we can see, the potential increases substantially as the number of general classes gets higher. This is because the number of possible intersections gets shown in larger, so the potential number of lower level classes that can be superseded grows exponentially.

IWe can use these figures to give us an indication of how multiple classification's potential for generalisation affects increases in scope and functionality. Assume that we double the number of objects in a totally generalised system. If we take the potential number of lower level classes as an indication of the scope and functionality of the system classes, then this doubling of size much more than doubles the scope and function-

© Copyright chris partridge chrispartridge.com

ality. For instance, when we double a system with 10 objects, we should have a theoretical 1000 fold (1048550/1013) increase in scope and functionality. This is an increase of three orders of magnitude. If we had started with a larger system, the increase would be even higher. Doubling from 100 to 200 objects gives a theoretical 30 orders of magnitude increase.

Number of general classes	Potential number of lower level classes
1	0
2	1
3	4
4	11
5	26
10	101
20	10485
100	1.27×10^{30}
200	1.61×10^{60}
N	2^N-N-1

Table 6.1: Theoretical power of generalisation

These figures obviously only indicate a theoretical potential; practically, they are not going to be reached. But even if substantially lower levels are reached, this still represents a significant increase in conceptual economy. As we discussed in the ***Prologue***, our experience with increasing the scope of business models is of significant increases in conceptual economy (shown in ***Figure P.5***). We also discussed how increasing the scope of a traditional system building project does not work in the same way. This is, in part, because it does not have access to the resources of multiple classification.

It seems difficult to understand why something that obviously makes common sense, such as generalisation, has not become a central feature of business modelling before. If getting a significant improvement in conceptual economy were easy, it would have been done long ago. There has to be a catch. These suspicions are well founded; there is a catch. It is that we can find it difficult to see the general patterns for the high-level classes that will supersede the original lower level classes.

I have found that the secret to seeing these high-level general classes lies in the object paradigm. This gives us a much better, much more conceptually accurate, understanding of what objects are, which unearths really powerful re-usable patterns. These naturally form the basis for very general classes. Without the paradigm, we cannot really exploit class's potential for compacting through generalisation properly. In the worked examples in Part Six, we see how this works; how using the object paradigm helps us to see the right general pattern and how this leads to compacting.

© Copyright Chris Partridge chris.partridge@BOROCentre.com

3.3 Strong sense of object

The preceding examples illustrate how the logical paradigm provides a relatively unrestricted environment for generalising and compacting patterns. One important reason for this is its strong sense of object. This means that everything in the paradigm is a logical object. We can this in its two new tools for capturing patterns, classes and tuples.

Classes describe collections of objects with similar patterns. Tuples, and their tuples classes, describe the connections between objects that make up the patterns. Both these tools have two important structural properties:

1. They can apply to any object. In other words, there is no universal structural restriction on an object belonging to a class or being a place in a couple.
2. The constructed classes and tuples (and their tuples classes) are objects. This is important because the tools cannot be applied to themselves or each other unless they are objects. In other words, class objects can be members of class and places in tuples, and so on.

These structural properties are of practical importance. Classes and tuples, just like individual objects, have patterns. If the pattern capturing tools could not be applied to them, then these patterns could not be described and the resulting model would then be less complete.

4 The problem with logical changes

We now look at how the logical paradigm deals with the last of the four key types of things we identified in ***Chapter 3***—changes happening to things. While the logical paradigm provides a good semantic framework for the first three types of things, it has problems with this fourth one.

4.1 Logical bodies persisting through change

As we noted in ***Chapter 5***, substance plays two roles. It is the home for attributes and also the means for bodies to preserve their identity over time. The logical paradigm deals with the first role—attributes are replaced with classes of tangible extensions. However, there is still the problem of explaining bodies' identity through change.

The problem, which the English thinker David Hume so precisely describes (but doesn't resolve) in his *A Treatise of Human Nature*, still exists for the logical paradigm. He sets out the objective:

> Our chief business, then, must be to prove, that all objects, to which we ascribe identity, without observing their invariableness and uninteruptedness, are such as consist of a succession of related objects.

And then uses a series of thought experiments to show that it is impossible to achieve:

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

> . . . suppose any mass of matter, of which the parts are contiguous and connected, to be plac'd before us; 'tis plain we must attribute a perfect identity to this mass, provided all the parts continue uninterruptedly and invariably the same, whatever motion or change of place we may observe either in the whole or in any of the parts. But suppose some very *small* or *inconsiderable* part to be added to the mass or subtracted from it; tho' this absolutely destroys the identity of the whole, strictly speaking; yet as we seldom think so accurately, we scruple not to pronounce a mass of matter the same, where we find so trivial an alteration. . . .
>
> There is a very remarkable circumstance, that attends this experiment; which is that tho' the change of any considerable part in a mass of matter destroys the identity of the whole, yet we must measure the greatness of the part, not absolutely but by its *proportion* to the whole. The addition of diminution of a mountain wou'd not be sufficient to produce a diversity in a planet; tho' the change of a few inches wou'd be able to destroy the identity of some bodies. . . .
>
> This may be confirm'd by another phenomenon. A change in any considerable part of a body destroys its identity; but 'tis remarkable, that where the change is produc'd gradually and insensibly we are less apt to ascribe to it the same effect. . . .
>
> But whatever precaution we may use in introducing the changes gradually, and making them proportionable to the whole, 'tis certain, that where the changes are at last observ'd to become considerable, we make a scruple of ascribing identity to such different objects. There is, however, another artifice, by which we may induce the imagination to advance a step farther; and that is . . . some *common end* or purpose. A, ship of which a considerable part has been chang'd by frequent reparations, is still considered as the same; nor does the difference of materials hinder us from ascribing an identity to it. . . .
>
> But this is still more remarkable, when we add a *sympathy* of parts to their *common end*, . . . This is the case with all animals and vegetables; . . . The effect of so strong a relation is . . . that in a very few years both vegetables and animals endure a total change, yet we still attribute identity to them, while their form, size and substance are entirely altered. An oak, that grows from a small plant to a large tree, is still the same oak; tho' there be not one particle of matter . . . the same. An infant becomes a man, and is sometimes fat, sometimes lean, without any change in his identity.

These puzzles over what makes something the same have a long history. In ***Chapter 4***, we looked at how the substance paradigm explained sameness over time with the mysterious notion of substance (see ***Figure 4.6***). In the last chapter, we saw how the logical paradigm has replaced mysterious substance with the more physical notion of extension. However it cannot solve the problems of identity persisting through time. Our notions of sameness and identity need to evolve into something more sophisticated before we can see a solution. We have to wait for the next stage of evolution, the object paradigm, for this to happen.

4.1.1 The whole–part pattern persisting through time

This problem with identity spills over to the whole–part pattern (the study of which is known as mereology). This was and is an important pattern. It was analysed by Aristotle and plays an important part in current O-O analysis and design, along with the class–

© Copyright Chris Partridge chris.partridge@boroprogram.com

member pattern. Interestingly, in Frege's time what we call classes and wholes were both called classes: one a collective class, the other a distributive class. And both types of classes were studied by mathematicians—mereologic was a part of logic. Nowadays mathematicians tend to only be interested in the class–member pattern.

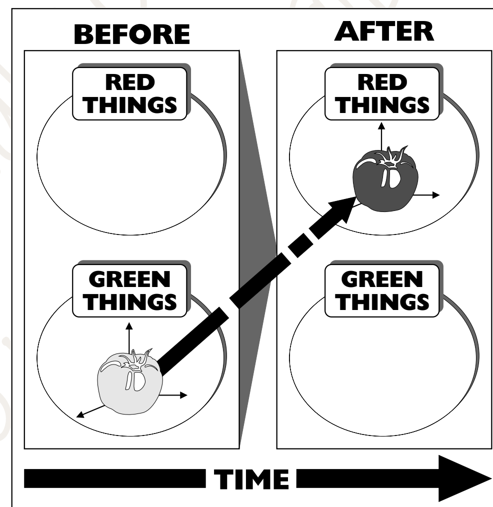
The logical paradigm's notion of extension may seem, at first sight, to offer a semantic explanation of wholes and parts. After all, is a part not something whose extension is contained in the extension of the whole? This works for a point in time, but—like identity, and for the same reason—does not work over time. My hand, now, is part of my arm, now. But is my hand, now, a part of my arm tomorrow? Is being a part tied into a moment of time?

For hands and arms this is not much of a problem, but the issue goes deeper when we move from parts in space into parts over time. We talk about the morning being part of the day, of our childhood being part of our life. These whole–part patterns do not fit into the logical paradigm's extension-based explanation. We need to wait for the object paradigm's more sophisticated notions of extension and sameness to resolve these difficult puzzles.

4.2 Logical changes

The notions of bodies and changes are intimately connected. The problem of identity as bodies persist through time has another side. This is working out what the changes that happen to the body are. In the substance paradigm, they were not a thing, but what happened to accidental attributes. The logical paradigm's replacement for changing attributes is dynamic classification. This is the same as we discussed earlier where an object shifts from one class to another (shown in ***Figure 6.12***).

Figure 6.20: Dynamically classifying extensions



It would have been ideal if the logical paradigm could have found a way to make dynamic classifications objects. Then, they could have participated in object's useful patterns; for instance, we could have had classes and tuples of dynamic classification objects. The problem is that if dynamic classifications are objects, they have to have

© Copyright Chris Partridge chris@BOROCentre.com

extensions, and it is unclear what the extension could be. This thought experiment illustrates the problem.

Assume that there is a tomato on the table. It is currently green, so a member of the class, green things. Assume it instantaneously turns red, so becoming a member of the class, red things. In logical-speak, it is dynamically re-classified. What is happening from an extension viewpoint? As extensions exist at instants, we could say that at every instant up to a certain point, the instantaneous extension belonged to the class green things. And then at every instant after that point, the instantaneous extension belonged to the class red things. This is an acceptable explanation, but, like the substance paradigm, it puts the dynamic classification on a different dimension. Dynamic classification happens to objects, it is not an object (illustrated in ***Figure 6.20***).

If we were keen to make the change an object, to say it is the instantaneous extension of the tomato when it changes colour would be a strong temptation. We can use the logical paradigm's version of Zeno of Elea's paradox to show this leads to contradictions. We looked at the substance paradigm's version in the discussions of Aristotle's notion of change in ***Chapter 4***. There, we used the paradox to show that motion—a type of change—could not be an attribute. Here, we use it to show that dynamic classifications cannot be extensions and so cannot be objects.

If the tomato's change is an instantaneous extension, then as the tomato is coloured, it must have a colour at that instant. This cannot be red or green, otherwise it would not be changing. So it must be some other colour. Even if we assume it has a colour, the problem re-surfaces. Let's assume it is blue; in other words, its instantaneous extension belongs to the class blue. We are back where we started, only with two dynamic classifications—one from red to blue and another from blue to green. However many instants we select, the same problem occurs. Dynamic classifications cannot be consistently translated into extensions and so objects.

There are no practical, operational problems with using dynamic classifications in business modelling. What is at issue is a missed opportunity. From a semantic viewpoint, dynamic classifications are inadequate and this compromises the conceptual power of the business model. We can see the semantic inadequacy of dynamic classifications from the way they breach the strong reference principle. They are not objects, they do not have an extension, so we cannot refer to them directly. The connection between the concept and object is mysterious, because the object is mysterious.

5 The four key types of things

We can see how far the logical paradigm has taken us from the substance paradigm by looking at the four key types of things we identified in ***Chapter 3***:

- particular things,
- general types of things,
- relationships between things, and
- changes happening to things.

©Copyright Chris Partridge chris.partridge@BORO Centre.com

In the substance paradigm, a thing was particular if it had primary substance and general if it had secondary substance. My car is particular because it has primary substance and car is general because it has secondary substance. In the logical paradigm, a thing is particular because it is an individual extension and general if it is a class of extensions. My car is particular because it is an individual extension and the cars class is general because it is a collection of extensions. The notion of primary substance has been replaced by the notion of extension, and the notion of generality by that of class.

In the substance paradigm particularity and generality applied to attributes as well as substance. From the logical paradigm perspective, the attributes divide into two types, non-relational and relational. Non-relational attributes, both particular and general, become classes. For example, my car's particular redness and redness in general become the class of red things.

Relational attributes take us onto the third key type of thing—relationships between things. The substance paradigm's relational attributes are re-engineered into something quite different—a tuple and a class. My car's ownership relational attribute becomes the couple <my car, me> and the owned by tuples class.

For the first three types of things, the logical paradigm has made a substantial change to (and a substantial improvement upon) the substance paradigm. However in the fourth type of thing—changes happening to things—it does not contribute much. In fact, as the ***Section 4, "The problem with logical changes"*** shows, the shift from substance to extension means that the advantage an unchanging substance has in explaining how objects persist through time has been lost.

Furthermore, neither the substance nor the logical paradigm offers an explicit description of what changes are. This problem with changes, along with the problem of what it means for bodies as extension to persist through change, is resolved by the object paradigm's semantics, which we look at in Part Four.

6 A new, conceptually more accurate, logical way of seeing things

Working within the logical paradigm's new semantics involves a new, conceptually more accurate, way of seeing things. In ***Chapter 4*** we discussed how the development of paper and ink technology led to a more accurate notion of signs. There is a similar development here.

6.1 A new, conceptually more accurate, whole–part pattern

For instance, the logical paradigm has developed a more accurate distinction between a whole–part and a class–member tuple than we are used to. Currently most of us would regard the expressions 'part of a group' and 'a member of a group' as interchangeable. In our ordinary everyday language there is no real distinction between:

John is part of the technical services group, and

John is a member of the technical services group.

©Copyright Chris Partridge

However within the more accurate logical paradigm there is a distinction—being a part and being a member are quite different. So which of the two is the person's connection with the group? We can work this out logically because:

If A is a part of B, and

B is a part of C, then

A is also a part of C

Consider a person's hand, it is undoubtedly part of the person. Therefore, if the person is part of the group, his or her hand would also be part of the group. In logical format:

If a person's hand is a part of a person B, and

A person is a part of a group, then

The person's hand is also a part of the group.

However, the person's hand is clearly not part of the group, so the person's connection to the group cannot be whole–part. But it can be class–member, as this logical format shows:

If a person's hand is a part of a person, and

A person is a member of a group, then

The person's hand is not normally a member of the group.

This implies that the group is a class and not a whole.

We saw (in ***Chapter 4***) why the Huichol Indians thought corn was the same as deer. They did not have as accurate an idea of signs and sameness as us (the shift to more accurate distinctions was illustrated in ***Figure 4.19***. Similarly, some people quite happily talk and think of people as part of groups in the 'same' way as they think of their finger as part of their hand. They have not yet acquired the logical paradigm's more accurate way of seeing whole–part patterns. This shift is illustrated in ***Figure 6.21***.

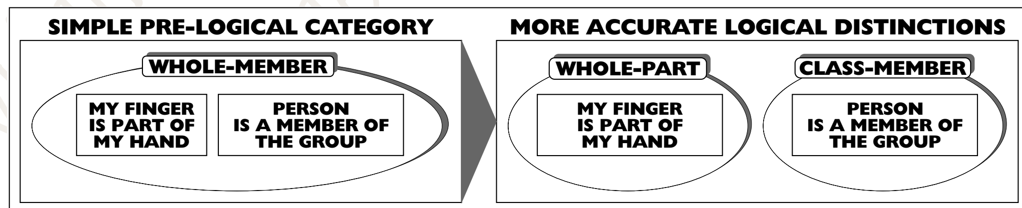


Figure 6.21: Shifting to more accurate logical distinctions

6.2 Other new, conceptually more accurate, patterns

The logical paradigm also leads to other new distinctions. In ordinary everyday language it is easy to confuse a sub-class tuple with a class–member tuple. For example, we do not normally say 'Rover is a *member* of the class dogs' and 'dogs is a *sub-class* of mammals'. Instead we say 'Rover is a dog' and 'a dog is a mammal' using the same connecting 'is a' phrase. This makes the two easy to confuse. Since the logical paradigm explained what the two are, logicians have been clear on the difference. However before the logical paradigm, even logicians had problems. For example, even the 17th

century thinker Gottfried Leibnitz, who has been called one of the all time great logicians, was confused about the difference at times.

The logical paradigm is much newer than the earlier paradigms; it is barely a hundred years old. This is an issue, because it can take centuries for a new way of seeing to be absorbed into a culture and become a standard part of the way people see things. It means that the logical paradigm has not really had time to sink in. This is particularly true with its notions of a class of classes and a tuples class. However, the slow overall progress of assimilation of class patterns is perhaps surprising given that most schoolchildren are taught them in mathematics lessons.

Most people find classes of classes an unnatural idea. The colours and car types examples in this chapter may seem logically correct but do not feel natural. When people start business object modelling they tend to find it difficult to spot a class of classes, even when it is staring them in the face.

The notion of a relational attribute is more insidious. We probably feel more comfortable with the idea of a connection as a separate tuple object, rather than an embedded relational attribute. For example, we are happy to see my ownership of my car as a separate object from both me or my car. However we tend to see each of these connected objects as having its own identity. We can demonstrate this with the example illustrated in ***Figure 5.16***. Most people naturally assume that the sentences 'Prince Charles is the father of Prince William' and the 'Prince Charles is taller than Prince William' refer to different connections. They do not naturally assume that both sentences refer to the same connection (a couple) that is a member of two tuples classes: father of and taller than. To them, the logical paradigm's more accurate way of seeing is counterintuitive.

All these new ways of seeing are inherited by the object paradigm. If we want to business object model, then we need to master them—this involves not just understanding the principles but actually seeing in the new way. This takes practice. The worked examples in Part Six are a good starting point, but for most people it will take some time before the new way of seeing is embedded so firmly that it becomes instinctive.

7 Summary

The logical paradigm provides us with a semantic replacement for the substance paradigm (or at least a replacement for the first three of the four key types of things). Because it is based on tangible extension, it is immune to the criticisms of mysterious unknowable substance.

The backbone of the paradigm is the strong reference principle that asserts that concepts can only refer to objects with extension. This was a guiding light in the re-engineering of the substance paradigm, which transformed the substance paradigm's particles into the simpler and more general particles of the logical paradigm, the class and the tuple.

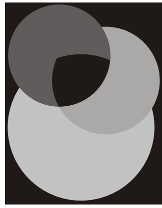
The re-engineering not only simplified the particles, it made them more general and so more powerful. The new particles can handle multiple 'inheritance', multiple classifica-

© Copyright chris partridge
chris.partridge@BORO.com

tion and dynamic classification. This enables a new logical environment that is much more generalisation-friendly than the substance or entity paradigms.

This shift to the logical paradigm takes us halfway along the evolution to object semantics. It has prepared the ground the shift to the object paradigm, which we now make in Part Four.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



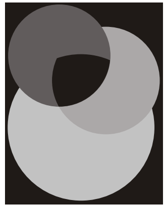
Part Four

Shifting to the Object Paradigm

Chapter 7 Physical Bodies as Four-Dimensional Objects

Chapter 8 Shifting to the Object Paradigm

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 7

Physical Bodies as Four-Dimensional Objects

1 Introduction
2 The logical semantics for physical bodies
3 The shift to object semantics
4 Physical stuff objects
5 Classes of four-dimensional objects
6 Tuples of four-dimensional objects
7 A new way of seeing bodies—a key type of thing
8 Summary

© Copyright Chris Partridge chris.partridge@BOROCentre.com

1 Introduction

To people familiar with object-oriented programming, it might appear that logical semantics has all that object-orientation needs. At an operational level they are right, but at an understanding level—the level of business modelling—they are missing something. As we saw in the final sections of the previous chapter, the logical paradigm's semantics are shaky for the last of the four key types of things—changes. The shift to the object paradigm is driven by the need to give change a firm semantics.

So in Part Four, we focus on the object paradigm's semantics for change. We do this in two parts. In this chapter, we deal with the semantics of physical bodies, persisting through changes. In the following chapter, we consider the semantics of the changes themselves

We start this chapter with a series of thought experiments that clarify the logical semantics for physical bodies, persisting through changes and the issues it raises. We then explore the shift to object semantics for physical bodies and see how it resolves the issues raised by logical semantics.

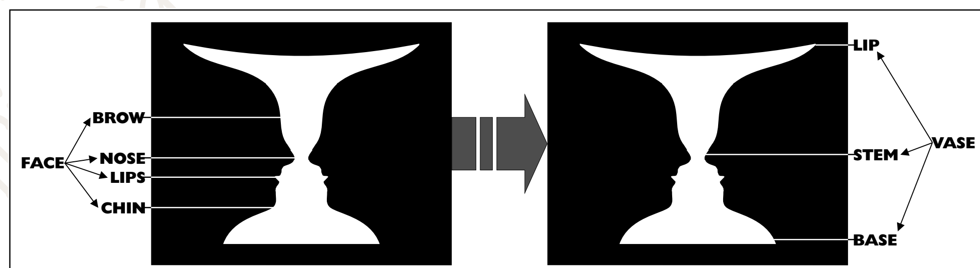
Then, we look at an example of how the new object semantics for physical bodies can transform our current notions. We see how our notion of 'stuff' is re-engineered into the semantically richer notion of stuffs as physical bodies.

Finally, we re-engineer the logical notions of class and tuple objects constructed from physical bodies.

2 The logical semantics for physical bodies

The evolution from logical semantics to object semantics involves a pure shift in our understanding of what objects, whether bodies or changes, are. In ***Chapter 1***, we used an ambiguous picture as an analogy for how paradigm shifts work. This is useful for explaining what happens in the shift to object semantics. In ***Figure 7.1*** (based upon ***Figure 1.2***), when we shift from seeing two faces to seeing a vase, nothing in the underlying picture changes. In the same way, the shift to object semantics does not involve any new facts, just a new way of seeing the old facts.

Figure 7.1: Shifting views



©Copyright Chris Partridge
chris.partridge@BOROCentre.com

This new way of seeing resolves a central problem for physical bodies, explaining how identity persists through change. We have illustrated this problem before with a lepidopter. Over time it goes through various stages. We need to be able to explain why and how these different stages are, in some sense, the same object; even though, for instance, the butterfly stage of the lepidopter is so obviously different from its caterpillar stage. We saw in ***Chapter 4*** how Aristotle's substance paradigm gave a consistent explanation (illustrated in ***Figures 4.5*** and ***4.6***), but one based on the now discredited notion of substance. In ***Chapter 6***, we saw that logical semantics cannot give an explanation; that something can both be the same and different at different times is a mysterious fact.

Before we make the shift away from logical semantics, we give ourselves a context by examining our current intuitions about physical bodies' identity over time. We do this in three thought experiments:

- The wrecked car,
- The car-minus, and
- The chairman experiments.

These reveal how we determine whether physical bodies are the same at different times and how 'two' physical bodies can be the same at one time and different at another time. We gain further insight by examining how the substance paradigm deals with these thought experiments.

2.1 Wrecked car thought experiment

We instinctively use a key criterion to decide whether an object is the same at different times—this is whether it has persisted continuously through time. In everyday life, we often make the decision on the basis of how the physical body looks and feels. This first thought experiment is designed to show how seriously we take the criterion of continuity and that the look and feel of a physical body are only practical stand-ins.

Assume I buy a brand new car. Using an advanced science fiction device, supplied to me for this experiment, I make a record of the type and position of every atom in the car. I then lend my car to a friend for a week.

At the end of the week, he brings me two cars. They are the same make and model, but one is brand new and the other is a smashed up wreck. He says that one of the cars is mine and asks me which one. The smashed up car does not look at all like my original new car; everything is either bent, torn or scraped. But the other car does. I double check by using the science fiction device to get a picture of its atomic structure, which I compare with the record I made at the beginning of the week. They match perfectly. With this evidence, it would only be natural for me to assume this is the car I bought at the beginning of the week.

Now my friend introduces me to a camera crew who have been filming my car over the last week. They show me their film. It starts at the beginning of the week and follows, without a break, everything that happens to the car and ends up with my friend bringing the two cars in to me. In the film, I see my car going through a number of accidents until it is the smashed up wreck that is before me now. Now I realise that the smashed up car

© Copyright Chris Partridge chris.partridge@BOROCentre.com

is, in fact, mine. Now I am not, and you wouldn't be, tempted to say that the new car is mine.

Why is this? It is because there is a continuous link between the car at the two times. This takes precedence over any evidence about how the car looks and feels. Continuity is the key criterion. It is the ultimate basis for our judgements on whether things are the same at different times. But it does not explain why they are the same.

2.2 Car-minus thought experiment

At any one time, two objects must either be the same or different. In this thought experiment, we see that the same is not true over time. Sameness can change over time; two objects can be different at one time and then the same at a later time.

Assume, again, that I bought a new car last week. An object is an extension, so I can construct an object by specifying an extension. Assume that I did this when I bought my car; assume I chose an extension, consisting of the car minus its back seats—and called it car-minus. Car-minus is obviously different from my car; they have different extensions. Car and car-minus are also both physical bodies that persist through time.

Now, assume that today I take the back seats out of my car and destroy them. Then, my car has changed; it is now without any back seats. But car-minus has not changed; the back seats were never part of it. It would appear that car and car-minus now have exactly the same extension; my car minus its back seats.

Under logical semantics, this means they must be the same object. Because physical bodies are instantaneous extensions, at a particular time, one can determine whether they are the same or different by seeing whether they occupy the same extension. We cannot meaningfully ask this question in the same way about physical bodies at different times. Because they change position, shape and size, their extension is not a reliable guide.

2.3 Chairman thought experiment

The car-minus experiment is contrived. It was meant to be, so that we could see the situation clearly. Because it is academic, I'm sure some (probably most) of you are tempted to dismiss it as irrelevant to anything commercial. But you should not. Any physical body could end up in a similar situation. We can see this in the following thought experiment that uses a modern version of an ancient puzzle; one that was known well before Aristotle's time. The puzzle was often expressed as a question—can two things be in the same place at once?

Consider Mr. Jones, the Chairman of NatLand Bank. Under logical semantics, if the concept 'Chairman of NatLand Bank' is legitimate it must refer to an object, similarly for the concept 'Mr. Jones'. In fact, we know they are both concepts and refer to the same object.

Technically speaking, under logical semantics, objects are extension. And if what appears to be two objects share the same extension (in other words, have the same

© Copyright Chris Partridge chris.partridge@BORO-Centre.com

height, width and depth), they are really the same object. Two objects *cannot* have the same extension at the same time. Because we know that the concepts ‘Chairman of NatLand Bank’ and ‘Mr. Jones’ refer to the same extension, they must, by the logical semantics’ definition, refer to the same object.

We now move on a week or two. Mr. Jones has resigned his chairmanship and Mr. Smith has been appointed the new chairman. From logical semantics’ perspective, the concept ‘Chairman of NatLand Bank’ now points to the same extension as the concept ‘Mr. Smith’ (shown in ***Figure 7.2***). It is plain that the concept ‘Chairman of NatLand Bank’ has changed its reference and now points to Mr. Smith. And this is not a special situation with an obscure case; it occurs in every business with every position, from tea boy up to managing director.

Figure 7.2: Changing reference

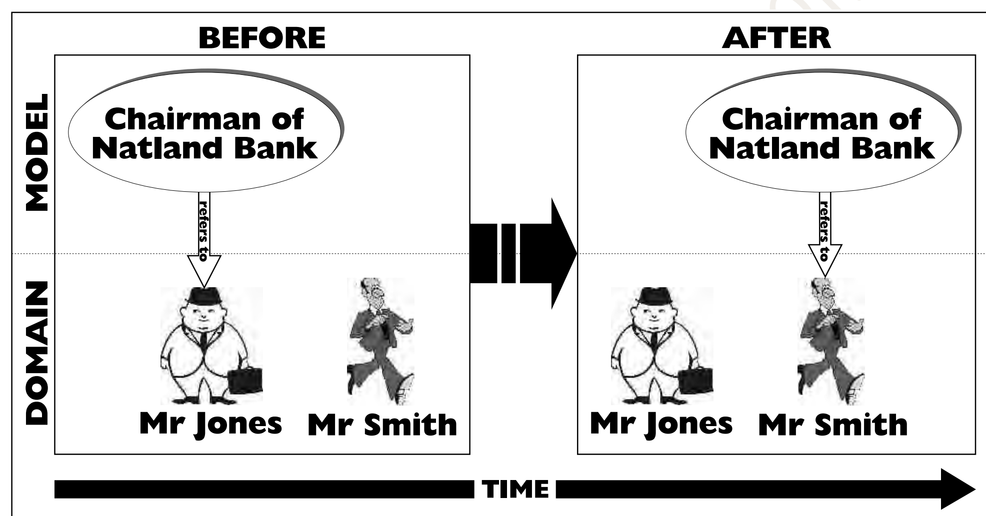
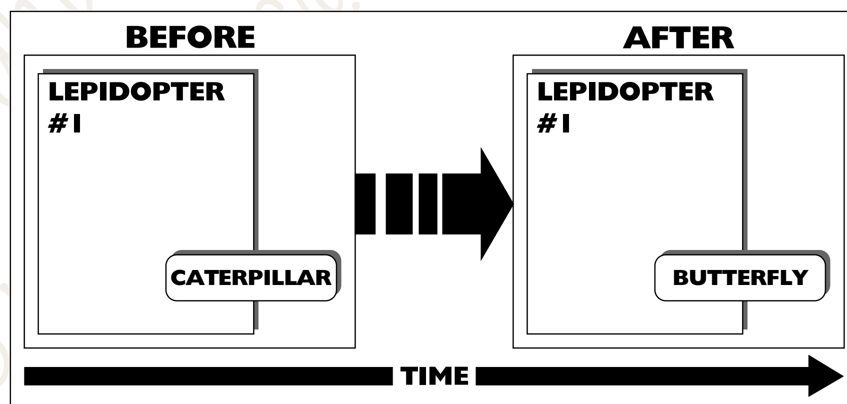


Figure 7.3: A lepidopter substance—caterpillar and butterfly attributes



2.4 Aristotle’s explanation of the experiments

These questions about how sameness over time works are ancient. In ***Chapter 4***, we saw how Aristotle developed his notion of substance in a way that, as far as he was

concerned, gave a clear explanation. We illustrated this with the example of the stages in a lepidopter's life. In the substance paradigm, the caterpillar and butterfly stages of the lepidopter were the same because they had the same substance. The reason they looked and felt so different is that they had different attributes (shown in ***Figure 7.3***). For Aristotle, one of substance's main purposes was to explain sameness over time. When logical semantics replaced the notion of primary substance, it could no longer use Aristotle's explanation.

Faced with the three thought experiments, Aristotle could use substance to give clear answers. In the wrecked car experiment, he would say that the two cars were the same because, like the lepidopter, they had the same substance.

In the car-minus thought experiment, Aristotle would say that my car and car-minus (if he accepted that car-minus existed) are different substances. He would explain what happened when I took the back seats out of my car as two substances starting to share the same place (shown in ***Figure 7.4***) where place is the Aristotelian equivalent of extension.

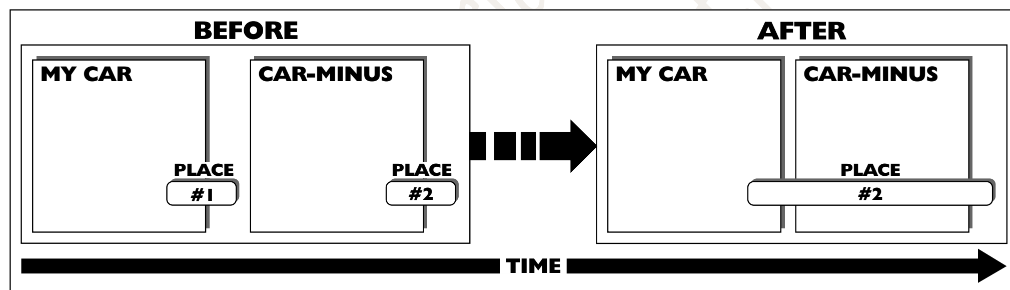


Figure 7.4: My car and car-minus sharing a place attribute

Aristotle would have offered a similar explanation to the Chairman of NatLand Bank thought experiment. He would have suggested that there is a third substance, the Chairman of NatLand Bank, in addition to the Mr. Jones and Mr. Smith substances. This shares a place attribute with the other two substances at different times (shown in ***Figure 7.5***).

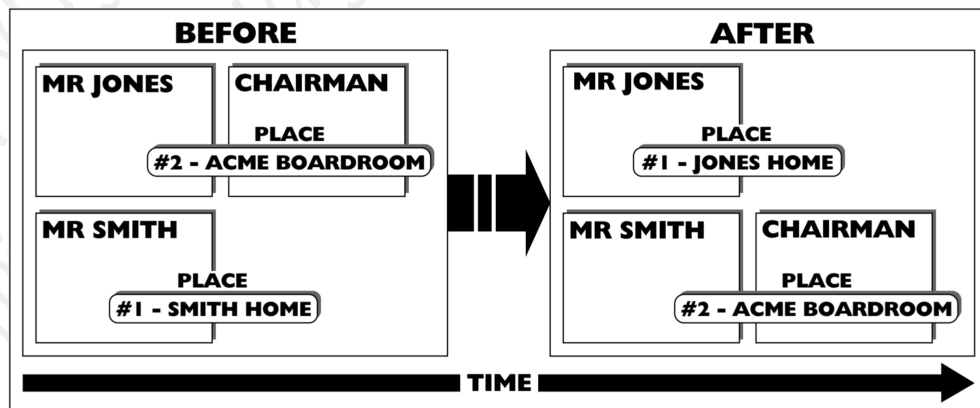


Figure 7.5: Chairmen sharing place predicates

© Copyright Chris Partridge ... borotre.com

3 The shift to object semantics

Aristotle's explanations of the three thought experiments show the benefits, when capturing change patterns, of having place (his version of extension) as an attribute category rather than a foundation for physical bodies. However, we should not be tempted to retreat back to substance. Object semantics' shift to a new particle for physical bodies provides us with a much better tool for capturing these change patterns.

3.1 The origins of object semantics

We start the re-engineering by looking at its origins in a new way of seeing developed in physics. The role model for this way of seeing was Albert Einstein's amalgamation of space and time to space-time in his theory of relativity. This is not normally applied in everyday life, to ordinary people-sized objects. However, a number of people (including one of today's leading philosophers, Willard Van Orman Quine) saw how Einstein's notion of space-time can be used to resolve the problems of identity of people-sized physical bodies. As Quine says:

> Our ordinary language shows a tiresome bias in its treatment of time.

This 'tiresome bias' is treating time as something completely separate from space. His answer is to follow Einstein and treat time as another dimension on a par with space's three. One of the benefits of this shift was that the patterns for space and time were amalgamated into general patterns for space-time. As we shall see, this proved particularly fruitful for whole–part patterns. The key shift we focus on here is in the central notion of extension; from space-based and three-dimensional to space-time based and four-dimensional.

3.2 Explaining our intuitions

This shift takes some getting used to. However, once we do, it seems natural; it appears to be an explicit explanation of what we have already grasped intuitively. It certainly fits in neatly with many of our intuitions. For instance, it not only respects our intuition that continuity is a key factor in identity, but explains why by giving continuity a physical embodiment.

3.2.1 A four-dimensional lepidopter

We can see how this works with an example. In logical semantics, individual physical bodies are extensions and extension is three-dimensional; height, width and depth. A lepidopter in a caterpillar state is a physical body with a three-dimensional extension. Similarly when a caterpillar metamorphoses into a butterfly, it is also an object and so is also a three-dimensional extension, but a different one. What makes these two extensions the same object is that there is a continuous link of time-stages between the caterpillar and the butterfly.

©Copyright Chris Partridge
chris.partridge@BoroCentre.com

The new object semantics follows Quine and Einstein's lead and assumes extension is four dimensional: space's three dimensions and the time dimension. In it, the lepidopter is one four-dimensional object. The two, three-dimensional objects from logical semantics are now just slices in time of the new four-dimensional object as illustrated in ***Figure 7.6***.

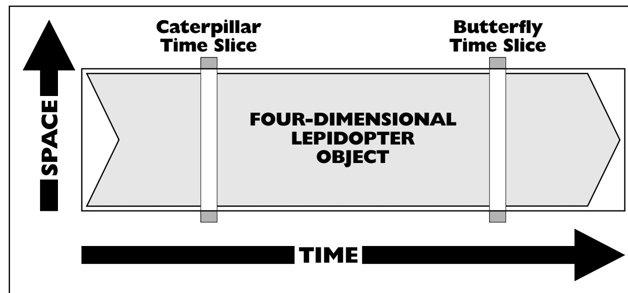


Figure 7.6: A four-dimensional lepidopter

In this four-dimensional way of looking at things, continuity across time is now the same as continuity across space. We can look up and down a lepidopter in space or backward and forward in time along it. The continuity in time that we merely intuitively grasped before is now transformed into something as physical and tangible as continuity across space.

3.2.2 The general trend away from egocentricity

We can see this shift to space-time is part of a general trend—a trend from an egocentric to a more 'objective' view of the universe. When children are young, they see the world revolving around themselves. As they grow up, they begin to realise it does not. In some ways, adults still retain an element of that egocentric attitude. For instance, most of us half believe in a variation of Murphy's law—that things happen when they are most inconvenient for us. So, we half suspect that it is raining because we forgot our umbrella or because we were going out to play tennis. Whereas, 'objectively' we know that the weather is not influenced by our future plans.

We can see a similar egocentric attitude in the earth-centred theory of early astronomy. This assumed that because people see the earth standing still and things moving in the skies, the earth must be standing still and the planets and stars moving. When, in the 15th century, Copernicus suggested the earth was just another planet moving around the sun, he was suggesting a less egocentric view of the cosmos. One in which humans lost their special position at the centre of the universe.

Copernicus' theory overturned an egocentric view of space. Quine and Albert Einstein's theories take this one step further and overthrow an egocentric view of time. We assume that, because we are at a particular point in time, our position must be special. This is egocentric. Why should the point in time that we inhabit (called the present) have a special quality? Most people do not think, when walking down a path, that that point on the path is special because they are there. The present is much like any other point in time. In fact, all points in time have been, or will be, at some time the present.

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

We tend to think of space and time as different because our experiences of them are so different. But we are interested in the things in themselves, not how we experience them. Just because something looks different, does not mean it is different. This is particularly clear when we can use two senses to 'perceive' the same type of object. When we touch one banana and taste another we get very different feelings; but we have no problems in recognising them as belonging to the same type of thing. We should think about perceiving time and space in the same way; that we are perceiving the same type of thing with different senses.

3.2.3 How business models have anticipated this shift

In one way, business modellers have already recognised the similarity of the space and time dimensions. They intuitively and instinctively translate the time dimension into a spatial dimension in their models.

We can see this by contrasting a business model with an engineer's working model of a steam engine. We expect the engineer's model to have pistons that move up and down when fuel is burnt in its combustion chamber. We judge the model by how accurately its movements reflect the movements of a real steam engine. If it did not move, we would say that it did not 'work'.

What is interesting is that a business model does not 'work' in the same way. Unlike the engineer's model, it is not expected to reflect changes in the business by moving or changing. Instead, it models one process following another in time as one process following another across a piece of paper. The changes in time are modelled by shapes in space.

Business modellers compact the four spatio-temporal dimensions onto a two-dimensional piece of paper; time is translated into space. This is analogous to the way an architect describes the three spatial dimensions of a building on a two-dimensional piece of paper—compacting three spatial dimensions into two.

There is an ancient precedent for this interchanging of time and space, one that we are all familiar with—writing. Its characters use space to describe the way speech's sounds change over time. They use a spatial dimension to represent speech's time dimension.

3.3 Re-interpreting the thought experiments

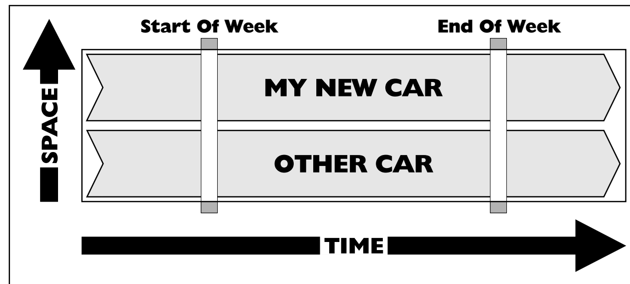
We now examine how object semantics resolves the problem of physical bodies' identity over time by looking at the earlier thought experiments through four-dimensional eyes. We need to be able to draw the four-dimensional objects that this reveals in some way. We cannot draw four-dimensions. But if we use the business modeller's technique of compressing dimensions to fit four dimensions into two, then we can draw a diagram called a space-time map. In it time, the most important dimension for us here, goes across the page and the three dimensions of space are condensed into one that goes up the page. (***Figure 7.6's*** four-dimensional lepidopter is an example.) Those people familiar with state-transition diagrams can see this as, in some ways, a simple O-O version.

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

3.3.1 *Wrecked car thought experiment*

We look at the wrecked car thought experiment first. When we did this experiment earlier, we gave a reason for seeing the two time-stages as stages of the same thing. We could trace a continuous link from the first time-stage to second. There was a continuous link between my car in its original new state at the beginning of the week, through all its mishaps during the week, to the battered wreck at the end of the week.

Figure 7.7: Space-time map of my new car



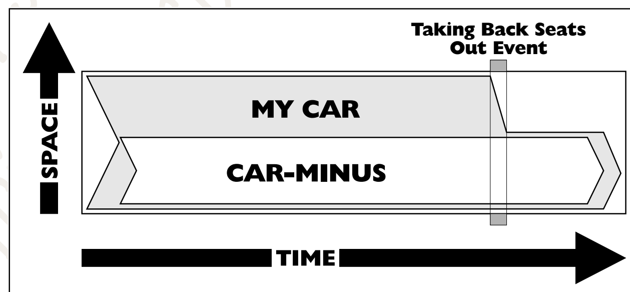
We now have a different way of interpreting this explanation based upon my car as a four-dimensional space-time object. The time-stage that was a brand new car at the beginning of the week is a part of the space-time object; the wreck at the end of the week is another part. The two time-stages are now slices in time of the new spatio-temporal physical body, as shown in ***Figure 7.7***.

This explains quite clearly why we regard the 'continuous link through time' as far more important than an object's look and feel. It is no longer a link through time but a line along the time dimension of a four-dimensional physical body. This four-dimensionality provides a simple and tangible explanation for a physical body's identity over time.

3.3.2 *Car-minus thought experiment*

Object semantics also leads to a consistent re-interpretation of the car-minus thought experiment. In the original experiment, we had the odd situation of my car and car-minus starting off as different objects and ending up as the same object. Look at the space-time map of the objects in ***Figure 7.8***.

Figure 7.8: Space-time map of my car and car-minus



We now see my car and car-minus as different objects, irrespective of time, because they occupy different bits of four-dimensional space-time. There is no temptation to see their sameness change over time. The parts of these two objects that fall inside the slice

of space-time called today, occupy the same bit of space (and small slice of time). But because these objects are now four-dimensional, we see this as the result of two four-dimensional objects with overlapping extension rather than as two three-dimensional objects occupying the same extension. The object paradigm gives us a consistent, simpler and more sophisticated notion of sameness. It enables us to coherently make distinctions that are impossible in the logical paradigm.

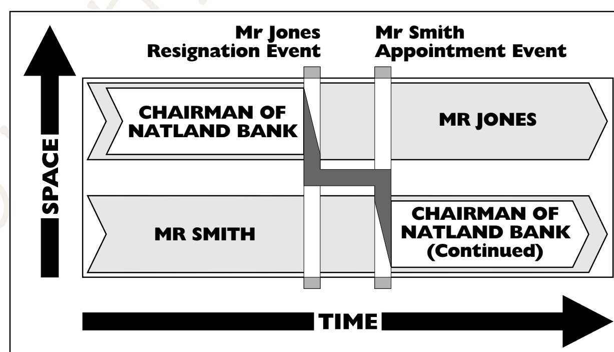
You may have noticed that we are treating temporal (time) parts in the same way as we treat spatial parts. We are re-using the patterns we have established for spatial parts on temporal parts. The steering wheel, gear stick and dashboard are all spatial parts of my car. My car today is a temporal part of my car. Car-minus is a spatio-temporal part of my car. Because time and space dimensions are on a par, all these varieties of car parts are regarded as the same type of thing—spatio-temporal parts of my car. We naturally extend the spatial whole–part patterns to spatio-temporal whole–parts. We shall see later on in this chapter (and in the worked examples in Part Six) that when we use object semantics; these more general, more powerful, patterns for whole–part crop up frequently. In fact, the next thought experiment uses them.

3.3.3 *Chairman thought experiment*

We now re-interpret the Chairman of NatLand Bank thought experiment into the new object semantics. Under logical semantics, the chairman seemed to be a physical body that changed sameness. It was the same as Mr. Jones at one time and then, later on, the same as Mr. Smith.

Now look at ***Figure 7.9***; it contains a space-time map of the four-dimensional chairman object. In this map, Mr. Smith and Mr. Jones have simple straight time-lines. The Chairman of NatLand Bank object is less simple. It is composed of temporal parts of Mr. Smith and Mr. Jones. A time-slice of Mr. Smith's time-line is his chairmanship. Similarly, a time-slice of Mr. Jones' time-line is his chairmanship. The fusion of these two chairmanships, and all the other chairmanships, is the Chairman of NatLand Bank object. This object does not change sameness. It is never the same object as Mr. Smith and Mr. Jones. What logical semantics interpreted as sameness is now re-interpreted as overlapping.

Figure 7.9: Space-time map of the Chairman of NatLand Bank



At first sight, this notion of a four-dimensional, multi-time-slice, chairman seems odd. It is made out of pieces of other objects and, a major sticking point, it is not continuous.

© Copyright Chris Partridge chris.partridge@BOROCentre.com

For example, there is a discontinuity between Mr. Jones' resignation and Mr. Smith's appointment. We intuitively expect physical bodies to be continuous over time. However, a radical paradigm shift, such as the shift to object semantics, is bound to lead to what seem initially like counterintuitive situations.

Object semantics provides a simple and powerful explanation of how Mr. Smith and then Mr. Jones 'are' Chairman of NatLand Bank without being the same object as the Chairman of NatLand Bank. We now see that they share temporal parts (slices of their time-lines), but do not have the same overall four-dimensional extension (and, so are not the same physical body).

We originally introduced this experiment with the ancient question—can two things be in the same place at once? Within logical semantics, there are reasons for wanting to answer both yes and no. Now, after the shift to object semantics, we can see why the question is ambiguous and that we really need to divide it into two separate questions. First, can two physical bodies overlap completely for a period of time? The answer to this is obviously yes. Examples are car-minus and the Chairman of NatLand Bank. Then second, can two physical bodies overlap completely? In other words, can they have the same four-dimensional extension? The answer to this is no. If they do then they must be the same object, the same physical body.

3.4 Characteristics of object semantics

These thought experiments provide our first sight of two important characteristics of object semantics, ones that we will meet again and again. These are:

- Timelessness, and
- Whole–part patterns.

3.4.1 Timelessness

We are accustomed to using one vocabulary and set of patterns for time and another, different, set for space. However, within object semantics, there is one general set of patterns for space-time. The four-dimensional perspective of these new patterns leads to one very important difference; we talk about (and see) objects in a 'timeless' way. We no longer say (using the thought experiments above) that car and car-minus occupied different extensions last week and the same extensions now. We now say that the four-dimensional car and car-minus objects share temporal parts. Similarly, we now say that a temporal part of Mr. Jones is also a temporal part of the Chairman of NatLand Bank. This new way of talking (and seeing) normally takes a while to become used to (as theoretical physicists who work with space-time in Einstein's theory of relativity will know). Making the change to this new perspective involves overriding some deeply embedded mental habits.

3.4.2 Timelessness and individual object identity

Reference and extension fit naturally into this space-time world. The breaches of the strong reference principle, which we discussed at the end of the previous chapter, dis-

© Copyright Chris Partridge chris.partridge@BOROCentre.com

appear. Reference and extension no longer vary to explain sameness over time for physical bodies—they are timeless. Reference is now unchanging, fixed forever to a timeless four-dimensional extension. And, as the extension is the object, sameness is no longer mysterious. It is being the same four-dimensional extension.

3.4.3 Re-using the spatial whole–part patterns in space-time

This shift to four-dimensional objects also enhances the power of the whole–part pattern. Shifting from three to four-dimensional extension extends the range of the whole–part pattern. Furthermore, the physical explanation of whole–part in terms of the extension of the whole containing the extension of the part is extended from spatial whole–parts to temporal and spatio-temporal whole–parts. The earlier thought experiments with their temporal whole–part patterns (such as Mr. Smith sharing a temporal–part with the Chairman of NatLand Bank) give us some idea of how useful this is.

Currently, most people do not see things in terms of spatio-temporal parts. If we are to feel comfortable working with object semantics, however, we need to. For instance, if my car was red last week and green this week, then we need to start instinctively seeing a red temporal part (stage) followed by a green temporal part (stage), where the temporal parts are time-slices of the whole car.

Becoming used to a pattern of temporal parts is not as hard as it might be, because the patterns for temporal parts are not really new. They are based on the familiar spatial whole–part pattern. The amalgamation of space and time into space-time means temporal parts now work under the same group of patterns as spatial parts. In other words, the patterns for spatial whole–part are now generalised to also cover temporal whole–parts—and spatio-temporal whole–parts. We are used to seeing things as spatial parts. For instance, we have no trouble recognising that a steering wheel is a (spatial) part of the car. All we need to do is learn to re-use these patterns on temporal and spatio-temporal whole–parts. Part Six contains useful worked examples of how they should be re-used.

4 Physical stuff objects

The object semantics for physical bodies does more than explain their identity through change. Through the use of the powerful whole–part patterns we have just been discussing, it also transforms some of our current notions. Here we look at one example; how it transforms our current abstract everyday notion of stuff into a down-to-earth physical body. (This explanation is drawn from the work of W.V.O. Quine. He and other philosophers have been using it for decades.)

People normally associate stuff with things. The patterns for the two are ancient and typically contrasted. For example: in standard grammar, nouns are divided into count and mass nouns. A mass noun, such as water, refers to stuff and a count noun, such as car, refers to a thing. The philosopher David Lewis has referred to the difference more light-heartedly as the hunk/gunk distinction.

What distinguishes things from stuff, hunks from gunk? A key difference seems to be that things are individuals; they stand by themselves. Whereas, stuff is more collective. If we put two bits of stuff together, then we have one bigger bit of stuff. If we divide a bit of stuff in two, then we have two smaller bits of stuff. Whereas, if we divide a thing, such as a car, in two, then all we get is two worthless pieces of junk.

The semantic problem that we set out to resolve is why a general stuff, such as milk in general, appears to be an abstract notion. Particularly, when bits of stuff are tangible and have extension. We see how object semantics' four-dimensional perspective gives general stuff a tangible physical basis.

4.1 Applying the strong reference principle to stuff

We start by applying the strong reference principle. We ask:

> What kind of object is a general stuff?

We focus in on one type of general stuff and ask:

> What kind of object is general milk stuff?

We have an idea of the kind of answer we are looking for. According to object semantics, general milk stuff should be a four-dimensional extension (or, maybe, a collection of extensions). The problem is —which four-dimensional extension? A glass or a jug can both contain milk. They are different bits of milk, but both bits are still the same stuff, milk. We are looking for a four-dimensional extension that can explain this.

4.2 Disconnected objects

Before we can see the answer, we need to develop a more sophisticated notion of what a physical body is. We started with the simple notion of it as necessarily connected in space and continuous in time (or in four-dimensional terms, connected in space-time). This was the point of the wrecked car example—the brand new car was connected through both space and time to the wrecked car—there were no gaps, no discontinuities.

This connectedness helps us recognise simple individual physical objects; it is a vital part of our early understanding. But as our world grows more sophisticated, it becomes a liability if taken as a fixed rule. We want to be able to have individual physical bodies that are not connected. For example, a United States of America, that has as a physically disconnected part, Alaska.

We saw another simple example of disconnectedness in the Chairman of NatLand Bank above. The chairman physical body object was not connected in space-time—there is a temporal gap between Mr. Jones' resignation and Mr. Smith's appointment. So individual physical objects are not necessarily always spatio-temporally connected. But the spatio-temporal gap in the chairman was small and temporal. We need to be able to tolerate wider, more substantial gaps in both time and space before we can see what milk, water and other general stuffs are.

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

4.3 Overall stuff

This is because milk, water, and so on are very disconnected objects. Milk, for instance, is the fusion of all the bits of milk in the world. (The fusion of two or more objects is the sum of their extensions, another extension, another physical body.) If there is a glass of milk on the table and a jug of milk in the fridge, then the fusion of these two is one physical body with part of its extension on the table and part in the fridge. The general stuff object, milk, contains all the bits of milk here and on the other side of the globe. It contains those that have been and those that will be; it stretches both back and forward in time. It is the fusion of innumerable bits of milk and so is incredibly disconnected. This is completely unlike connected physical bodies such as the one we started with, my car. I call this general stuff object an overall stuff object.

There is only one overall milk object; one overall water object; one overall brass object. Each of them are overall stuff objects. We talk about something being stuff, if it is part of the overall stuff object. So the milk stuff in a glass of milk is a part of the overall milk stuff object. And a brass statue is a (spatio-temporal) part of the overall brass stuff object.

Here, object semantics has given us a simple explanation of what stuff is. It may seem radically different from our intuitions, and in one sense it is. But it still accords with the way we talk about stuff. This notion of physical bodies of overall stuff gives an important role to the whole–part pattern. It is used to help define what counts as stuff; being stuff is being *part of* an overall stuff. So the water in my glass is water stuff because it is *part of* overall water stuff. In general, the shift to four-dimensional extension leads to an increase in the range of the whole–part pattern. We will find that it can be used to explain a number of different, previously unrelated, patterns.

5 Classes of four-dimensional objects

Looking at these patterns for physical bodies has reinforced our understanding of what one is in object semantics. This will help us understand how to re-interpret the patterns for the other types of logical object; classes and tuples. Classes are, in logical semantics, collections of objects. Some classes are constructed from collections of physical bodies. The shift to object semantics for physical bodies affects how we see these classes. It also resolves a problem that logical semantics' classes suffer from—the familiar problem of identity over time. It will help us to understand object semantics' notion of class, if we see how its resolves logical semantics' problem.

5.1 Logical semantics' problem with a class's identity over time

Once we understand what a class is, we instinctively see individual physical objects belonging to classes. For instance, we see a car as a member of the class cars; a person as a member of the class persons. This presumes that the class cars and persons are well-understood objects. However, in logical semantics, this presumption is not warranted. There is a gap between what the semantics says is an individual physical body and what our intuitions about class says a member is.

© Copyright Chris Partridge chris.partridge@BORO.Centre.com

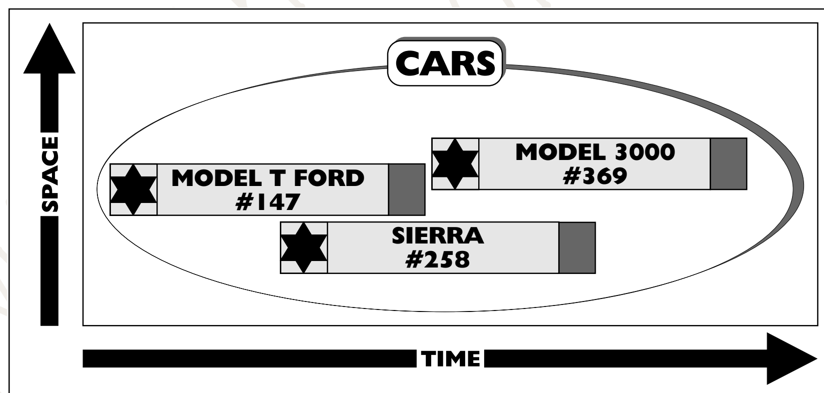
A class is a collection of objects. For example, the class cars is a collection of car objects. Car objects are physical bodies and so extensions. In logical semantics, a car object is a three-dimensional extension now and was another different three-dimensional extension yesterday and these are somehow the same car. What then is the class cars? It must be a collection of three-dimensional car extensions, but which collection? Is it all the extensions, historic and present, or only the present extensions? If we follow the lead of physical bodies and select only the present extensions, then a class (like a physical body) is continuously re-created. At each new moment of time, a new class with new members is re-created. We then have a problem explaining in what way these different classes with different members are the same. In logical semantics, classes share the same mysterious sameness over time as physical bodies.

5.2 Object semantics' view of a class's identity over time

In object semantics, we do not face this problem. A car object is a timeless four-dimensional extension. The class cars is the collection of these objects. Because they are timeless, it is timeless. This fits in well with our instinctive notion of what a class should be.

The cars class (like all classes) is timeless, it does not change. An object either is, or is not, a car (in other words, a member of the class cars)—time does not come into it. This applies to all objects wherever or whenever they exist. It includes the full four-dimensional extension of the first Model T Ford as well as all cars produced in the year 3000 AD, if there are any (illustrated in ***Figure 7.10***).

Figure 7.10: Class of spatio-temporal extensions



This object shift does for classes what it did for individual objects. It fixes the reference of classes to a single extension (in this case a collection of four-dimensional extensions). This also clears up the explanation of a class's identity. Two classes are the same if, and only if, they have the same extension (in other words, if they have the same collection of extensions). We no longer have to explain how the 'same' class has a different extension at different times.

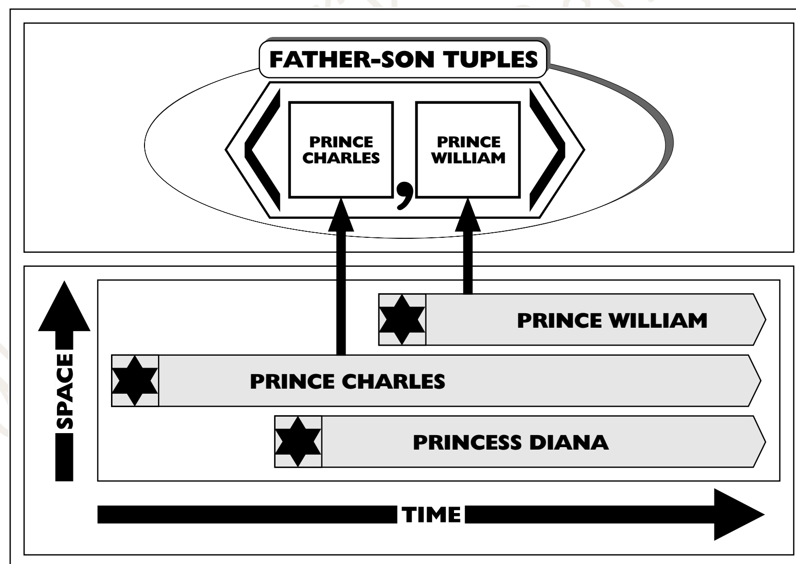
6 Tuples of four-dimensional objects

Tuples like classes are often constructed out of physical bodies. These tuples, like classes, have an extension that depends on the extensions of the physical bodies from which they are constructed. This means that within the logical paradigm, they also suffer from the problem of identity over time.

We can see this by looking again at the example we originally used to explain tuples in logical semantics. Consider the couple <Prince Charles, Prince William> which belongs to the father–son tuples class. It is constructed out of the physical objects Prince Charles and Prince William. If, at different times, these have different three-dimensional extensions (as they do in logical semantics), then the couple must also have different extensions at different times.

As with classes, this problem disappears after the shift to four-dimensional extensions. Then the tuple's places are filled with timeless four-dimensional extensions, as illustrated in ***Figure 7.11***. When the physical bodies are given a more solid foundation, then the objects constructed out of them (such as tuples and classes) also share in it.

Figure 7.11: Tuples with four-dimensional places



7 A new way of seeing bodies—a key type of thing

The shift to four-dimensional extension gives us a radically different and better foundation for the area of semantics we are looking at now—bodies. It also—as the previous section explained—gives us a more solid foundation for both classes and tuples of physical bodies. However, it involves a radically new and different way of seeing things, one that is much newer than the logical paradigm and so has had much less time to make its way into the general consciousness.

At least with the logical paradigm, the new way of seeing things has worked its way into our language. For example, we have words for ‘part of’ and ‘member of’, even if we do not use them as accurately as the logical paradigm demands. Whereas, we have no obvious words to describe overlapping four-dimensional objects such as Mr. Smith and the chairman. We have to describe them in a roundabout way—saying ‘Mr. Smith is *currently* chairman’. Not many people would understand what we meant if we said ‘the Mr. Smith and chairman objects currently overlap’.

Intriguingly, the shift to object semantics gives us a more accurate way of seeing sameness. Modern western civilisation has a more accurate way of seeing sameness than oral cultures such as the Huichol Indians (I described this in ***Chapter 4***). We find it difficult to understand why the Huichol Indians say that corn and deer are the ‘same’. Now the boot is on the other foot. Someone steeped in the object paradigm finds it difficult to understand why modern westerners say that Mr. Smith and the Chairman of NatLand Bank are the ‘same’. The object paradigm has developed a more accurate notion of sameness that renders this way of speaking obsolete.

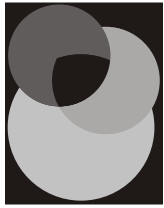
From language’s point of view the shift to the object paradigm involves immense changes. In language, time is currently described using tense. If language is to reflect the object paradigm’s amalgamation of time and space, we need to develop a tenseless language that describes space-time. This would be a substantial change.

8 Summary

This chapter has given you an insight into how object semantics works. It has shown how logical semantics’ time-bound notion of physical bodies as three-dimensional extension can be re-engineered into a notion of timeless four-dimensional extension. We have seen how this new notion resolves logical semantics’ problem with explaining the nature of identity ‘over time’ for physical bodies, classes and tuples.

However, physical bodies are only the first of the two semantic areas unresolved by the logical semantics that we identified at the beginning of this chapter; the second area is changes. In the following chapter, we see how object semantics uses spatio-temporal extension to explain why changes are objects. When this is done, we will have finished examining the semantic framework of the object paradigm.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 8

Changes as Three-Dimensional Objects

1 Introduction
2 States as physical body objects
3 Events – a new kind of physical object
4 The time-based 'consciousness' of information systems
5 A new way of seeing changes—a key type of thing
6 What's next

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction

In ***Chapters 5*** and ***6***, we found the logical paradigm was not able to consistently explain one of the key types of things; changes. In the last chapter, we focused our attention on object semantics' consistent explanation of physical bodies persisting through change. In this chapter, we shift our focus to its radically different, much improved and consistent explanation of changes. It uses two very different types of change objects to do this:

- States, and
- Events.

States are types of physical bodies, much like the physical bodies of which they are states. Events, on the other hand, are a new type of individual physical object. The patterns of connections between these two types of objects both explain and transform our current notions of change. The introduction of the new physical event objects extends the structure of the paradigm (illustrated in ***Figure 8.1***). This only affects the individual objects level. The structure at the constructed objects level, which contains classes and tuples, is unaffected.

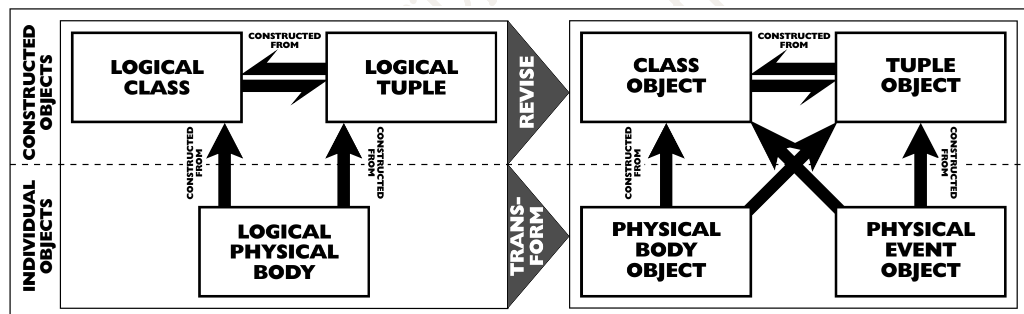


Figure 8.1: Structural extension in the shift to objects

2 States as physical body objects

We start by looking at object semantics' explanation of states. The substance paradigm had a clear vision of what a state is; so, we use it as our starting point. We look at what it describes as states and then use object semantics to transform these into objects.

We then look at some of states' common patterns. We start with the common and intuitive state–sub-state and state–sub-class patterns. We then get a feel for state's counter-intuitive nature by looking at two odd patterns; components as fusions of states and objects that are states of themselves.

Changes are patterns in time, and states form patterns in time that reflect the changes. We look at a key element of these patterns, the time ordering of the states.

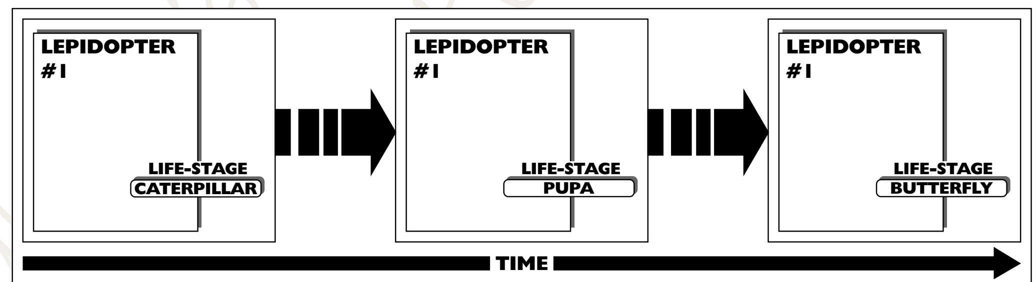
Finally, we look at state tuples. These are re-engineered from changes of a substance paradigm's relational attribute. In object semantics, these attributes have to be re-engineered into tuples of states. They cannot be re-engineered directly into couples, as they were in the logical paradigm.

2.1 Substance paradigm's view of changing states

In ***Chapter 4***, we looked at the substance paradigm's consistent and coherent view of changes (based on the now discredited notion of substance). In it, states were not explicitly particles; but the substance framework gave a clear and accurate explanation of what they are. If you remember, the attributes of a primary substance can be divided into two main types: essential and accidental. Essential attributes cannot change, but accidental attributes can and do change. A state (which comes from the Latin *status*, to stand) is what a substance is in when it possesses a particular accidental attribute. We can see it as the substance for the period of time that the particular accidental attribute belongs to it. Where an attribute can have a range of values and each value corresponds to a state, we sometimes talk of the state of an attribute.

We can use the lepidopter in ***Figure 8.2*** to explain what a state is, and how it relates to an attribute value. In the figure, lepidopter substance #1 starts life with an accidental attribute of caterpillar-ness, which changes to pupa-ness and then butterfly-ness. We see this as a single attribute that changes value; we have called it the life-stage attribute. This has as values; caterpillar, pupa and butterfly. Lepidopter substance #1's changes are then changes in the value of the life-stage attribute. When the life-stage attribute has a particular value, we talk of the substance being in a particular state. When it has the value caterpillar, the substance is in a caterpillar state. It remains in this state as long as the life-stage attribute continues to have a caterpillar value. When the life-stage attribute changes to a pupa value, it moves into a pupa state, and so on.

Figure 8.2: The lepidopter #1's states



2.2 Applying object semantics to changing states

Object semantics provides a radical re-interpretation of this view of states. We have some pretty strong pointers to what this re-interpretation will be. States, in object semantics, have to be objects. As objects they can either be physical bodies, classes or tuples (or some new type of object). Whatever they are, they have to have four-dimensional extension (either directly or as a collection of extensions) and so be time-less; change cannot enter the picture.

2.2.1 Re-interpreting the lepidopter example

With these pointers, physical state objects are not difficult to find. If we look at the space-time map of the lepidopter example (shown in ***Figure 8.3***), the states stick out like sore thumbs. They are temporal parts of lepidopter #1. The caterpillar state #2,

pupa state #3 and butterfly state #4 objects divide lepidopter #1 along the time dimension into three. The three state objects are not a new type of object, but physical bodies, just like lepidopter #1. What makes them a state is that they are part of another physical body—in other words, the whole–part pattern connection with lepidopter #1.

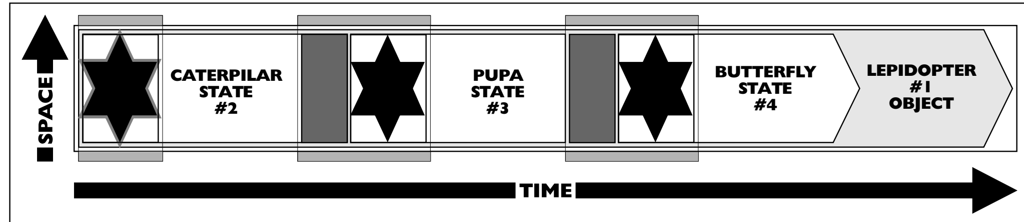


Figure 8.3: Lepidopter #1's state objects

Not every part of a physical body is a state. A state object has to be *all* the spatial extension of an object over a period of time. For example, a butterfly's wings are part of the butterfly, but we do not see them as state objects. Furthermore, if we take a time-slice of a butterfly, but leave out the wings (illustrated in ***Figure 8.4***), then this is also not a state.

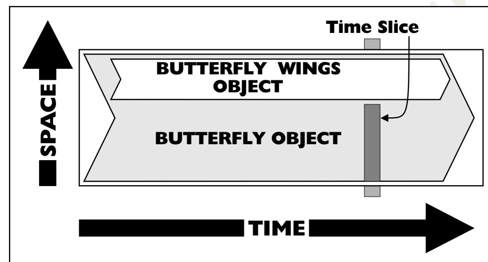


Figure 8.4: A non-state part

Once we realise what a state object is, we begin to see them everywhere. This should not be a surprise. If we look at the population of things in the substance paradigm, states have a big representation. A substance almost always has more than a few attributes. Most of these are accidental, with at least a few states. So, inevitably, states are more numerous than substances or attributes.

2.2.2 State identity

This object-oriented way of looking at states as physical objects gives a more accurate meaning to a state's identity. It provides a clear and simple way of deciding whether two states are the same. We can see this from the following thought experiment. Imagine a young boy with tonsillitis. Assume I meet him twice and on both occasions he is ill with tonsillitis. On the second meeting, I ask his parents:

Is he in the same state as he was when we first met?

His parents need to interpret the question. It might mean:

1. Is the disease as bad as it was last time I saw him?

Or perhaps:

2. Is this the same disease as he had last time?

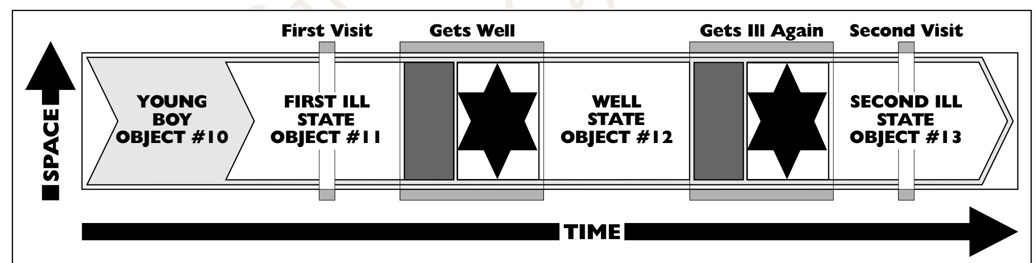
© Copyright Chris Partridge BORO Centre.com

It is unlikely that I am asking whether the diseased state the boy is in now, is the same thing as the diseased state he was in the last time we saw him. Our everyday notion of state is not strong enough to give it an identity.

To see this, imagine that in the period between the two meetings, the boy had recovered from the first bout of tonsillitis and succumbed to a second, and was now as ill as before. Then, if the parents interpreted the question as (1) above, they would answer 'yes'. However, if the parents interpreted the question as (2) above, there are two possible answers. If the same underlying strain of tonsillitis caused the first and second bouts of illness, then they would answer 'yes'. If, on the other hand, there were two different strains, they would answer 'no'. In everyday language, even though my original question appears to be about a physical state object, it is really just a way of speaking.

In the object paradigm, states are objects with an identity. Let's assume that disease state objects are relatively continuous over time. Then, in our example, there are exactly two ill state objects, each with a clearly defined extension. These are objects #11 & #13 in the space-time map of ***Figure 8.5***. With these state objects, we do not need to work out what my original question might 'mean' to decide on an answer. The states are well-defined objects and the answer is unambiguously 'no'. The object paradigm has given us a more accurate notion of sameness for states.

Figure 8.5:
Ill state objects



2.2.3 State hierarchies

In business object models, I have found that state objects often fall into one, other or both of two closely linked hierarchy patterns; the state–sub-state and state–sub-class patterns. We now look at these and see how they are based on two of object semantics' fundamental patterns: the mereological whole–part and logical super–sub-class patterns.

2.2.3.1 State–sub-state pattern

States can themselves have states and this leads to a state–sub-state hierarchy. For example, assume that biologists divide the caterpillar state of the lepidoptera's life-cycle into an early and a late stage. The space-time map in ***Figure 8.6*** shows this division for caterpillar state #2. Notice that the early and late stages are state objects (#'s 5 and 6)—they are temporal slices of caterpillar state #2. This is a state–sub-state hierarchy pattern. It is perhaps easier to see in the hierarchy diagram. of ***Figure 8.7***. You probably recognised that the state–sub-state hierarchy pattern is based on object semantics' strengthened spatio-temporal whole–part pattern.

© Copyright chris.partridge@...

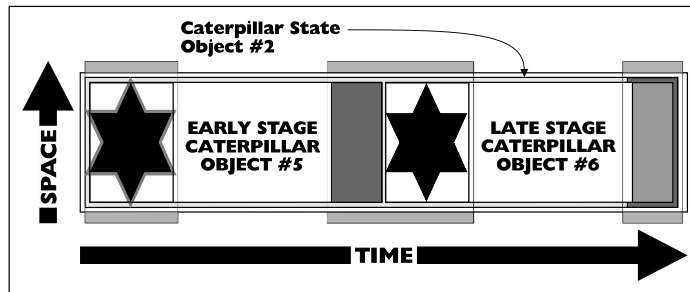


Figure 8.6: The caterpillar state's state objects

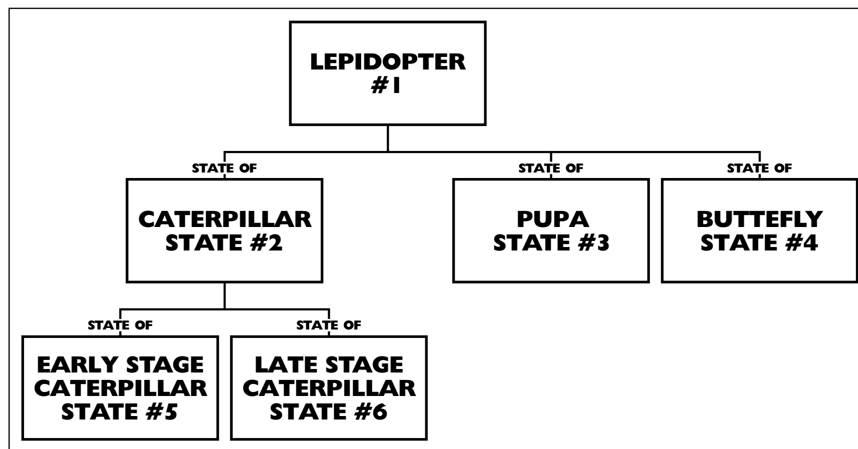


Figure 8.7: State–sub-state hierarchy diagram

2.2.3.2 State–sub-class pattern

The state–sub-state pattern should be distinguished from the closely linked, but different, state–sub-class-pattern. We can use caterpillar states to illustrate this. Assume that biologists classify caterpillar's states by colour. Assume also that there are red and green caterpillars and that they do not change colour. This means that red and green are not states of the caterpillar. So, for example, a red caterpillar state will be the same object as the caterpillar state, and so have the same extension. However, it does lead to a distinction at class level—the class of caterpillars has a red caterpillar and a green caterpillar sub-class. This pattern is shown in the Venn diagram in ***Figure 8.8***.

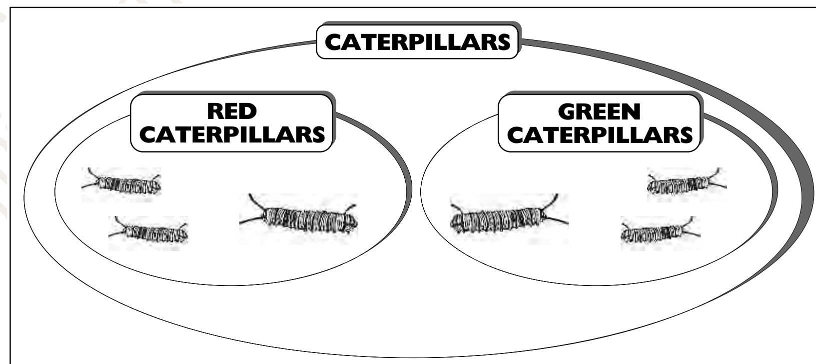
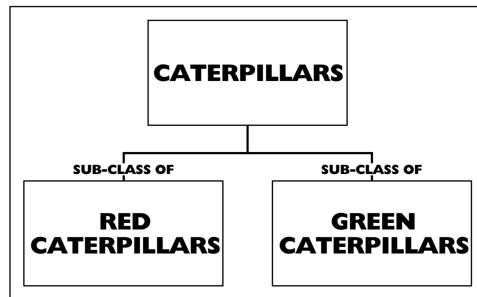


Figure 8.8: The caterpillar (state) class's sub-classes

Notice that these are not, like the early and late stages, sub-states of the caterpillar state, but sub-classes of the caterpillar (state) class. These sub-classes have a state–sub-class hierarchy pattern. This can be seen more clearly in the hierarchy diagram in ***Figure 8.9***. This time the pattern is based upon the super–sub-class pattern, not the whole–part pattern (as with the state–sub-state hierarchy).

Figure 8.9: State–sub-class hierarchy diagram

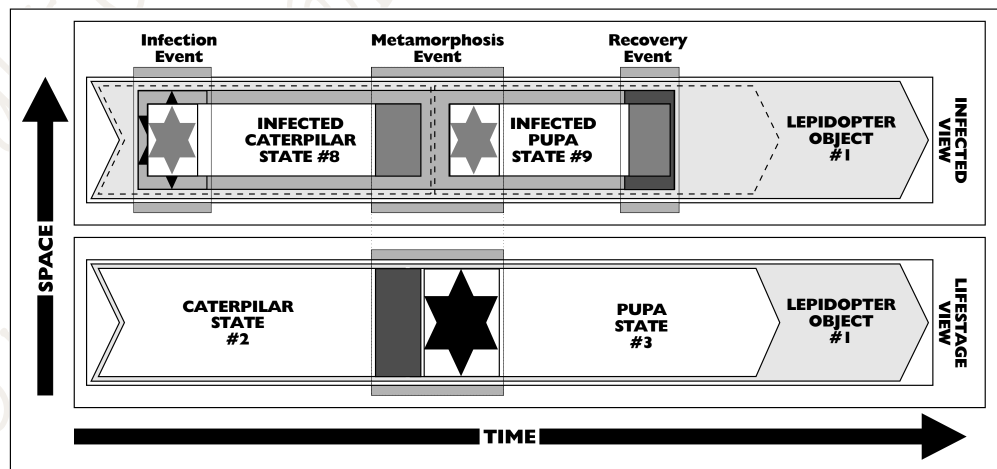


2.2.3.3 Distinct states

In the last two examples, the states we looked at were all distinct; they did not overlap. They were distinct on two levels—the whole–part and the super–sub-class levels. From a whole–part perspective, the four-dimensional extensions of the individual early and late stage caterpillars do not overlap. There is no part of one extension that is also a part of the other's extension. This is plain to see from ***Figure 8.6's*** space-time map.

The red and green caterpillar state classes are also distinct, but from a super–sub-class perspective. No member of the red caterpillar class is also a member of the green caterpillar class, and vice versa. This is plain to see from ***Figure 8.8's*** Venn diagram.

Figure 8.10: Overlapping sub-states space-time map



2.2.3.4 Overlapping states

States, however, do not have to be distinct at either the whole–part or the super–sub-class levels. For example, individual sub-states can overlap; in other words, they can

have parts in common. Take the lepidoptera example again. Consider a lepidopter (#1) that becomes infected while it is a caterpillar (in caterpillar state #2). It is still infected when it metamorphoses into a pupa state (#3). However, it recovers before it turns into a butterfly state. This introduces a new 'infected' state (#7) that overlaps both the caterpillar and pupa states. This means there is an infected caterpillar sub-state #8 and an infected pupa sub-state #9, as illustrated in space-time map in ***Figure 8.10***.

As before, these states form a state–sub-state hierarchy pattern—shown in the hierarchy diagram in ***Figure 8.11***). However, unlike the distinct sub-states that formed a tree hierarchy, these overlapping sub-states form a lattice hierarchy.

Figure 8.11: Overlapping sub-states hierarchy diagram

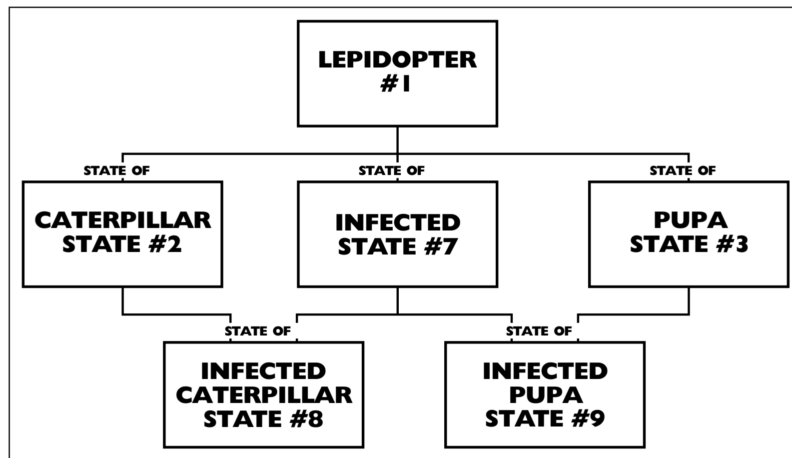
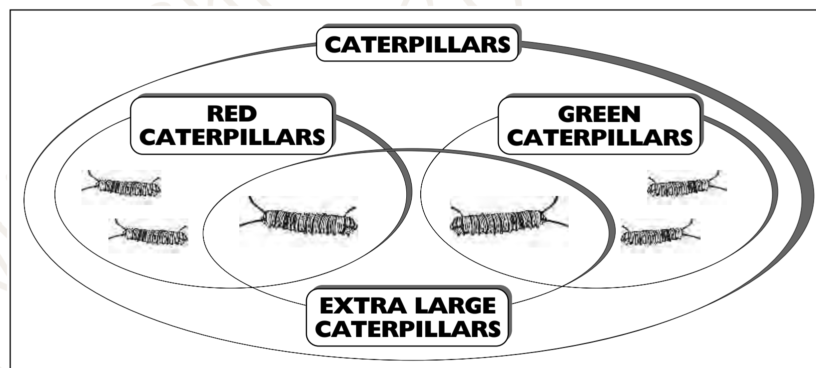


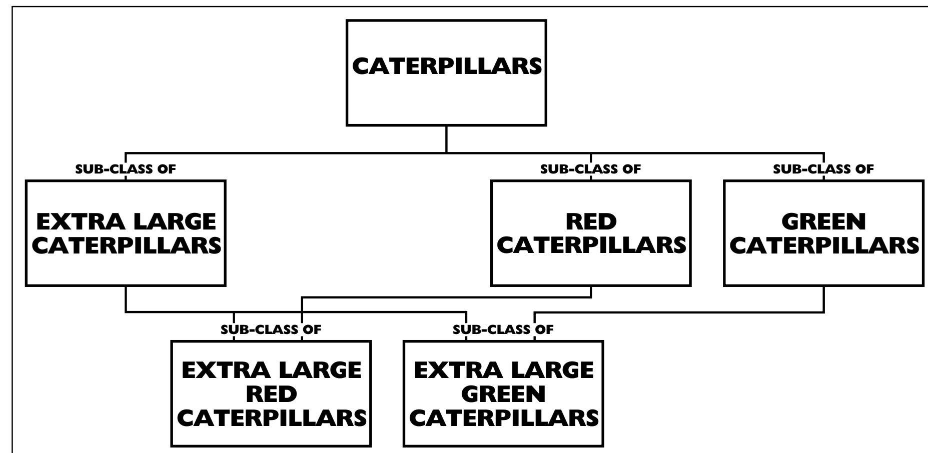
Figure 8.12: Overlapping sub-classes Venn diagram



State–sub-classes can overlap as well. Assume, in the caterpillar example, that biologists also classify some caterpillars as extra-large and that both red and green caterpillars can be so classified. As ***Figure 8.12*** shows, the caterpillar state's sub-classes overlap.

These overlapping state classes form a super–sub-class hierarchy pattern with the extra large red and extra large green caterpillar sub-classes at the bottom. This has a lattice structure (shown in the hierarchy diagram in ***Figure 8.13***).

Figure 8.13: Overlapping sub-classes hierarchy diagram



2.3 Consequences of timeless state objects

The notion of states as objects that are temporal parts of physical bodies leads to a new way of seeing them as physical bodies that do not change. This has some counterintuitive consequences. To get a better understanding of states, we look at two of them:

- Components as fusions of states, and
- Objects that are states of themselves.

2.3.1 Components as fusions of states

It is a truism that a whole is the sum of its parts. So it would seem reasonable to expect a thing to be the sum (the fusion) of all its components. However, object semantics reveals an inherent ambiguity in such everyday talk of components. At any point in time, it seems quite clear what a thing's components are. But it becomes much less clear when we consider different points in time.

Here is an example that illustrates the problem. We expect some of a car's components to change. For instance, it is customary to change a car's tyres when they are worn; it is illegal not to. When we change a car's tyre, it stays the same car. It still has its full complement of components. It is just that one of its components has been changed. But what is this component we are talking about? It is one tyre before the change and another tyre after the change.

Object semantics gives a clearer and more accurate answer. Look at the space-time map of the car object #20 in ***Figure 8.14***. This shows that the four-dimensional extension of the car contains a temporal part of one tyre (#21) followed by the temporal part of another tyre (#22). At any one time, the car overlaps with only one tyre;. but, over time, it overlaps with two tyres. (You may recognise this as a similar pattern to the chairman thought experiment.) The two tyres have state objects that are 'components' of the car.

© Copyright Chris Partridge chris.partridge@BORO Centre.c

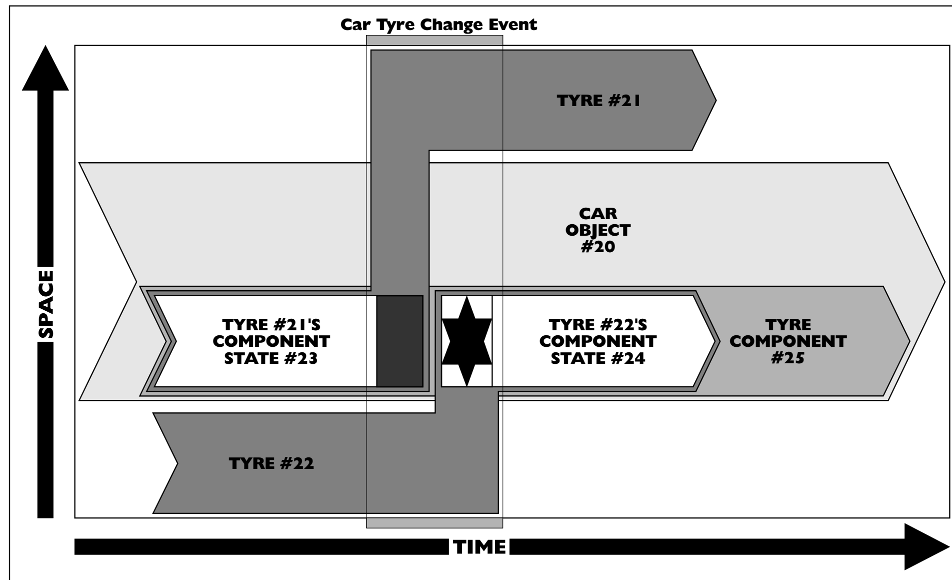


Figure 8.14: Car tyre change space-time map

If this is as far as we go, then the car could be said to have a different component before and after the change. But this is not at all satisfactory, because it would mean that the 'components' change over time—an anathema in our timeless object paradigm. We need a timeless explanation. We get it by constructing a tyre component from the fusion of all the 'component' tyre states. This is shown in the space-time map as the car's tyre component—object #25. It is a component of the car; it is a part of the car; it is a fusion of the tyre state objects (#'s 23 & 24); and most important of all, it is timeless. This more sophisticated object-oriented component has none of the inherent ambiguity of our everyday notion.

2.3.2 *Objects that are states of themselves*

A different and more counterintuitive situation arises for states that do not change—the object appears to be a state of itself. In the substance paradigm, a state existed where there was an attribute that had the potential for change; it need not actually change. If we translate this into object semantics, it means that a physical body can be a state object of itself. We can illustrate how this 'happens', using the notion of the well state of a person.

Consider someone who has been ill and is now well—such as the boy with tonsillitis in the example illustrated in ***Figure 8.5***. He is in a well state that is one of a number of well state objects whose extensions are time-slices of his overall time-line. Now consider a super-fit girl with a superb constitution. Her four-dimensional extension, stretching from birth to death, is a member of the persons class. Assume also that she was permanently in good health (in other words, in a well state) from the day she was born until the day she died. As her well state's time-slice stretches from birth to death, it fills her four-dimensional extension exactly. Extension is the basis of object identity; so it follows that she is her own well state object (see object #32 in ***Figure 8.15***).

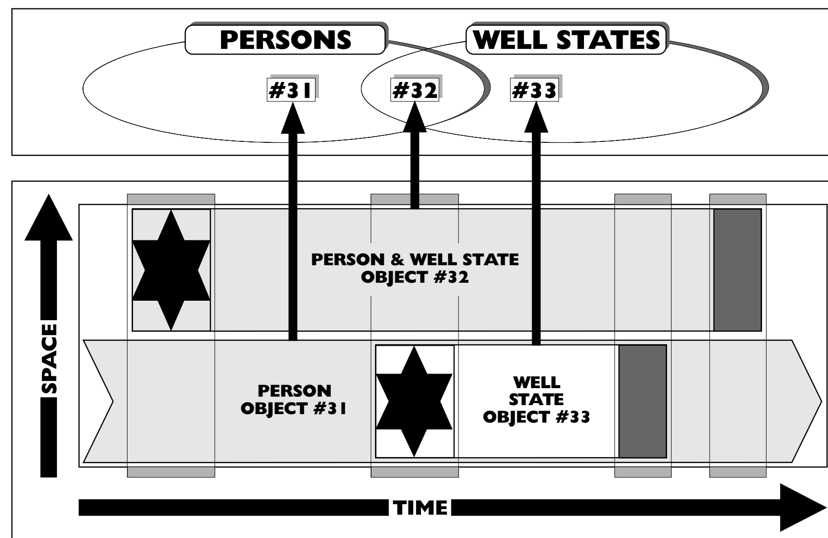


Figure 8.15: Person as a well state

To see what this means, we need to recognise that what makes a physical body a well state object is that it is a member of the class of well state objects. And that what makes it a person object is that it is a member of the class of person objects. So all we are really saying is that the super-fit person is a physical body (in the object paradigm sense) that is a member of both the class of well state objects and the class of person objects. This may feel a bit counterintuitive at first, but it does not lead to any contradictions and it is a necessary result of treating states as four-dimensional objects.

This is a contrived example—used to make a point clearly. A more common example, at least nowadays, is gender. Most people stay the same gender throughout their life. In other words, most women belong to the class female and most men the class male. However, those people who have gender–changes will have a male state belonging to the class male and a female state belonging to the class female. This means the gender classes (male and female) contain both whole person objects and person state objects.

Gender provides a good example of the usefulness of taking a flexible view on whether a class contains individuals or states. There are some species that, unlike us, naturally change gender and some, like the earthworm, that can be both genders at once. For these, gender is naturally a state. If we generalise the gender pattern from humans to animals, it needs to be able to handle these species. If we were flexible about allowing the male and female humans classes to have states as members, then the generalisation is trivial.

2.4 State object's time-ordered connections

Even though time and space share many similar patterns that can be generalised, time has one useful pattern that space does not. It has a well-defined absolute direction—from the past to the future. Space's directions are not so well-defined. The direction that I, in England, call 'up', people on the other side of the world; in Australia, call down; and people in North Africa, halfway around the globe, call along. There is no absolute direc-

tion in any of space's three dimensions. Time's arrow, however, points in only one direction (leaving aside the extreme conditions considered by modern physicists).

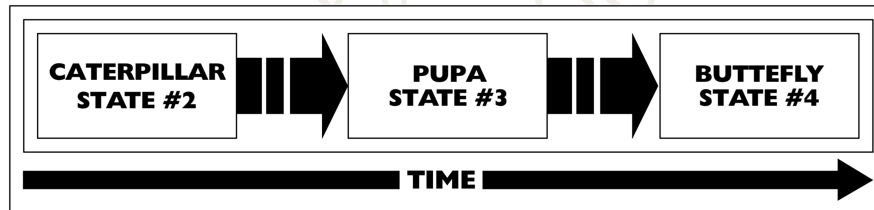
In object semantics, time's absolute direction is used to enhance patterns, originally developed for space, to describe the time dimension in space-time. We now look at how we use these patterns to describe time-ordered connections for state objects.

2.4.1 Sequences of states

A common time-ordered pattern for states is a sequence, where one state naturally follows another. We tend to talk of one state being before another. We have a natural image of something being in a particular state, then something happens and it moves into another state.

However, in object semantics, things do not 'happen'; the world is timeless. So we borrow a pattern from space, generalise it to timeless space-time, and use it to describe these time-ordered happenings. In space, we can put a number of things in a line, and then talk about one object being after another. This same pattern, generalised to space-time, applies to sequences in time of state objects. In the lepidopter example, its three states can be considered as objects following one after the other, in sequence, along the time dimension (illustrated in ***Figure 8.16***).

Figure 8.16: State objects laid out in space-time



2.4.2 Alternating states

Another common pattern is alternating states. We can use the young boy modelled in ***Figure 8.5*** to illustrate it. He alternates from an ill (tonsillitis) state to a well state. To model this using object semantics, we again have to revise our time-oriented everyday way of speaking. As in the last example, we use the 'things in a line' space pattern generalised to space-time. We see these alternating state objects following, one after the other along the line of the time dimension (illustrated in ***Figure 8.17***). This shows quite clearly the state objects alternating between the ill and well state classes

This kind of pattern is common where an object can switch between two states; for example, when a bank balance alternates between being in credit and overdrawn. Or the shelf on a warehouse alternates between holding stock and being empty.

© Copyright chris.partridge@BORO Centre.com

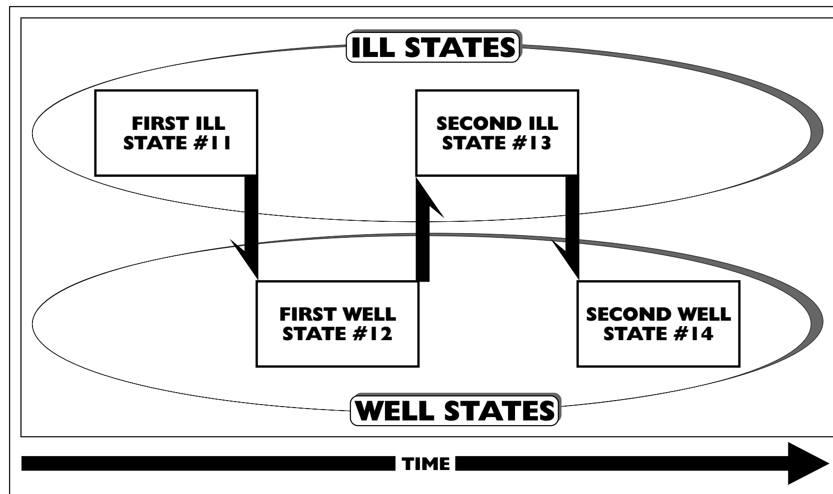


Figure 8.17: State objects alternating between state classes

2.4.3 Contiguous states

We can borrow another distinction from space to describe time patterns such as these—contiguity or, in less technical terms, touching. When a series of objects follow each other in space, each pair of objects can either be touching or have a gap between them. This same spatial pattern occurs along the time dimension in space-time. In everyday language, we say that sequential states either follow each other immediately or after a time. In object semantics' timeless view of things, these state objects are either contiguous (touching) or not.

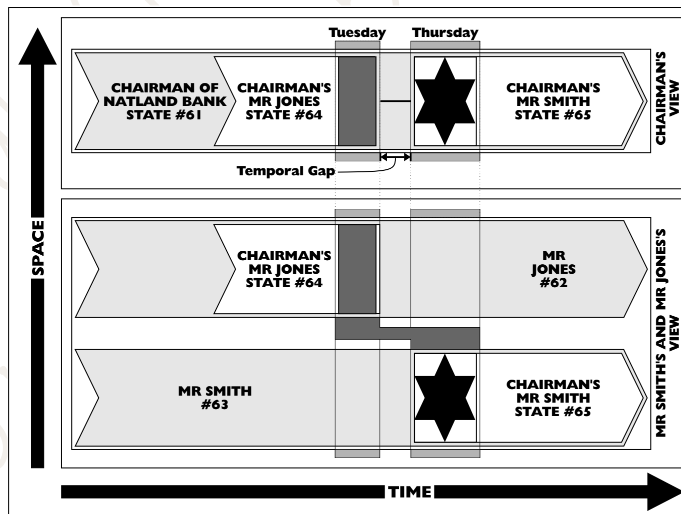


Figure 8.18: Chairman state objects

For example, the lepidopter's state objects in ***Figure 8.16*** are contiguous. There is no time gap between the caterpillar state and the pupa state. Contiguity is common in time-ordered patterns, but by no means universal. The chairman thought experiment, from the previous chapter, provides us with a counter-example. Mr. Jones as chairman and Mr. Smith as chairman are two states of the chairman object (shown in the space-time

© Copyright chris partridge BORO Centre

map in ***Figure 8.18***). However, Mr. Jones resigned as chairman on Tuesday and Mr. Smith was appointed the new chairman on Thursday. So there is a temporal gap between the two states. Because of no intervening chairman state, the same temporal gap exists for the chairman object. It is disconnected with no four-dimension extension between the resignation event on Tuesday and the appointment event on Thursday.

2.5 State tuples—tuples with state object places

So far we have not considered the impact of states on tuples; we do so now. In the substance paradigm, relational attributes were attributes and so could, in principle, change. For example, consider a car owned by a garage. In substance-speak, this is a car substance with an owned by relational attribute. This attribute can change. In fact, as the garage is trying to sell the car, it is likely to change. Assume the garage does sell the car to Ms Brown. The owned by attribute changes; it no longer points to the garage, it points to Ms Brown (illustrated in ***Figure 8.19***).

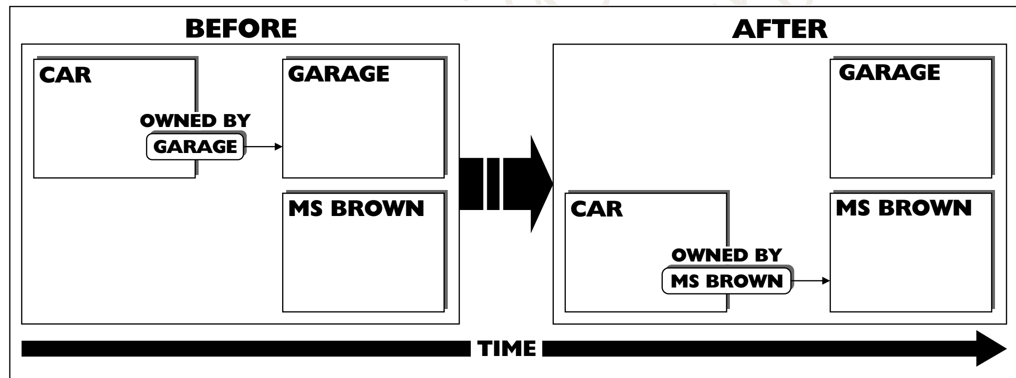


Figure 8.19: Changing car ownership attribute

How do we interpret the ‘owned by’ attribute in object semantics? We cannot simply follow the logical paradigm’s treatment of relational attributes. Then we would re-engineer the attribute into a tuple object belonging to an ‘owned by’ tuples class. The tuple would start with the three-dimensional extension <car, garage> and then switch to the three-dimensional extension <car, Ms Brown>.

The problem is that this logical tuple changes, which tuples should not do in the object paradigm. If we are a little more sophisticated, we can resolve this problem. We need to construct the tuple from states of the car object, rather than the car object itself. Then, we have an object that captures the change pattern.

We divide the car object into states either side of the sale event. It has a car ‘owned by garage’ state (object) before the sale and an ‘owned by Ms Brown’ state (object) after the sale (illustrated in ***Figure 8.20***). We then use these state objects to construct two couples:

- <car owned by garage state, garage>, and
- <car owned by Ms Brown state, Ms Brown>.

These are the couples that belong to the ‘owned by’ tuples class. This neatly captures the change in a time-less way. You will have noticed that the car owned by state objects

(like the lepidopter state objects in ***Figure 8.16***) fall into a natural time-ordered sequence.

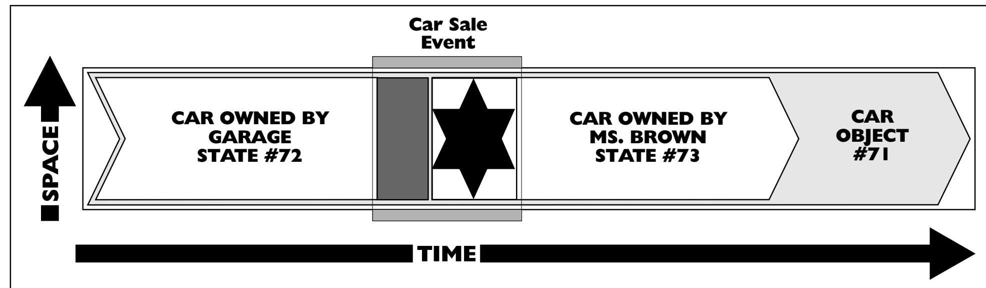


Figure 8.20: Car ownership state objects

3 Events – a new kind of physical object

So far in this chapter, we have looked at states. We saw how, under object semantics, these state objects are four-dimensional physical bodies—just like the physical bodies of which they are states. For example, the caterpillar state is as much of a physical body as the lepidopter object it is a state of.

We now look at the second type of object that the object paradigm uses to model changes, events. Unlike state objects, these are a new type of fundamental particle. What pattern underlies this particle? We touched upon it at the end of ***Chapter 6***. There we talked about how, within the logical paradigm, dynamic classification was not an object and so could not make use of the class and tuple patterns.

We now look at how object semantics transforms dynamic classifications into a new kind of object—event objects. We first look at what event objects are and the patterns they generate. Then we see how they capture and transform our ordinary notions of cause and effect—and much more—giving us an insight into understanding.

3.1 The object paradigm's shift to event physical objects

We now look at the shift to event objects from the logical paradigm's dynamic classifications. We identify the extension of these events, establishing them as objects. Then we look at the following patterns:

- The object versions of the happens–to and happens–at patterns,
- The encapsulation of complex events, and
- The object version of state change events.

3.1.1 Physical events as three-dimensional objects in a four-dimensional world

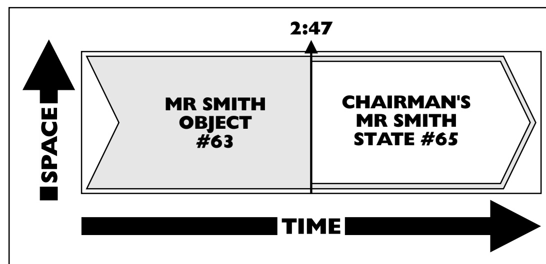
In some ways, our everyday intuitions about events anticipates object semantics. We say that the car accident happened at 10:00 am or that Mr. Smith was appointed Chairman of NatLand Bank at 2:47 pm. We see these events as happening at a point in time.

© Copyright Chris Partridge
chris.partridge@BoroCentre.com

This contrasts with the physical bodies that the changes happen to, objects such as the car in the accident and Mr. Smith. These we instinctively see as persisting through time.

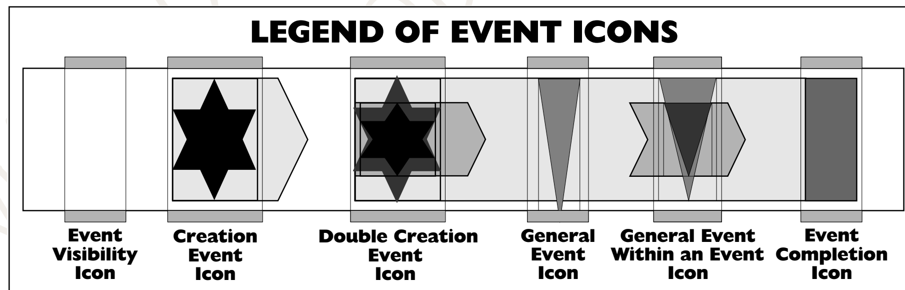
Object semantics respects this distinction. In it, events (unlike physical bodies) do not persist through time. To see what they are (what extension they occupy), consider the Chairman of NatLand Bank example again. Assume that the Mr. Smith's appointment to chairman event occurred at exactly 2:47 pm. Look carefully at the space-time map of this event in ***Figure 8.21***. The only candidate for the event is the moment in Mr. Smith's time line that he is appointed chairman. This is a slice of his four-dimensional extension at precisely 2:47 pm.

Figure 8.21: Mr. Smith's appointment event space-time map



While event and states are both time-slices, unlike states, events do not persist through time. They have zero thickness along the time dimension, because they only occupy an instant in time. This gives us a very neat distinction between physical bodies and physical events. Physical bodies persist through time; whereas, physical events do not. This makes bodies four-dimensional and events three-dimensional, but three-dimensional in a four-dimensional world. This gives a clear and simple way of distinguishing events (changes) from bodies; the first and fourth of our key types of things.

Figure 8.22: Legend of event icons for space-time maps



3.1.1.1 Drawing events on space-time maps

It is not easy to see the event in ***Figure 8.21***; it is a line at the very edge of a box. To get around this problem, I adopt a policy of turning the events in space-time maps into icons. This has the disadvantage of appearing to give them extension along the time dimension, but I find that this is more than outweighed by the advantage of being able to see them clearly. To signal that the time dimension is suspended for the icons, I put them in event visibility boxes. The most common events are the 'creation' or start and 'completion' or end events for physical bodies, and these have their own icons; a star for creation and a rectangle for completion. These icons have been used in most of the

© Copyright chrispartridge.com / BORO Centre.com

space-time maps in this chapter, for instance in ***Figures 8.18*** and ***8.20***. There is also a general event icon, which we have not used yet. To help you identify the icons, ***Figure 8.22*** gives a legend.

Most people will initially find it odd seeing an event as a three-dimensional slice of a physical body. Part of the problem is that the extension by itself does not seem like an event. However, we need to remember Frege's definition of meaning as composed of sense and reference (discussed in ***Chapter***). The three-dimensional extension is only the reference; the sense is the event's relevant connections to other objects. In other words, the pattern of connections between it and other extensions some of which are modelled in the space-time maps. The extension and the sense combine to make up the meaning.

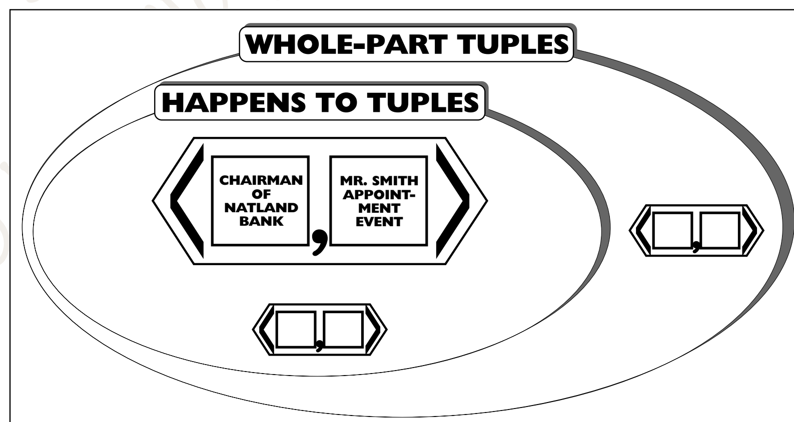
For example, the appointment event time-slice of Mr. Smith does not stand by itself. It acquires meaning by being put into context with the other objects; some of which we have not shown. For example, the Board appointed Mr. Smith chairman, so there is a connection between the Board and the appointment event. We look in more detail at these types of 'causal' connections in a later section.

3.1.2 The happens–to (whole–part) tuple

Seeing an event object as a three-dimensional extension in a four-dimensional world enables us to see a number of new patterns. One such pattern is the 'happens–to' tuple. This neatly illustrates how, in object semantics, analysis often becomes a matter of mapping patterns of connections between extensions, typically involving the whole–part pattern.

We loosely say that the appointment event 'happens to' the chairman. In the substance paradigm, this would be explained as Mr. Smith's substance acquiring a chairman attribute. In the logical paradigm, as Mr. Smith being dynamically classified as a chairman. In both paradigms, the 'happens–to' connection is not captured by its fundamental particles and so, in a sense, is outside their scopes. It is neither a substance, an attribute, a tuple or a class—it is certainly not a physical body.

Figure 8.23: Happens–to tuples class



© Copyright Chris Partridge
chris@BORO Centre.com

In object semantics, the ‘happens–to’ tuple is an extension and so an object within the scope of the paradigm. To understand it, we need to, at least, map its pattern of connections with other extensions. In the case of the chairman’s appointment, the most important connection is that the extension of the event is a part of the extension of the chairman. This is visible in the space-time map in ***Figure 8.21*** This means that the <chairman, Mr. Smith’s appointment event> couple not only belongs to the ‘happens–to’ tuples class, but also the whole–part tuples class. This is generally true of all ‘happens–to’ couples, which means the ‘happens–to’ tuples class is a sub-class of the whole–part tuples class (illustrated by ***Figure 8.23***.

3.1.3 The happens–at (whole–part) tuples

There is another useful connection in our example, the happens–at pattern. When we described the earlier example we said:

> Mr. Smith was appointed Chairman of NatLand Bank at 2:47 pm.

In other words the appointment event happened at 2:47 pm. This raises the interesting question of:

> What is the 2:47 pm object?

We need to know because it occupies one of the places in the happens–at couple (which is <2:47 pm, Mr. Smith’s appointment event>).

Object semantics approach to this is, as usual, simple but radical. It proposes an extension for the instant 2:47pm. But what extension? We know its temporal dimensions. Because it is instantaneous, it has zero time dimension. What are its spatial dimensions? The object paradigm proposes that it is the whole of space (at that instant 2:47 pm). So it is the instantaneous time-slice through the whole of space-time at 2:47 pm. Because it is an instantaneous time-slice, it is three-dimensional with zero time dimension. Under object semantics’ distinction between bodies and events, this makes it an event. This interpretation of 2:47 pm means that Mr. Smith’s appointment event is part of the 2:47 pm instant event. So the happens–at tuples class, like the happens–to tuples class, is a sub-class of the whole–part tuples class (in pattern-speak, the happens–at pattern is part of the whole–part pattern).

3.1.3.1 Time objects

We now have the key to explaining what a day, a month and a year are in object semantics. Let’s take 25th May 1999 as our example. We want to find out what its extension is. We intuitively think of this day lasting for twenty-four hours. In object semantics, this means that the time dimension of the day object is twenty-four hours long; starting at just after midnight on the 24th and finishing at midnight on the 25th. We now know its time dimensions, but what about its spatial dimensions? It follows the same pattern as the 2:47pm instant object; it is all of space between those two times. Similar transformations into physical bodies are made to months and years. Object semantics physicalises time. We shall look at these time patterns in more detail in the worked example in ***Chapter 17***.

We have some intuition that spatial whole–part and temporal whole–part patterns are similar. We routinely use the same prepositions for both
patterns—saying:

I went to Brighton *in* 1999. and

The hat is *in* the box.

However, our use of prepositions does not always tie in with the object paradigm's understanding. We don't say;

I went to Brighton *in* the 25th May.

but

I went to Brighton *on* the 25th May.

How this temporal 'on' is related to the spatial 'on' is unclear.

Furthermore we do not normally see days as spatio-temporal objects. If we did, we might say:

My trip to the Brighton object is *in* the 25th May object.

As we can see our use of language has not caught up with the object paradigm's more general and conceptually coherent notion of whole–part.

It should come as no surprise that the notion of event we have just examined bears a strong similarity to Einsteinian physics' definition of an event as a point in space-time—something with zero spatial and temporal dimensions. After all, the notion of space-time was borrowed from Einstein's theory to begin with. To explain the types of events that happen to the people-sized objects that business modelling deals with, we extended the physicists' definition of an event to encompass spatial dimensions.

3.1.4 Encapsulating complex events

So far we have been looking at examples of simple events occurring to one particular physical body. We now look at the encapsulation of more complex series of events. We see how object semantics explains complex events as encapsulations of simple events.

When two or more events are encapsulated into a single more complex event object, this new object is the fusion of the extensions of the encapsulated events. In a similar fashion to overall stuff (discussed in the last chapter), the complex event has a disconnected extension. For example, assume two people reach an agreement (also assume that this is done over the phone to make sure their extensions are disconnected). In object terms, there is an overall agreement event for both people. This is the fusion (or encapsulation) of the two agreement events for the individual people. ***Figure 8.24*** illustrates this. The encapsulated event has a single disconnected extension composed of the fusion of the extensions of the two component events.

© Copyright Chris Partridge chris.partridge@BORO Centre.com

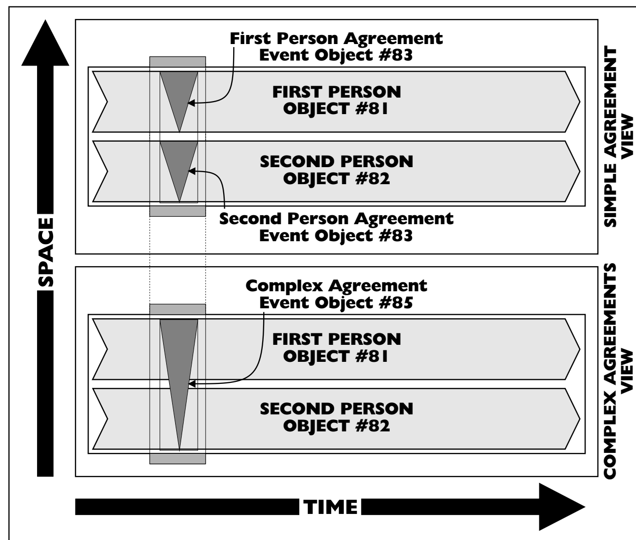


Figure 8.24: Encapsulated events

This same principle of encapsulation (fusing the component extensions) applies to much more complex events. Consider the Second World War. This is a single complex event object, but it is also a very complex network of events. It is the fusion of the extensions of a large number of simple events that happened to physical bodies. These form the base of an encapsulation (whole–part) hierarchy of more and more complex events, with the Second World War at its apex. Each of the smaller events is encapsulated into (a part of) one or more of the larger events. So, for example, the evacuation from Dunkirk and the D-Day landings are both encapsulated into (parts of) the overall complex Second World War event.

At first sight, it may seem that a complex event such as the Second World War persists through time. We (in Britain) talk about it starting in 1939 and ending in 1945. But there is a distinction to be drawn here. While the complex event may have parts in both 1939 and 1945, this does not mean it persists between 1939 and 1945. Because each of the simple parts has zero thickness along the time dimension, the total thickness of the fusion of these parts is the sum of the thickness of its parts. Now 0+0=0, even (mathematicians tell us) if we do it an infinite number of times. In the Second World War's case, we are adding a large, but finite, number of zeros. So no matter how many events make up the complex Second World War event, it still has zero time dimension and so stubbornly remains a three-dimensional event.

3.1.4.1 Complex events without a body

The new way of looking at complex events leads to a conclusion that is obvious but incapable of being captured properly in previous paradigms. Complex events do not always happen to a physical body. Indeed most of them, like the complex agreement event in ***Figure 8.24*** and the Second World War, do not. In the substance paradigm, a change always happened to an attribute belonging to a substance. In the logical paradigm, a change happened to an object that was dynamically classified. In object semantics, simple events happen to a physical body, such as Mr. Smith's appointment happens to Mr. Smith. But a more complex event does not have to. As in the complex

agreement event in Figure 8.24, its constituent simple event parts each happen to a physical body, but not usually the same one. As more events are encapsulated into a complex event, it gets more and more unlikely that they will have a physical body in common. In other words, it is unlikely that the extensions of all the individual events would be temporal parts of a single physical body. Typically, they are spread over a number.

These complex events without bodies have interesting repercussions for current O-O programming languages (OOPLs). Methods are, in some ways, OOPLs equivalent of events and 'objects' its equivalent of bodies. In most OOPLs, methods are firmly tied to 'objects' (in the object semantics' way of speaking, events are tied to bodies). In this environment, complex events, such as the Second World War, have to be squeezed into the framework. A technique often used is to create a pseudo-object (in our terms, a body) for the event to happen to. So there would be a Second World War 'object' for the complex Second World War event to happen to. As we have noted in the ***Prologue***, O-O programming is, in some ways, a halfway house with elements of both the substance and object paradigms; insisting events have to happen to 'objects' is one example.

3.1.5 Object-version of state change events and Zeno's paradox

It would seem that we now have a consistent and coherent picture of what event objects are and how they have extension. We have seen how events differ fundamentally from bodies. They only have three-dimensional spatial extension, with zero temporal extension—unlike bodies, which have a temporal dimension, and so persist through time.

However, there is a small area left, state change events, with an outstanding problem—Zeno of Elea's paradox. We first met this paradox in ***Chapter 4*** where we saw how the substance paradigm resolved the paradox using the now discredited notion of substance. We looked at it again in ***Chapter 6*** , where we saw the problems it caused in the logical paradigm if change was treated as an object with extension. If we do not see certain types of events in the right way, the paradox appears to cause problems for object semantics as well.

We can see why by looking again at the change example in ***Chapter 6*** (see ***Figure 6.21***). This assumes that there is a tomato changing colour—from green to red. The problem is that the instantaneous colour change has extension, albeit three-dimensional, and so has a colour. And the colour cannot be either green or red; otherwise, it would not be a change.

The object paradigm does not seem to have resolved this problem. The simple event we re-engineer from the instantaneous colour change has the same problems as its logical predecessor. We need to be more sophisticated in our re-engineering. We need to re-interpret the instantaneous colour change as a complex encapsulation of two events. This is the encapsulation of the completion event of the before state and the creation event of the after state (illustrated in ***Figure 8.25***). Zeno's paradox is no longer paradoxical because the encapsulated event does not have to have a single colour.

© Copyright Chris Partridge chrispartridge@BORO Centre.com

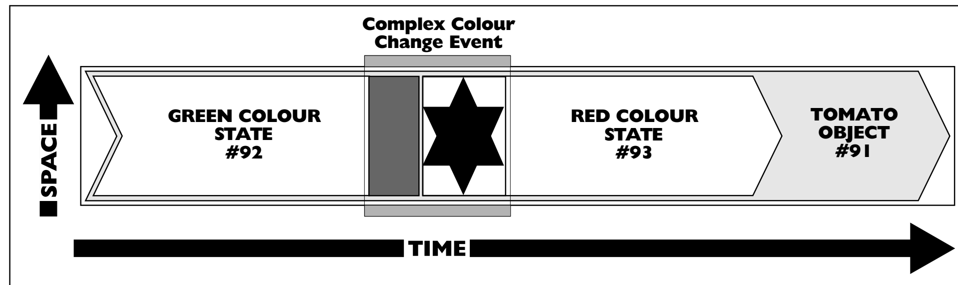


Figure 8.25: Complex event of the tomato changing colour space-time map

3.2 Events, causes and effects

We now have a consistent semantics for events as a new type of physical object. Unlike the dynamic classifications of the logical paradigm, they are objects, and so they can make use of the class and tuple patterns. They can be collected into classes or arranged into ordered tuples, just like any other object. They can have whole–part and super–sub-class patterns. They have, however, another equally important aspect. They are the basis for time-ordered patterns that capture and transform our ordinary notions of cause and effect. In the patterns, events explain the link between causes and effects.

In object semantics, the cause and effect connections are used to describe and explain a far wider range of patterns than is traditional in modern times. The semantics' notion of cause has more to do with understanding (and explaining)—the objective of business modelling, as we recognised in the ***Prologue***—than operation. It turns out that this approach has many similarities with the wide ranging ancient framework for cause originally brought together by Aristotle. We now look at this framework and see how it develops into the object paradigm's.

3.2.1 Aristotle's approach

Aristotle saw causes as explaining an event, helping us to understand it. So, to us, his classifications of cause seem to be explanatory principles; for example, he includes what we see as effects (the results or consequences of the event) as causes. He synthesised his framework from a number of traditions and the result was four types of cause or explanation:

- The efficient - that which makes a change happen,
- The material - what the change happens to,
- The formal - what the change results in, and
- The final - the end or purpose of the change.

Aristotle believed all of these were needed to give a proper explanation and criticised his predecessors for emphasising some to the neglect of others. A similar criticism could be made of our modern attitude, which often restricts us to the efficient cause—that which makes a change happen.

To see how Aristotle's approach works, consider a sculptor who has carved a statue from a block of marble. In this case, the types of cause are:

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

- The sculptor is the efficient cause, because he carves the block of marble into the statue.
- The marble is the material cause, because it is what the change happens to.
- The statue is the formal cause, because this is what the sculptor wanted to carve.
- The sale is the goal or final cause, because sculptor made the statue to sell.

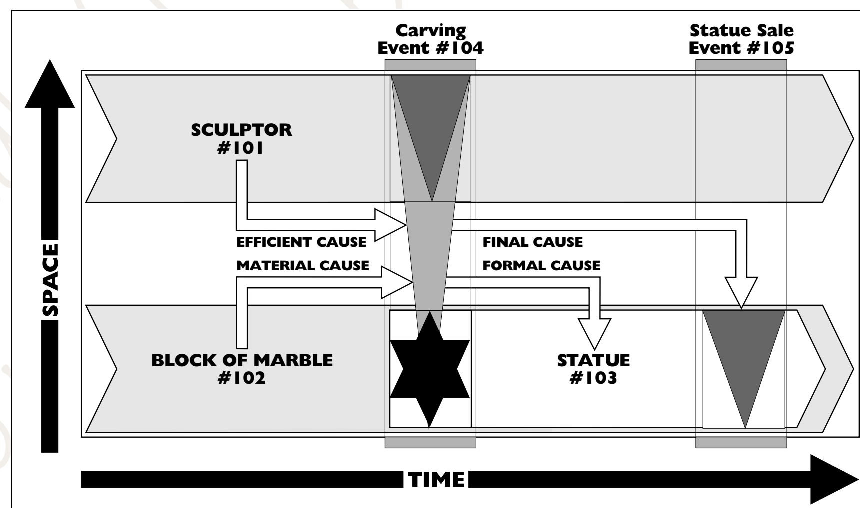
Aristotle was suggesting that when we describe all four types of cause (as we have just done here), we are giving a description of everything we need to know to understand the event.

3.2.2 *Object semantics' approach*

In object semantics, Aristotle's types of cause translate neatly into time-ordered patterns involving events. These are the event object's equivalent of the state object's time-ordered patterns we looked at earlier. As with the state object's patterns, we can illustrate the patterns with space-time maps.

Look at Figure ***Figure 8.26***. It is a space-time map for the sculptor carving a statue—with an additional event, the sale of the statue by the sculptor. To make the patterns more visible, we assume that the complex encapsulated process of carving the statue and the sale are both simple instantaneous events. (People versed in current O-O thinking can see this as the business modelling's equivalent of OOPL's 'information hiding'. The cause is a connection with the complex encapsulated event not its simple parts.)

Figure 8.26: Sculptor carving a statue space-time map



Each of Aristotle's four types of change appear in the space-time map. Their links to the carving event are illustrated with arrows. Underlying each arrow is a pattern of connections between the extensions of the objects involved in the statue carving event

© Copyright chrispartridge BORO Centre.com

The efficient cause is the sculptor who carved the statue. In more modern terminology, the sculptor is a pre-condition for the carving event. This is analysed as a tuple between the sculptor physical body and the carving event (the couple belonging to the cause tuples class). In time ordering terms, the sculptor physical body extension must 'exist before, during and after' the carving event extension.

The formal cause is the statue that is a result of the carving event. In modern terminology, the post-condition of the carving event. This is analysed as a connection between the carving event and the statue physical body. The carving event is a complex (encapsulated) event. The statue is a state object of the marble object, whose creation event is part of the carving event. This means the connection between the statue and the carving event is one of overlapping parts.

The material cause is the block of marble that is carved. This is also a pre-condition for the carving event, though one with a different time pattern to the efficient cause. This is analysed as a connection between the block of marble and the carving event. The connection has an overlapping pattern because the block of marble contains the statue object's creation event, which is part of the carving event. Like the efficient cause pre-condition, the block of marble extension must 'exist before, during and after' the carving event extension.

Object semantics leads to a counterintuitive situation for the material cause, where the cause connection pattern is also a whole–part pattern. The cause is a couple, <block of marble, statue>, which belongs to the cause tuples class. This couple also belongs to the whole–part tuples class, because, as we can see from the space-time map, the statue is part of the block of marble. In other words, the connection is both cause and whole–part. We instinctively differentiate between cause with its roots in time and whole–part with its root in space. However, as this example shows, in space-time our instinctive reactions are misleading.

Last, the final cause is the eventual sale of the statue. This is analysed as a connection between the carving event and the sale event. In time ordering terms, the carving event 'precedes' the sale event. There are also other less important patterns; for instance, elements of the sale event are part of the efficient cause and the material cause.

This shows how working out the Aristotelian causes is part of the overall task of mapping the sense of an event. When we analyse the pattern of connections between the extensions of the objects involved in the statue carving event, we naturally unearth them. This also shows that Aristotle and his predecessors intuitively understood the physical time patterns that object semantics make explicit. Their categorisation reflects the various aspects of the different underlying time patterns that explain the event. It by no means exhausts the time patterns that occur, but it does give us some idea of the most common patterns. It also gives us a feel for how analysing the patterns of connections between extensions can explain an event.

4 The time-based ‘consciousness’ of information systems

One of the prime characteristics of the object paradigm is the time-less nature of its objects. They give us an ‘objective’ view, independent of any particular information system at any particular time. This is an extremely powerful way of seeing the world. However, there is one aspect of an information system that cannot, whatever we do, be captured in a totally timeless way. This is its shifting position in time. From the information system’s perspective, its ‘consciousness’ exists at a point in time that is moving inexorably along the time dimension.

Computer systems are information systems and so they have a time-based ‘consciousness’. They reflect this in their information; when the computer system’s ‘consciousness’ is in the 24th May 1999, the leg of a deal that settles on the 25th May 1999 is classified as awaiting settlement. When its consciousness moves onto the 25th May 1999, it is re-classified as due today. This is a change in the computer’s ‘consciousness’; nothing has happened to the settlement. We need to be able to capture this in our business models.

4.1 The dynamic ‘here’ event

We do this by introducing a new kind of class object—the dynamic class or dynaclass. To reflect the information system’s consciousness moving down the time dimension, we use the dynamic ‘here’ event class. To explain what this object is, we first need to identify the system itself. It is a simple physical body, the four-dimensional extension of the system. We can then construct the new type of object that represents the moving consciousness—the dynamic ‘here’ event class. It is a class with a single member, the three-dimensional time-slice of the system at the instant of time that the consciousness is aware of. The time-slice’s zero length time dimension makes it an event.

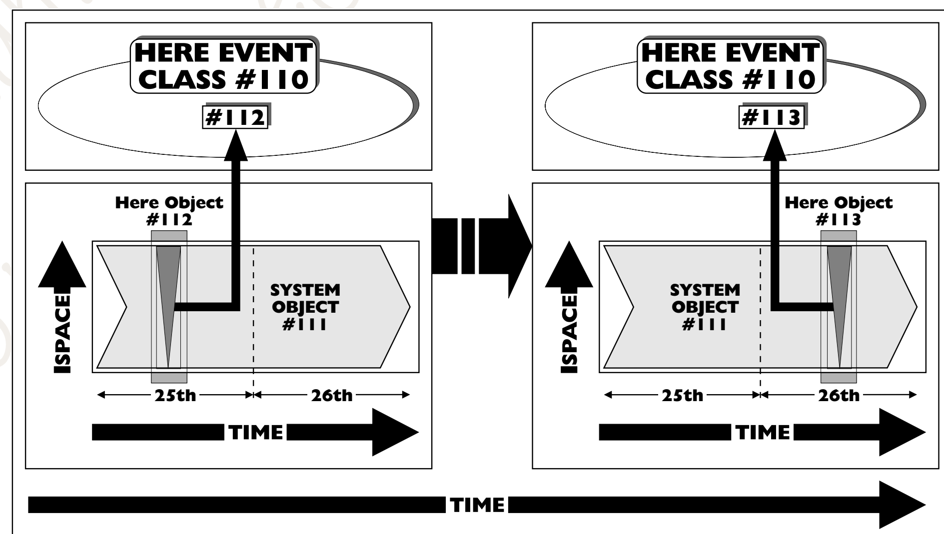


Figure 8.27: The dynamic ‘here’ event class space-time map

© Copyright Chris Partridge chris@BOROCentre.com

This event behaves in a similar way to the three-dimensional extensions of physical bodies in the logical paradigm. As the system's consciousness moves down the time dimension, the 'here' event class dynamically changes its member to the system's current three-dimensional time-slice. It is called dynamic because, unlike other objects, it changes. Two 'versions' of the dynamic 'here' event class are illustrated in the system's space-time map in ***Figure 8.27***.

4.2 The dynamic 'now' event and the dynamic 'current' tuples class

We need to find a way to link the dynamic 'here' event class to non-dynamic objects. We do this through another dynamic class,—the 'now' event class. This is also a class with a single member, the instant that the 'here' member event occupies—in other words, the whole of space for that instant. Like the 'here' event class, it is dynamic, with its member tracking the system's consciousness.

Once we have the 'now' event class, we can use it to construct a dynamic current tuples class for any class of physical objects. Consider, for example, the class cars. Some of the members of this class will exist now. Speaking timelessly, they have a temporal part that is part of the now object. For each of these cars we can construct a couple, <now, car> (illustrated in the space-time map in ***Figure 8.28***). All these couples belong to the current tuples class. Those cars that do not overlap with the now object, do not have a couple in the current tuples class.

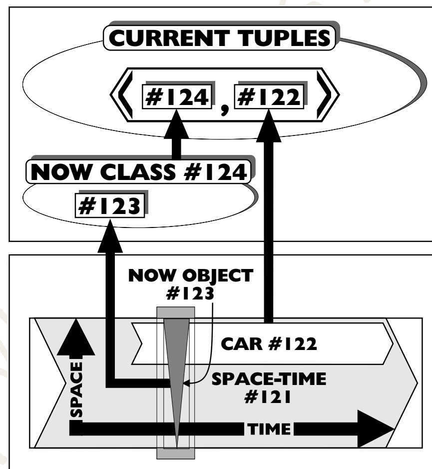


Figure 8.28: The dynamic current tuples class space-time map

The current tuples class is dynamic, because one of the places of its couple is dynamic, making the couple dynamic and so the class it belongs to. These dynamic couples provide us with the link between the time-bound 'consciousness' of the system and the timeless world of object semantics.

4.3 Implementing dynamic (state) classes

The objective of business modelling is understanding and so I try to keep the dynamic classes to a minimum. However, when building the system, there may be good opera-

tional (as opposed to understanding) reasons for designing dynamic classes for implementation. For example, a system may only need to keep a record of all the current state objects and have no interest at all in historical state objects. In this case, implementing a dynamic class that only reflects the current state makes sense. It would be a waste of information storage space to hold details of the previous states. However, we do not have to consider these issues when constructing the business model.

5 A new way of seeing changes—a key type of thing

This chapter provides us with a revised semantics for the fourth and final key type of thing—changes. Unlike previous paradigms, the object paradigm brings changes explicitly within its remit. Changes are event objects and share the patterns common to objects.

This is a radical change;, one that, as we expect by now, requires a completely new way of seeing, thinking and talking about things. This enables us to see the world more accurately. The tyre component example illustrates this (see ***Figure 8.14***). Where, under the logical paradigm, we would see a tyre as simply a part of a car, object semantics reveals a more accurate and sophisticated pattern of overlapping parts.

Furthermore, the new way of seeing is really new. Unlike the logical paradigm's 'member of' and 'part of' patterns that have begun to work their way into ordinary everyday language, object semantics has made next to no inroads.

Object semantics' timeless view of the world has had some impact. Expressions like 'time-line' for four-dimensional objects have been imported from Einsteinian physics. But we still see and talk of them in a time-oriented way. We talk of things moving down their time-line, bringing time into the four-dimensional world. Furthermore, most people still think of a period of time starting and ending. They do not see it as a physical body containing all of space for period of time. They certainly do not see an hour as a physical object that is a spatio-temporal part of a day object, though they may talk of hours being 'in' a day.

Otherwise, there is very little evidence of object semantics impact on everyday language. There are no words for:

- An event as an instantaneous time-slice of a four-dimensional object,
- A complex event as a fusion of extensions of simpler events, and
- An instant as a time-slice through space-time.

This is hardly surprising. Object semantics is under a hundred years old and things as fundamental as semantics can take much longer to work themselves into the popular consciousness.

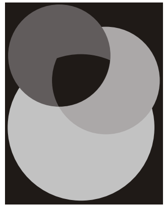
We can talk about objects in a timeless way by twisting our language. The traditional way of dealing with time in language is through tenses. We can describe the timeless four-dimensional world in a tenseless way by only using one tense, the present tense. We can start saying sentences such as 'the well state extended along the time dimension as far as 25th May'. We have done this, to some extent, in this chapter.

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

6 What's next

This discussion of language leads neatly onto the subject of the next chapter. Language is not a suitable format for describing a formal and sophisticated semantics, such as the object paradigm, with any accuracy. Once people start seeing and thinking in the new object-oriented way, they need an accurate means of modelling their four-dimensional world. That is the topic of the following chapters in Part Five. There we will look at the object syntax, and its notation, that together with object semantics form the object paradigm. People generally find that working through this gives them a more 'substantial' feel for object semantics.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



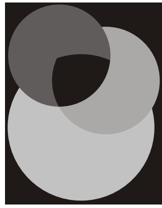
Part Five

Constructing Signs for Business Objects

Chapter 9 Constructing Signs for Business Objects

Chapter 10 Constructing Signs for Business Objects' Patterns

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 9

Constructing Signs for Business Objects

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction
2 Constructing signs for individual objects
3 Constructing signs for classes of objects
4 Constructing signs for tuples
5 Constructing signs for whole–part tuples
6 Constructing signs for dynamic objects
7 Signs as objects—modelling the model
8 What's next

1 Introduction

In Part Four, we developed an understanding of what business objects are. However, that was only a precursor, albeit an important one, to the real business of modelling. The accuracy and flexibility of object semantics give us a powerful way of seeing the business. We harness this power by building models that describe what we see.

Here, in Part Five, we focus on object syntax—in other words, on how we 'write' the signs for objects and their patterns. We learn a notation for describing business objects in models. In this chapter we look at the individual signs for the main types of business object. These are the signs with which we build the business object model. We focus on what they mean and how they work. Then, in the next chapter, we look at signs for business objects' patterns.

Together Part Five's two chapters help us to develop an understanding of object syntax and the notation for business object models. They also deepen and broaden our understanding of object semantics. Using a notation for describing objects naturally leads to a better understanding of them. For example, because the notation explicitly signs the key structural patterns (super–sub-class, class–member and whole–part) these are clearly visible and so easier to understand. And because the notation gives each pattern a different sign, it helps us see that they are different.

Learning this notation is essential for Part Six, where we work our way though examples of re-engineering existing computer systems into business objects. As well as providing useful illustrations of both object semantics and object syntax, these examples will provide us with further experience of how the notation is used.

1.1 Main types of business object

In this chapter we look at the object notation's basic signs for the following main types of object:

- Individual objects,
- Class Objects,
- Tuple objects (and, more specifically, whole–part tuple objects), and
- Dynamic objects.

We see how the signs are constructed, what they mean and how they are used.

1.2 Why use a two-dimensional notation for a multi-dimensional model?

Before we look at the notation, I should explain why it is two-dimensional. I have asserted a number of times that objects free us from the two-dimensional constraints imposed by paper and ink technology. I have suggested that this enables us to take advantage of computer technology's ability to handle multi-dimensional structures. However, the notation we are about to look at is on paper and so only two-dimensional. Why is this so and why doesn't it constrain the overall business model to two dimen-

© Copyright Chris Partridge chris.partridge@BORO Centre.com

sions? To understand the answers to these questions, we need to look closely at the ‘technology’ that we use when business modelling.

Modelling the business is currently done by humans. It is human brains, and not computers, that construct and revise the business model. This means that the human mind needs to ‘interface’ with the business model. The object notation has to be easily read by humans.

Human biotechnology and computer technology both constrain how we can ‘process’ a business model. (Processing currently means ‘see’—we do not touch or hear business models, let alone taste or smell them.) Computer technology constrains our visual ‘interface’ to two dimensions. The ‘inputs’ we receive from a computer system, whether on a screen or a print-out, are two-dimensional. The biotechnology of human eyes’ retinas is also constrained to two-dimensional images. Furthermore, human brains are trained to process the kind of information in business models on two-dimensional surfaces.

From a practical point of view, this means that the sensible solution is to construct and review the multi-dimensional business model through two-dimensional views. Digestible two-dimensional chunks are an easy and effective way for the human brain to absorb the model. And its multi-dimensionality is not affected.

This solution can give computer technology an important role. Business models are static—in both traditional and object modelling; they map the time dimension onto the spatial dimensions. This means that the business model is not itself an information processing system; it is only stored information—data. However, producing a two-dimensional view of a multi-dimensional business model does take processing. So, at least in theory, we need the power of a computer to store the multi-dimensional model and produce the two-dimensional views.

In practice, using a computer with good CASE tool software can make the administration of storing the model and the processing of views easier, but it is not essential. I have found that constructing a multi-dimensional business model from two-dimensional paper views (in other words, using paper and ink technology) is a practical option—particularly when working with small models. The vital decisions about the construction and review of the business model happen in the brain of the business modeller, which is not excessively hampered by a paper model.

This is just as well because CASE tool software is not yet fully geared up for business object modelling. At a more mundane level, I have found a computer graphics package an invaluable aid to producing the paper views; the results are much more legible than hand-drawn ones. While computers are not essential at the business modelling stage, it is a different story when the model is turned into a working system. Then, computer technology becomes essential.

2 Constructing signs for individual objects

Let’s now look at the two-dimensional notation. In object semantics, an individual object is a plain and simple extension. This is referred to (directly mapped onto) by a sign in

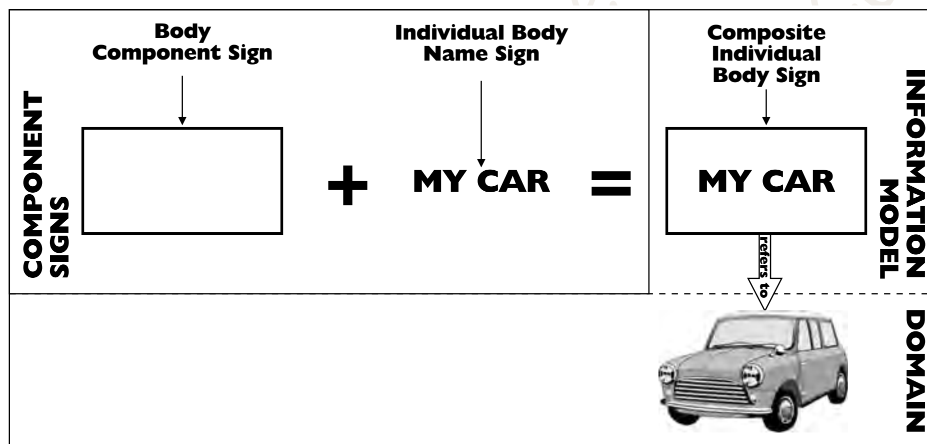
the model. We use different signs for the different types of individual objects:

- Individual body, and
- Individual event.

2.1 Constructing a sign for an individual body

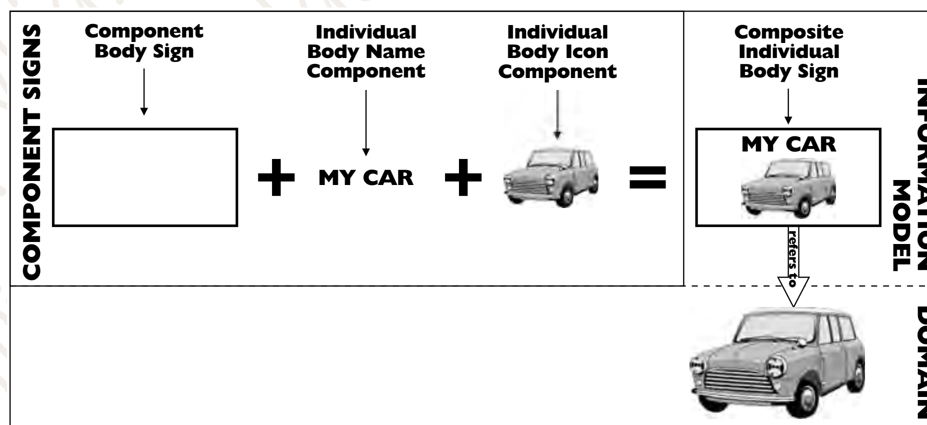
The sign for an individual body is constructed out of two components. A body sign, which is a rectangle, and a name sign that is the name of the body. We put the name sign inside the body sign (shown in ***Figure 9.1***). This figure also diagrams the extra-model reference link between the individual body sign in the model and the body object in the domain.

Figure 9.1: Individual body sign



Sometimes, to aid recognition, we include an icon of the individual body inside the body sign (shown in ***Figure 9.2***).

Figure 9.2: Alternative individual body sign



2.2 Constructing a sign for an individual event

We construct the sign for an individual event in a similar way out of two components. An event sign, which is an ellipse, and a name sign, which is the name of the event. We again put the name sign inside the event sign (shown in ***Figure 9.3***).

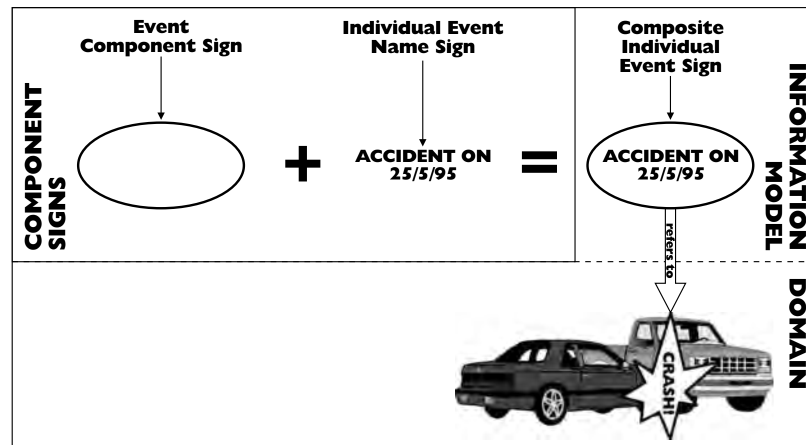


Figure 9.3: Individual event sign

2.3 Constructing individual object name sign components

The shape of the component body and event signs show their type; so, all signs of the same shape are the same type. We use everyday language for the name components. These differentiate between signs for different objects. They help us recognise which object a particular composite sign refers to. To avoid confusion, a convention, within each model, indicates that the name signs are unique; no two individual objects have the same name sign.

3 Constructing signs for classes of objects

We now look at how to construct the signs that refer to classes. We also look at the signs for the class pattern's two important tuples connecting classes:

- Class–member, and
- Super–sub-class.

3.1 Constructing a sign for a class of individual objects

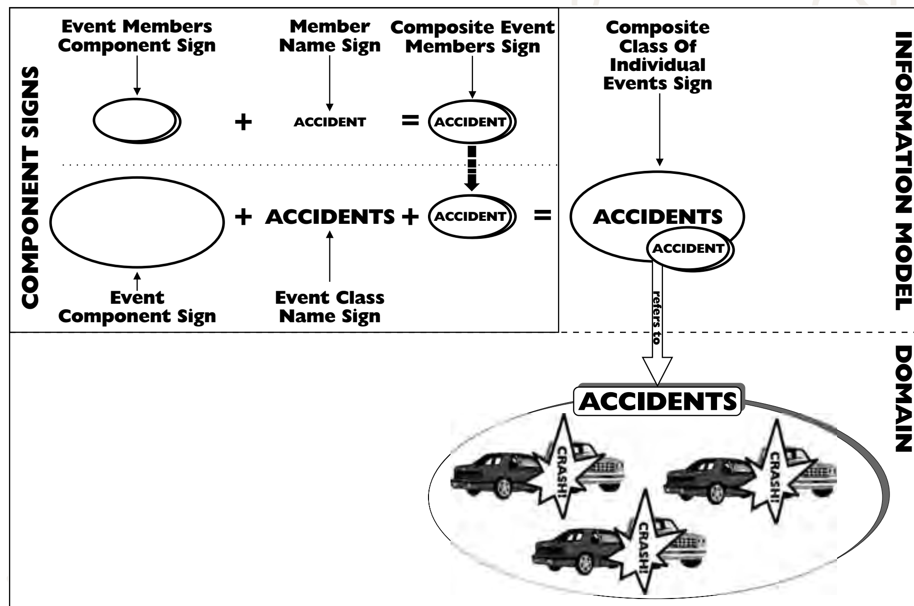
We first look at how to construct a sign for a class of individual objects. Just as there are different signs for an individual body and an individual event, there are different signs for a class of individual bodies and a class of individual events.

3.1.1 Constructing a sign for a class of individual events

Remember that we construct a class of individual objects by collecting together the extensions of those objects and treating the collection as a single object. This single object is what the class sign refers to.

A class of individual events only contains events, so we use the same elliptical event sign as a component. We put the name of the class in this ellipse. We then indicate that we have constructed the event class out of individual events by putting two smaller superimposed ellipses—signs for the member events—in the bottom right corner. ***Figure 9.4*** gives an example. In this example, we have also put the name sign for a member of the class, 'accidents', in the smaller ellipse. Often, however, a member name sign takes up too much space and we have to leave it out.

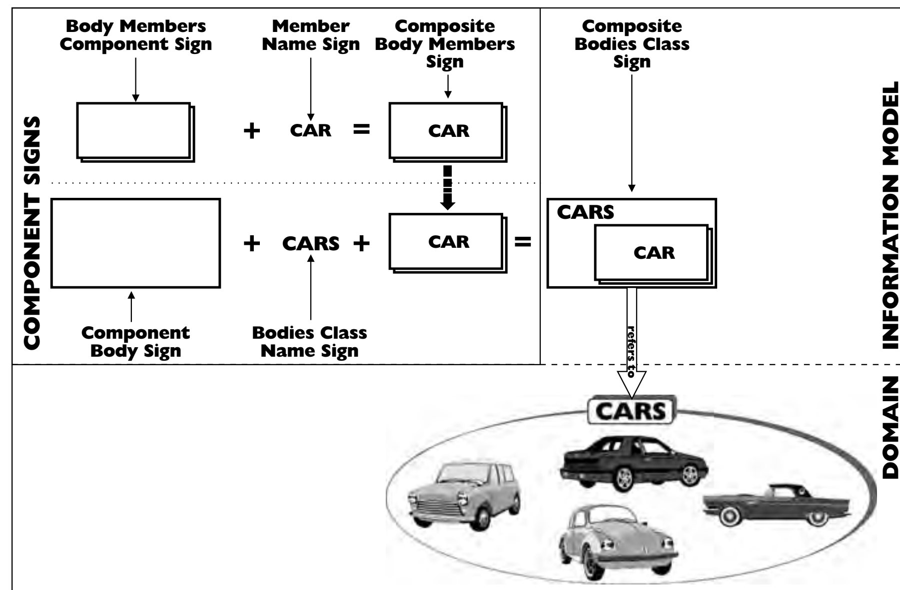
Figure 9.4: A class of individual events sign



3.1.2 Constructing a sign for a class of individual bodies

The sign for a class of individual bodies follows the same pattern. We use the same rectangular box sign that we used for individual body signs and show the class has members using two smaller superimposed rectangular boxes in the bottom right corner. Again we name the class and, if there is enough space, the potential members. The name sign for the class is in the larger class rectangle and the name sign for a member of the class is in the smaller member rectangles (shown in ***Figure 9.5***).

Figure 9.5: A class of individual bodies sign



3.1.3 *Constructing class name and member name sign components*

We use class names, as we used individual object names, to differentiate the signs (and so identify the classes). As before, we keep the names unique within each model. Unlike some notations, we use different names for a class and its members. These other notations, will, for example, call both the cars class and its individual members 'car'. I have found that this causes confusion. In object semantics, a clear distinction is made between the class and its members. In object syntax, this is reflected in different names for the class and its members— often, as here, the plural and singular forms of a noun. Using the car example, the class is called (and so the class name sign is) 'cars' and an individual member is called a 'car'.

3.2 Constructing a sign for a class–member tuple

Individual objects (whether bodies or events) that are members of a class belong to that class; that is, there is a class–member tuple connecting each member object and the class. This tuple is central to the notion of a class, so we need to have a sign for it in our notation.

3.2.1 *Classes and members*

The class–member tuple is, strictly speaking, a couple <individual object, class> that belongs to the class–member tuples class. We model it by drawing a class–member tuple sign. This is a dashed line joining the relevant class and member signs. It has, at the member end, a semi-circle with a line through it (shown in ***Figure 9.6***). This is intended to look like the Greek character epsilon 'Î'—the mathematical sign for class

membership. We show that the connection is a tuple by putting a black diamond, the sign for a tuple, on the line.

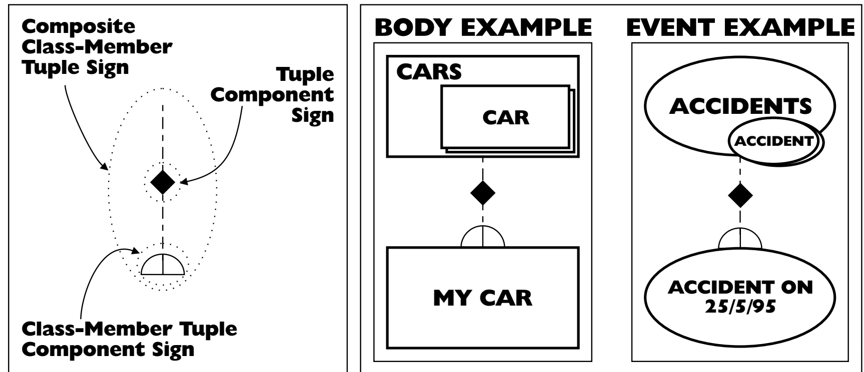


Figure 9.6: Class–member tuple sign

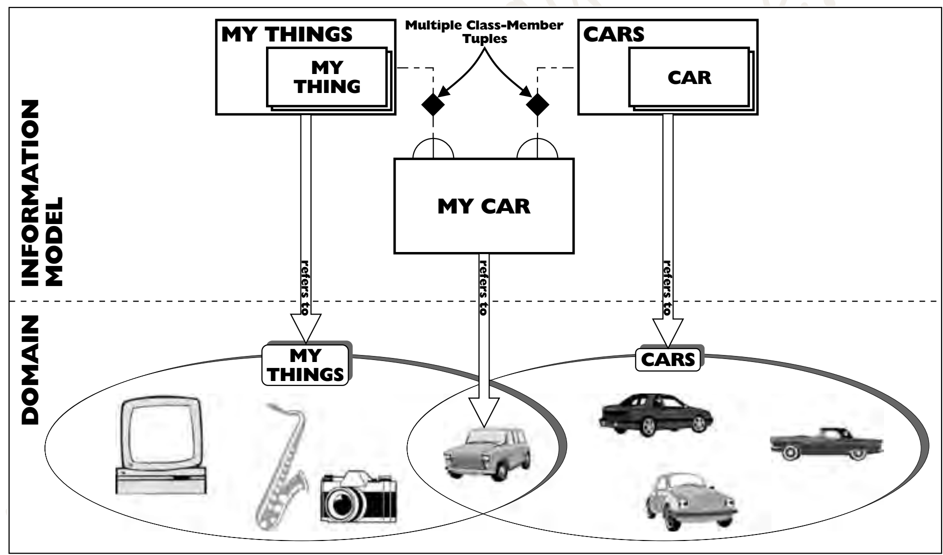


Figure 9.7: Multiple class–member tuple signs

As you can see from the figure, we use the same class–member tuple sign for body and event classes. This is understandable because the underlying pattern is the same. There is also an informal convention (followed in Figure 9.6) that we draw classes higher up the page than their members; though in some complicated diagrams, it is not possible to do this.

3.2.1.1 Members of more than one class

Unlike some notations, this can easily model an object that is a member of more than one class—what we called multiple classification. We just join the object's sign to each of the relevant class signs with class–member tuple signs (shown in ***Figure 9.7***).

3.2.1.2 An accurate class–member sign pattern

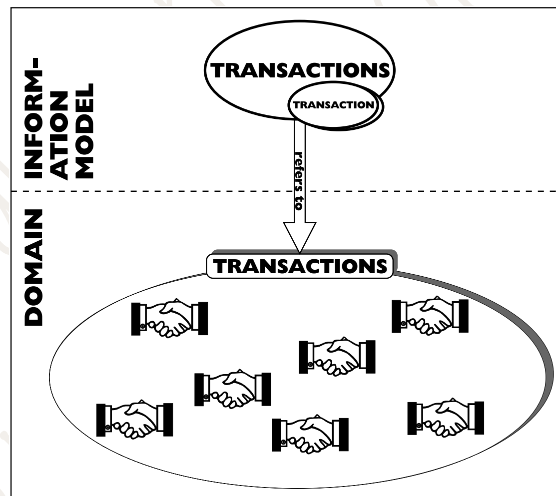
The class–member pattern is a very strong pattern; one that is central to object semantics. So is its reflection in the information model, the class–member sign pattern. Because one is a reflection of the other, they have similar patterns. However, a common mistake is to assume they have the same pattern. This is not so. The information model's 'ignorance' leads to differences. We look at one of these now.

It is natural and normal to assume a class has members. A class is a class because it captures some common patterns of its members; so, it is reasonable to assume it has members. Because a class sign's purpose is to model a class, it also appears reasonable to assume that it will reflect this characteristic—to think that a class sign is always linked to some member signs (sometimes called instances).

But this is wrong—it turns out that it is natural and normal in an information system for a class sign to have no member signs. In fact, it is quite common. For example, we talk about types of wild animals without having any notion of a particular animal. We talk about elephants (the class elephants) or gorillas (the class gorillas) without ever knowing a particular elephant or gorilla (members of the classes elephants and gorillas). Our minds, as information systems, have no member signs for the class signs we are using.

Member sign-less (instanceless) class signs are an almost universal rule in the shipped versions of business computer system packages. For example, accounting packages are usually shipped with a transactions file (a class sign) that has no individual transactions (in object terms, member signs)—the situation shown in ***Figure 9.8***.

Figure 9.8: The instanceless transactions class sign



3.2.2 Modelling lack of membership information

No information system is completely 'informed'. This includes human minds, which are considered information systems. We now illustrate this with two types of ignorance that arise when modelling the class–member pattern:

- Unknown members, and

- Unknown membership.

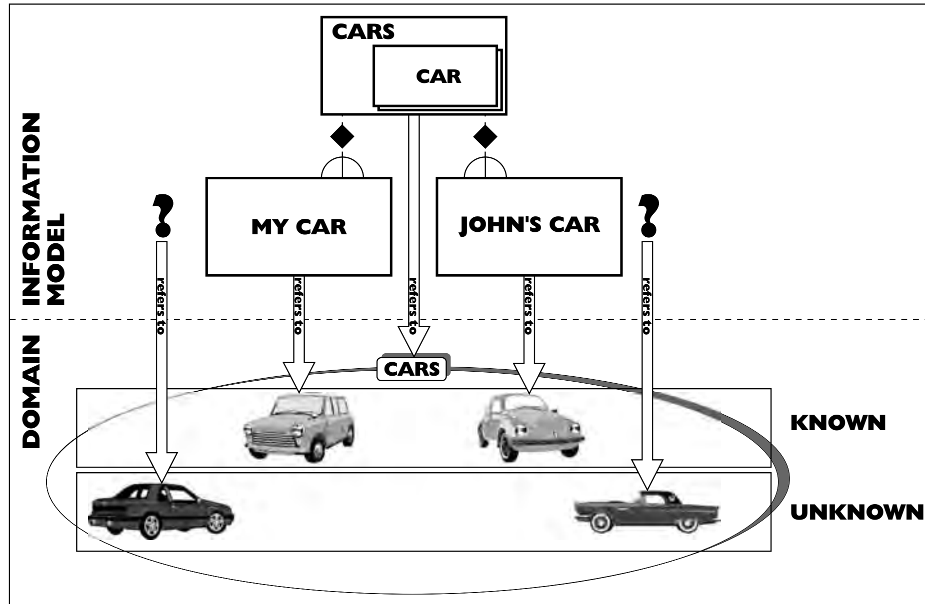


Figure 9.9: Known and unknown class–members

3.2.2.1 *Modelling unknown members*

For every class in an information system, when we look at it objectively from outside the system, we can divide its members into known and unknown. Known if the information system has a sign for them; otherwise, unknown (or, more accurately, unknown by the information system but known by us—otherwise, we could know that they were unknown). This distinction has nothing to do with the class or its members. It is a feature of the information model (shown in ***Figure 9.9***.)

It is common for a member to be unknown because it has not yet come into existence. When it does, the information system can then construct a sign for it. This happens, for instance, when a new country is created. It happened recently for the Czech Republic and Slovakia; ten years before they were created, no-one would have known about these two countries. But when Czechoslovakia decided to separate into two countries, people began to become aware of them. The extension of the class countries in the real world did not change. All that changed was the construction of new signs for the Czech Republic and Slovakia in information systems.

3.2.2.2 *Modelling unknown class membership*

It is important to remember that, not only do signs have to be constructed in the system for the class's members, but also for the class–member tuples connecting members to the class. Our minds automatically and unconsciously supply this link; so, it is easy to forget that it needs to be explicitly constructed. We can illustrate this with an example where the system starts off knowing about the member of a class but not its membership of the class.

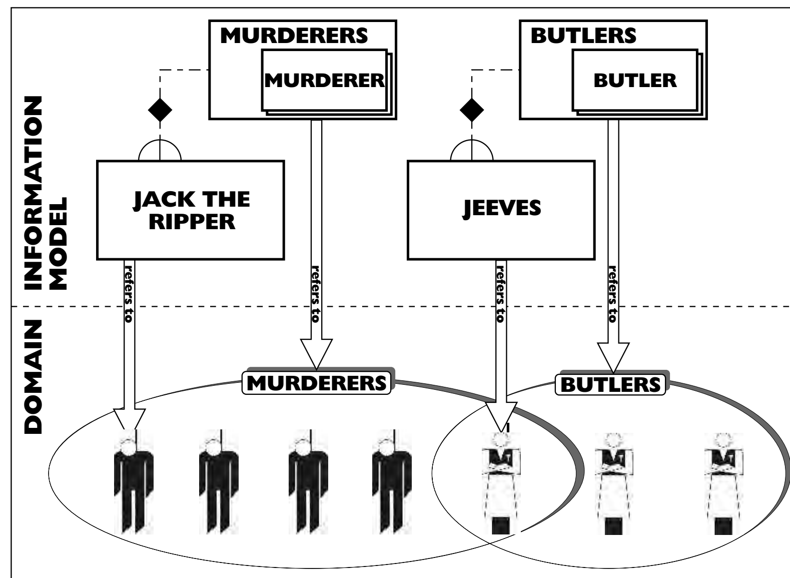


Figure 9.10: Jeeves the butler as an unknown member of the class murderers

Consider an Agatha Christie type of detective novel, in which a murder has been committed in a country house. At the beginning of the novel, we are introduced to each of the characters; the butler, the lord of the manor, the chambermaid, and so on. We know that, by convention, one of these is the murderer. Assume that Jeeves the butler is the murderer—in other words, a member of the class of murderers. Now, when we start reading the book we know Jeeves and know the class murderers, but have not (yet) found out that Jeeves is a member of the class murderers. ***Figure 9.10*** shows the state of our knowledge.

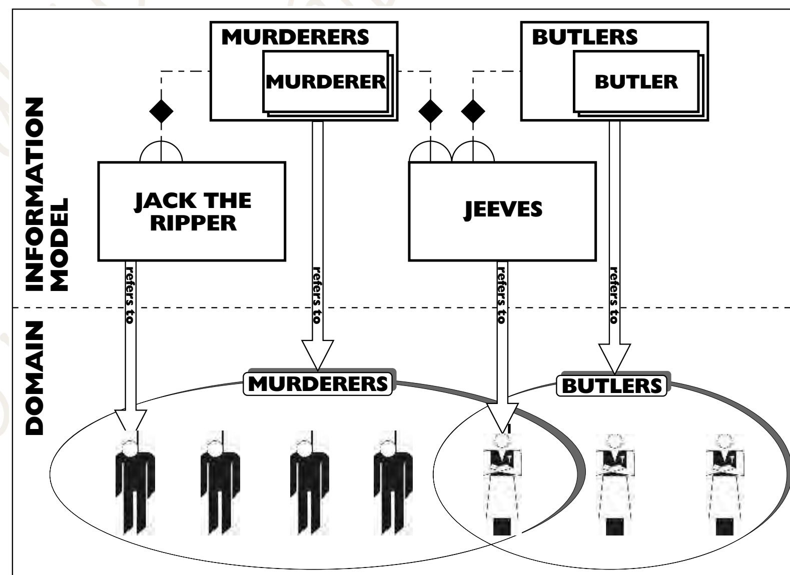


Figure 9.11: Jeeves the butler as a known member of the class murderers

© Copyright Chris Partridge chris@BORO-centre.com

At some stage, as the plot unfolds, we realise the butler is the murderer. As we already have signs for the butler and the class murderers, all we need to do is construct the class–member tuple sign between the two. The result is shown in ***Figure 9.11***. Notice that there is no change in the domain. The butler belonged to the class murderers all along, what happens when we solve the mystery is that we learn of his membership.

3.2.2.3 The constructive nature of modelling

This and the previous example of ‘ignorance’ have highlighted what might be called the constructive nature of signs and so information. Signs only exist if we construct them. This is obvious when we start to think about it. How could a sign exist that has not been constructed? We shall see, as we work through this chapter, the fundamental impact this constructive nature has on information modelling.

3.2.3 Classes as members of classes

So far we have only considered classes of individual objects. However, we recognised in the logical paradigm that classes were objects and so could, like individual objects, be collected together into classes—giving us classes of classes objects. This means that the class–member tuple, with its <class, member> format, can have any type of object (individual, class or tuple) in its member place. We now look at how we sign the class-member pattern for classes of classes, and, in the process, see how we capture class–member hierarchy patterns in the model.

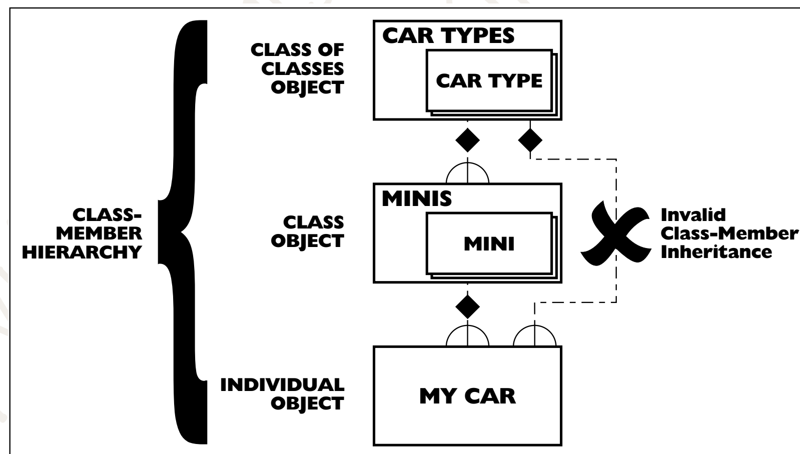


Figure 9.12: Car types—an example of a class–member hierarchy

3.2.3.1 Class–member hierarchy

The sign for describing a class as a member of a class is exactly the same as that for describing an individual object as a member of a class. The example shown in ***Figure 9.12*** is taken from our original example of classes of classes in ***Chapter 6*** (illustrated by ***Figures 6.8*** and ***6.9***). As we can see, the class–member sign is used in the same way for members that are classes as for members that are individual objects. ***Figure 9.12*** is also an example of how we model a simple class–member hierarchy.

3.2.3.2 Class membership inheritance

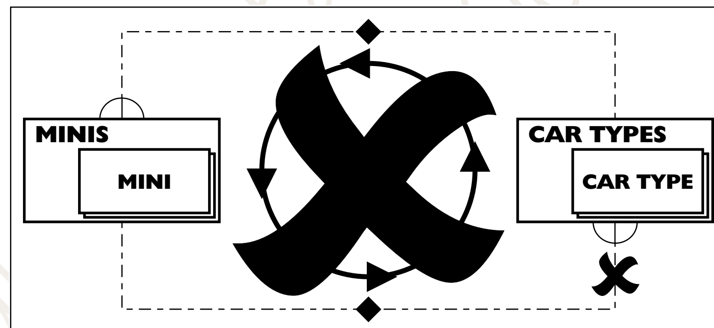
We shall see later on that various patterns are inherited down (and sometimes up) the different hierarchies. However class membership is not one of these. It is inherited neither up nor down the class–member hierarchy. Consider my car in ***Figure 9.12***. It is a member of the class minis, which is itself a member of the class car types. But this does not imply that my car is automatically a member of the class car types. In fact, as shown, it is not a member.

This should not be surprising. Classes capture patterns by collecting together similar objects. It is unlikely that a collection of similar classes, such as car types, would share their car type pattern with their members. For example, that my car (a member of minis) would behave like a car type.

3.2.3.3 Ban on circularity

Because we construct classes from extensions, we cannot construct a class with itself as a member. Furthermore, we cannot construct a class that is a member of a class lower down the class–member hierarchy. The impossible situation is shown in ***Figure 9.13***. We recognise this impossibility in the information model. We do not allow class signs to be instances of class signs lower down the class–member sign hierarchy.

Figure 9.13: Impossible circular class–member hierarchy



It is the nature of our understanding of space (and time and space-time) that makes this circularity impossible. This can be shown using the reference diagram in ***Figure 9.14***.

3.3 Constructing a sign for a super–sub-class tuple

We have just looked at the signs for capturing the class–member patterns. However, this is only one of class's two main structural patterns. The super–sub-class connection is the other. We now look at the signs for this second main structural pattern. Together these two provide a framework that helps give classes their enormous power.

3.3.1 The super–sub-class pattern

The super–sub-class pattern resembles a whole–part pattern for classes. It is about classes containing other classes. For example, horses are animals—or, in class-speak,

© Copyright chrispartridge.com

the class horses is a sub-class of (is contained in) the class animals. This containment or sub-class connection is between the super-class and the sub-class. Strictly speaking, it is the couple <super-class, sub-class> that belongs to the super–sub-class tuples class.

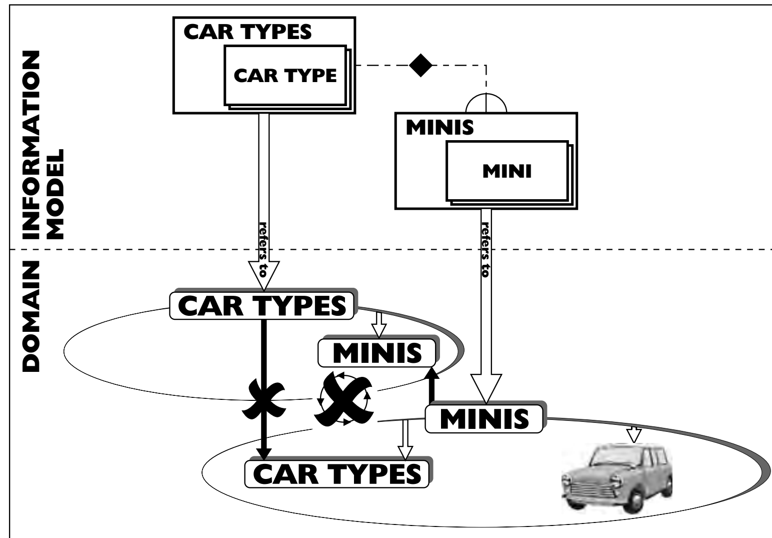


Figure 9.14: Impossible circular class–member hierarchy reference diagram

3.3.1.1 *Super–sub-class sign*

We model this super–sub-class pattern with a sign. It consists of a line joining the two relevant class signs with a semi-circle at the sub-class end. This is intended to look like the mathematical notation for sub-class—'É'. Because it reflects a tuple, it also has a black diamond tuple sign on the line. As we can see from ***Figure 9.15***, the same sign is used for body and event classes. There is no need for different signs because the connections have the same pattern.

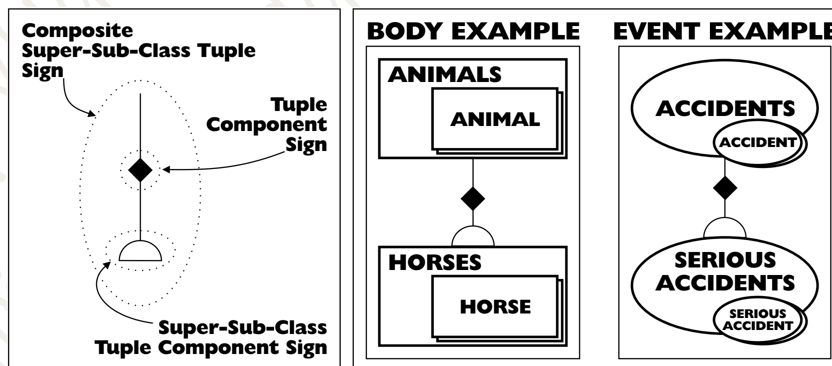
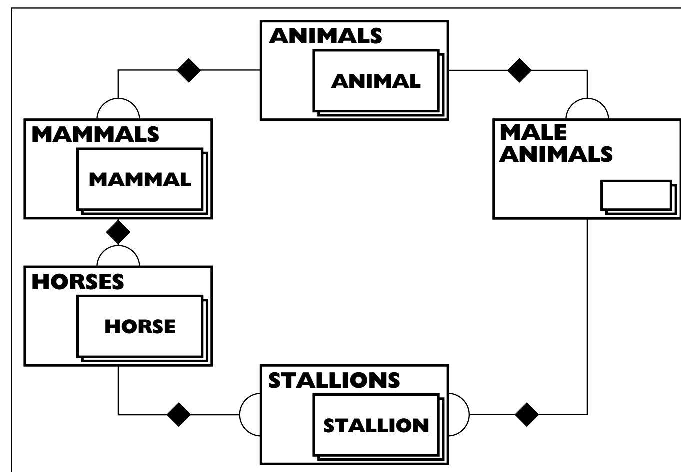


Figure 9.15: Super–sub-class tuple sign

3.3.2 *Super–sub-class hierarchies*

Typically, in a business model, classes are linked into a lattice hierarchy of super- and sub-classes. As we saw, when looking at the logical paradigm in ***Chapter 6*** (see ***Figure 6.3***), a tree hierarchy is too constraining to provide an undistorted reflection of reality.

Figure 9.16: Natural super–sub-class hierarchy structure



3.3.2.1 *Natural super–sub-class hierarchy structure*

In this lattice hierarchy, a super-class may have multiple sub-classes and a sub-class may have multiple super-classes. For instance, the schema in ***Figure 9.16*** models the super-class animals as having the sub-classes, mammals and male animals. It models the class stallions as having the classes horses and male animals as its super-classes.

When I construct a model of a super–sub-class hierarchy like this, I tend to automatically order the classes into a structure like the one in ***Figure 9.16***. As you can see, this follows an informal convention whereby super-classes are higher up the page than their sub-classes (though I find that in some complicated hierarchies it is not possible to do this).

3.3.2.2 *Modelling descendant–sub-classes*

The natural structure in Figure 9.16 subtly ignores the fact that the super–sub-class tuple can be inherited. The class stallions is a sub-class of the class horses and so contained in it. The class horses is a sub-class of the class mammals, which is a sub-class of the class animals. So, the class stallions is contained in the class animals. This means that we can, if we wish, recognise it as a sub-class and construct a sub-class sign in the model linking them.

Though we may need to do this for some classes, it is not a good idea to do it for all of them in a single schema. Why is this? Consider what the model for our simple example would look like if we included signs for all the possible sub-class tuples.

© Copyright Chris Partridge chrisPartridge@BORO Centre.com

Figure 9.17 illustrates the problem—the hierarchy becomes cluttered. If the super–sub-class hierarchy were larger, the problem would be worse because the number of potential sub-class tuples would increase dramatically. Modelling all these possible sub-class tuples would result in an impossibly cluttered schema.

Figure 9.17: All possible sub-class tuples

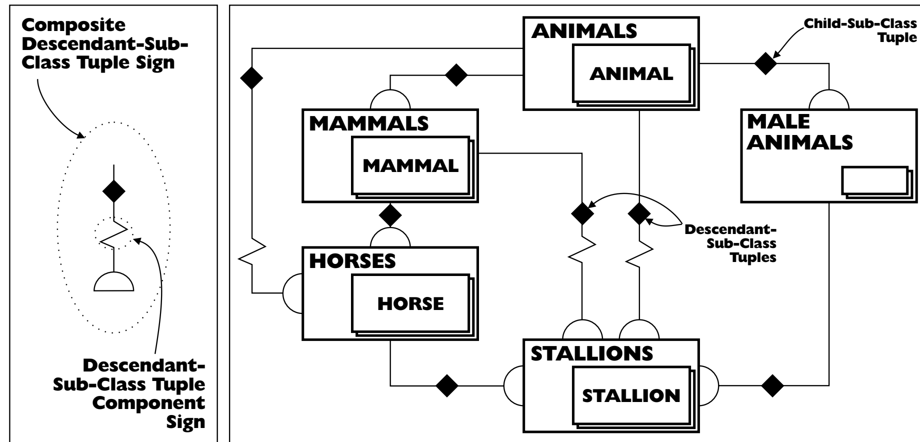


Figure 9.17 also, quite usefully, distinguishes between two types of sub-class tuples; child and descendant. The stallions-to-horses tuple is a child–sub-class tuple because there are no intermediate sub-classes explicitly modelled. On the other hand, the stallions-to-animals tuple is a descendant–sub-class tuple because the sub-classes, mammals and horses, are explicitly modelled as intermediate sub-classes. The sub-class sign we have been using until now is really the sign for the child–sub-class tuple. The descendant–sub-class tuple sign is a modified version of it, with an additional zigzag in its line (as shown in ***Figure 9.17***).

3.3.2.3 Deducing descendant–sub-class signs

Descendant–sub-class tuples logically depend on child–sub-class tuples, because we can 'logically' construct their signs from the signs for the child–sub-class tuples. More generally, we can logically deduce the sign for a descendant–sub-class tuple from a combination of sub-class tuples. This deduction has the following pattern:

A is a sub-class of B
B is a sub-class of C
C is a sub-class of D
D is a sub-class of E
Therefore:A is a descendant–sub-class of E

Where there can be any number of sub-class lines (except zero and one of course).

This is not a new logical deduction pattern. It is the same as one of Aristotle's syllogisms—one that looks like this:

All Spartans are humans,
All humans are animals,
So all Spartans are animals.

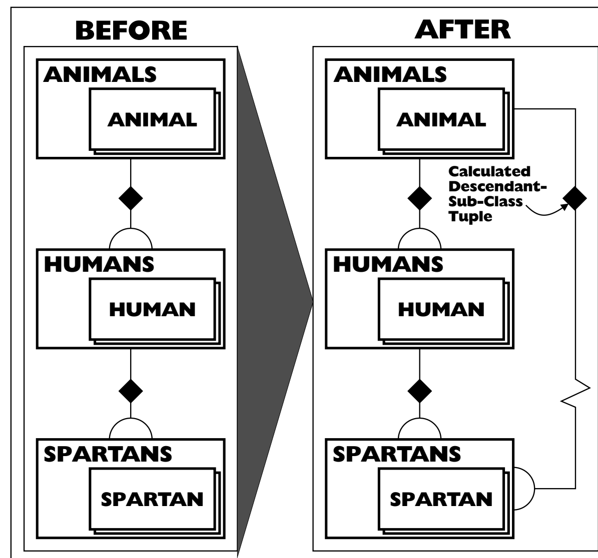
© Copyright Chris Partridge chris.partridge@BorOGroup.com

It might be easier to see the resemblance when the syllogism is translated into class-speak—as below:

> The class Spartans is a sub-class of the class humans,
> The class humans is a sub-class of the class animals,
> So the class Spartans is a descendant–sub-class of the class animals.

Figure 9.18 shows this descendant calculation graphically.

Figure 9.18: Descendant–sub-class calculations



3.3.2.4 Virtual descendant–sub-class signs

This descendant deduction pattern provides an opportunity to tidy up the sub-class clutter problem. The descendant signs can be virtual, calculated as required. As we discussed in ***Chapter 2*** , there is no reason why processes in the information system cannot represent business objects. This gives us the benefit of having signs for all the descendant–sub-class tuples without having to bear the cost of storing them—a significant compacting.

The power of computing makes this 'virtual' strategy more reliable. In a paper and ink environment, the information processor that deduces the descendant tuple signs is our minds. They are not particularly reliable processors, particularly of these sorts of logical calculations. However, in computer processing, we have a reliable logical processor. It can accurately and consistently calculate the signs.

I normally adopt a strategy of making most descendant–sub-class signs virtual. I construct views of the business model that only show the signs for child–sub-class tuples and those descendant–sub-class tuples that are essential. I make the signs for the other descendant–sub-class tuples virtual. This reduces the clutter in even the most complicated super–sub-class hierarchy to an easily manageable level.

© Copyright Chris Partridge@BORO Centre.com

3.3.2.5 Non-circular super–sub-class hierarchy structure

There is a logical constraint upon the super–sub-class hierarchy. Like the earlier class–member hierarchy, it cannot be circular. For example, animals from ***Figure 9.18*** cannot be a sub-class of Spartans, as (falsely) indicated in ***Figure 9.19***. Because classes are built up out of extensions, it is impossible for any circularity to exist. A class, such as animals, cannot even potentially, be a sub-class of itself—in other words, contained in itself.

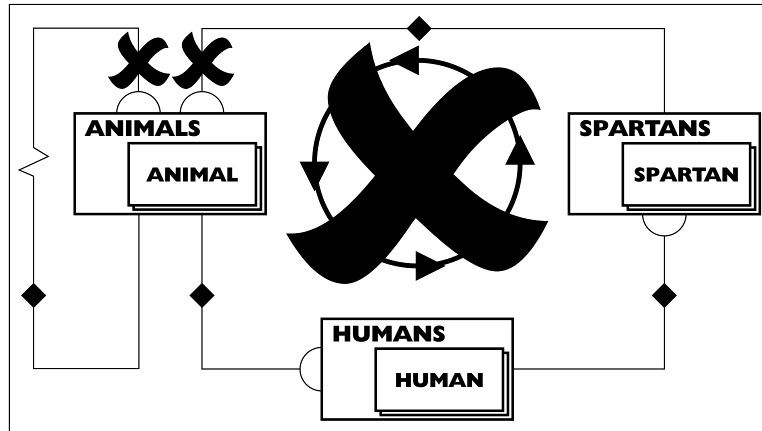


Figure 9.19: Impossible circular super–sub-class hierarchy

Like the class–member hierarchy, it is the nature of our understanding of space and time (and space-time) that makes this circularity impossible. This is shown by the reference diagram in ***Figure 9.20***. Normally, we illustrate the super–sub-class structure by having one class contained in another. However, as the figure shows, this will not work for a circular structure; instead, we show the sub-class connection using an arrow.

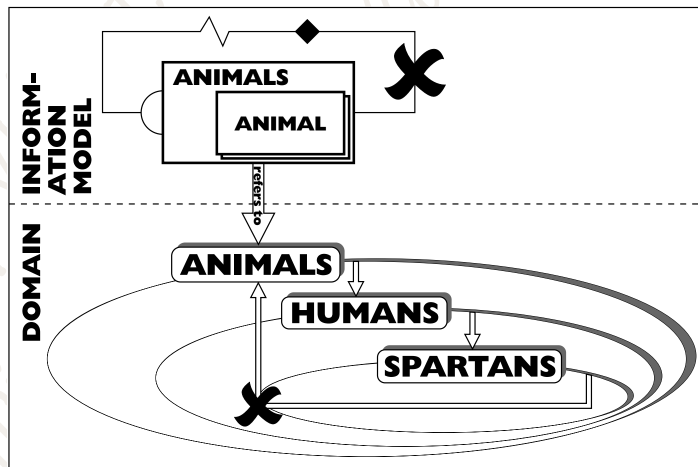


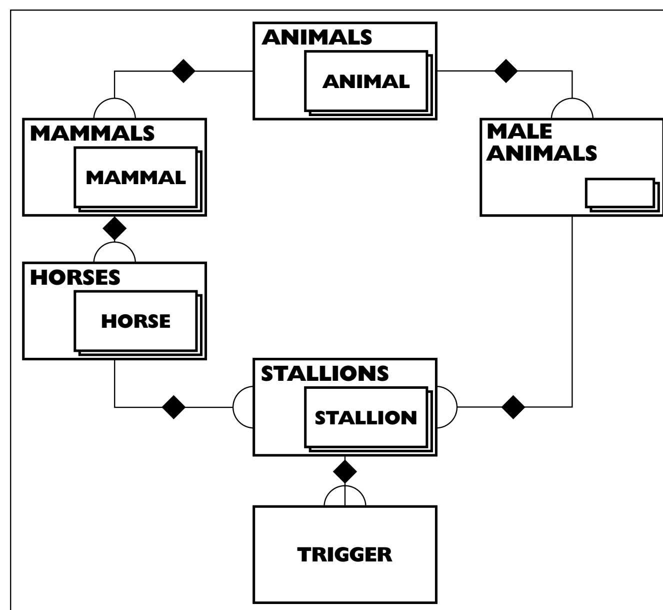
Figure 9.20: Impossible circular super–sub-class reference diagram

3.3.2.6 *Inheriting class membership*

As we would expect, the patterns for classes are webby; in other words, the patterns for super–sub-class and class–member intertwine. One pattern is particularly important; it is the inheritance of class membership up the super–sub-class hierarchy.

To see how this works, we introduce Trigger the horse into the model in ***Figure 9.21***. We naturally tend to make him a member of the class stallions (shown in ***Figure 9.21***). Stallions is the hierarchy's lowest class. However, Trigger is potentially a member of all the hierarchy's higher classes, but our natural instinct is not to model these possibilities.

Figure 9.21: Natural position for Trigger the horse

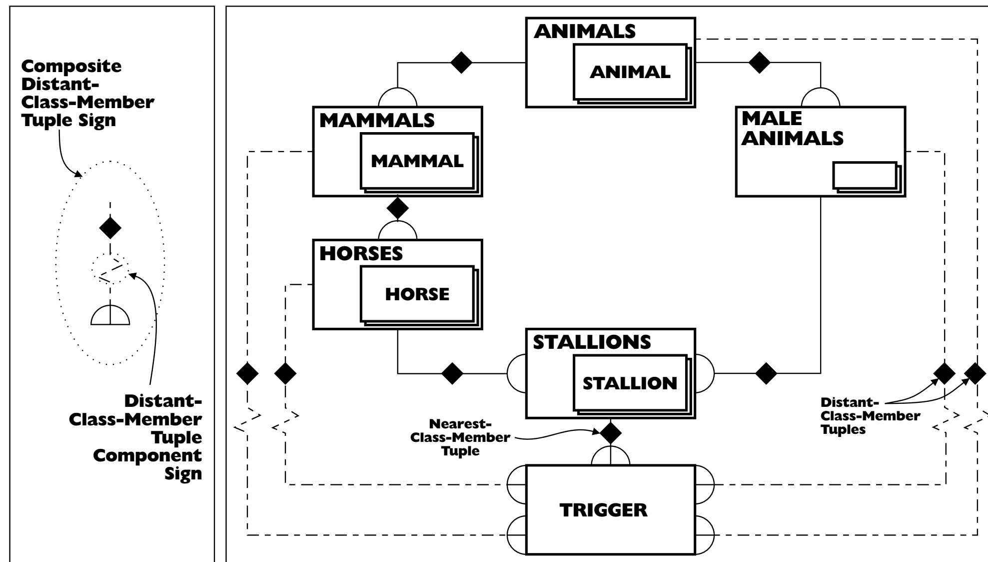


Why don't we model these potential higher class–member tuples? We had a similar situation to this earlier with child– and descendant–sub-classes. And the answer is the same here—they would clutter up the schema and the model. We can see this in ***Figure 9.22***, which shows the results of constructing all the class–member tuples for our example. The model is pretty cluttered and this is only a small hierarchy. A much larger hierarchy would be impossibly cluttered. We have a class–member, as well as a sub-class, clutter problem.

Figure 9.22 also illustrates a distinction between the sign for the lowest class–member tuple—now called the nearest-class–member sign—and the sign for other class–member tuples—now called distant-class–member signs. The distant-class–member signs use the same zigzag component sign as the earlier descendant–sub-class signs. Trigger's nearest class is stallions because there is no class below stallions in the super–sub-class hierarchy to which he belongs; so, the class–member tuple is a nearest-class–member tuple. Trigger is a distant-class–member of each of the classes horses, mammals, male animals and animals because there is a class below them in the class–member hierarchy, the class stallions, of which he is a member.

© Copyright chrispartridge.com

Figure 9.22: All Trigger the horse's member possible class–member tuples



3.3.2.7 Deducing more distant-class–member signs

As with child–sub-class signs, we can deduce and construct distant-class–member signs from the nearest-class–member sign. Like before, this is done logically, without involving any analysis of what the signs refer to. The converse is, of course, not true. We cannot work out a nearer class–member sign from a more distant sign. This makes the nearest-class–member sign key; from it we can calculate all the distant-class–member signs.

More generally we can construct a distant-class–member sign from the class–member sign and a chain of super–sub-class signs up from its class sign. The deduction has the following pattern;

A is a class–member of B
B is a sub-class of C
C is a sub-class of D
<u>D is a sub-class of E</u>
Therefore:A is a more distant-class–member of E

There can be any number of sub-class lines (except zero of course) in this calculation.

As with the earlier descendant–sub-class calculation pattern, this has the same pattern as one Aristotle's syllogisms (called barbara), which looks like this:

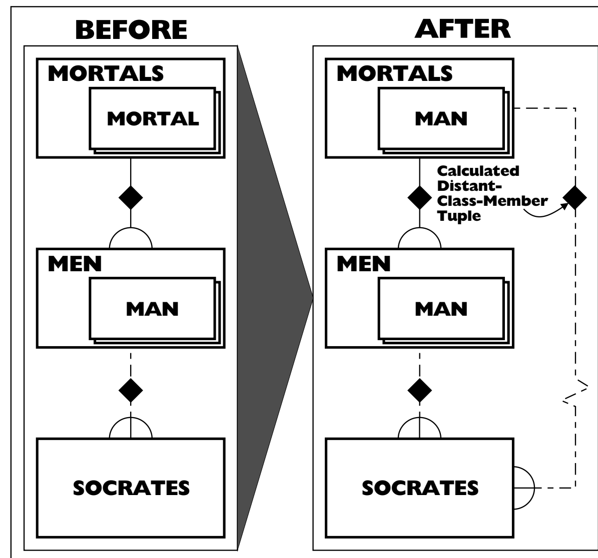
Socrates is a man,
<u>All men are mortal,</u>
So Socrates is mortal.

It is easier to see the resemblance when it is translated into class-speak—as below:

Socrates is a class–member of the class men,
The class men is a child–sub-class of the class mortals,
So Socrates is a more distant-class–member of the class mortals.

The distant calculation for Aristotle's syllogism is shown in ***Figure 9.23***.

Figure 9.23:
Aristotle's barbara syllogism



People who have not yet developed a clear idea of the difference between the class–member and super–sub-class patterns often see this distant-class–member calculation process as the same as the earlier descendant–sub-class calculation process. When they develop a clear understanding of the differences between the two patterns, they then begin to see the differences between, as well as the similarities in, the two processes.

3.3.2.8 Compact class–member hierarchy models

The distant-class–member deduction pattern works in a similar way to the earlier descendant–sub-class pattern. This provides us with an opportunity to use virtual signs again and tidy up the class–member clutter problem. An opportunity to get the benefit of having signs for all the descendant–sub-class tuples, without having to bear the cost of visibly recording them. Once I model the nearest-class–member tuple, I can assume that all the distant-class–member tuples also 'virtually' exist.

I can then adopt the strategy of only modelling the nearest-class–member tuples and essential distant-class–member tuples. The signs for the many other distant-class–member tuples are virtual. This can reduce the clutter in even the most complicated class–member hierarchy to an easily manageable level.

© Copyright Chris Partridge chris.partridge@BORO Centre.com

3.3.3 Super–sub-class tuples class

We naturally see the super–sub-class tuple as connecting classes. This is correct in one sense, the tuple can only connect classes. But, it does not mean the super–sub-class sign only connects class signs. If a class sign refers to a class of classes, then its member component refers to the member classes. Because these are classes they can be connected with their sub-class signs using the super–sub-class sign. Furthermore, because the member component sign refers to a class of members, the super–sub-class sign refers to a class of super–sub-class tuples.

Here is an example. Consider the Linnaean biological scheme used to classify individuals into species and species into genera (singular—genus). This two-level structure is reflected in the Linnaean names for species. For instance, our species is *homo sapiens* where *homo* is the genus and *sapiens* the species within genus. This gives us an example of a super–sub-class tuple class between classes' members.

At the classes of classes level we have two classes; genera and species. The class genera has individual genus classes, such as *homo* as members. The class species has individual specie classes, such as *homo sapiens* as members. At the classes of individual objects level, we also have two classes; *homo* and *homo sapiens*. The class *homo sapiens* is a sub-class of the class *homo* as shown in ***Figure 9.24***. This particular super–sub-class pattern is just a particular example of a more general pattern. The members of the class species are sub-classes of the members of the class genera. This is a super–sub-class tuples class between the classes' members. Because it refers to all the different individual tuples between the various members, it is a class of tuples not an individual tuple. This is reflected in its sign, which uses a tuples icon instead of a tuple icon (shown in ***Figure 9.24***).

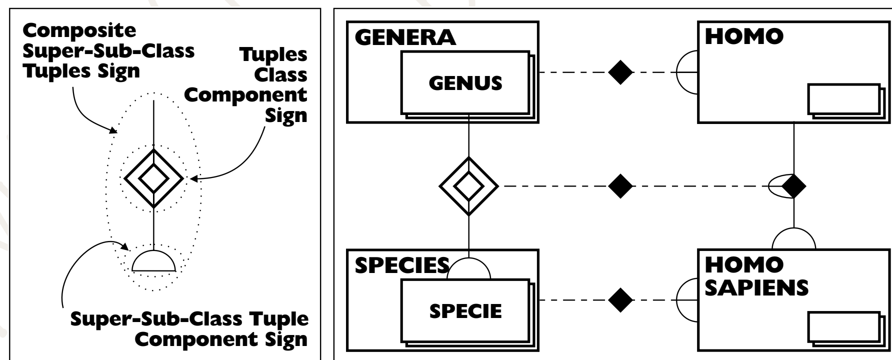


Figure 9.24: The super–sub-class tuples class sign

4 Constructing signs for tuples

We have finished looking at the notation for classes, an object with internal structure resulting from its construction from other objects. We now look at another constructed object with internal structure, the tuple object.

© Copyright chris.partridge BORO Centre.com

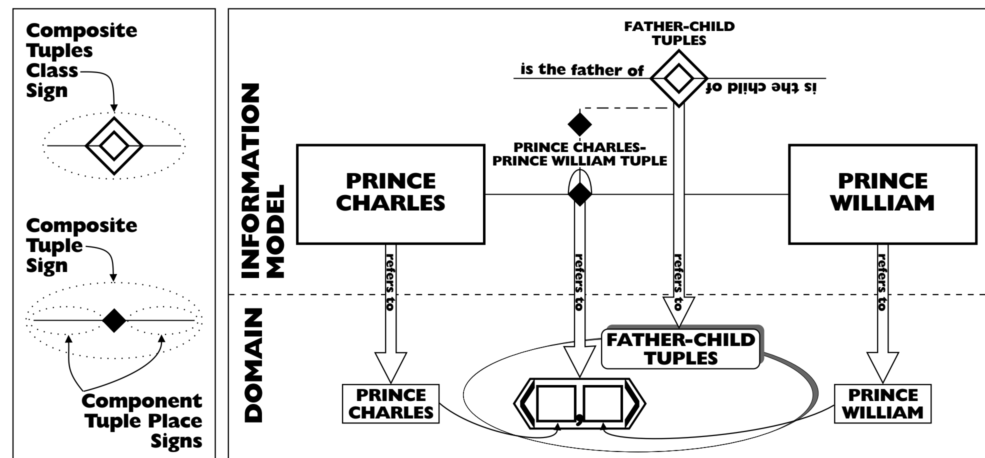


Figure 9.25: Tuple and tuple class signs

4.1 Constructing a tuple of individual objects and a tuples class

For our purposes, tuples exist with an associated tuples class. So we model the sentence ‘Prince Charles is the father of Prince William’ with a tuple and a tuples class (shown in ***Figure 9.25***).

Note these points:

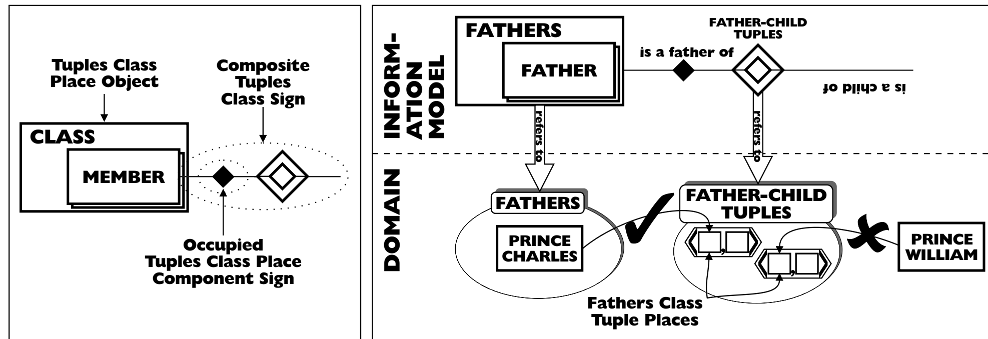
- The solid black diamond is the component sign for the tuple.
- The connecting lines from the tuple component sign to other signs are called tuple place component signs.
- The tuple place component signs connect the sign for the tuple with the signs for the objects out of which it is constructed. These are called the tuple place objects. In the Prince Charles—Prince William tuple, the places are occupied by individual objects, but they could be occupied by any type of object.
- The component sign for a tuples class is two hollow diamonds, one inside the other.
- The lines from the component tuples class sign are called the (tuples) class place component signs.
- The father–child tuples class is a class object and so uses the standard class–member sign to link to its member tuple sign.

4.1.1 Occupied class place signs

A (tuples) class place is said to be occupied when its tuples class sign is connected to another object. For instance, the fathers class is connected to the father–child tuples class in ***Figure 9.26***. This occupation is signed by adding the component tuple sign, a solid black diamond, to the (tuples) class place component sign. The object to which the tuples class is connected is called a (tuples) class place object, an example is the fathers class in ***Figure 9.26***. Notice that the ‘is a child of’ class place sign in ***Figure 9.26*** does not have a black diamond component because it is not occupied.

© Copyright Chris Partridge

Figure 9.26: Class place constraints on tuple places

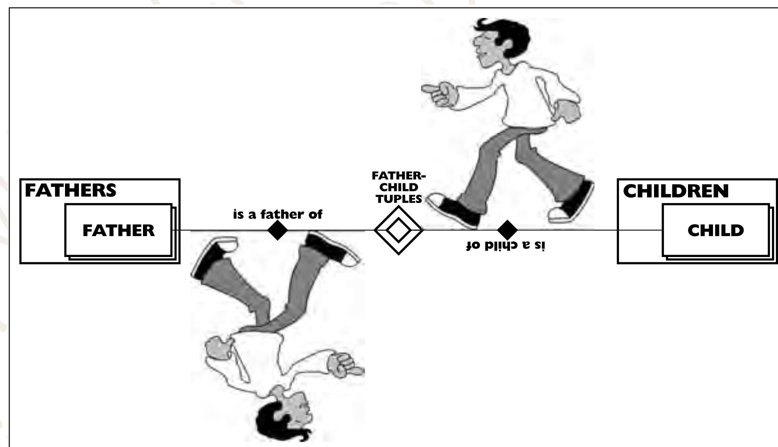


4.1.2 *Occupied class place constraints on tuple places*

A class place object constrains which tuples can be members of its tuples class. In the example in ***Figure 9.26***, the fathers class is a place object, which implies the existence of a fathers tuple place in the tuple members of the tuples class. In simpler language, this means that one of the places of each tuple member of the tuples class is designated a father tuple place and must be occupied by a member of the father's class.

In the example illustrated in ***Figure 9.26***, the first place in the couple is designated the father couple place. This means that a couple with Prince Charles in its first place (<Prince Charles, ?>) can be a member of the father-child tuples class because Prince Charles is a member of the class fathers. But any couple with the format <Prince William, ?> cannot be a member, because Prince William is not a member of the fathers class.

Figure 9.27: Convention for reading tuple names



4.1.3 *Tuple and tuples class names*

Tuples classes have names in the same way as other classes. However, in addition, both tuples and tuples classes have a name constructed from the names on their class place signs. The convention for constructing these names is that one of the class place signs is picked and then a mental walk is taken along the class place (or place) sign to

the tuples class (or tuple) sign reading the text on the left and then along to the next class place (or place) sign.

In the example in ***Figure 9.27***, there are two 'walks'. We can start at the father member sign, and mentally walk past the father–child tuples to the child member sign, reading the 'is the father of' text on the left. This gives us the name 'father—is the father of—child'. Mentally walking the other way, from child to father, would give us the name 'child—is child of—father'.

4.2 Tuples classes inheriting patterns from classes

We can now begin to take advantage of the power that re-using patterns brings. We can re-use the class pattern on tuples classes. As we have just seen, they are classes—in class-speak; they are a sub-class of the class classes. So they inherit all the characteristics of a class and share all its patterns. For instance:

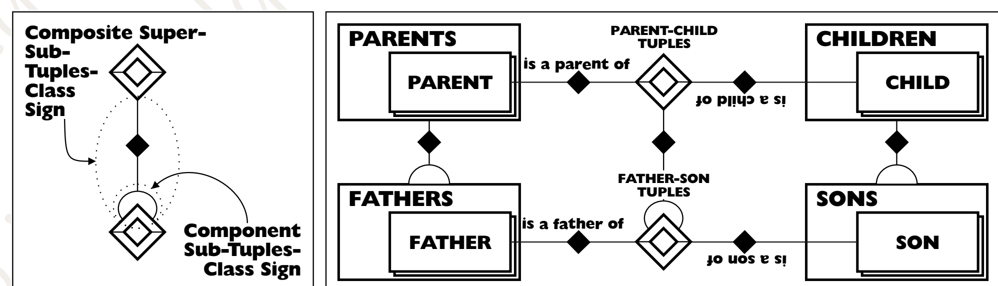
- They have tuple super–sub-class and tuple class–member hierarchies.
- They have child– and descendant–sub-tuples-classes.
- They have nearest– and distant– class–member tuples.

They will also automatically inherit any new class patterns we construct (for example, the distinct and overlapping patterns we examine in the next chapter). Tuples classes inherit all this as the result of being a class object. We now look at an example of a class pattern being re-used for tuples classes, the tuple super–sub-class hierarchies.

4.2.1 Tuple super–sub-class hierarchies

As tuples classes are classes they can also have super– and sub-classes. For instance, parent–child tuples is a super-class of father–son tuples (shown in ***Figure 9.28***). Notice that the super–sub-class tuple uses the standard super–sub-class tuple sign.

Figure 9.28: Super–sub-tuple-class sign



4.2.1.1 Modelling super-sub place classes

Care needs to be exercised when working out the super–sub-class tuples between the place classes of tuple super- and sub-classes. This tends to come with practice. In particular, as one moves from a tuple subclass to a tuple supercools, the place classes

should either remain the same or move up to a super-class. This is easiest to explain with an example.

Look at ***Figure 9.28***, the father–son tuples class has as one of its class place objects, the fathers class. Its super-class, parent–child tuples, has as its corresponding class place object, the parents class. As parents is a super-class of fathers, we can go in a full circle. Starting from father–son tuples we go along to fathers, up to parents, along to parent–child tuples and back down to where we started, father–son tuples. This works because place classes of a tuples super-class need to be either the same as or super-classes of the corresponding place classes of their tuples sub-class.

For an example of incorrect modelling, look at ***Figure 9.29***. We cannot trace a full circle here because mothers is (rightly) not signed as a super-class of fathers. The problem here is that mother–child tuples has been signed incorrectly as a super-class of father–son tuples.

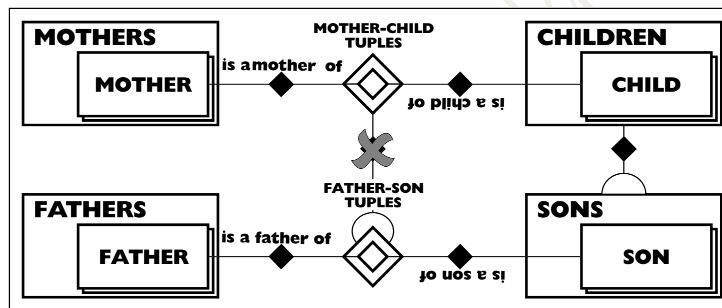


Figure 9.29: Incorrect super–sub-class tuple

5 Constructing signs for whole–part tuples

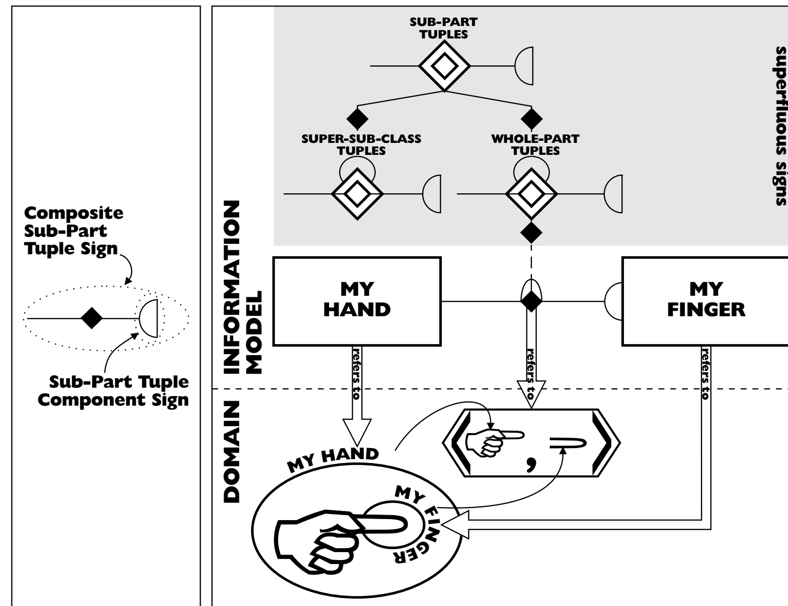
In Part Four, we noted how important whole–part tuples were to object semantics. This is recognised in the notation by giving whole–part tuples their own sign. We look at it now, along with the patterns of the underlying whole–part tuples that it is used to sign. We noted, in ***Chapter*** , that the whole–part pattern is similar to the super–sub-class pattern. Here we see more evidence of this.

5.1 What are whole–part tuples?

But first let us remind ourselves how whole–part tuples fit into the class framework. Consider this example. My fingers are part of my hand. This means that there is a connection between my fingers and my hand. Using the same analysis as we used for general tuples above, this is a couple object <my fingers, my hand>, which is a member of the whole–part tuples class. This analysis is shown in ***Figure 9.30***. Because the whole–part couple has its own sign, the whole–part tuples class sign and its class–member sign are redundant. They are only included in this model to make absolutely clear what the whole–part tuple is.

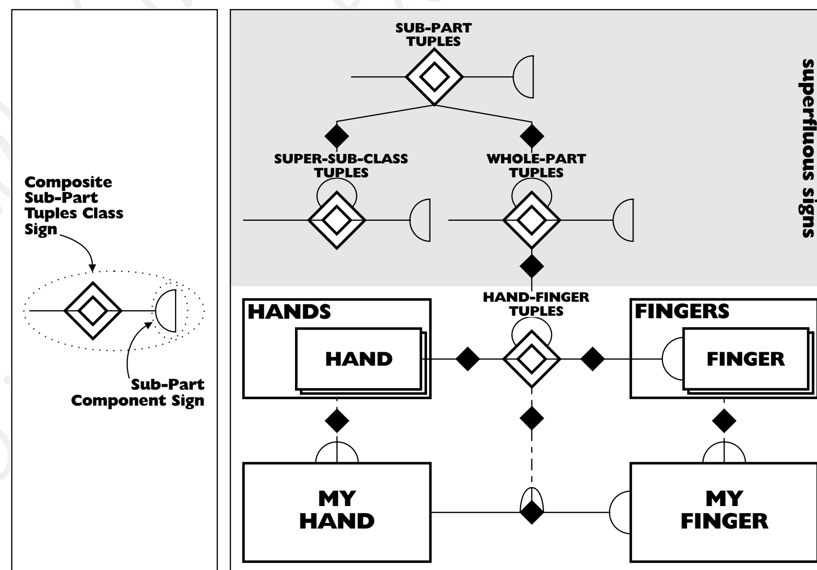
© Copyright Chris Partridge christhepartridge@BORO-Centre.com

Figure 9.30: My fingers are part of my hand



The composite whole–part sign is constructed from familiar components. Because the couple is a tuple, we use a tuple sign for it. As we mentioned above, the whole–part and the super–sub-class tuples are similar kinds of tuples, operating at different levels. So they have the same composite sign. Until now, we have called this the sub-class sign, but as we are generalising it across sub-class and whole–part, we rename it the sub-part sign.

Figure 9.31: Fingers are part of hands



The example above is of particular individuals. There are also classes whose members have whole–part patterns. We can extend the example to illustrate this. Fingers are parts of hands—in class-speak; the individual members of the fingers class are parts of

the individual members of the hands class. How do we sign this whole–part tuple between members of a class? We use the individual whole–part sign, but suitably amended to show that it is the sign of a tuples class instead of a tuple – as shown in ***Figure 9.31***). Like the tuple level sign, the whole–part tuples class sign and its class–member sign are superfluous because the hand–finger tuples is signed as a whole–part tuples class.

5.2 Individuals whole–part tuple hierarchy

The individual whole–part tuples create an individual whole–part hierarchy. For instance, my fingers are part of my hand, my hand is part of my arm, and my arm is part of my body (shown in ***Figure 9.32***). As we can see, this is, in many ways, a super–sub-class hierarchy for individual objects.

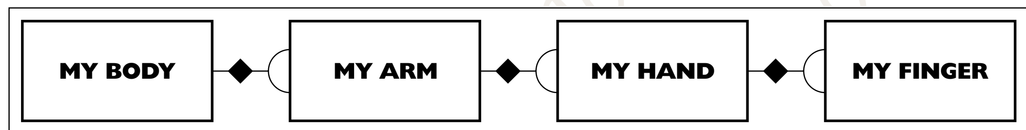


Figure 9.32: Individual whole–part tuple hierarchy

5.3 Classes whole–part tuple hierarchy

Individual whole–part hierarchies can be generalised into whole–part tuples class hierarchies. For instance, the individual whole–part hierarchy shown in ***Figure 9.32*** can be generalised to the class level (shown in ***Figure 9.33***.)

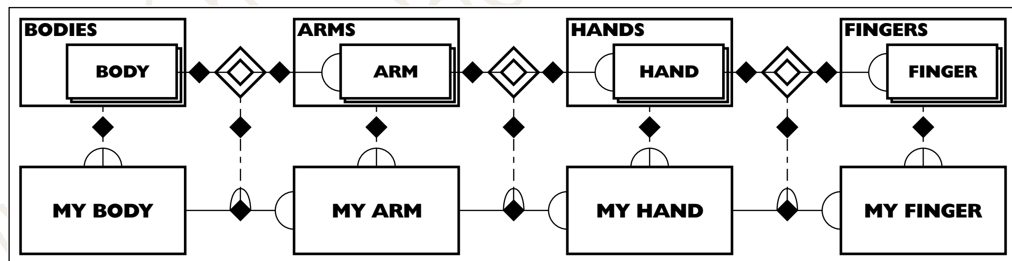


Figure 9.33: Whole–part tuple class hierarchy

5.4 Child– and descendant–parts

Just as we drew a distinction between child– and descendant–sub-classes in the super–sub-class hierarchy, we draw a corresponding distinction here between child– and descendant–parts. To see this, we add all the potential whole–part tuples to the model in ***Figure 9.32*** (see the result shown in ***Figure 9.34***).

A child–part is one that has no intervening parts (in the particular model being considered). Descendant–part tuples are ones with intervening parts. For example, in Figure 9.34, 'my fingers are part of my hand' is a child–part tuple. Whereas, as 'my finger is part of my arm' is a descendant–part tuple because it has my hand as an intervening part.

© Copyright chris partridge, BoroCentre.com

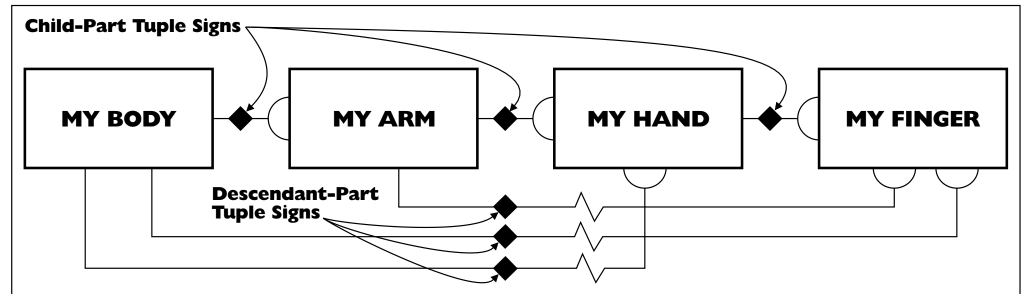


Figure 9.34: Child– and descendant–part tuples

5.5 Deducing descendant–part signs

Like descendant–sub-class tuples, a descendant–part tuples sign can be deduced from the child–part tuple signs and more generally from whole–part tuple signs. This deduction has the same pattern as the descendant–sub-class deduction:

A is a whole–part of B
B is a whole–part of C
C is a whole–part of D
<u>D is a whole–part of E</u>
Therefore:A is a descendant–part of E

Where there can be any number of whole–part lines (except zero and one of course). An actual example is:

My finger is a part of my hand, and
<u>My hand is a part of my arm</u>
ThenMy finger is a (descendant–) part of my arm.

Figure 9.35 shows this deduction graphically.

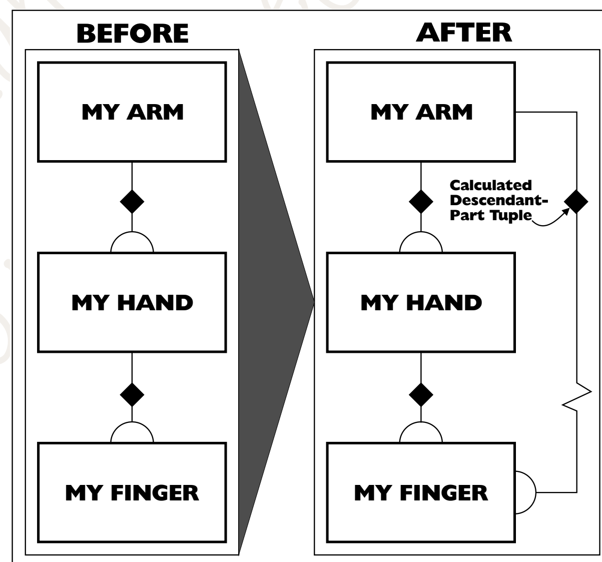


Figure 9.35: Descendant–part tuple deduction

© Copyright Chris Partridge
chris@BOROCentre.com

6 Constructing signs for dynamic objects

So far we have been constructing signs for timeless objects. We now look at signs for the time-bound dynamic objects described in ***Chapter 8***:

- the here event class,
- the now event class, and
- current couples.

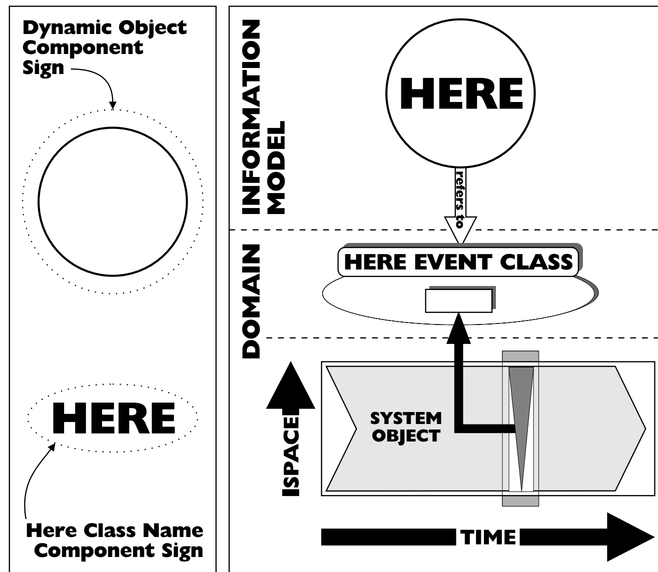


Figure 9.36: Sign for the 'here' event class

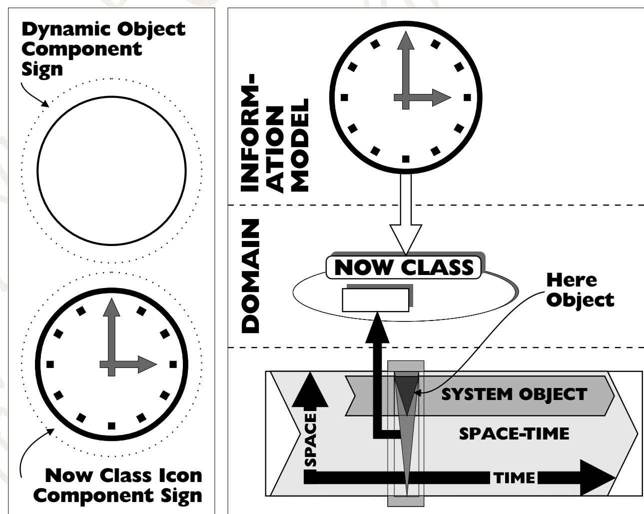


Figure 9.37: Sign for the 'now' event class

© Copyright Chris Partridge @ BORO Centre.com

6.1 Constructing a sign for the 'here' event class

The here event class has as its single member an instantaneous time-slice of the system object—a physical body. This member moves, like all the dynamic events, along the time dimension with the system's 'consciousness'. The composite sign for the here event class is a circle, the component sign for a dynamic object, with the name 'HERE' in it (shown in ***Figure 9.36***).

6.2 Constructing a sign for the 'now' event class

The 'now' event class has as its single member the instant that contains the 'here' event class's member. It is signed using a circle containing a clock face – as shown in ***Figure 9.37***.

6.3 Constructing a sign for a current tuple

We now construct the signs for the current tuples class and its members, current tuples. However, the current tuples class sign is, to an extent, superfluous because any tuple signed as current automatically belongs to the current tuples class. This is done using a component dynamic circle sign (illustrated in ***Figure 9.38***). As you can see, one of the sign's current tuple places is linked to the now event class, the other(s) to the object(s) currently classified as current.

You can also see the current tuples class signed in ***Figure 9.38*** as a tuples class with the dynamic circle component sign around it.

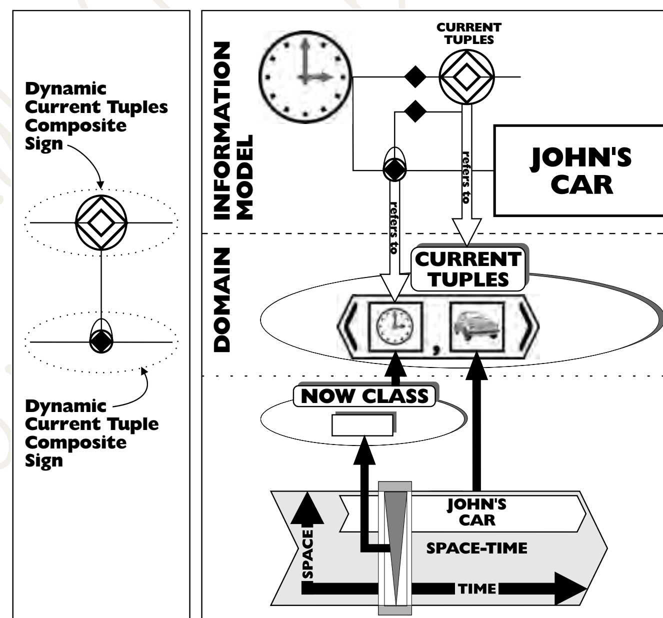


Figure 9.38: Sign for a current tuple

7 Signs as objects—modelling the model

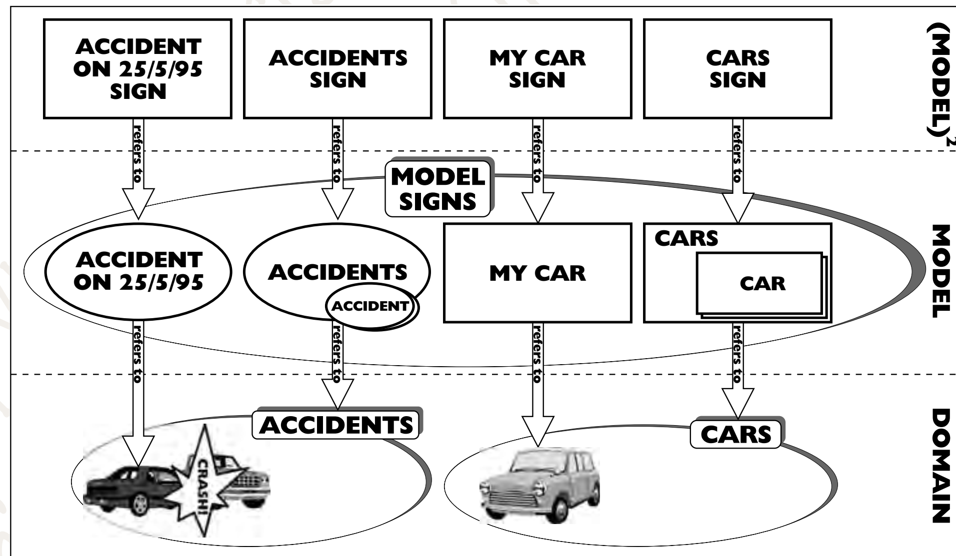
Object semantics applies to signs in the model as well as the objects that are modelled. According to object semantics, everything is an object with four-dimensional extension. Even the signs in the model are objects—they are model objects. We can see this clearly when we start modelling the model. This is not meta-modelling, this is more like modelling x modelling or (modelling)^2.

This (modelling)^2 clarifies one aspect of modelling that people sometimes find confusing. This is that the type of an object (for example, body or event), and the type of the model object that models it, are often quite different. This confusion about types sometimes manifests itself as a belief that the distinction between data and process in information systems (the signs in the model) reflects the distinction between objects and events in the real world. This resolves itself into a belief that data reflects objects and process reflects events. We discussed how mistaken this belief is in ***Chapter 2*** .

7.1 A (modelling)^2 model

Look at the (modelling)^2 model in ***Figure 9.39***. It models examples of the four major types of signs in our object notation; the individual body and event objects and the bodies and events classes. As the model shows, all these signs (model objects), are all individual physical bodies, whatever they refer to—whether event, class or body.

Figure 9.39: Modelling body and event model objects



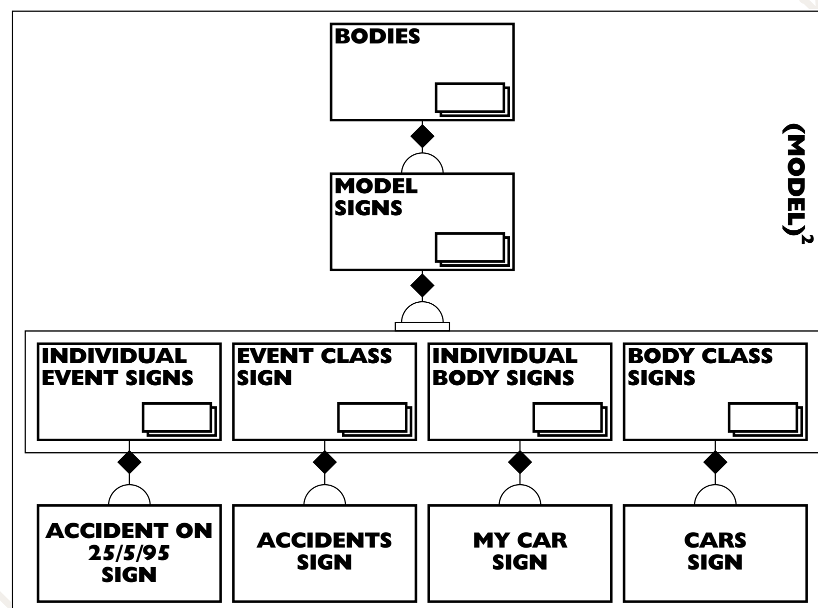
The model object for my car is an individual body sign. This sign is, like the body object it refers to, an individual body object in its own right. It has extension, it persists through time—though maybe not for as long as the body object it refers to. Individual body signs are the only type of model object where the model object and the object it refers to are of the same type.

© Copyright chris partridge BORO Centre.com

The individual event sign is, like the individual body sign, an individual body. It has extension and it persists through time—it has spatial and temporal dimensions. However, the event model object, unlike the individual body object, is not of the same type as the object it refers to. The 'accident 25/5/95' sign is a body object with temporal extension; the accident it refers to is an event that does not persist through time.

Model class objects, like the model individual objects, are individual bodies and so different in type from the objects they refer to. The event and body class examples in ***Figure 9.39*** illustrate how constructed objects with an internal structure, such as the two class objects, are flattened out in the object model into individual physical body objects. This (model)^2 structure is illustrated in ***Figure 9.40***.

Figure 9.40:
(Model)^2 objects

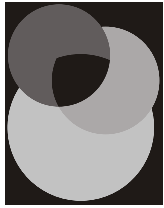


8 What's next

We have now looked at signs for all the major types of individual objects that we need to business object model. We have got a feel for what they mean and how they work. We are well on our way to being ready to start business modelling. In the following chapter, we look at the syntax of business object patterns. We see how we can use the object notation to model patterns of business objects.

© Copyright chris partridge.com

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 10

Constructing Signs for Business Objects' Patterns

1 Introduction
2 Patterns for the connections between extensions
3 State hierarchy patterns
4 Time ordered temporal patterns
5 Cardinality patterns for tuples classes
6 A pattern for compacting classes
7 Where we are

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction

In the previous chapter, we saw how to construct signs for the basic types of objects in object semantics. In this chapter, we step up a level. Instead of looking at individual signs, we look at the syntax of signs that describe patterns of business objects. We examine how this syntax works using the following examples of fundamental patterns found in our investigations of object semantics in Part Four:

- Patterns for the connections between extensions,
- State hierarchy patterns,
- Time ordering patterns,
- Cardinality patterns for tuples classes, and
- Patterns for compacting classes.

2 Patterns for the connections between extensions

Extension is a central notion of logical and object semantics. Many of the patterns we have analysed so far turn out to have structures based on it. For instance, the sub-part tuple (the generalised whole–part and super–sub-class tuple) is based on the extension of one of the related objects containing the other.

Closely related to sub-part tuples are two other patterns based on structural connections between extensions: the distinct and overlapping patterns. These two patterns occur at two levels:

- The individual object level, and
- The class level.

This is similar to the sub-part pattern, which is the whole–part pattern at the individual object level and the super–sub-class pattern at the class level. Let's now investigate these patterns, starting at the individual object level.

2.1 Individual object level patterns

At the individual object level, any number of individual objects can have the distinct or overlapping pattern, but the pattern is at its simplest when only two objects are involved. So we start by looking at pairs of distinct and overlapping objects then move onto larger groups of objects. Finally, we examine the following associated patterns:

- Inheriting distinct and overlapping patterns,
- Known and unknown distinct and overlapping individual objects,
- Partition patterns for distinct individual objects,
- Intersection pattern for overlapping individual objects, and
- Fusion pattern for individual objects.

We also work out what objects the signs for distinct and overlapping individual patterns refer to.

© Copyright Chris Partridge chris.partridge@BOROCentre.com

2.1.1 *Distinct pairs of individual objects*

Two individual objects that do not have any spatio-temporal parts in common are distinct. For example, my car and me are distinct—no part of my car is also a part of me. We model this distinct pattern with the sign shown in ***Figure 10.1***.

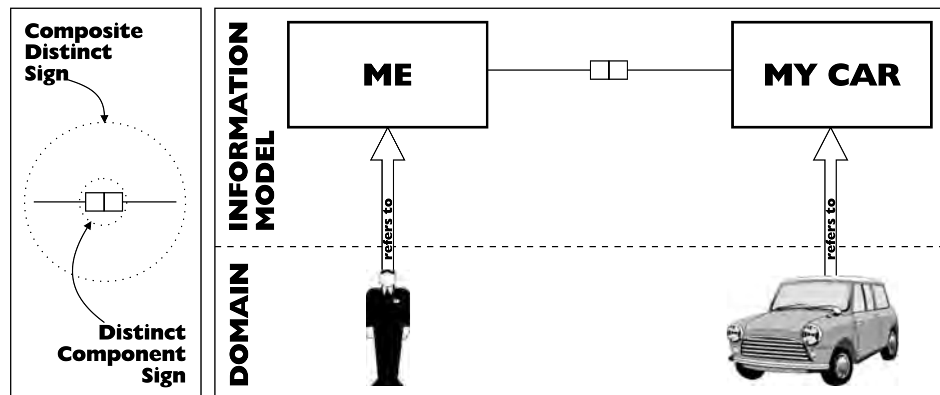


Figure 10.1: Distinct individual objects sign

2.1.2 *Overlapping pairs of individual objects*

A pair of individual objects that have parts in common overlap. For example, the island of Ireland and the country United Kingdom overlap; the country of Northern Ireland is a part of both individual objects. We model this overlapping pattern with the sign shown in ***Figure 10.2***. (This and subsequent examples involving countries use our simple intuitive view of country objects. We re-engineer a more sophisticated view in Part Six's worked examples.)

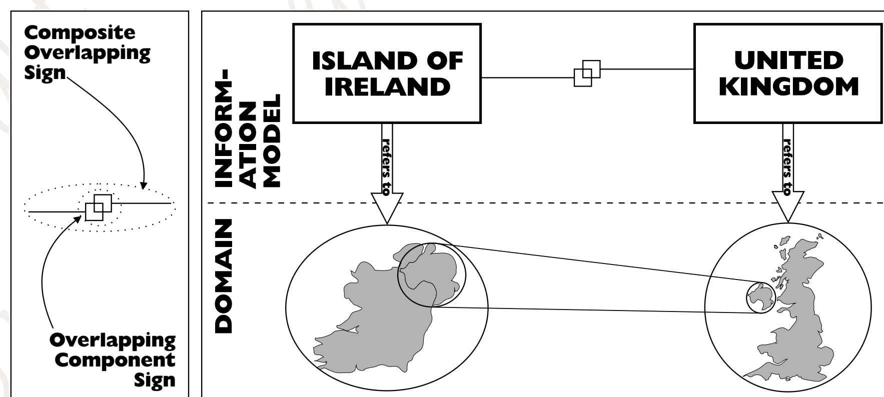


Figure 10.2: Overlapping individual objects sign

2.1.3 *Three main types of connection for pairs of individual objects*

We have now looked at what are, from an extension point of view, the three main patterns of connection between pairs of individual objects; distinct, overlapping and whole–part. As illustrated by ***Figure 10.3***, a pair of individual objects must fall under one of these patterns. It could be argued that the whole–part pattern, where one individual

object completely contains another, is an extreme case of overlapping. However, the convention is to consider these as separate patterns with their own signs.

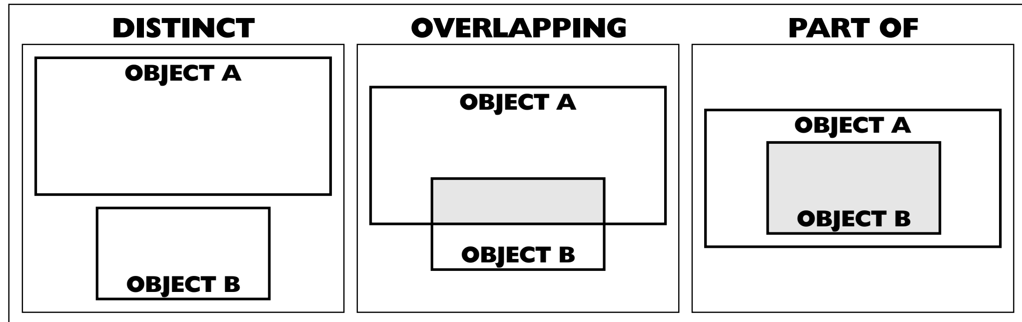


Figure 10.3: Pattern for individual objects

2.1.4 *Larger groups of individual objects*

Groups of individual objects larger than two can have a variety of patterns of connection. All the individual objects can be distinct [as shown schematically in ***Figure 10.4*** (a)]. Or they can all overlap [shown in ***Figure 10.4*** (b)]. It is also possible that some will be distinct and others will overlap. Furthermore, it is possible that even if every pair in a group of individual objects overlap, the whole group will not overlap [shown schematically in ***Figure 10.4*** (c)]. The same is not true for distinctness; if every pair is distinct, then the whole group is distinct.

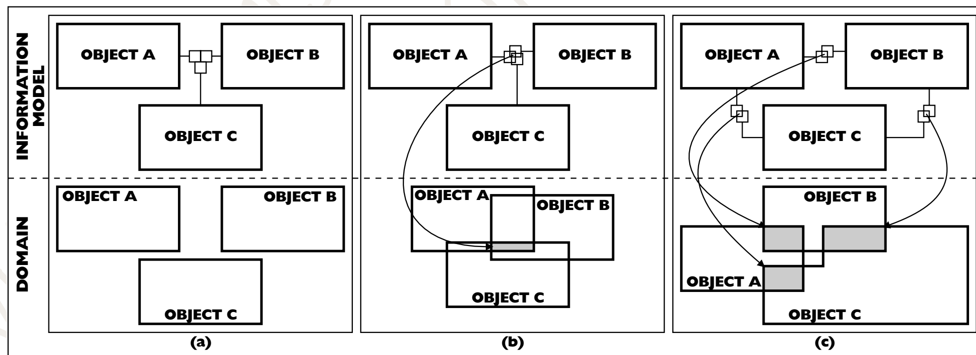


Figure 10.4: Schemas for larger numbers of individual objects

2.1.5 *Inheriting distinct and overlapping patterns*

Distinct and overlapping patterns for individual objects are inherited in opposite directions along the whole–part hierarchy. Distinctness is inherited down the hierarchy. So, as the United States and France are distinct, their parts—for example, Texas and Bordeaux—are also distinct. This is modelled in ***Figure 10.5***. The model also shows NATO and the EEC (which have the United States and France as parts) overlapping, proving that distinctness is not inherited up the whole–part hierarchy.

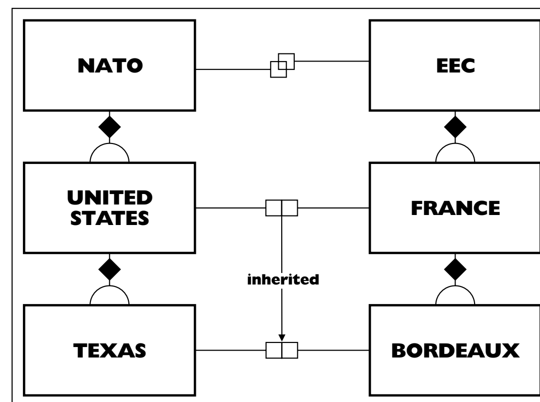


Figure 10.5: Inheriting distinctness

Overlapping is inherited up the whole–part hierarchy. So, as London and the River Thames overlap, any wholes of which they are parts also overlap. For instance, South-East England and the River Thames and its tributaries overlap. Overlapping, however, is not inherited down the hierarchy (illustrated by the distinct City of London and Thames Estuary in the model in ***Figure 10.6***).

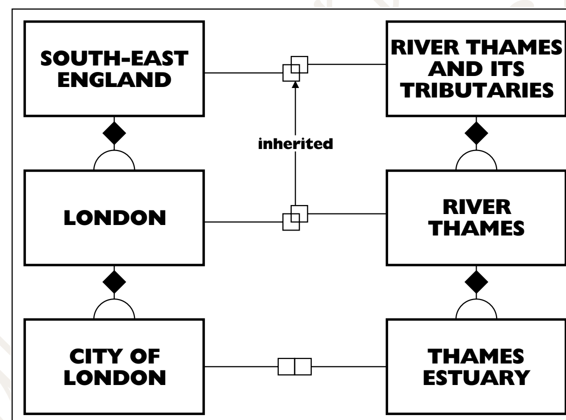


Figure 10.6: Inheriting overlapping

This inheritance has implications for how we model. I have found it useful to push the distinct connections as far up the whole–part hierarchy as they will go and the overlapping connections as far down the hierarchy as they will go. This increases the number of objects that can inherit the pattern and so automatically increases the functionality of the model. It also compacts the model as it replaces a number of lower-level distinct connections (higher-level overlapping connections) with a single connection.

2.1.6 Known and unknown distinct and overlapping patterns

As mentioned earlier, a pair of individual objects must either be distinct, overlap or one part of the other. However, we do not always know which pattern holds and sometimes cannot find out without considerable analysis. In many cases, it is not worth the effort of finding out and we can leave the point unresolved. In this situation, we model our igno-

© Copyright chris.partridge@borovie.com

rance with a lack of signs.

A more subtle ignorance occurs when two individual objects are signed in the model as overlapping, but appear distinct because no common part objects are signed. For example, the model in ***Figure 10.2*** signs the island of Ireland and the country, the United Kingdom, as overlapping but does not contain an overlapped object that is a part of the two objects. However, this does not imply that the objects are distinct, just that the model does not 'know' any of the parts in the overlap.

2.1.7 The distinct and overlapping individual objects pattern objects

According to object semantics, we should be able to point to the objects referred to by a model's signs. None of the signs should refer to mysterious unknowable objects. This raises the question of what objects the distinct and overlapping signs refer to. Take, for example, the distinct sign in ***Figure 10.1***. What object does this refer to?

The distinct and overlapping individual object signs work in a similar way to the individual whole–part sign and most other pattern signs. They refer to an object and its class. It is tempting to suggest that as we talk about distinctness as a connection, that the distinct sign should, like the whole–part sign, refer to a tuple object. This will not work because the distinct and overlapping patterns are, unlike the whole–part pattern, symmetric.

This means that saying 'A is distinct from B' is no different from saying 'B is distinct from A'. (Saying 'my hand is part of my arm' is different from 'my arm is part of my hand'.)

We can see how this causes a problem with an example. Consider the distinct sign in ***Figure 10.1***. Assume that this refers to the tuple <me, my car>. The couple <me, my car> is belongs to the distinct tuples class. We have no guarantee that the couple <my car, me> also belongs to the distinct tuples class. This raises the decidedly contradictory possibility of me being distinct from my car, while at the same time my car is not distinct from me.

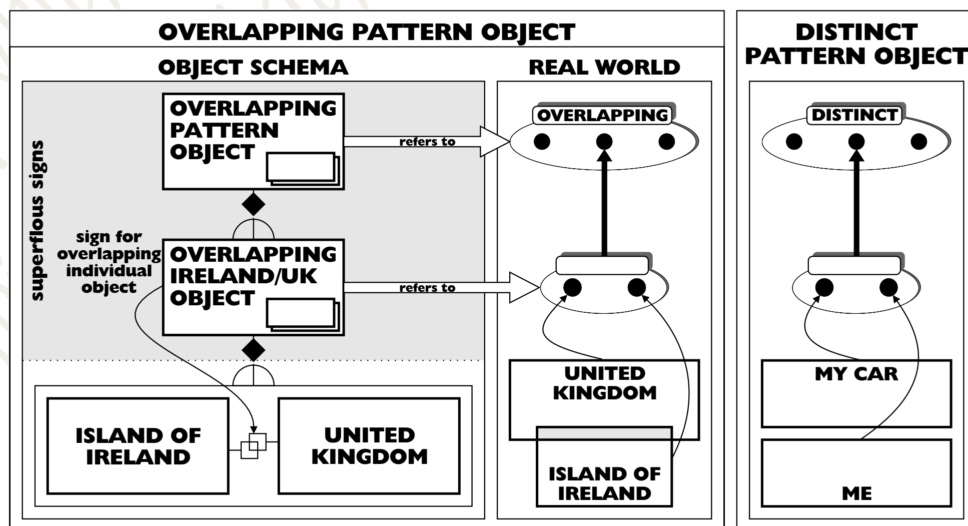


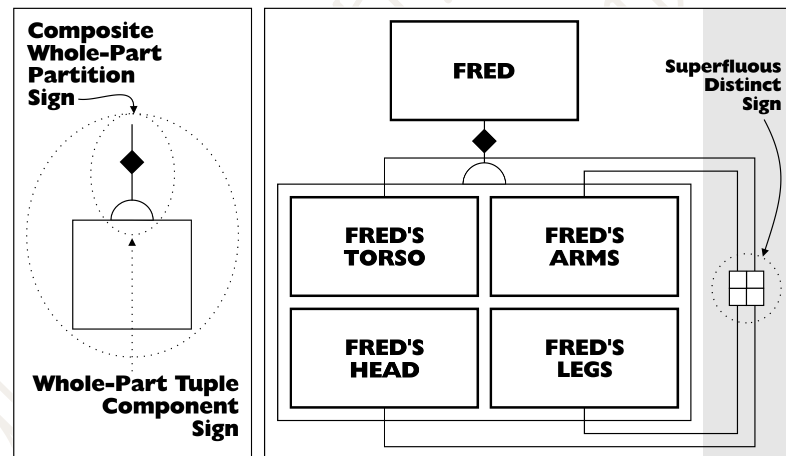
Figure 10.7: Individual object examples of distinct and overlapping pattern objects

The distinct and overlapping objects are actually the classes of the distinct (or overlapping) objects. In the example, the distinct object is the class {me, my car}. Our earlier problem is resolved, because, unlike a tuple, members of a class are not ordered; {me, my car} is the same class as {my car, me}. The distinct and overlapping pattern objects, are then classes of classes—the class of distinct class objects and the class of overlapping class objects. Examples of the two pattern objects are diagrammed in ***Figure 10.7***.

2.1.8 Partitioning patterns for distinct individual objects

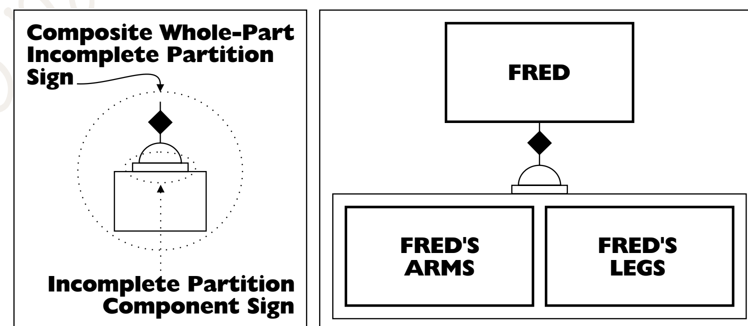
Distinct patterns, particularly useful distinct patterns, frequently arise from the partition of an object into distinct parts. We find this a natural way of seeing. For instance, when we see a person, we are almost instinctively already partitioning them—arms (hairy), legs (long), face (round), etc. The partitioning objects are distinct parts of the whole object and we can model this by combining the whole–part and distinct patterns into a partition pattern. The sign for the composite pattern is shown in ***Figure 10.8***. The component whole–part tuple sign describes the whole–part element and the partition box, the distinct element. Individual objects contained within the partition box are distinct.

Figure 10.8:
A partitioned individual object



When we model, we often do not want to partition an individual object completely; we only want to look at some of its parts. Then, we use a partial or incomplete partition. We sign the incompleteness with a partial sign (a small flat rectangle) between the whole–part sign and the partition box (shown in ***Figure 10.9***).

Figure 10.9:
An incompletely partitioned individual object



2.1.9 Inheriting partition patterns

Individual object partitions are inherited down the whole–part hierarchy as the example in ***Figure 10.10*** shows. The partition of the United States into the Northern United States and the Southern United States is inherited by the Western United States—giving us the North-Western and South-Western United States.

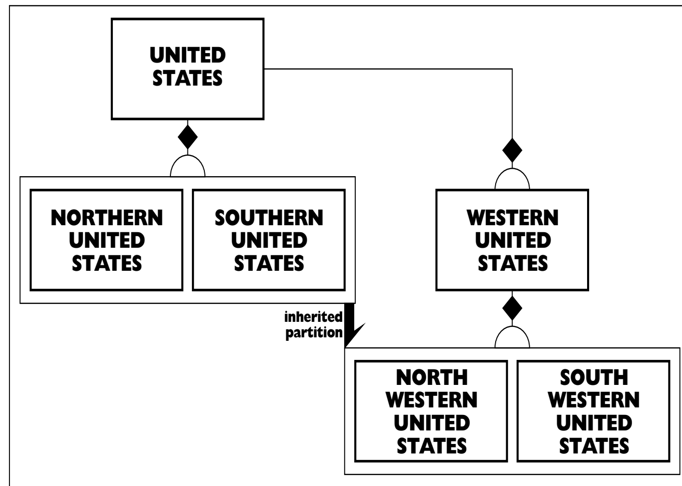


Figure 10.10: Individual object partition inheritance

As with distinct and overlapping inheritance, this has implications for how we model partitions. I have found it useful to push the partitions as far up the whole–part hierarchy as they will go. This increases the number of objects that can inherit the pattern, and so automatically increases the functionality. It also compacts the model, eliminating the need for a number of lower level partitions.

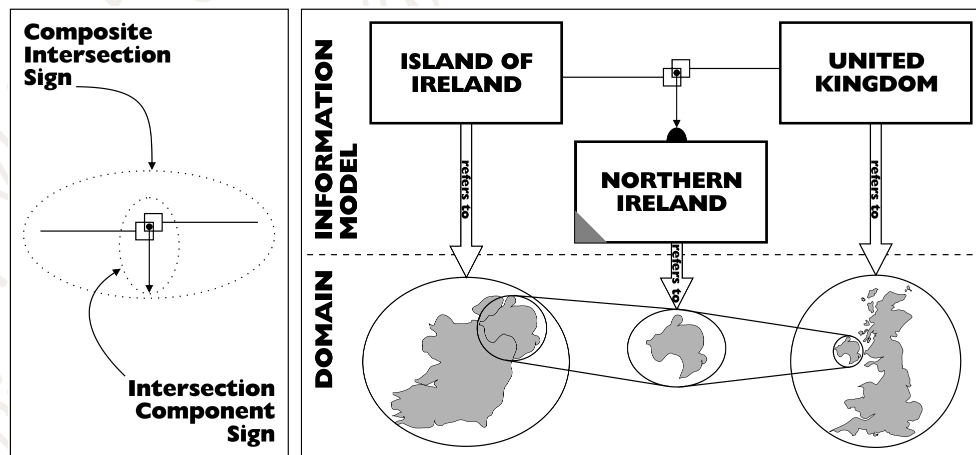


Figure 10.11: Intersected individual object

2.1.10 Intersection pattern for overlapping individual objects

Sometimes we take two overlapping individual objects and recognise their overlapping part as an object. This pattern is called an intersection and is signed in the model. In the

example shown in ***Figure 10.11***, we sign the intersection of the island Ireland and the country, the United Kingdom, to give the country Northern Ireland.

The intersecting object, the country Northern Ireland, is logically dependant on the intersected objects. This is signed in the model in two ways. First, this is shown by a logical dependency component sign. This is a black semi-circle at the intersecting object end of the composite intersection sign (shown in Figure 10.11). Second, Northern Ireland is signed as derived with a grey triangle in the bottom left corner of the Northern Ireland sign. This derived component sign is needed because, when the Northern Ireland sign appears on a schema that does not have both the Ireland and the United Kingdom signs, we cannot draw its intersection sign and so its logical dependency sign. Then, the derived sign reminds us that it is logically dependent.

2.1.11 Fusion pattern for overlapping individual objects

Sometimes we construct a new object by fusing a number of overlapping individual objects. The extension of the new object is the fusion of the extensions of the individual objects. If the individual objects were distinct (as in ***Figure 10.10***) then we would have a partition pattern. Where they overlap, we have the potential for a fusion pattern. For example, NATO and EEC overlap and so we can fuse them to get NATO & EEC. This is the geographic area covered by countries that are members of both NATO and the EEC. This fusion is recorded in the model in ***Figure 10.12*** with a fusion sign. As with the intersecting pattern, the fusion pattern creates a logical dependency. This is signed with the same logical dependency and derived signs.

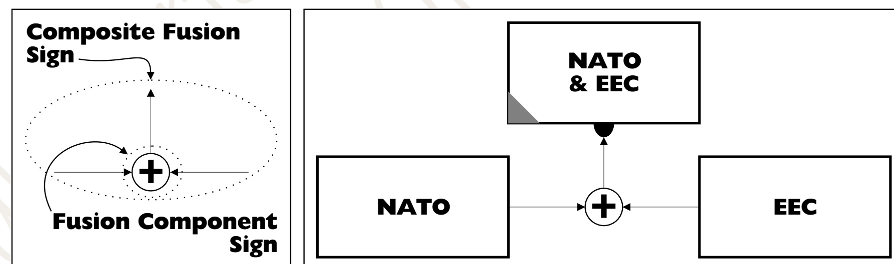


Figure 10.12: Fusion sign

2.2 Class object level patterns

The distinct and overlapping patterns between extensions, which we have just examined for individual objects, appear again at the class level. Although, at that level, the super–sub-class hierarchy plays the role of the whole–part hierarchy. We now analyse the class level patterns in the same way as we analysed the individual object level ones. Like before, we start with the simple patterns that hold between pairs of distinct and overlapping classes before moving onto larger groups of classes.

We then examine a similar set of associated patterns:

- Inheriting distinct and overlapping class patterns,
- Known and unknown distinct and overlapping classes,
- Partitioning patterns for distinct classes,

- Intersection pattern for overlapping classes, and
- Fusion pattern for classes.

We also work out what objects the distinct and overlapping class signs refer to.

2.2.1 Distinct pairs of classes

A pair of classes that does not have any members in common is distinct. For example, the classes birds and bees are distinct. A member of the class birds is never a member of the class bees. As shown in ***Figure 10.13***, we sign this pattern with the same distinct sign we use for individual objects. Distinctness is a connection between classes; so, the class signs, and not their member signs, are linked.

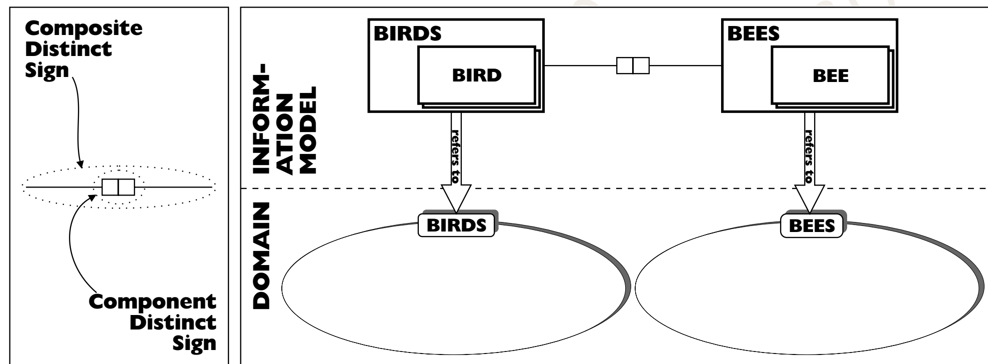


Figure 10.13: Distinct sign

2.2.2 Overlapping pairs of classes

A pair of classes that has members in common overlap. For example, the classes blondes and Germans overlap—there are Germans with blonde hair. As shown in ***Figure 10.14***, we sign this pattern with the same overlapping sign that we use at the individual object level. Overlapping, like distinctness, is a connection between classes, so the overlapping sign links class signs.

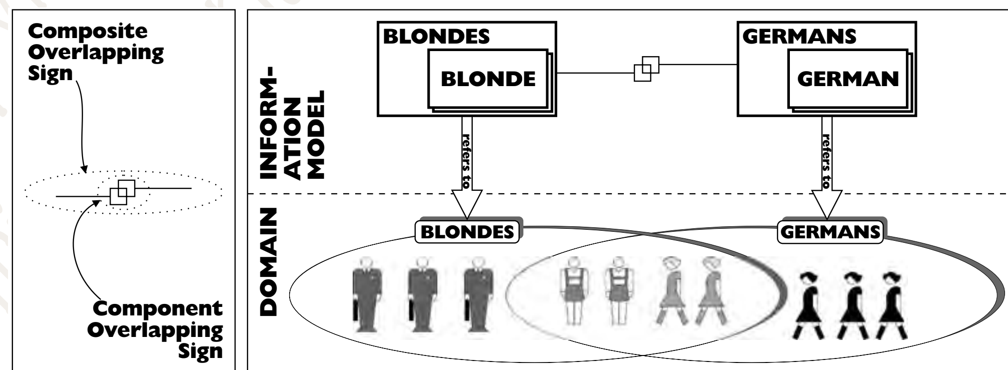


Figure 10.14: Overlap sign

2.2.3 *Three main types of connection for pairs of classes*

From an extensional point of view, pairs of classes have a similar set of structural patterns to individual objects. These are the distinct, overlapping and sub-class (matching individual object's whole–part) patterns shown in ***Figure 10.15***. A pair of classes must fall under one of these patterns. We could regard the super–sub-class pattern, where one class completely contains another, as an extreme case of overlapping. However, in a similar fashion to individual objects, the convention is to consider this a sub-class and not an overlapping pattern.

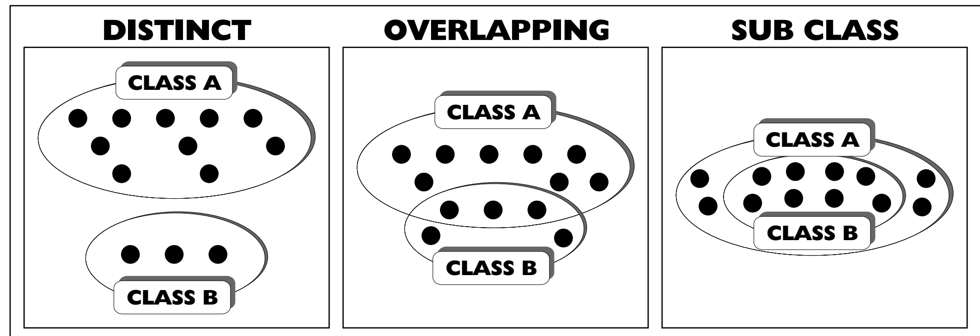


Figure 10.15:
Pattern for classes

2.2.4 *Larger groups of classes*

As with individual objects, for groups of classes larger than two, there are a wider variety of possible patterns of connection. It is possible for them all to be distinct [shown in ***Figure 10.16*** (a)]; or for them all to overlap [shown in ***Figure 10.16*** (b)]. It is also possible that some will be distinct and some will overlap. Even if every pair in a group of classes overlaps, the whole group may not overlap [shown in ***Figure 10.16*** (c)]. However, the same is not true for distinctness. If every pair of classes in a group is distinct, then the group is distinct.

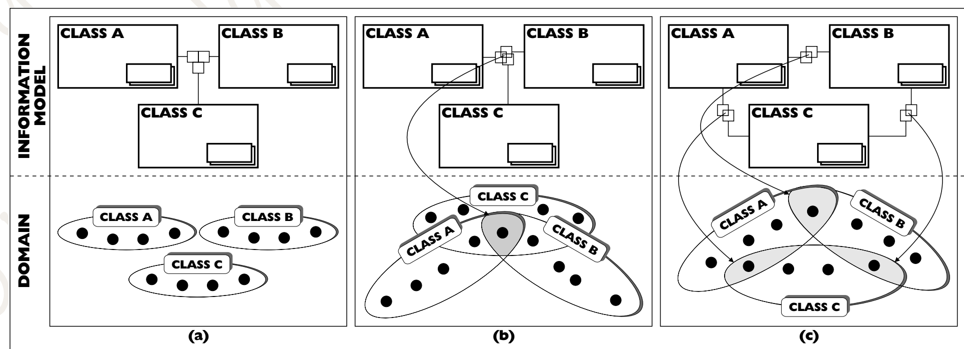


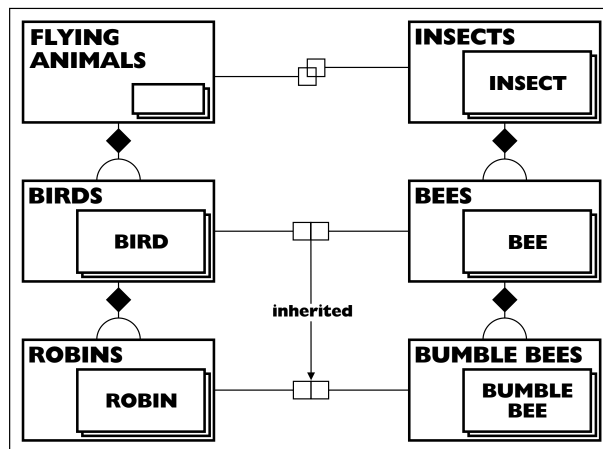
Figure 10.16:
Schemas for larger numbers of classes

2.2.5 *Inheriting distinct and overlapping patterns*

Both the distinct and overlapping class patterns are inherited along the super–sub-class hierarchy, but in opposite directions (matching the patterns for the individual object

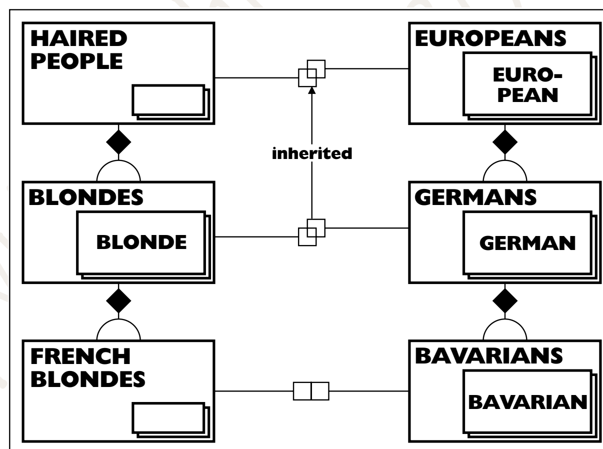
level's inheritance along the whole–part hierarchy). The distinct pattern is inherited down the hierarchy. For example, the classes, birds and bees, are distinct and so their sub-classes, robins and bumble bees, inherit that distinctness. But, as ***Figure 10.17*** illustrates, their super-classes flying animals and insects do not, thus proving distinctness is not inherited upwards.

Figure 10.17: Inheriting distinctness



Overlapping is inherited up the hierarchy. For example, as illustrated in ***Figure 10.18***, the classes blondes and Germans overlap and so their super-classes, haired people and Europeans do as well. However their sub-classes, French blondes and Bavarians are distinct proving that overlapping is not inherited down the hierarchy.

Figure 10.18: Inheriting overlapping



As with the individual level connections, this inheritance has implications for how we model. We push the distinct connections as far up the super–sub-class hierarchy as far as they will go and the overlapping connections as far down the hierarchy as they will go. This compacts and increases the functionality of the model.

© Copyright chris.partridge.com / BORO Centre.com

2.2.6 *Known and unknown distinct and overlapping patterns*

We often do not know all the members of a class. So we cannot always say whether a group of classes is distinct or overlapping and sign this in the model. This lack of information is not necessarily a problem. We only need to know the relevant distinct or overlapping patterns. Working out every pattern, relevant or otherwise, would be a waste of time.

However, when we want to model a group of classes as overlapping, it helps to know at least one common (overlapped) member. There is, in principle, nothing wrong with signing them as overlapping when we do not know a common member. However, this is not a good policy. Finding a common member is a sure way of confirming that the classes do indeed overlap. Even if we are reasonably sure that they do, it makes sense—as a safety check—to follow a policy of confirming our intuitions. We can do this simply and effectively by finding a common member. ***Figure 10.19*** illustrates this process of confirmation. If we cannot find a common member, this should make us suspect that the classes do not, in fact, overlap.

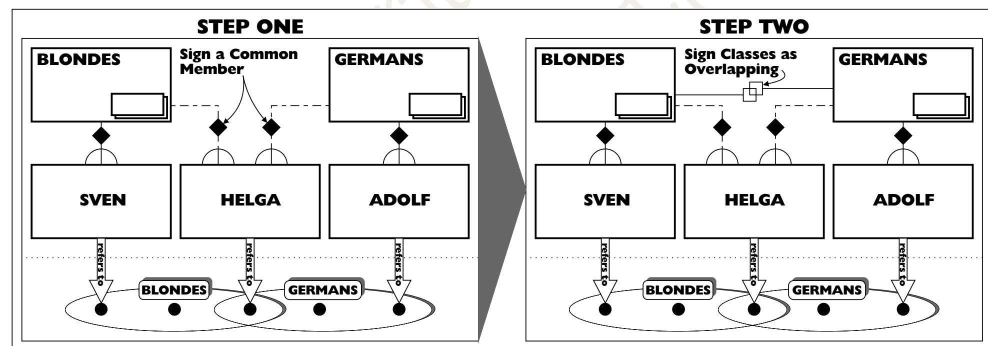


Figure 10.19: Constructing confirmation of overlapping

Things are not so easy for distinct patterns. No matter how many distinct instances two class signs may have, this does not prove that their classes are distinct. There may be an unknown object that is a member of both classes. So a group of classes cannot be logically proven to be distinct in the same way as they can be proven to be overlapping. This means we need to exercise caution before signing classes as distinct in the model.

2.2.7 *The distinct and overlapping class pattern objects*

The strong reference principle requires that, as we have signed distinct and overlapping class patterns, the signs refer to objects. These are constructed in the same way as their individual object cousins. They are the classes of the distinct (or overlapping) classes. We can illustrate this with the distinct birds and bees classes from ***Figure 10.13***. Its distinct sign refers to the class {birds, bees}, which has the birds and bees classes as its only members. Furthermore, this class is a member of the distinct class. This is shown in ***Figure 10.20***, which also shows an example of the pattern for the construction of the overlapping class.

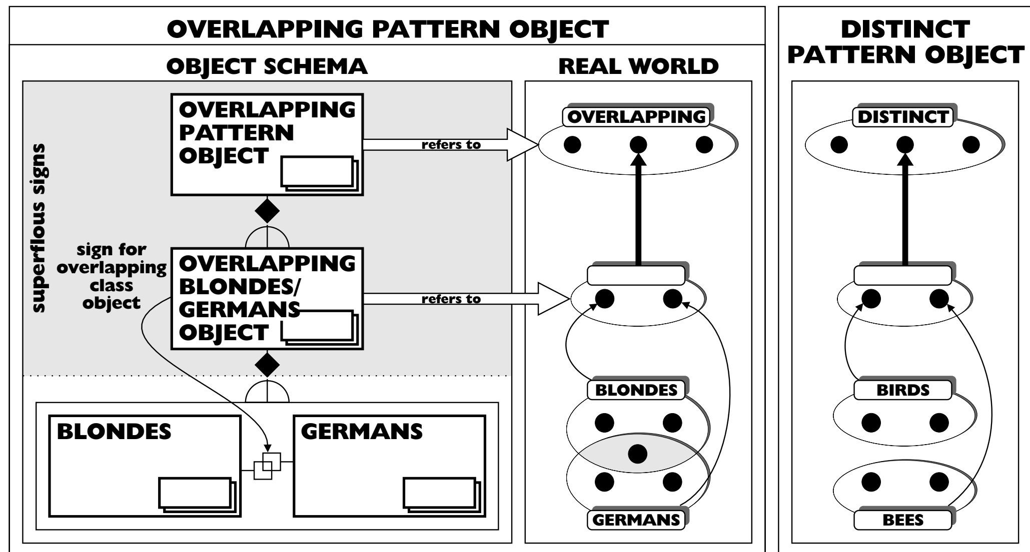


Figure 10.20: Class examples of overlapping and distinct pattern objects

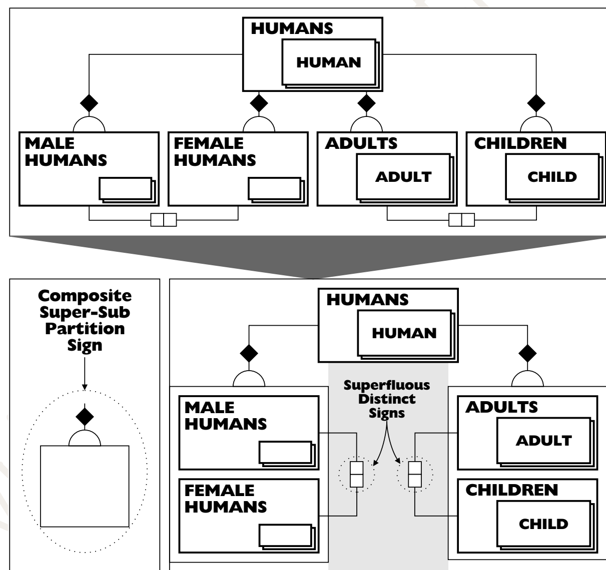


Figure 10.21: Incompletely partitioned classes

2.2.8 *Partitioning patterns for distinct classes*

Like individual objects, where a distinct pattern is often part of a larger individual partition pattern, distinct class patterns are often part of a larger class partition pattern. A type of partitioning class pattern has been a natural way of seeing since well before the emergence of the substance paradigm and its secondary substance hierarchy. For example, when we think of the class humans, we almost instinctively start partitioning it, maybe by gender. Then, even though it contravenes the substance paradigm's single classification restriction, some of us also start thinking of alternative ways of partitioning, for example into adults and children. In the class partition pattern, the partitioning class is divided into distinct partitioned sub-classes. As shown in ***Figure 10.22***, the notation is

similar to the individual object partition sign—with the component super–sub-class sign replacing the whole–part sign.

Often, the partition pattern does not partition a class completely, partitioning only some of its members into distinct classes. This is a partial or incomplete partition and is signed by adding an incomplete partition component sign to the composite partition sign. As shown in ***Figure 10.21***, this is a small flat rectangle that is put between the super–sub-class sign and the partition box.

Figure 10.22: Partitioned classes

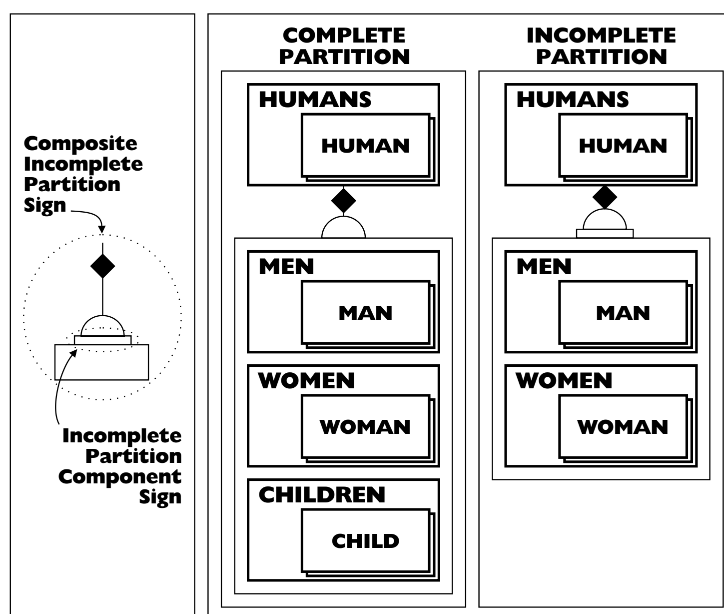
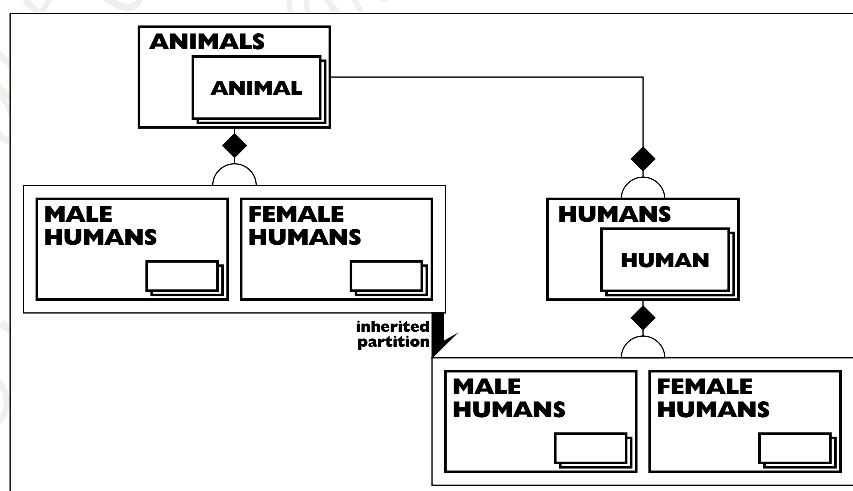


Figure 10.23: Partition inheritance



2.2.8.1 Partition pattern inheritance

As the example in ***Figure 10.23*** shows, partition patterns (like distinct patterns) are inherited down the super–sub-class hierarchy. The partition into distinct male and female animals classes is inherited down the super–sub-class hierarchy to the distinct male and female humans classes partition. (You will notice that the inherited partition is the more general male/female partition from ***Figure 10.22*** rather than the human specific men/women partition from ***Figure 10.21***.)

As with individual level partitions, this has implications for how we model class partitions. It is useful to push them as far up the super–sub-class hierarchy as they will go, increasing the number of classes that can inherit the pattern. This compacts and increases the functionality of the model.

2.2.9 Intersection patterns for overlapping classes

Sometimes we want to work with a class constructed from objects that are members of the overlap of a group of classes. This is an intersection pattern, which goes one step further than the overlapping pattern and constructs the class of the overlapped members. The intersection pattern only applies to overlapping classes, it cannot apply to the other two types of class patterns: distinct and sub-class. Distinct classes have no members in common and so have no use for the intersection pattern. Sub-classes have all their members in common with their super-class, and so the intersection pattern would not produce a new class.

We can see how the intersection pattern works with an example. Assume we are targeting a group of companies for a sales campaign and we are going to select the group from a comprehensive list. The list identifies whether companies are large and whether their headquarters are in the north or south of the country. If we target large companies in the north (in other words, the class of companies whose members belong to both the large companies class and the northern companies class) then we need the intersection pattern shown in ***Figure 10.24***. This illustrates the intersection sign, which is an enhanced version of the overlap sign.

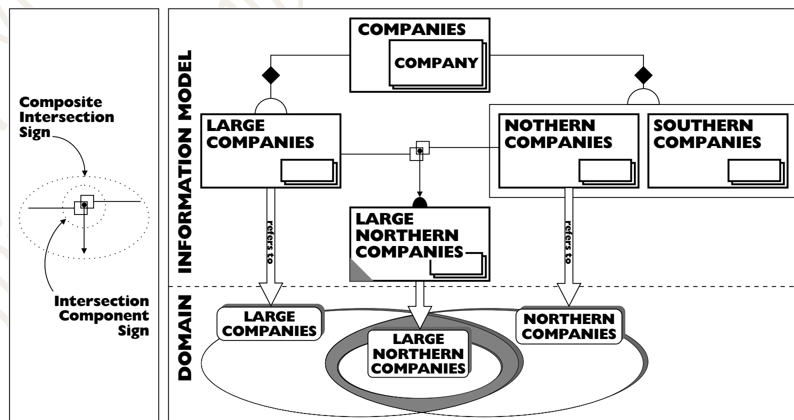


Figure 10.24: Intersected classes

The example in ***Figure 10.24*** also confirms that only overlapping classes can be intersected. It is pointless intersecting the classes northern companies and southern companies because they are distinct (as they are part of a partition). So we know in advance that the intersecting class would be empty. The intersected classes in the intersection pattern must be overlapping so that the intersecting class has members.

The intersecting class, large northern companies, is logically dependant on the intersected classes, large companies and northern companies. The logical dependency is shown by a black semi-circle sign at the end of the intersection sign (seen in ***Figure 10.24***. The class is derived by the logical dependency. This is shown by the derived sign, a small grey triangle in the bottom left corner of the class sign. Again this is visible in Figure 10.24. This derived component sign becomes an integral part of the composite sign for the class. It needs to be because the class sign can appear in other schemas without the intersection sign and so the logical dependency sign. Then the derived sign reminds us of the logical dependency.

2.2.10 Fusion patterns for overlapping classes

Sometimes every member of a group of overlapping classes has an interesting characteristic and this is captured by a class that pools all the members of the group of classes. For example, at some future date it may be decided to make the citizens of France and Germany citizens of a new Western Alliance state. The class Western Alliance citizens is the pooling of the members of the classes French citizens and German citizens. This pattern is called a fusion and is modelled using a fusion sign (shown in ***Figure 10.25***). You will notice that the classes French citizens and German citizens overlap; it is possible to have dual citizenship. If they did not (the classes were distinct), this would be a partition pattern. The fused class, Western Alliance citizens, is logically dependant on the classes French citizens and German citizens. This is shown in the same way as for intersected classes, with a logical dependency and a derived sign.

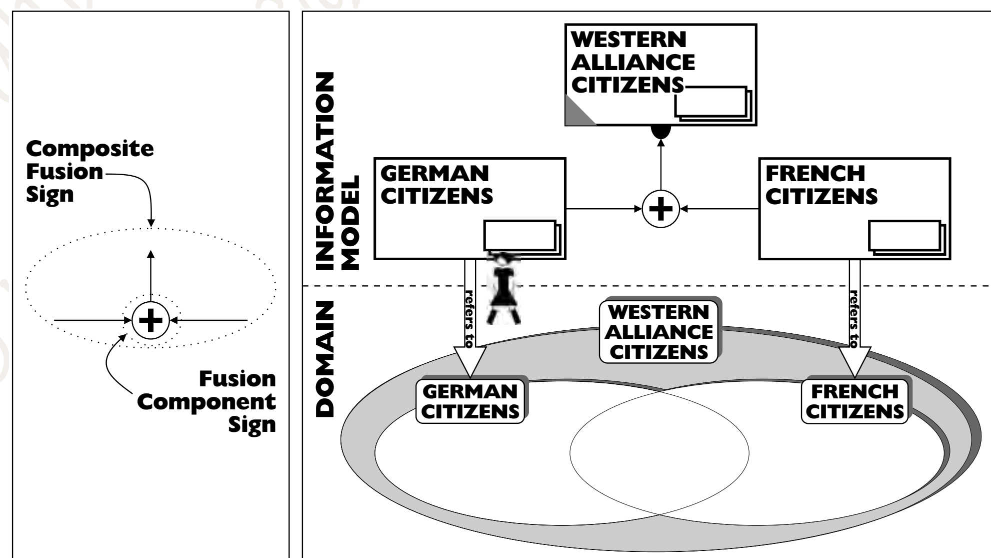


Figure 10.25: Fusion sign

2.2.11 *A close-knit family of extension patterns*

This examination of the patterns of connections between extensions has revealed a close-knit family of patterns. We have seen how patterns at the individual object level repeat themselves at the class level. How patterns are inherited up and down the super–sub-class and whole–part hierarchies. How we can and should generalise the connections along their inheritance hierarchies to compact and increase the functionality of the model. The examples have given us a feel for how these patterns work with one another. As we get more experience of business object modelling, they will become second nature.

3 State hierarchy patterns

In ***Chapter 7***, we examined how object semantics explained substance's states as physical bodies that are temporal parts of other physical bodies. Here, we look at the basic object syntax for states. We look at the sign for a state and how to model the following state patterns:

- State–sub-class hierarchy patterns,
- State–sub-state hierarchy patterns,
- Distinct state patterns,
- Partitioned state patterns, and
- Overlapping state patterns.

These are all spatio-temporal patterns. In the next section, we look at temporal (time ordered) patterns.

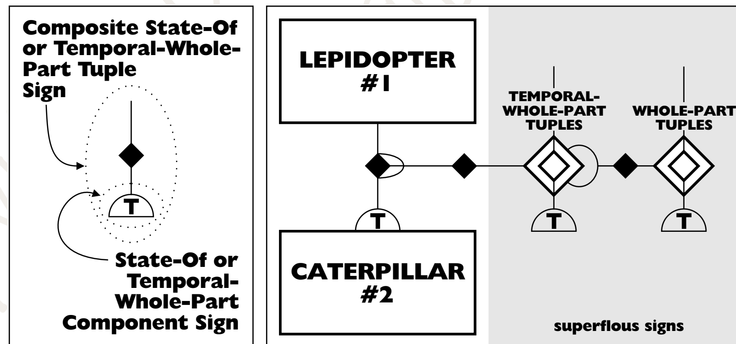


Figure 10.26: Temporal–whole–part or 'state–of' sign

3.1 The state–of sign

A state is a physical body that is a temporal part of another physical body. This link between the state and the physical body is a particular type of whole–part tuple. Consider the lepidopter example from ***Chapter 7*** (illustrated in ***Figure 7.2***), where caterpillar #2 is a state of lepidopter #1. As ***Figure 10.26*** shows, the state–of tuple is a couple <lepidopter #1, caterpillar #2> belonging to the temporal–whole–part tuples class. (This is the states tuples class; all states are, by definition, temporal parts of physical bodies.)

© Copyright Chris Partridge chris.partridge@BOROCentre.com

The temporal–whole–part tuples class is, in turn, a sub-class of the whole–part tuples class.

As the couple belongs (distantly) to the whole–part class, we sign it with a whole–part sign. To reflect the fact that caterpillar #2 is a temporal part (state) of lepidopter #1, the composite state–of sign has a state-of or temporal component sign. In ***Figure 10.26***, the whole–part and temporal–whole–part tuples classes are drawn. However, these are normally left out of the schemas because they are superfluous, implied by the state–of or temporal–whole–part sign.

Figure 10.27:
State–sub-state hierarchy pattern

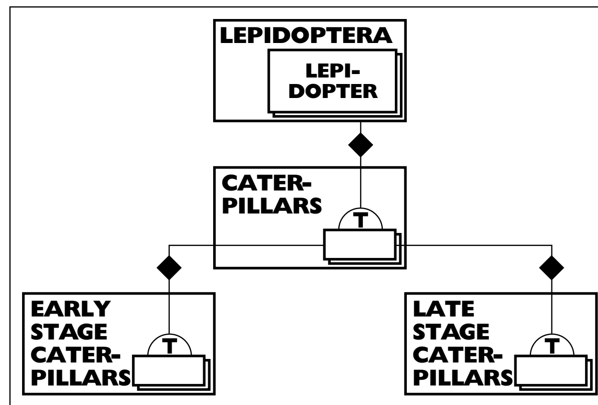
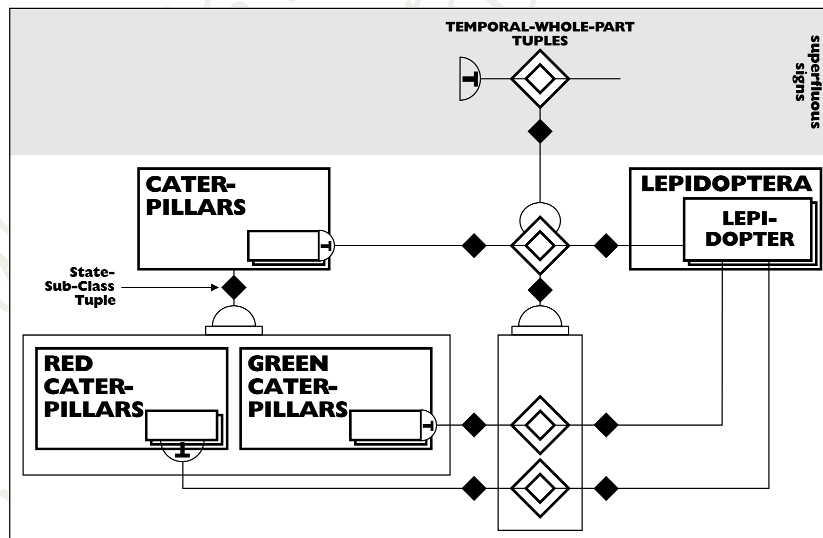


Figure 10.28:
State–sub-class hierarchy pattern



3.2 State–sub-state hierarchy patterns

We saw in ***Chapter 7*** that states can have states and this leads to a state–sub-state hierarchy pattern. In the example illustrated in ***Figures 7.6*** and ***7.7***, caterpillars had early and late stage sub-states, where a substate is defined as a temporal part of a temporal part. So, as shown in ***Figure 10.27***, the pattern is signed using the state–of sign.

You should notice that this pattern is at the member level, with the state tuples signs connecting the classes' member signs.

3.3 State–sub-class hierarchy patterns

Chapter 7 also shows us that states are collected into state classes that can have state–sub-classes. This state–sub-class pattern is just a super–sub-class pattern, where the classes are state classes. ***Figure 10.28*** shows this using the example illustrated in ***Figures 7.8*** and ***7.9***, where the caterpillars (state) class has red and green (state) sub-classes.

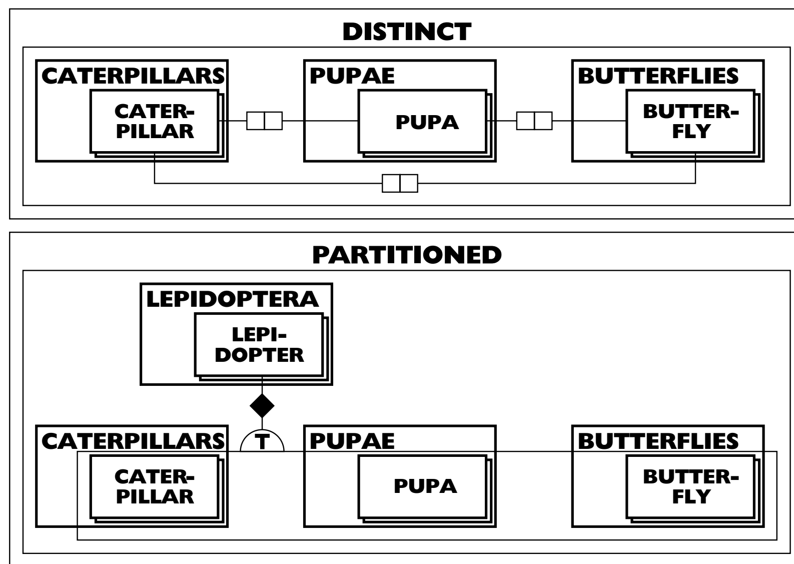


Figure 10.29: Distinct and partitioned states

3.4 Other extension-based state patterns

States, as physical bodies, fall into the same extension-based patterns as other physical bodies. For instance, they have the distinct, overlapping and partitioned patterns we examined in the beginning of this chapter. We illustrate this using the lepidopter example again. Its states are distinct and also completely partition the lepidopter object. ***Figure 10.29*** models these two patterns. You can see that the partition is modelled connecting the classes' members icons, this is because it operates at the member level.

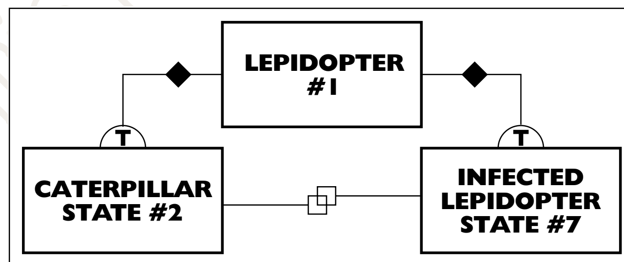


Figure 10.30: I
Overlapping states

In ***Figure 10.30***, we have used the overlapping caterpillar and infected lepidopter states from ***Figures 8.10*** and ***8.11*** to illustrate how we sign an overlapping state.

4 Time ordered temporal patterns

In ***Chapter 8***, we examined how object semantics explains changes using these two types of object:

- States, and
- Events.

We now look at the object syntax for their time ordered temporal patterns.

4.1 State changes

In object semantics, states are objects and often ordered in time. This ordering can take a number of patterns; we only look at this sample here:

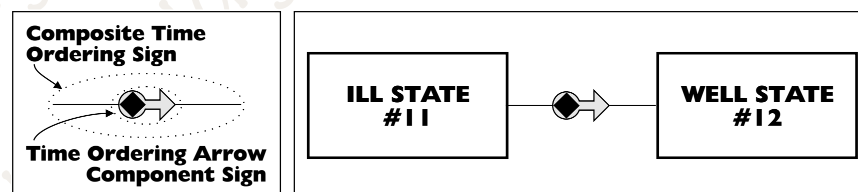
- Simple state 'change' patterns,
- Sequence of states pattern, and
- Alternating states pattern.

We then investigate how the state life history of an object is constructed from a states' time ordering patterns.

4.1.1 A simple state 'change' pattern

The simplest state change involves a 'change' from one state to another—for instance, a change from an ill state into a well state. The states are ordered in time – one after the other. To describe this pattern, we construct a tuple of the two states and sign its order with a component time ordering arrow sign (shown in ***Figure 10.31***).

Figure 10.31: Sign for time ordering



4.1.2 A time sequence of states pattern

Often the states of an object fall into a time sequence pattern. We can describe this pattern at an individual object level or generalise it to a class level—as in the chairman and lepidopter examples below.

© Copyright Chris Partridge chris.partridge@BOROCentre.com

4.1.2.1 Individual object level sequence

The chairman thought experiment from ***Chapter 8*** (illustrated in ***Figure 8.18***) provides a good example of a time sequence pattern of individual states. Each new resignation and appointment leads to a new chairman state. If we extend the pattern in the thought experiment we get a sequence, over time, of chairman states—all states of the chairman object. In this case, the sequence of states has a temporal gap. This is modelled with the time 'gap' ordering arrow component sign shown in ***Figure 10.32***.

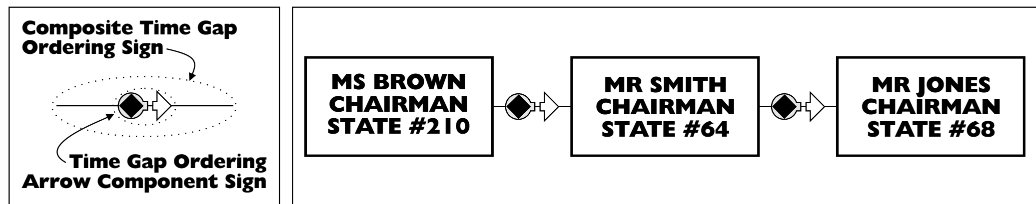


Figure 10.32: Individual object level sequence of states

4.1.2.2 Class level sequence

The ubiquitous lepidopter provides us with an example of a class level sequence pattern. Caterpillars develop into pupae that develop into butterflies. It is the members of the classes that develop, not the classes themselves, so the ordering sign is linked to the class members' signs (shown in ***Figure 10.33***), not the class signs.

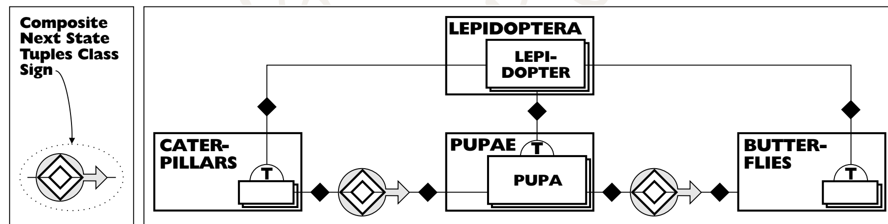


Figure 10.33: Class level sequence of states

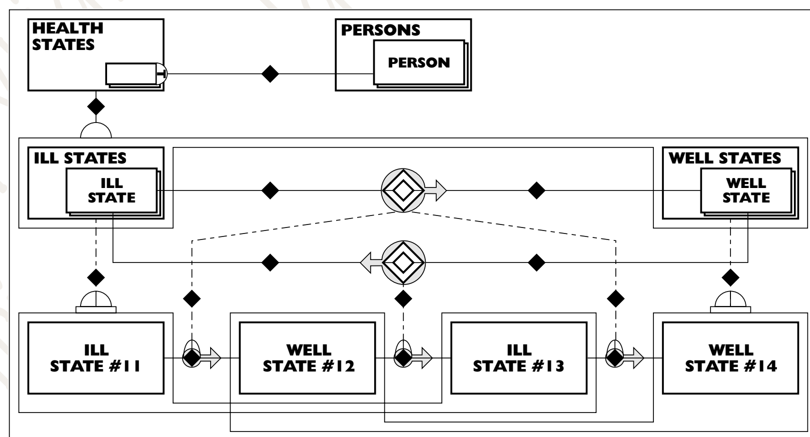


Figure 10.34: Alternating state patterns

4.1.3 Alternating state patterns

States also fall into an alternating pattern, as shown in the well and ill states example in ***Chapter 8*** (***Figure 8.17***). We model this using the sign for time ordering (shown in ***Figure 10.34***). You should notice that, in this case, the model shows both the individual object and the class level ordering.

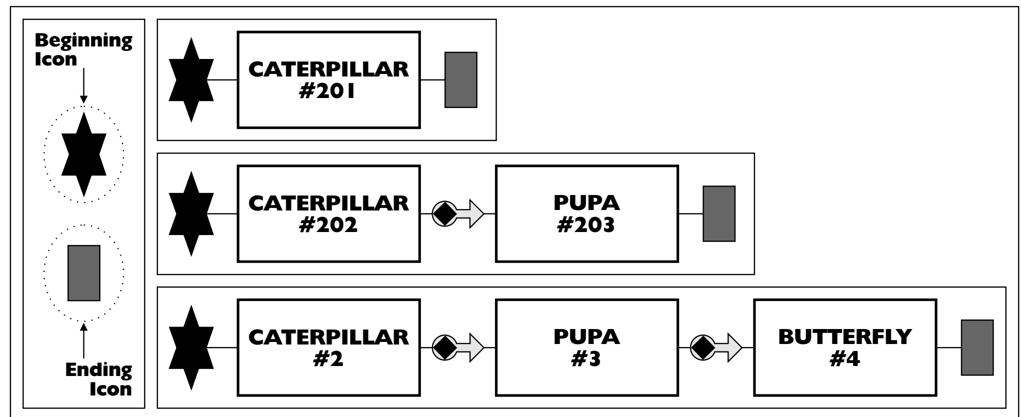


Figure 10.35: Three individual lepidopter state life histories

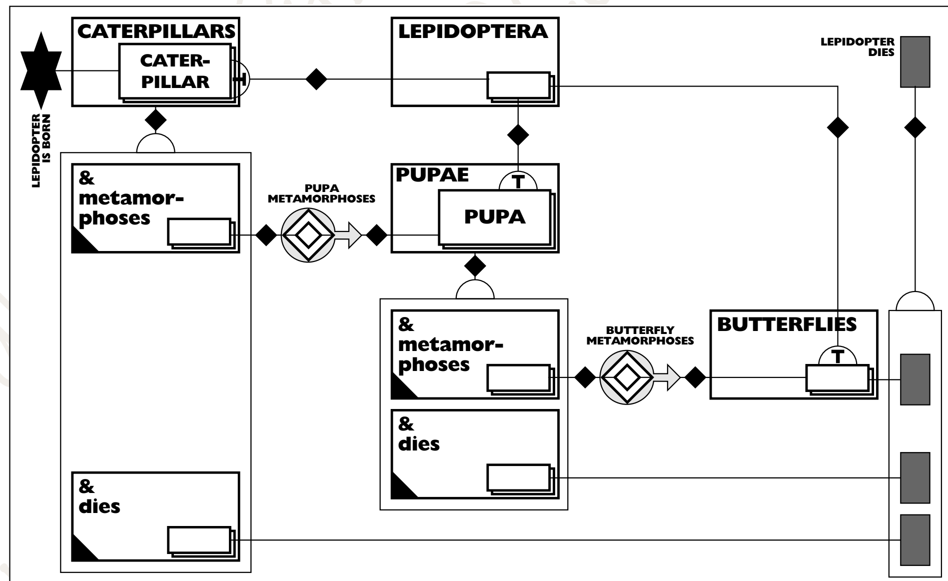


Figure 10.36: A class level lepidoptera state life history

4.1.4 An object's state life history

These signs for states' time orderings allow us to tell an individual object's state life history (or indeed, a class of objects' state life histories). Consider the lepidoptera example again. To determine its state life history we first need to find all the possible patterns for its individual states. ***Figure 10.35*** provides a simplified version of these in the form of state life histories for three individual lepidoptera—each one dying at a different stage of

development. Notice the beginning and ending signs. These are, as you can see, based on the space-time map icons.

We generalise these individual level patterns into a class level history; the result is ***Figure 10.36***. Notice that as the state life histories are of the individual states of the physical object, the time ordering pattern is between the members and not the classes. This is a very simple example. Normally, an object would have a number of different state partitions, across which states would overlap (illustrated in ***Figures 8.10*** and ***8.11***).

People familiar with traditional modelling may recognise this as object syntax's version of the entity paradigm's life history diagrams. Getting a picture of something's life history is an extremely useful part of business modelling. However, the entity life history has to work within the confines of the entity paradigm, which typically constrains it to a tree-like structure. Using the more powerful and sophisticated object semantics enables us to construct a much more accurate, and so useful, picture of a life history.

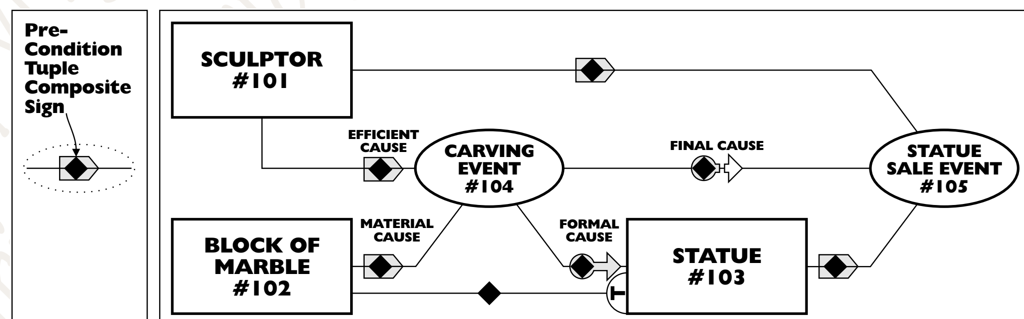
4.2 Event cause and effect time orderings

As well as a life history perspective on objects, object syntax offers a cause and effect perspective centred on events. In ***Chapter 8*** we discussed how Aristotle analysed understanding into the following four types of cause:

- efficient cause,
- material cause,
- formal cause, and
- final cause.

We now look at how these are modelled with time ordering signs. We do this by example. We model, using object syntax, the 'sculptor carving a statue' example illustrated in ***Figure 8.26***. The result is ***Figure 10.37***. We use a new sign (the pre-condition sign) for the efficient and material causes because the causes are not ordered before or after the event, but around it. The efficient and material causes are differentiated because the material cause has a temporal–whole–part connection with the formal cause.

Figure 10.37: Object syntax's event perspective



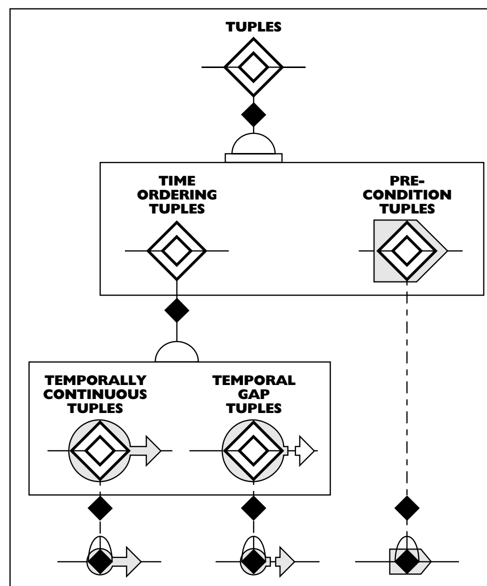
The life history and event perspectives complement one another. The life history fits the states into a pattern. The event perspective then explains that pattern by mapping what 'causes' the events that change the states.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

4.3 Time ordering tuple objects

We have looked at various time ordering (and pre-condition) signs. We now examine, in deference to the strong reference principle, the objects that these signs refer to. They are tuples that belong to the appropriate pattern's tuples class. At the individual level, they are couple objects as indicated by the two place links to the diamond tuple component sign. These are members of one or another of the time ordering or pre-condition pattern's tuples classes (illustrated in ***Figure 10.38***).

Figure 10.38: Time ordering and pre-condition tuples classes



5 Cardinality patterns for tuples classes

We now move from time ordering tuples to a particular aspect of tuples classes. We look at a group of useful modelling patterns—cardinalities. Traditional information modelling uses cardinality patterns for its relational attributes and we re-engineer a version of the patterns here. A few differences arise because the tuples class and the occupied class places are objects in their own right in object semantics. This is a change from traditional modelling, where cardinalities are implicit parts of relational attributes.

In many cases, it is useful to describe the cardinality patterns of a tuples class, but this notation does not insist on it. A number of notations are used for describing cardinality in traditional information modelling; most of which can be adapted to object semantics. I prefer to use the simple one described below, but it does not really matter which one is used. I suggest that you use the notation you feel most comfortable with, though remember it will probably need some amendments to cope with object semantics.

© Copyright Chris Partridge chris.partridge@BORO Centre.com

5.1 Types of cardinality pattern

In the last chapter, we looked at the signs for tuples classes and their occupied class places. These occupied class places are the basis for the cardinality patterns. Cardinality is a pattern that, in object syntax, applies to occupied class place objects. When a cardinality pattern is signed, both an upper and a lower bound are specified. There are two levels for the lower bound—optional or one. There are also two levels for the upper bound—one or multiple. These upper and lower bound levels can be combined in four ways to produce four different cardinality patterns for the occupied class place:

- Optional-to-one pattern,
- One-to-one pattern,
- Optional-to-multiple pattern, and
- One-to-multiple pattern.

We now look at each of these in more detail.

5.1.1 Optional-to-one cardinality pattern

Consider ***Figure 10.39***, which models the person-born-in-Britain tuples class. What is the cardinality pattern of the class place occupied by the class persons? I have found that it is important when determining cardinality to confirm one's intuitions with specific instances. I go through this confirmation process step by step in this example.

Prince Philip and Queen Elizabeth are both members of the class persons. Prince Philip is a person and was not born in Britain. So it must be optional for a person to be born in Britain. Or, in object-speak—it must be optional for members of the class persons to occupy the person place in a couple that is a member of the person-born-in-Britain tuples class. So the lower bound for the occupied class place is zero. This is signed in a similar way to traditional modelling with an '0' on the line between the occupied class place and the tuples class sign.

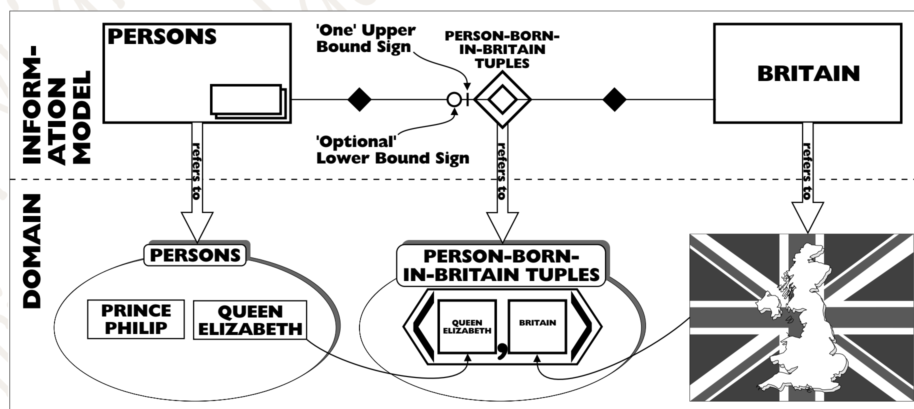


Figure 10.39: Optional-to-one cardinality pattern

Queen Elizabeth is a person and was born in Britain. So a person can be born in Britain (a member of the class persons can occupy the person place in a couple that is a member of the person-born-in-Britain tuples class). It is safe to assume that a person cannot

© Copyright chris partridge chris@BORO Centre.com

be born more than once, in Britain or anywhere else. So the maximum number of times a person can appear in the person place of a person-born-in-Britain couple is once. This means the upper bound for the occupied class place is one, which is noted by a ‘1’ sign on the class place link. By convention we draw the upper bound sign closer to the tuples class sign than the lower bound sign. Both these upper and lower bound signs are shown in ***Figure 10.39***.

5.1.2 One-to-one cardinality pattern

If we now model the son–father tuples class with its place classes son and father then we get the schema shown in ***Figure 10.40***. A son always has one and only one biological father; so, every son appears once and only once in the son place of a father–son couple. This means the upper and lower bounds are both one. So two ‘1’ signs are put by the occupied class place sign, next to the tuples class sign.

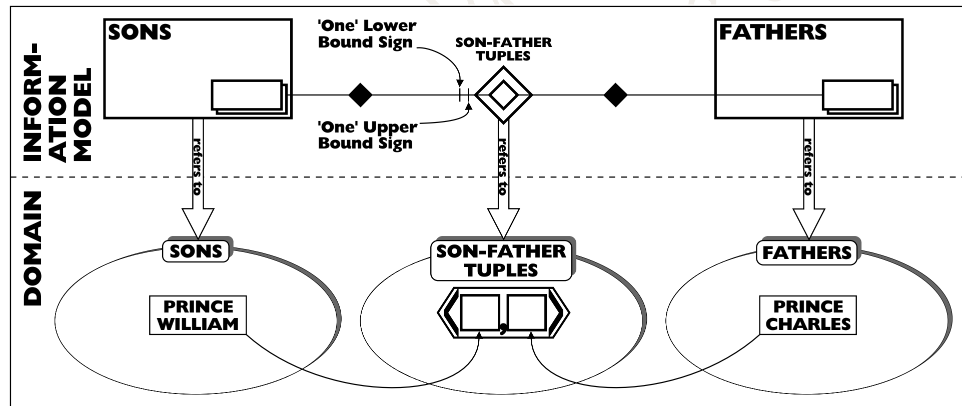


Figure 10.40: One-to-one cardinality pattern

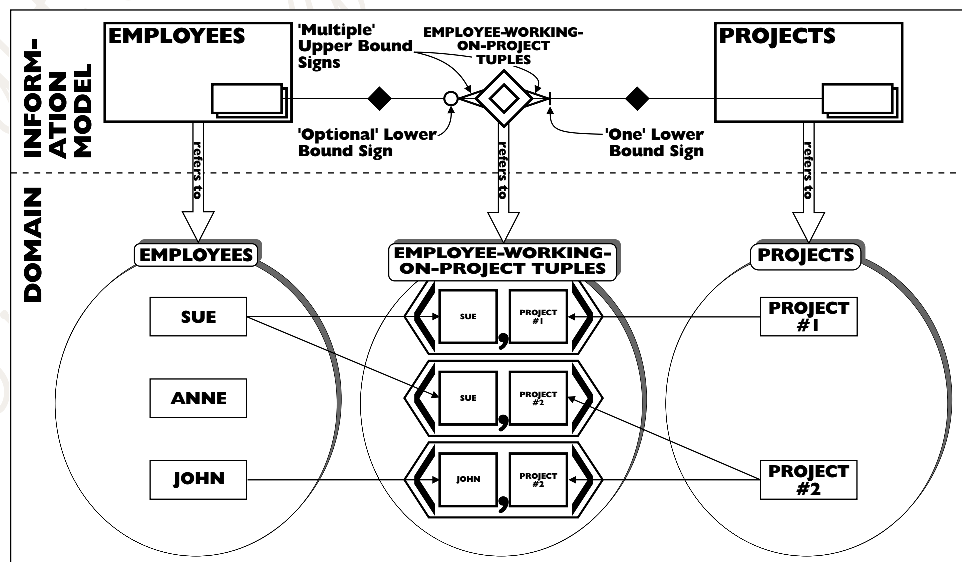


Figure 10.41: Figure 10.41 Optional- and one-to-multiple cardinality patterns

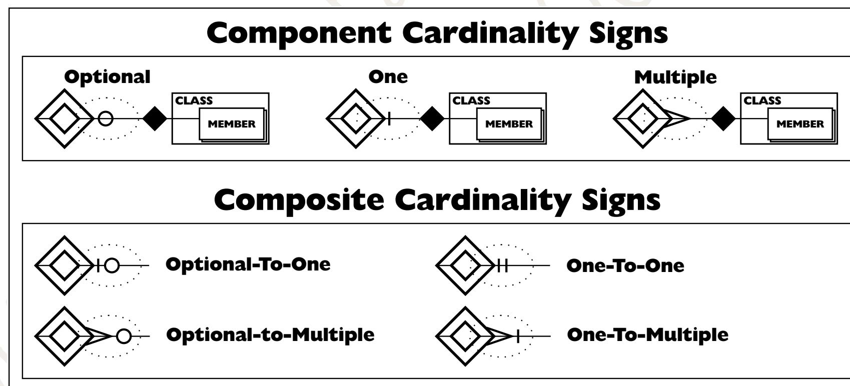
5.1.3 *Optional-to-multiple and one-to-multiple cardinality patterns*

Now consider the model in ***Figure 10.41***, this shows the employee–project tuples class originally illustrated in ***Figures 3.23*** and ***3.24***. An employee will sometimes work on a number of projects. This means the upper bound for the occupied class place must be greater than one. For this, we use the multiple sign. As you can see, it looks like a crow's foot. Some employees, such as secretarial staff, will never work on a project. So the occupied class place has a lower bound of zero. We use the same '0' sign that we used in Figure 10.40 for this. All projects have one or more employees working on them. So the lower bound of the occupied class place link is one and the upper bound is multiple. These optional-to-multiple and one-to-multiple cardinalities are is signed in ***Figure 10.41***.

5.1.4 *Cardinality pattern signs*

These examples cover the only four possible signs for the cardinality of an occupied class place. A full list is given in ***Figure 10.42***. If we are going to sign the cardinality of an occupied class place then we will use one of them. Remember, however, that unlike some traditional modelling notations, each occupied place of a tuples class can be given a cardinality pattern. So a tuples class can have as many cardinality patterns as it has occupied class places.

Figure 10.42: The four composite cardinality pattern signs



5.2 Cardinality patterns as objects

Cardinality signs, like the distinct and overlapping signs, refer to class objects. But, which class objects? If we analyse the model carefully we can see the members of the cardinality classes—occupied class place objects.

5.2.1 *Occupied class places as objects*

We looked at the signs for occupied class places in the last chapter (see ***Figures 9.25*** and ***9.26***). We now work out what these signs refer to. We start with the signs for individual tuples and work up to the occupied class place signs.

The sign for individual tuples, such as <Prince Charles, Prince William>, is a black diamond. This component has a number of lines, called (tuple) place component signs joining the diamond to the signs for the objects that make up the tuple. For example, in ***Figure 10.43***, a component place sign joins the black diamond tuple sign to the Prince Charles sign.

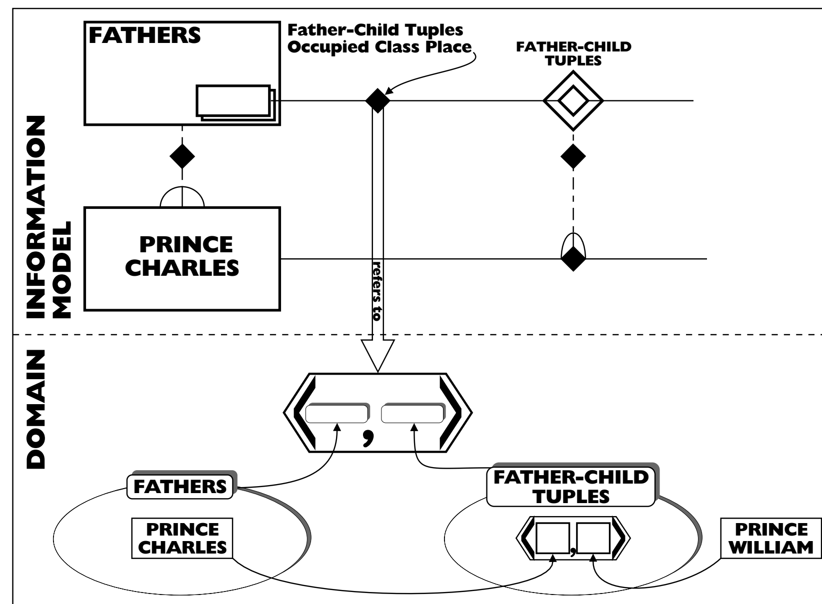


Figure 10.43: What occupied class places signs refer to

The tuples class signs have a component that looks similar. This is the (tuples) class place sign. Like the place component sign, it is a line. Unlike it, the line does not have to join the tuples class sign to another sign. For example, the class place sign on the right of the father-child tuples class in ***Figure 10.43*** is not joined to anything. The class place sign can join the tuples class sign to another sign—as shown by the class place sign joining the fathers class sign to the father-child tuples class in Figure 10.43. When this happens, the class place is said to be occupied and a black diamond (the tuple sign) is added to the line.

What object does this occupied class place sign refer to? Despite the similarity of the signs, it cannot reflect a simple construction relationship as the tuple's place sign does. The connection between the father-child tuples and fathers classes is not one of a tuple constructed from an object. Instead, it is a tuple, <father-child tuples, fathers> (illustrated in ***Figure 10.43***). That is why the occupied class place component sign is a black diamond, the sign for a tuple. In general, occupied class place signs refer to a couple with the format <tuples class, place class>.

5.2.2 Cardinality classes with occupied class places as members

These occupied class places are the members of cardinality classes (shown in ***Figure 10.44***). For example, the one-to-one cardinality sign refers to the one-to-one bound cardinalities tuples class. This has as members all occupied class places with a one-to-one cardinality, including the one shown in the figure.

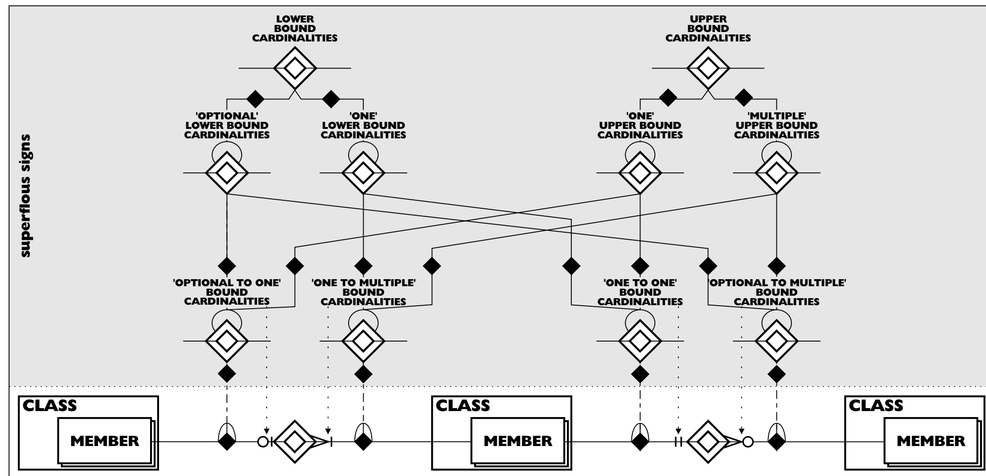


Figure 10.44: Underlying cardinality model

The figure also illustrates how, once the composite cardinalities are seen as classes, they can be generalised into their elements. The one-to-one cardinality class can be generalised as a sub-class of the 'one lower bound cardinalities' and the 'one upper bound cardinalities' tuples classes. We can also see quite clearly how much easier it is to use the cardinality signs than the more long-winded couple and tuples class–member signs.

5.3 Inheriting cardinality patterns

There are constraint patterns for the inheritance of cardinality patterns up and down the super–sub-class hierarchy. They are easy to work out; so try doing it for yourself.

6 A pattern for compacting classes

So far, in this chapter, we have looked at how object syntax helps us model objects. Now, we turn our attention to a pattern that helps us generalise classes and so compact the model. This is the pattern of tuples classes defining their place classes.

Once we identify the pattern, we generalise the place classes up the super–sub-class hierarchy. We can then eliminate the original, less general, place classes. This compacts the model without compromising its information content. This is a good illustration of one way in which compacting works and how we handle it within object syntax. We shall re-use this compacting pattern in the worked examples in Part Six.

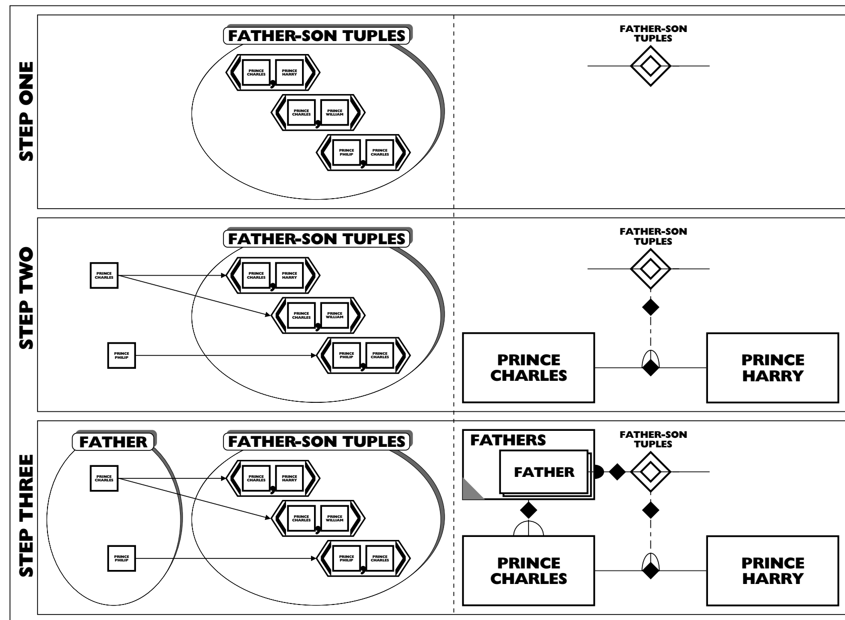
6.1 Constructing an example of the pattern

To illustrate the compacting, we need an example of the pattern. We get one by constructing a derived place class from a tuples class. Step one, shown in ***Figure 10.45***, is taking the father–child tuples class. At this stage, it has no occupied class places. Step two is identifying in each of the member tuples, the object that occupies the father

© Copyright Chris Partridge
chris.partridge@BOROCentre.

place. Step three is collecting all these objects into a class. This gives us a fathers class that occupies one of the father–child tuples class places.

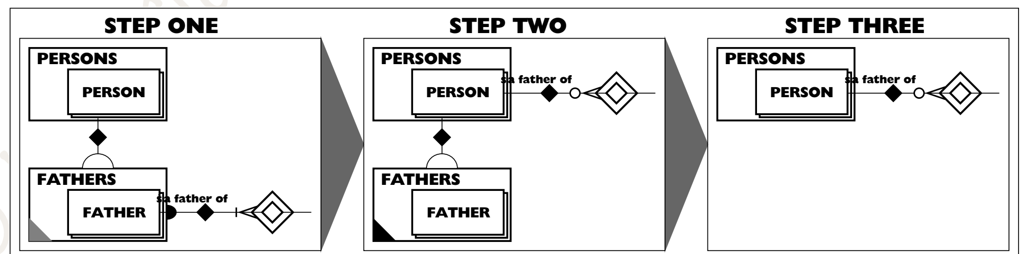
Figure 10.45: Constructing the logically dependent place class fathers



The class fathers is defined as those persons who have a father–child tuple linking them to a child—so it is logically dependent on the father-child tuples class. This makes it derived. This is modelled in the usual way; with logical dependency and derived signs (shown in ***Figure 10.45***).

We will spot this pattern frequently if we keep asking whether there is a logical dependency between a tuples class and its place classes. Until now, we have tended to assume that they are logically independent. In this father–child tuples case, and many other cases, if we had asked ourselves the question, we would have realised that there is a logical dependency.

Figure 10.46: Making a derived place class redundant



6.2 Using the pattern to compact the model

We now have an example of the pattern of tuples classes defining place classes. So we can illustrate the compacting. We do this in the three steps shown in ***Figure 10.46.*** In the first step, we generalise the fathers class (the occupied class place) up the super–

sub-class hierarchy to the persons class. In the second step, we generalise the 'is a father of' occupied class place from the father class to the persons class. At this stage, the fathers class no longer has a role to play; so, we classify it as redundant. The grey derived component sign in step one becomes a black redundant component sign. In the third and final step, we eliminate the now redundant fathers class from the model.

Often, when we are growing a business model, we construct classes that are logically dependent on tuples classes. These normally serve a purpose during the early stages. But, in most cases, they are redundant and so do not need to be implemented. As the model matures, we compact it by eliminating the redundant classes.

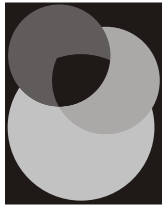
In this example of the compacting process, we eliminated the fathers class. However, it is sometimes useful to keep a record of redundant classes. Then, we do not eliminate the class but leave it in the model flagged as redundant. It then occupies a kind of limbo, kept in the model for reference purposes only.

7 Where we are

Compacting the model is an important part of business modelling, and generalising a class place's link up the super–sub-class hierarchy is a useful pattern for compacting. The other patterns we looked at are also useful when business modelling. We will find ourselves (re-)using most of them. As well as constructing object models of useful patterns, this chapter has helped us develop a clear idea of how the object notation captures patterns of business objects, an essential part of good business object modelling.

This is the end of the first half of the book, the part that concentrates on helping you understand what business objects are. In the next half of the book, Part Six, we use worked examples to demonstrate an approach that uses business objects to re-engineer the entity formats of existing systems.

© Copyright chris partridge
chris.partridge@BORO Centre.com



BORO

Part Six

Applying Business Objects

Chapter 11 The REV-ENG: An Approach to Applying Business Objects

Chapter 12 Re-Engineering Country's Entity Format

Chapter 13 Generalising Country's Re-Used Patterns

Chapter 14 Re-Engineering Our Conceptual Patterns for Country

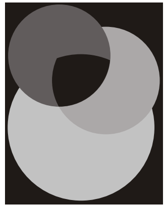
Chapter 15 Re-Engineering Region

Chapter 16 Re-Engineering Bank Address

Chapter 17 Re-Engineering Time

Chapter 18 Starting a Re-Engineering Project

© Copyright Chris Partridge chris.partridge@BOROCentre.com



Chapter 11

The REV-ENG: An Approach to Applying Business Objects

1 Introduction
2 The REV-ENG approach
3 The worked examples
4 A systematic approach to re-engineering
5 A framework for the model
6 Generalisation and compacting
7 What's next

© Copyright Chris Partridge : chris.partridge@BOROCentre.com

1 Introduction

In Parts One to Five we developed an understanding of the object paradigm and business object modelling. We now know what a business object is and how to model it. Here, in Part Six, we develop a feel for how to apply business objects to re-engineering entity-based computer systems. We do this by working through several examples using the systematic REV-ENG method. In this introductory chapter to Part Six, we take an overview of the REV-ENG method and the content of the worked examples.

2 The REV-ENG approach

The REV-ENG method is designed to re-engineer the business paradigms embedded in existing computer application systems—ones that are used for such tasks as accounting or foreign exchange trading. It is here that we start to see the actual benefits of re-engineering.

The previous parts of the book have been, in some ways, a preparation for this. When we re-engineered the entity paradigm into the object paradigm, we gained a clear understanding, at a general level, of:

- What the object paradigm is,
- What kind of business objects the world contains, and
- How to describe these in a model.

These are worthy achievements, but by themselves they do not actually deliver any business benefit. We harvest the benefits when we re-engineer our entity-based computer application systems.

Our re-engineering of the entity paradigm in the previous parts of the book involved challenging fundamental assumptions—the more fundamental the better. Now that we have established the object paradigm, we start working within its fundamental assumptions. In this part of the book, we re-engineer the entity business paradigms embedded in computer systems into object business paradigms (illustrated in ***Figure 11.1***). We transform their entity-formats into an object model, using the object paradigm as a foundation, rather than the subject of the re-engineering. This re-engineering, by its very nature, is less fundamental and more structured. This has certain advantages. Because we are working within a framework, we can guarantee a practical result. This takes the uncertainty out of re-engineering.

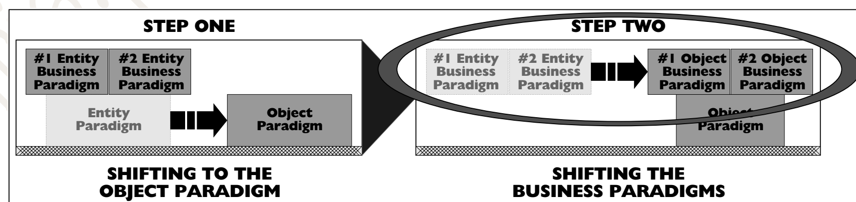


Figure 11.1: Re-engineering entity business paradigms

© Copyright Chris Partridge chris.partridge@BORO Centre.com

Using entity-based computer systems as the starting point for our re-engineering enables us to define a systematic method for re-engineering (the REV-ENG method). This makes the re-engineering both effective and efficient. This is very different from discovering a fundamental shift, such as the shift to the object paradigm, which involves the kind of inspiration that cannot be planned in advance. With the REV-ENG method, the re-engineering of the entity business paradigms can be planned and executed with a reasonable amount of precision. This makes it a viable commercial approach.

3 The worked examples

The easiest and best way to learn the approach is by working through examples. That is what we do in the following chapters.

3.1 Picking an element of an existing computer system

In each example, we pick a small element of an existing computer system; typically:

- An entity type,
- And, its associated;
- Attribute types,
- Entities, and
- Attributes.

We use these as the starting point for the re-engineering process. For each of them, we work through a series of steps that transmutes their primitive entity formats into a powerful and sophisticated object model.

The worked examples serve two purposes:

- First, they help us to understand what re-engineering a business paradigm involves and how the REV-ENG method works.
- Second, they provide us with basic object models, which we can re-use again and again in subsequent re-engineerings.

Because the examples include fundamental business patterns (ones that are used in a wide range of businesses), the basic object models are likely to be reusable in most re-engineerings.

3.2 The worked examples' business patterns

The worked examples are organised into groups based on two types of business pattern:

- Spatial patterns, and
- Temporal patterns,

The first group—spatial patterns—is based on three simple entity formats that are found in many business computer systems:

©Copyright Chris Partridge
chris.partridge@BOROCentre.com

- Country,
- Region, and
- Address.

We re-engineer these into a comprehensive object model of our spatial patterns. In the process, we also re-engineer a general object model for naming patterns.

The second group—temporal patterns—is based on two simple entity formats:

- Bank holiday, and
- Weekend.

We re-engineer these into a general object model of our temporal patterns.

The early examples are designed to help you become familiar with the systematic re-engineering method by carefully taking you through each step in the process. We begin with a simple example—country—whose re-engineering is straightforward. This provides a comfortable atmosphere in which to start establishing the basic principles of the method.

The later examples, building on the established principles, are designed to help you understand what is happening in the re-engineering and to develop a feel for how entities are transformed into objects. You will learn how to construct simple general objects that, through re-use, compact the object model. And, you will begin to appreciate how accurate your analysis needs to be within the object paradigm.

3.3 Entity formats based on working computer systems

All the entity formats in the examples have come from working computer systems. I have changed some details, for reasons such as confidentiality and ease of understanding. They are common formats that could have come from any number of systems. In the examples, we assume that they come from a single working computer system (this is not too far from the truth).

4 A systematic approach to re-engineering

A systematic approach to re-engineering entity formats is made possible by the foundation work done on the re-engineering of the entity paradigm into the object paradigm. Over the years, I have refined the systematic approach into a more formal method called REV-ENG. This is a nick-name that a modelling team gave to an early version of the method; it is a contraction of REVerse ENGineering. (When we first started, we thought that what we were doing was reverse engineering. It was only later that we realised we were re-engineering; hence, the name.) The method simplifies the re-engineering process by formalising the approach into the step-by-step system illustrated in the worked examples.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

4.1 Re-engineering data

The method focuses on re-engineering data—in particular the entity formats in an existing legacy system. In my experience the data in most systems contains enough of the important patterns to enable us to construct the general patterns needed for a comprehensive business model. (Remember, as we discussed in ***Chapter 2***, that the data contains patterns for both business bodies and events.)

Process is deliberately ignored by the method. In the early days, we attempted to re-engineer both data and process. However, we soon found that it is practically impossible to effectively re-engineer process into a business model. The problem is that process has a language-like linear structure. (That is why we talk about programming *languages* in contrast to database management systems.) Like everyday language (and unlike paper tables and computer data), these linear constraints so seriously distort process's view of the business that it cannot be re-engineered systematically. This has never caused us a problem. We just simplified the re-engineering by ignoring process. This was no loss; the more explicitly structured data contained more than enough of the object patterns we needed (including event patterns).

4.2 Stages in the method

The worked examples make more sense if we first look briefly at an outline of the method. The method divides the re-engineering into two main stages:

- Re-engineering the entity format of the existing entity system, and
- Re-engineering our conceptual patterns for the entity format.

These are illustrated in ***Figure 11.2***.

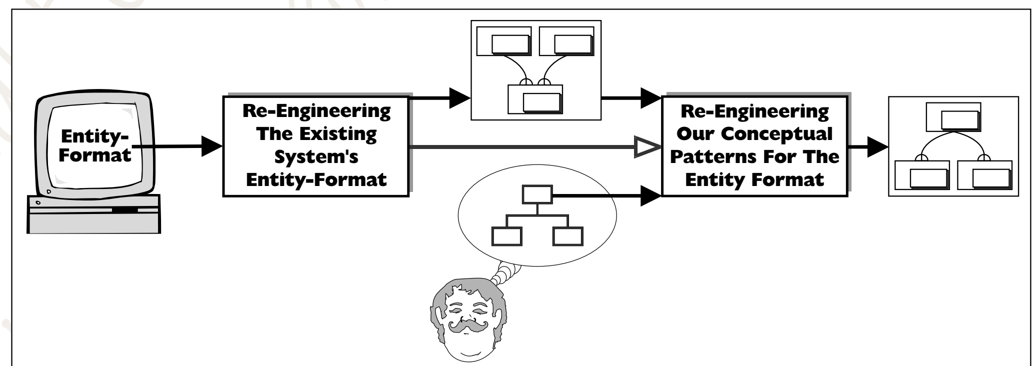


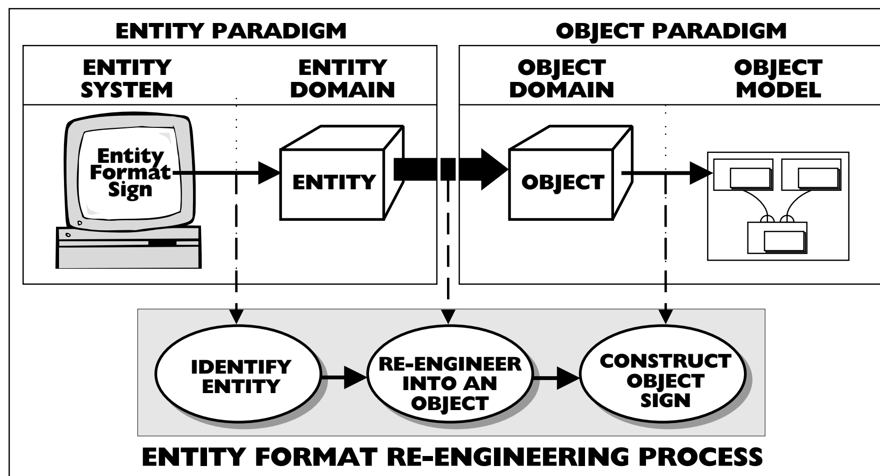
Figure 11.2: Two stages of re-engineering

4.2.1 Re-engineering the entity format of the existing entity system

The elements of the entity format are re-engineered one by one. Each re-engineering starts with a sign in the entity format; from it an entity in the 'real world' is identified. This entity is then re-engineered into an object and a sign constructed for the object in the model. This process is illustrated in ***Figure 11.3***.

© Copyright Chris Partridge chris.partridge@BORO Centre.com

Figure 11.3: Re-engineering the existing system's entity format



4.2.1.1 Rules for ordering the elements of the re-engineering

I have found that it is worth following two simple rules when choosing the order in which to re-engineer the various elements of the entity formats. This makes the whole process of re-engineering much more straightforward.

The first rule is:

> Re-engineer the individual entity and entity type signs before their associated individual attribute and attribute type signs.

This is only common sense. We obviously need to work on the entity sign, before we can move onto its dependant attribute signs.

The second rule—for individual entities and entity types—is:

> Re-engineer a couple of individual entity signs and use the patterns to re-engineer their entity type sign.

This again is common sense. It is much easier to establish the patterns with more tangible individual entities than with 'abstract' entity types. This is sometimes called working by example.

The second rule only needs a change of name to apply to attributes. After the change, it reads:

> Re-engineer a couple of individual attribute signs and use the patterns to re-engineer their attribute type sign.

4.2.2 Re-engineering our conceptual patterns for the entity format

In the second stage, of the process we re-engineer our conceptual patterns (in other words, the patterns our brain uses) for the entities in the entity format. This follows similar steps to the first stage, with the addition of one initial step. We:

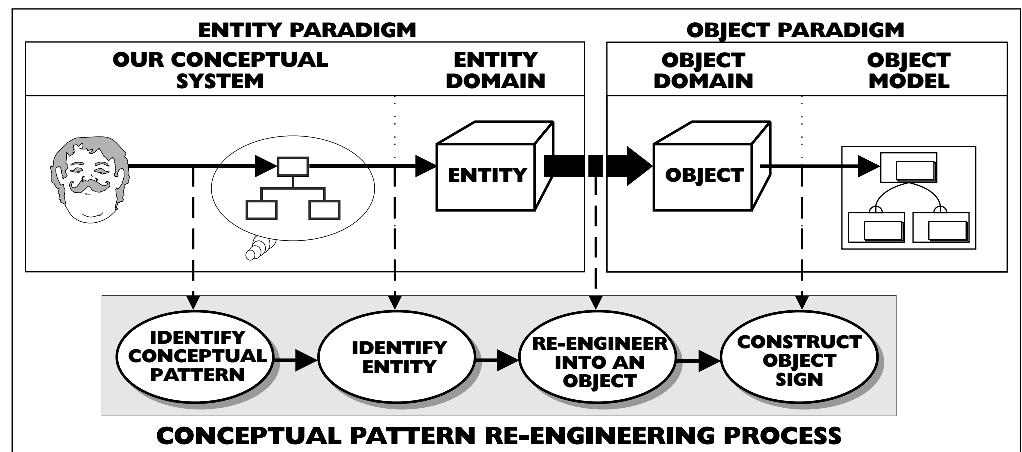
- Find a relevant conceptual pattern,

© Copyright Chris Partridge chris.partridge@BORO Centre.com

- Identify the entity (entities) the conceptual pattern refers to,
- Re-engineer it (them) into objects, and
- Then construct signs for the objects in the model.

These steps are illustrated in ***Figure 11.4***. The reason we need the extra first step of finding a relevant conceptual pattern is that—unlike the entity formats in the existing computer systems, these are not formally listed. It involves some effort, and in some cases, ingenuity to find relevant ones.

Figure 11.4: Re-engineering our conceptual patterns



4.3 Following the systematic method

In the worked examples, these two stages (and the steps within them) occur repeatedly. However, once we have firmly established the steps in the process, we focus on the nature of the re-engineering. This means that, in later examples, we do not work through each step in detail.

This should not be seen as a licence to miss out steps when you re-engineer. It is tempting to do so, but you should resist. Go through every step in the process, especially the first few times you re-engineer. Even someone who is experienced can easily take a wrong turning. Relying on intuition or a gut feeling, particularly at the start, is a sure recipe for slipping back into the old (entity) way of seeing and almost bound to lead you astray. Following the full process helps keep you on the straight and narrow.

Furthermore, and perhaps as important, when the model is being reviewed, your detailed workings will make it much easier for the reviewer to see any wrong turnings you may have taken. With the workings revealed, the re-engineering will not look like a series of magic rabbits appearing from nowhere out of top hats.

5 A framework for the model

In the worked examples, you will see that the objects have been allocated to one of these three broad levels:

- Framework level,
- Application level, and
- Operational level.

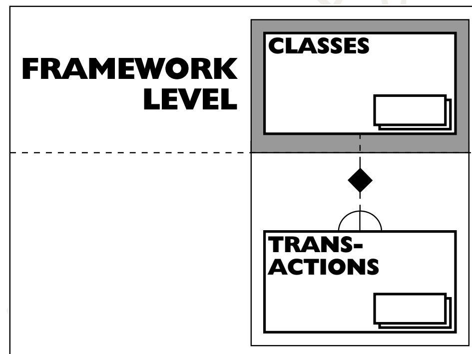
I have found that this not only makes the model easier to understand but helps in the system building process.

5.1 The framework level

The framework level contains the object meta-model. I have found it helps to have this explicitly described in the model. So, rather than start each project with a blank sheet of paper, I start with a framework meta-model.

In the object schemas, I identify the framework level with a background shading. ***Figure 11.5*** has a sample. Any model object within the shading is a framework object.

Figure 11.5: An example of framework level shading



5.2 Other levels in the model

There are two non-framework levels:

- The application level, and
- The operational level.

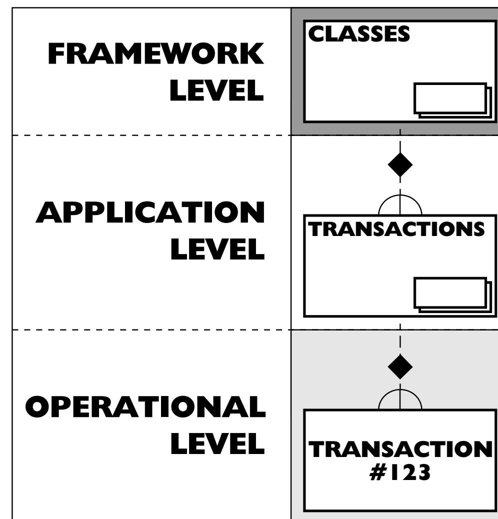
I call those model objects that will be built during the system development, application level objects. I call those that are typically set up during live operation by the users of the finished system, operational objects.

I use a similar shading method of identification to differentiate these two levels of objects. I shade the background of the operational level objects; so, by default, the application level objects have no background shading. ***Figure*** shows an example of the different shadings. In it the class transactions is in an area with no shading signifying it

© Copyright Partridge chris.partridge@BOROCentre.com

is application level; whereas Transaction #123 is in an area of light shading signifying that it is an operational level. As you can see in the figure, the framework shading is darker than the operational shading.

Figure 11.6: An example of operational level shading



I find it useful to start modelling with individual objects, particular examples of the more general classes. These are more tangible and help me (and others) to see the underlying patterns more clearly. These individual objects are often operational level, objects that the users would set up. Consequently, I usually end up with quite a number of operational objects in my working model. When I tidy up the model for the next stage of system building, I purge them because they are not needed.

5.3 Assuming that all classes are application level objects

Working in the entity paradigm, it is easy to assume that the application-operational and type-individual distinctions are the same or similar. In other words, that all entity types are application level and all individual entities are operational level. This assumption is wrong.

Some people carry this mistaken assumption across into object modelling. They assume that all classes are application level and all individual objects are operational level. This places an unnecessary restriction on their modelling. We can show this with examples of classes that are at the operational level and individual objects at the application level.

Consider the car types class modelled in ***Figures 6.8*** and ***6.9***. This has a member class, Minis. If we assume all classes are application level, we would classify the Minis class as application level. However, if someone was building a general package for car manufacturers, it would make no sense for them to construct in the system 'individual' car types, such as Minis, that vary from manufacturer to manufacturer. They would let each manufacturer set up their own. So Minis, despite being a class, is an operational object.

© Copyright Chris Partridge chris.partridge@BORO Centre.com

Consider a general Bank of England reporting system for banks. We would expect this to contain information about the Bank of England as standard. Because it is an individual object, not a class, we might be tempted to allocate it to the operational level of our business model. Yet, it is not something that the users at individual banks would be expected to set up. So, despite being an individual object, it is an application level object. This and the last example are modelled in ***Figure 11.7***.

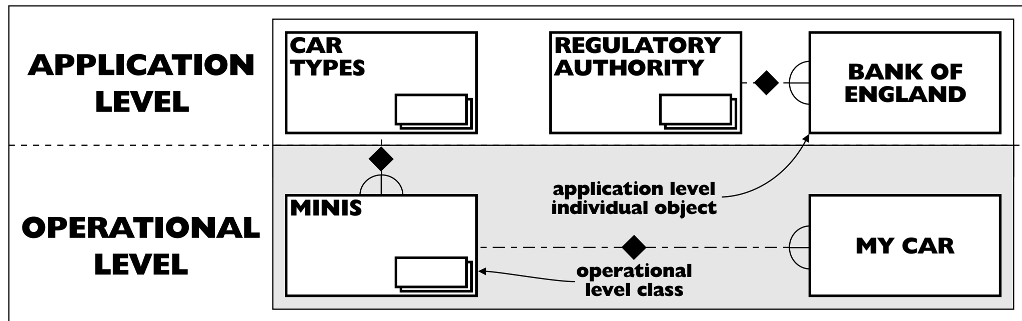


Figure 11.7: Examples of application and operational level objects

5.4 Levels as objects

In the object paradigm, with its strong reference principle, the level shadings are signs referring to objects—level objects. The framework level object is the meta-class of all the framework objects. Each level object is the meta-class of all objects in that level. These meta-classes are modelled in ***Figure 11.8***, which also shows the individual application and overlapping operational objects illustrated in ***Figure 11.7***—though without their connections.

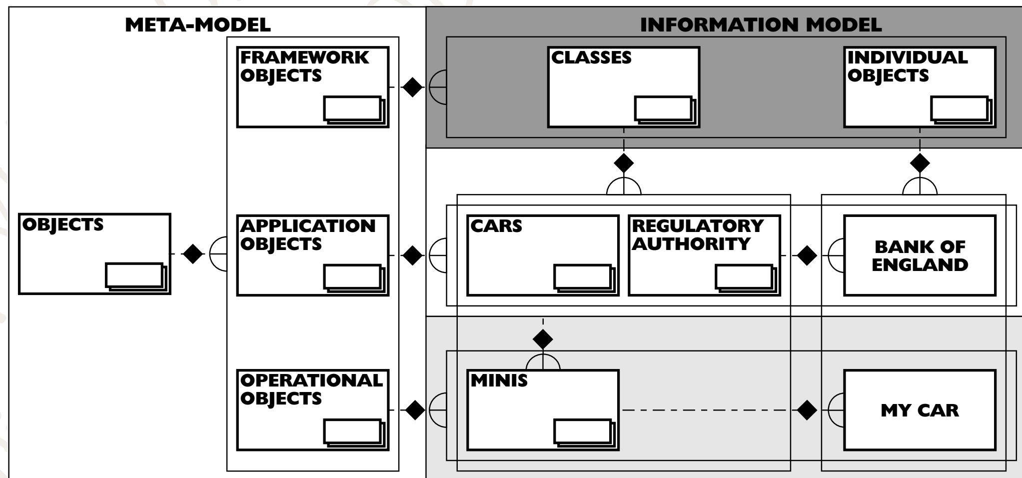


Figure 11.8: Level objects

5.5 Expanding the framework level—the general lexicon

I find it useful to expand the framework level into a 'starter pack' for future re-engineering projects. I call this a general lexicon.

5.5.1 *What is a general lexicon?*

In everyday language, a lexicon is a store of words or concepts. Dictionaries can be seen as, and in some cases, have been called lexicons. As far as a business object modelling is concerned, the general lexicon is a store of general re-usable model objects.

I have found that I can manage generalising and identifying similar objects across projects more effectively, if I use a general lexicon. This stores the model objects that have fruitful patterns, ones that are re-used in most models. I classify all these as framework level. So, for all practical purposes, the general lexicon is the framework level.

Sometimes I have found, and you will find, it useful to create specialist lexicons for specialist areas of the business. For instance, if you were to work on a number of accounting systems, you might construct an accounting lexicon. These would contain models of the accounting objects that you found were re-used in most of the accounting models.

5.5.1.1 *General lexicon as a transformation of the Aristotelian categories*

The general lexicon can usefully be seen as the object paradigm's transformation of the Aristotelian categories (discussed in ***Figure 4*** and illustrated in ***Figure 4.14***). The object meta-model embedded in the framework level is the transformed version of Aristotle's framework of categories—substance and types of attribute (quality, quantity, etc.). The rest of the general lexicon is the transformed version of the lower level Aristotelian categories. It records the same types of things, such as the humans class being a sub-class of the more general animals class.

5.5.2 *Constructing a general lexicon.*

An essential precursor to the construction of a useful general lexicon is identifying model objects that will be re-used. If, over a sufficiently large number of models, the model of a particular object (or group of objects) is re-used frequently, then it is a likely candidate for the general lexicon. The naming pattern that appears in all of the first group of worked examples is a good example.

With experience, modellers find that they can judge whether model objects are candidates for the general lexicon. There is a useful rule of thumb that helps them make that decision. It is the more general a model object, the more likely it is to be re-used; the less general, the less likely it is to be re-used. Here, we measure generality in terms of how high up either the class–member or super–sub-class hierarchies the object is. The class objects is the limiting case; it is at the top of both hierarchies. As we move down the hierarchies from objects, we get closer and closer to the border of the general lexicon (framework level) and the application level (illustrated in ***Figure 11.9***).

© Copyright Chris Partridge
chris.partridge@BORO-Centre.com

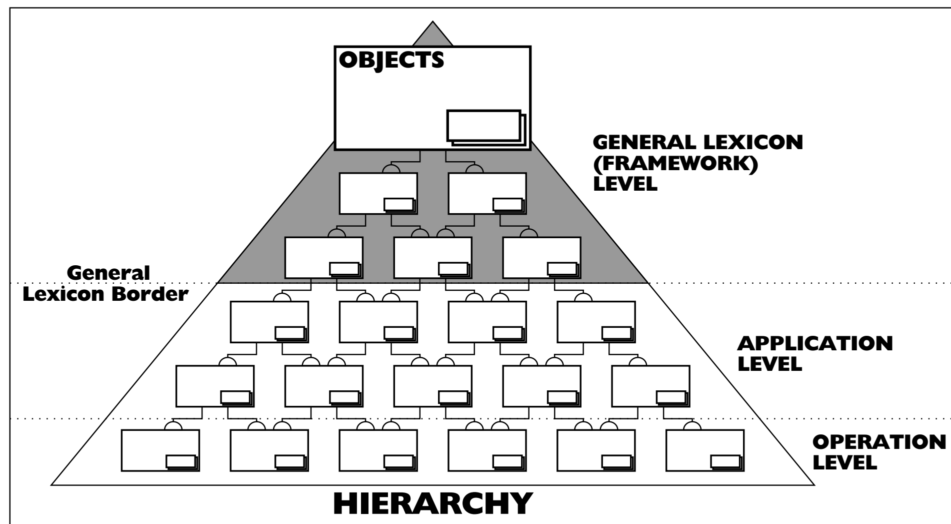


Figure 11.9: The general lexicon border

The exact location of the general lexicon border is a matter of opinion. Different people sometimes have slightly different views. In the end, it does not matter exactly where the border is, so long as the majority of really re-usable model objects are inside it, ready for re-use.

When we decide a model object belongs to the general lexicon, implementing this decision is trivial; we classify it as framework level. Because the general lexicon is tied into the framework level, whenever we classify a model object as framework level, it automatically belongs to the general lexicon.

6 Generalisation and compacting

A key objective of business object modelling is generalising objects. This is what delivers compacting and its associated benefits.

6.1 Spotting that objects share the same patterns

An important element in generalisation is spotting that objects share the same patterns. The schemas are a useful tool for this. They describe the pattern of relations between objects. This is what Frege called the sense element of meaning—distinguishing it from the reference element (we looked at Frege's description of meaning in the ***Figure 5***). The object schemas (and so the business model) map Fregean sense patterns. Mapping these sense patterns creates an environment that encourages generalisation. The object schemas play a key role. They can make the sense patterns explicit in a way that makes similar patterns easier to spot.

The way in which the schema is drawn influences how it is understood. In well drawn schemas, similar patterns have similar shapes, making them easier to spot. However, a badly drawn schema can make two similar patterns look dissimilar. So, it is worth

© Copyright Chris Partridge's BORO Centre
chris.partridge@BORO...

spending some time drawing an object schema properly. I find that I sometimes go through five or more drafts before I arrive at something I am happy with. This is worth doing because it encourages substantial generalisation, and so substantial compacting.

6.2 Compacting metrics

I find that I can get a rough idea on how successful the re-engineering is by monitoring the scale of generalisation and compacting.

6.2.1 Population, re-use and generalisation metrics

I usually measure the compacting directly with population metrics that compare the number of items in the existing entity system with the number in the object model. I also measure the actual amount of re-use and generalisation that has occurred in the re-engineering of the business model. By comparing the two sets of metrics, I can see the correlation between generalisation, re-use and compacting.

6.2.2 Model levels

I normally divide the metrics into levels. I do this for both the entity system and object models' metrics. The entity system's metrics are divided into the following two levels:

- Type, and
- Individual.

The object model's metrics are divided into the following three levels:

- Framework,
- Application, and
- Operational.

The framework level objects are separated so that they can be excluded from the overall metrics. We exclude them because they are not constructed during the re-engineering. They are part of the framework model rather than the model for the particular re-engineering.

The application and operational level objects are included in the overall metrics, but a distinction is made between the two because only the application level objects are passed onto the systems analysis stage. By contrast, operational objects are eventually purged from the business model. These are just examples of the types of objects that the user will construct after the system has gone live. So these two levels can be used to separate the impact on compacting of generalisation during development—the application level—from that during operation—the operational level.

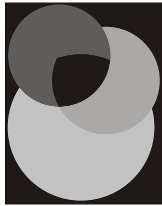
Even though we formally purge the operational objects from the business model, we often use them during the later stages of system building. For example we might use them for illustrative examples during the systems analysis and also as the basis for the test data used in system testing.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

7 What's next

Enough of this 'abstract' discussion of re-engineering. It is time to look at the worked examples, including their metrics. We start in the next chapter with the first group—spatial patterns—and its first entity format—country.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 12

Re-Engineering Country's Entity Format

1 Introduction
2 The systematic re-engineering process
3 The context for re-engineering
4 Re-engineering country entity type sign
5 Re-engineering attribute type signs
6 Basic elements of the re-engineering completed

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction

We now start on the re-engineering of country—the first of three spatial patterns (listed in ***Figure 12.1***). In this worked example, we start growing our object model for spatial patterns by re-engineering the country entity format. We will extend the model in the next two worked examples by re-engineering the region and address entity formats. The final model then forms a very basic building block that we can re-use repeatedly when we re-engineer (and build) business systems.

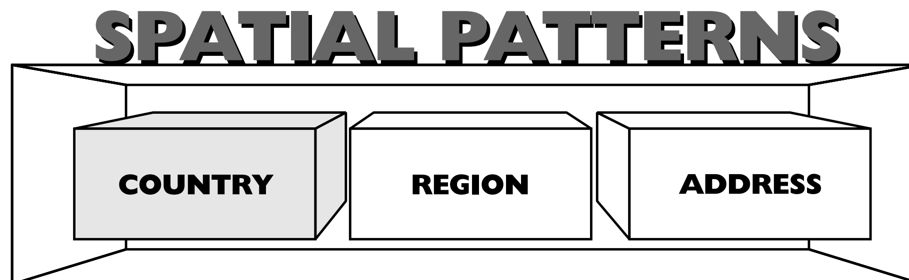


Figure 12.1: First of three examples of spatial patterns

We start by re-engineering the spatial patterns embedded in the country's entities and entity type. This is a simple and straightforward task. It will appear mostly as just good common sense—careful, accurate, and maybe a little pedantic—but still common sense. This establishes the patterns of re-engineering, which we can then (re-)use later without too much explanation.

We then re-engineer the country format's two attribute types, both names, to reveal a common name pattern. Most entities have name attributes and so, in a re-engineering of any size, the name pattern is bound to crop up many times. This makes it is a very useful pattern.

The name pattern had to be stretched and twisted to fit into the entity paradigm's framework. So the example provides us with a good illustration of how the process of re-engineering unwinds distorted entity formats, and the kind of unfamiliar objects this can produce. As we shall see, this involves paying particular attention to the accuracy of the analysis.

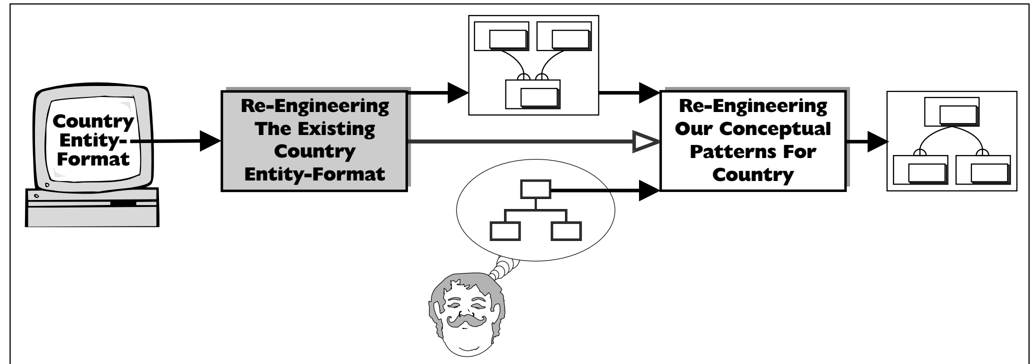
Once we have found the new name pattern, we need to absorb it into our way of seeing. Unfortunately, most of us have the current distorted entity name pattern deeply embedded in the way we see. So, even though the undistorted object name pattern is clearer and simpler, it can appear odd and awkward at first sight. But once we learn to see with the new pattern, we appreciate its superiority.

2 The systematic re-engineering process

We follow the systematic re-engineering process, working in two stages as described in the previous chapter and shown in ***Figure 12.2***. In the first stage, we re-engineer the

existing system's country entity format into a model that maps business objects. At the second stage, we re-engineer our (typically entity-based) conceptual patterns—in other words, the ideas in our heads—into business objects and include them in the model.

Figure 12.2: The two stages of re-engineering



In this first example, we build up our understanding of how the re-engineering works, by going through it in some detail. This takes up quite a lot of space and so is spread over three chapters. In the first two chapters, we go through the first stage—entity format re-engineering. In this first chapter, we re-engineer the elements of the country's entity format. In the second chapter, we generalise the country's re-used patterns. In the third chapter, we go through the second stage—conceptual pattern re-engineering.

3 The context for re-engineering

We now get down to the nitty-gritty of entity format re-engineering. The first step is getting a context, finding out what we are going to re-engineer. We know it is the country entity, but we need to know more than this..

Country Name	Country Code
Germany	DM
Italy	IT
Japan	JP
Turkey	TK
United Kingdom	UK
United States	US

Table 12.1: Partial Country Listing

People can understand a lot about what a particular entity format is by looking at examples of its entities and their associated attributes. In this worked example, we use the Partial Country Listing given in ***Table 12.1***. This is the sort of 'file listing' a computer system constructed using the entity paradigm would produce. Here, we are only interested

© Copyright Chris Partridge chris.partridge@BORO Centre.

in the two name attributes shown in the listing; we ignore any other attributes country may have. From ***Table 12.1***, we can deduce that the relevant part of the country format looks like ***Table 12.2***

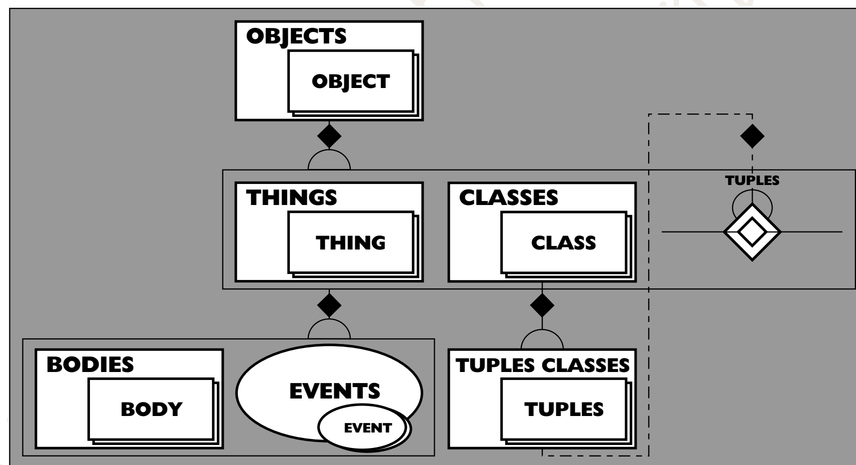
Attribute Type #1	Attribute Type #2	Etc.
Country full name	Country code	-

Table 12.2: Country Entity Format

3.1 The re-engineering framework

When we start the re-engineering, we do not have to grow the new model from scratch. We use a framework model or general lexicon as our starting point (see ***Chapter 11***). We do not need the full framework for this example; we only take the partial framework shown in the object schema in ***Figure 12.3***, and grow the country object model underneath.

Figure 12.3: Partial framework object schema



3.2 The systematic approach to re-engineering

The three steps in the systematic approach to re-engineering the country entity (type)—one of the country entity format's elements—are illustrated in ***Figure 12.4***. Notice the process starts with the country entity sign, not the country entity, and ends with a country model object, not the country object. This is because we are re-engineering information; so we re-engineer from entity signs to object signs. The heart of the re-engineering process is, however, entities, objects and their semantics, not their signs. Before we can construct the object sign in the object model, we have to:

- Identify the entity that the entity sign refers to, and
- Re-engineer it into an object (or, in some cases, a number of objects).

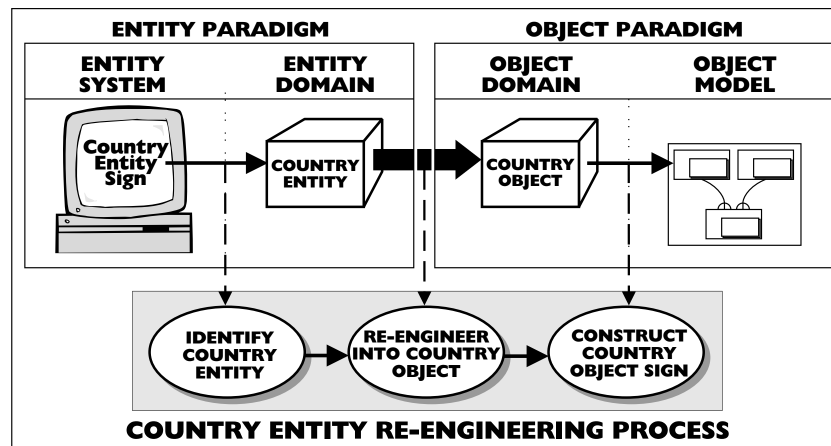


Figure 12.4: First stage – re-engineering the country entity

3.3 Following the ordering rules for re-engineering

We now follow the two simple rules for ordering the re-engineering of entity formats set out in the previous chapter. The first rule applied to country is:

> Re-engineer the country individual entity and entity type signs before their associated individual attribute and attribute type signs.

The second rule—for the country entity type—is:

> Re-engineer a couple of the country individual entity signs and use the patterns to re-engineer the country entity type sign.

4 Re-engineering country entity type sign

Following the rules, we start with the country entity type sign and select one of its entity signs from the Partial Country Listing in ***Table 12.1***. It does not matter which one, because they should all exhibit country patterns. We pick the last entry in the table as our first sign—the United States entity sign.

4.1 Re-engineering the first individual entity sign

This table entry is a record of a United States entity sign on our existing computer system. We have no real problem finding the sign, it is the United States record on the Country File. We now need to work out what individual entity this sign refers to.

4.1.1 Identifying the United States entity

As ***Figure 12.4*** shows, at this stage we are working within the entity paradigm. In ***Chapter 4*** , we looked at how the substance paradigm's semantics explained what individual entity signs referred to. That they refer to individual entities constructed from an under-

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

lying primary substance, to which attributes are attached (illustrated in ***Figure 4.1***). We apply the explanation to this particular example. The United States individual entity sign refers to the 'primary substance' of the country we call the United States. The primary substance and its attributes, in some sense, make up the United States. This is shown diagrammatically in ***Figure 12.5***.

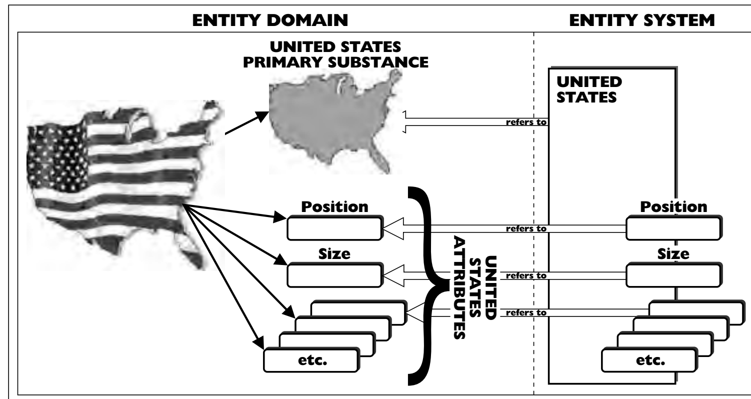


Figure 12.5: United States primary substance

4.1.2 Re-engineering the United States object

We now re-engineer the country primary substance into a country object. In the object paradigm, we see in four-dimensional terms, so we see the United States object as occupying both space *and* time. The re-engineering is relatively simple. We take the United States current three-dimensional extension—in entity paradigm terms, the position or place attribute. We then follow the United States' three-dimensional extension back and forward in time, filling in a four-dimensional extension. This process is illustrated in ***Figure 12.6***. The resulting four-dimensional extension is the United States object.

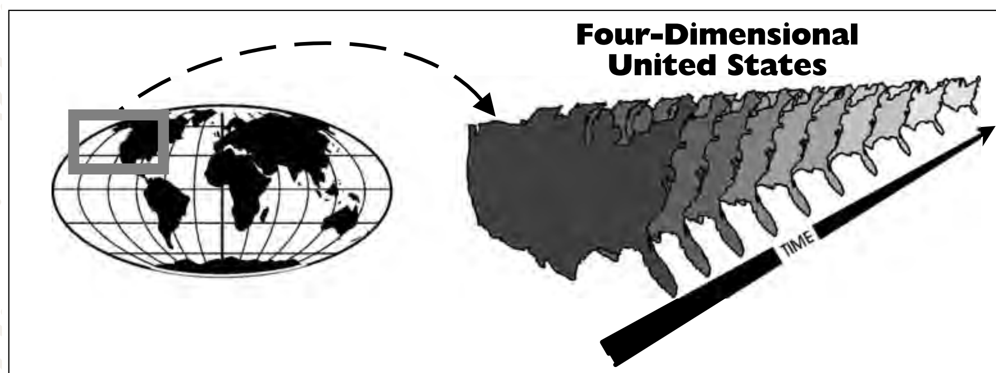


Figure 12.6: Four-dimensional object, the United States

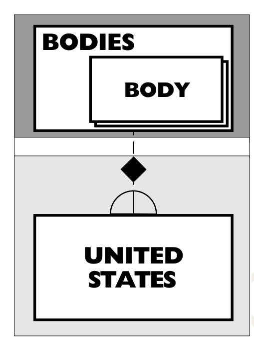
To increase coherence, we need to find a connecting pattern between the United States object and the framework objects (those shown in ***Figure 12.3***). To do this we ask—what type of object is the United States object? The answer is clear; it persists through time and so is a body object. This gives us our pattern. There is a class–member pat-

tern linking it to the framework's bodies class; in other words, it is a member of the bodies class. This introduces a second object; the class–member tuple constructed from the two objects.

4.1.3 Constructing the United States object sign

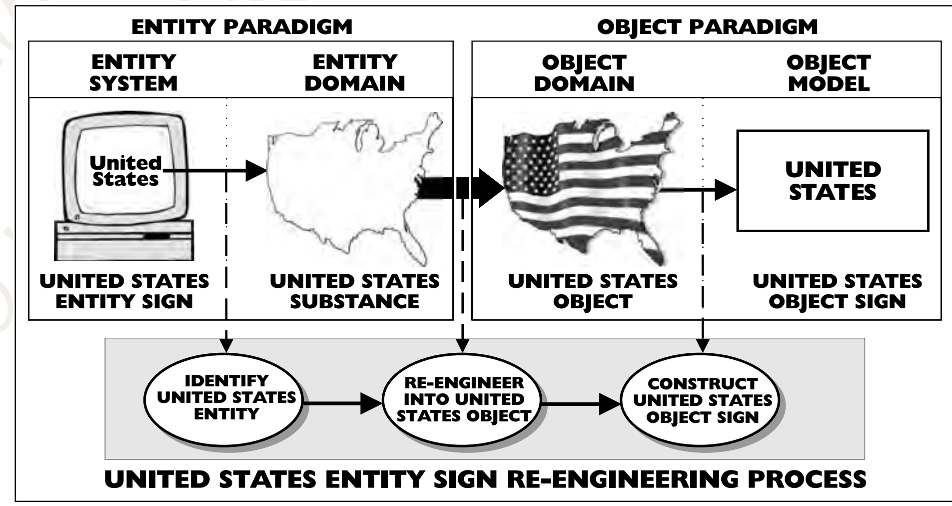
We carry out the final step in re-engineering by constructing signs for the two objects. We construct an individual body object sign for the United States object and a class–member sign for the tuple object. The resulting model is shown in the schema in ***Figure 12.7***. This also shows how we used a class–member tuple to graft the new United States object onto the framework's bodies class.

Figure 12.7: United States object schema



In ***Figure 12.7*** the framework, application and operational levels are signed using the shading convention described in ***Chapter 11*** (and illustrated in ***Figure 11.6***). The United States sign is in the operational level. This is because the users of the system are expected to decide what countries to set up on their system, and so whether to set up the United States. Operational objects, like this one, naturally emerge during the re-engineering. We eventually purge them when we produce an application version of the model for the system designers to work on.

Figure 12.8: Re-engineering the existing system's United States entity sign



© Copyright Chris Partridge
chris.partridge@BORO Centre.com

4.1.4 *The re-engineering of the United States entity sign*

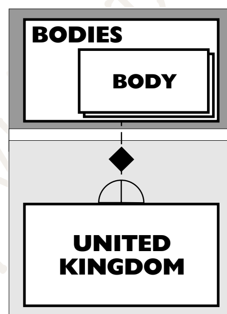
We have now worked through the three steps of entity format re-engineering. We have re-engineered the United States entity sign into the United States object sign. This is shown in ***Figure 12.8***. In sign terms, this might not look like much of a change, but, in semantic terms, things in the domain have changed dramatically. These differences emerge in the model as it captures the pattern of connections between objects.

Some people may find the analysis above far too detailed, thinking that we are wasting time dealing with the obvious. They might think it is clear what the United States is; though I suspect it may not be clear to many of them that it is a four-dimensional object. However, it is important to recognise that we are shifting to a full blown object paradigm. Remember that the whole point of a paradigm shift (particularly one as fundamental as this) is that it *makes* us look at things in a new and different way. What we previously thought clear and obvious now becomes suspect. We have immersed ourselves in the entity paradigm for so many years that we will need to work hard to overcome the habits and prejudices we have built up. Only careful and systematic analysis of exactly what things are, as in this example, will keep us on the straight and narrow.

4.2 Re-engineering the second individual entity sign

Following the rule about re-engineering a couple of entity signs, we pick a second entity sign and re-engineer it. We pick it from the partial listing of countries in ***Table 12.1***. This time we pick the United Kingdom sign. We follow the same pattern of re-engineering as we used for the United States entity sign. Because the pattern is the same, the details are not repeated here. We end up constructing a sign for the United Kingdom object and including it in the object model. This gives us an object schema that looks like ***Figure 12.9***.

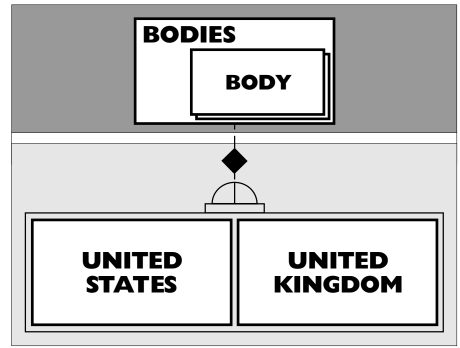
Figure 12.9: United Kingdom object schema



Not surprisingly, it has the same pattern as the United States object schema in ***Figure 12.7***. We now integrate the two schemas and get ***Figure 12.10***. Because the individual countries are only two of many, the class–member tuple sign to the partition box is partial (shown by the small rectangle).

© Copyright, Chris Partridge chris.partridge@BoroCentre.com

Figure 12.10: Merged United States and United Kingdom object schema



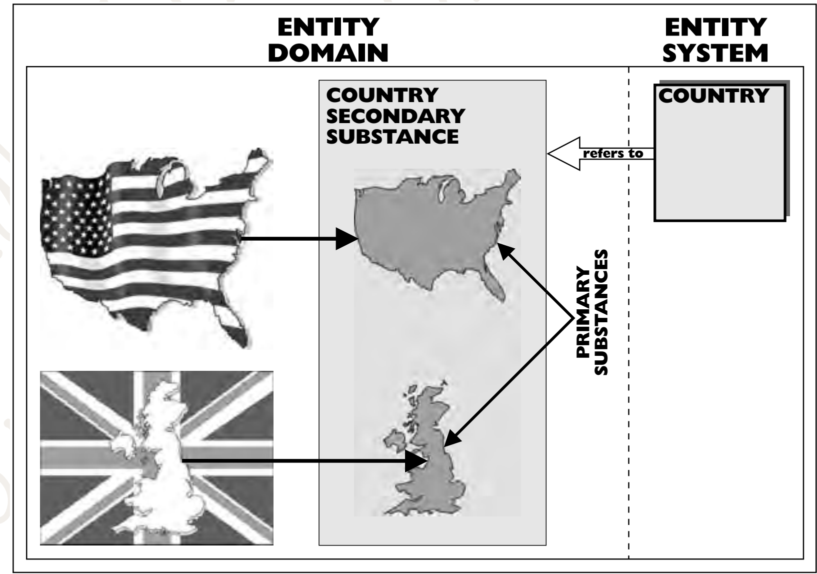
4.3 Re-engineering the entity type sign

We are now ready to re-engineer the country entity type sign. We follow the same three steps we used for the entity signs.

- We identify the country entity type the sign refers to.
- Then we re-engineer the country entity type into a country object.
- Then we construct a country sign in the object model for the country object.

We looked at what an individual entity sign referred to when we re-engineered the United States entity sign earlier in this chapter. It referred to the primary substance to which attributes attach. We now ask—what does the country entity type sign refer to?

Figure 12.11: Country secondary substance



In ***Chapter 3***, we saw that the entity type sign refers to a secondary substance. One that we can see as the sum of primary substances. Using this pattern, the country entity type sign then refers to country secondary substance (shown in ***Figure 12.11***). We can

© Copyright chris partridge BORO Centre com

see this as the sum of the primary substances of countries like the United States and the United Kingdom.

We now need to re-engineer the entity type—the country secondary substance—into an object. In ***Chapter 4*** (on the logical paradigm), we looked at a pattern where secondary substances evolved into classes. We use this pattern here and transform the country secondary substance into the class of countries. Because the individual countries have four-dimensional extensions, so does the class.

We now work out the pattern of structural interconnections with the existing model. The countries class has as members the individual physical body objects: United States and United Kingdom. It is a class; so, it is a member of the framework class, classes. Its members are all physical body objects; so, it is a sub-class of the bodies class.

Now that we have re-engineered the class, we construct the countries class sign and the signs for its patterns of structural connections in the object model. ***Figure 12.12*** shows the results. Because we are working in the object paradigm's notation, the countries class sign (unlike the entity type sign) differentiates between the name sign for the class—countries—and the name sign for its members—country. Note that the countries class is an application, rather than operational, level object—one that we will use to specify the system.

Figure 12.12: Countries object schema

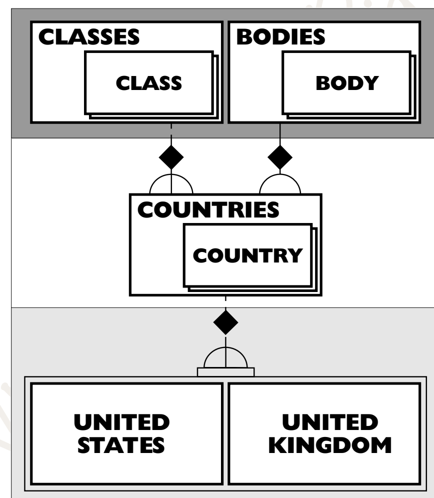


Figure 12.13 illustrates the overall re-engineering process for the entity type.

We step back and look at both the objects and the object model from outside using the reference diagram in Figure ***Figure 12.14***. There, we can see how the pattern of connections between the countries class sign and other signs in the object model reflects the pattern of connections between the objects.

The re-engineering of the country entity type and its associated entity is now completed. This step-by-step walk through the process should have given you a clear idea of how the systematic REV-ENG method works.

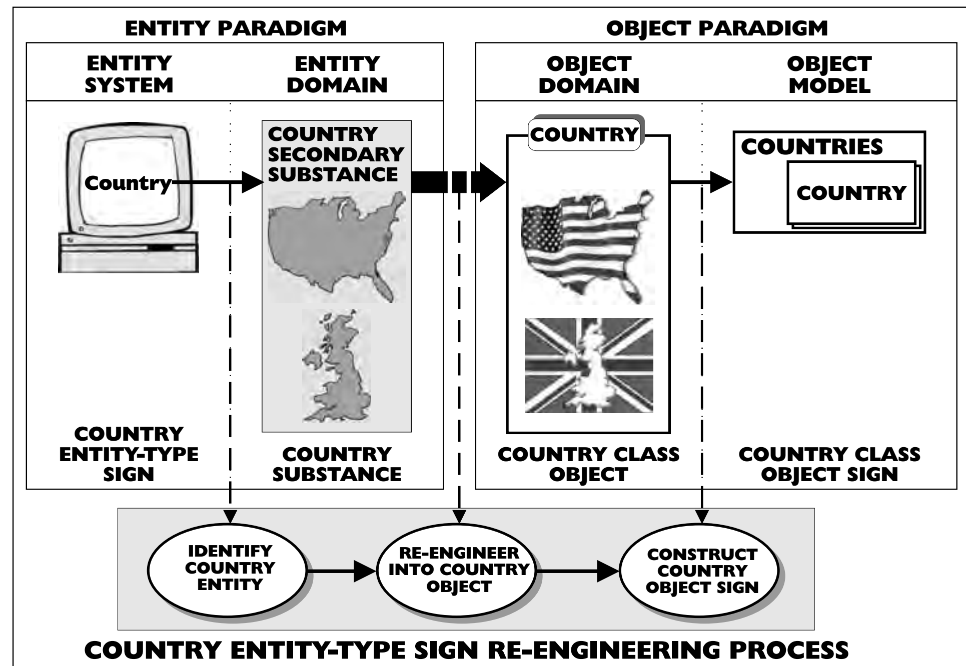


Figure 12.13: Re-engineering the country entity type

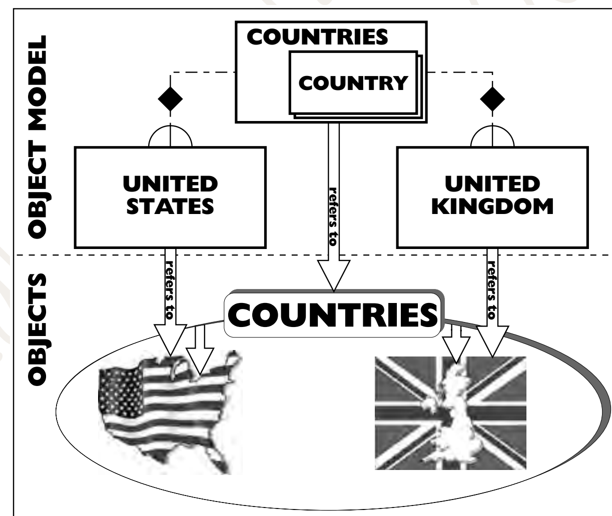


Figure 12.14: Countries class reference diagram

5 Re-engineering attribute type signs

We are now ready to start re-engineering the country entity format's attribute type signs. Both of these are names but, before we start, I warn you again that names work in a very different way in the entity and object paradigms. Some of us are so acclimated to the successful way they work in the entity paradigm that we are going to need to work hard to get to grips with the new perspective—even though it is simpler.

5.1 Re-engineering the country full name attribute type sign

We start the re-engineering with the country full name attribute type sign. The attribute version of the second rule applies:

> Re-engineer a couple of individual attribute signs and use the patterns to Re-engineer their attribute type sign.

We need to pick individual attribute signs for our attribute type sign. We start by picking the United States entity sign's country full name attribute sign 'United States'.

5.2 Re-engineering the first individual attribute sign

We again follow the three steps in our systematic approach. We start with the actual attribute sign itself in the computer system. This is the full name field of the United States record and contains the character string 'United States' (recorded in the first column of the last entry in ***Table 12.1***).

5.2.1 An implicit relational attribute sign

This full name field is an attribute sign and so refers to an attribute, but identifying which one is not straightforward. Although the attribute sign looks like a non-relational attribute sign, it is actually an implicit relational attribute sign related to an implicit United States full name entity sign (shown in ***Figure 12.15***). It is not uncommon to find these implicit signs when re-engineering. They are the result of trying to use the entity paradigm to describe patterns that do not fit into its constrained structure. We re-engineer this implicit relational attribute sign in two steps. We start by re-engineering the implicit 'United States' full name entity sign and then re-engineer the relational attribute.

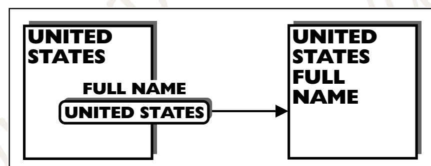


Figure 12.15: Implicit relational attribute

5.2.2 United States full name entity

The 'United States' full name entity sign refers to the 'United States' full name entity. This is not particularly informative; it is easier to understand what the entity is by working out what object it is re-engineered into. What object can this be? It cannot be a single character string, such as the string in the full name field on the computer's United States record or the string in the last entry of the partial country listing in ***Table 12.1***. It somehow refers to both of these, as well as all other 'United States' character strings. This gives us our clue—it is the class of these character strings—as shown in ***Figure 12.16***.

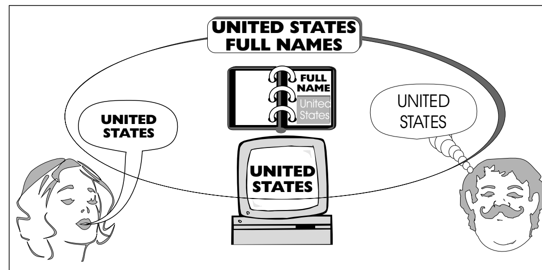


Figure 12.16: United States full names class

To tie the class's reference down, we need to tie down the reference of the members of the class—in this case, the individual character strings. We need to see these as four-dimensional objects, persisting through time. The individual character string objects created by different 'technologies' have different characteristics. Characters in the written strings produced by pen and paper technology persist through time together. For example, the 'U' and 'n' of 'United States' exist at the same time, as shown in ***Figure 12.17***.

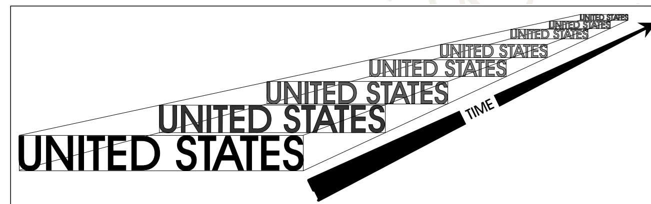


Figure 12.17: Example of a particular four-dimensional written United States full name

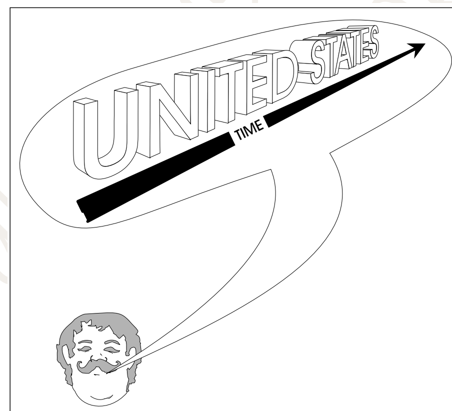


Figure 12.18: Example of a particular four-dimensional spoken United States full name

Spoken characters in strings, the product of speech technology, have their characters spread through time. In this case the 'U' exists before the 'N', as shown in ***Figure 12.18***.

5.2.3 United States full names class

These four-dimensional objects are the members of the United States full name class. Every time someone says, writes down or, most importantly for us, keys into a computer system, the character string 'United States', and this names the United States; then the character string belongs to the United States full names class. This analysis means that in the object scheme of things, we read the sentence;

The full name of the United States is 'United States'

as

'United States' is a member of the United States full names class.

We now follow the same pattern as before and work out the structural connections for the class object. We have already done some of the analysis; we have worked out some of its members. Because the object is a class, it is a member of the framework class, classes. We then construct the sign for the object and its patterns in the object model. The result is shown in ***Figure 12.19***.

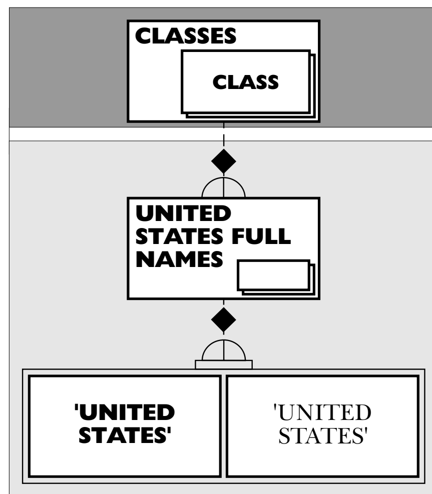


Figure 12.19: United States full names object schema

5.2.4 Full naming tuples class

We now start re-engineering the second part of the attribute—the implicit relational attribute sign that points to the 'United States' full name entity sign. This refers to the full names relational attribute that relates the United States substance to the United States full names entity (shown in ***Figure 12.15***). In ***Chapter 5*** (where we introduced tuples), we looked at the basic re-engineering pattern for relational attributes. It transforms them into a couple belonging to a tuples class. Applying the pattern to this example, the relational attribute re-engineers into the couple <United States body object, United States full names class> belonging to the country full naming tuples class. This is shown graphically in ***Figure 12.20***.

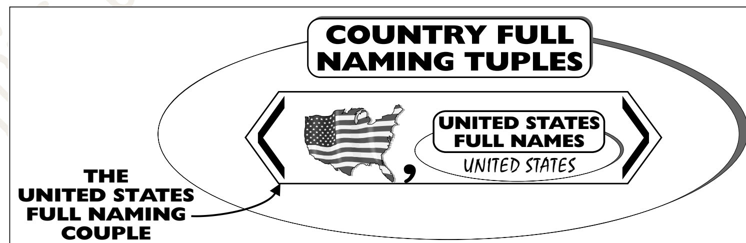


Figure 12.20: The United States full naming couple

© Copyright Chris Partridge chris.partridge@BOROCentre.com

We look for the tuples class and couple's structural patterns. The tuples class is a class of couples and so a sub-class of the framework class, tuples. We construct the signs for the tuples class and couple and its structural patterns and include them in the object model. The resulting object schema looks like ***Figure 12.21***.

Figure 12.21: United States full naming tuples object schema

You will have noticed that the United States full names class is classified as derived. This is shown by the shaded grey triangle at the bottom left of the rectangle. It is defined by the United States full naming couple. This is because a name is only a name if it names something; without the naming couple it is not a name.

5.2.5 The basic naming pattern

The re-engineering of the attribute sign uncovered the tuples class that links the United States to its full name class. But it has not fully explained what this object is and so we cannot see the basic naming pattern clearly. We need to dig a bit deeper.

Even though the United States is connected to the full names class, and we call the connection a naming pattern, we never actually use it to refer to the United States. What we use is the naming pattern class's members; for example, we say 'United States' names the country the United States. This more particular naming pattern is shown in ***Figure 12.22***.

Figure 12.22: United States naming pattern

Every time we use a United States full name (a member of the United States full name class), we get the same naming pattern. In object terms, this particular pattern translates into a couple object, with the country the United States and the character string 'United States' as components. This couple object belongs to the United States full naming tuples class (shown in ***Figure 12.23***). This is an example of the objects behind the basic naming pattern.

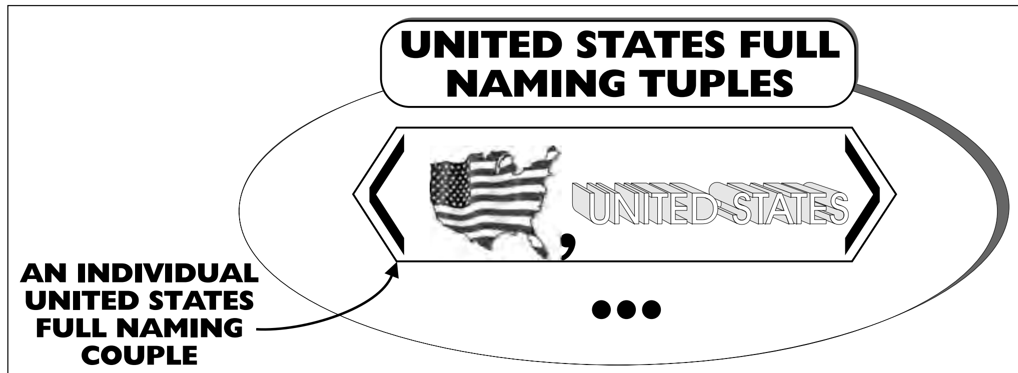


Figure 12.23: United States full naming tuples class

Every member of the United States full names class has the naming pattern shown in ***Figure 12.22***. This suggests a new and better way of re-engineering the full name relational attribute. Might it not be re-engineered into the United States full naming tuples class, instead of the couple <United States body object, United States full name class> we re-engineered earlier? This offers a much more satisfactory explanation of what is going on.

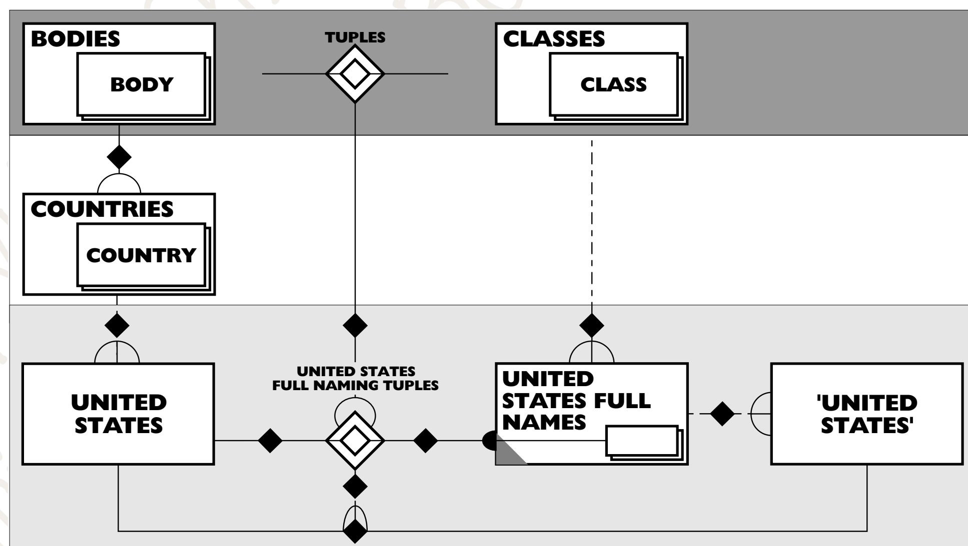


Figure 12.24: United States full naming tuples object schema

We work out the pattern of structural connections, construct the objects and revise the object model. This gives us the object schema in ***Figure 12.24***. The model shows that the link between the United States body and the United States full names class is now a tuples class rather than a couple. It is an unusual tuples class in that one of its links is to

an individual body object instead of to the more usual class's member. You will have noticed that the United States full names class is still classified as derived—shown by the shaded grey triangle at the bottom left of the rectangle. This time it is defined by the United States full naming tuples class.

5.2.6 Exemplar United States full name

The analysis is not yet complete; there is one last bit of investigation we need for the naming pattern. We need to take a more accurate than usual look at how names refer. This will make us revise a simple commonly held notion—that the signs in a model are clearly separated from the things they refer to in the domain (shown in ***Figure 12.25***).

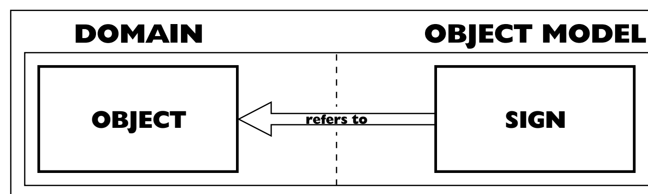


Figure 12.25: Simple view of object model's separation from the domain

This is an accurate enough view most of the time; but for the element of the name pattern we are looking at now, it is a little too simplistic. Most of us assume that ideas are mental and objects physical; so, assume that ideas and signs are totally separate from the objects that they refer to. However, in the object paradigm, signs in the object model are not necessarily mental. Furthermore (as we saw in ***Chapter 9*** when we constructed a (model)^2 model (illustrated in ***Figures 9.39*** and ***9.40***) signs are as much 'objects' as the objects they refer to in the domain. In this scheme of things, there is, in principle, no absolute separation between signs and objects. In fact, signs are objects—model objects. So there is no reason why one of these model objects cannot be in both the object model and the domain.

We can see an example of this if we accurately examine the United States full name attribute again. The attribute is actually an example of a full name. In other words, it is not only an example of the attribute type but also a sign for it. Mixing entity and object domains, we can say the United States full name attribute sign (an element of the existing system) is a member of the United States full name class (an element of the domain).

This is naming by example—a very simple and effective way of naming. It eliminates the need to construct the 'meaningless' signs normally used in language. Coin-operated food dispensers often use a similar system; for instance, the button for dispensing a particular chocolate bar has one of the bars displayed behind the button. The advantage of this way of naming is that there is less chance of misrepresentation. The dispensed chocolate bars can easily be compared with the bar on display.

This is the motivation behind the United States full name attribute sign. When a character string is keyed into the computer system, it can be compared with the sign's character string to determine whether it also belongs to the United States full name class. ***Figure 12.26*** illustrates the state of affairs. It also illustrates a new, more sophisticated view of how the domain and the object model relate. The United States full names class

is partly outside the object model and partly inside. This means the United States exemplar name is in both the domain and the object model; so, the domain and object model are not distinct, they overlap.

Figure 12.26: United States exemplar full name

One aspect of the United States full name attribute sign is that it is an exemplar name for the United States entity. This means that it does not follow the normal pattern of re-engineering. It cannot because the sign is also the entity and the entity is re-engineered into an object that is also an object sign. So, in this case, the first and last steps of the systematic re-engineering process are superfluous.

Figure 12.27: United States exemplar full name object schema

The sign in the object schema for the United States exemplar name needs to reflect its special status as an example referring to a class. We do this in two ways. First, we insert at the end of the name sign the text '[EXEMPLAR]'. And second, we put an arrow over the class–member tuple sign linking the exemplar name to the sign of its class

(shown in the object schema in ***Figure 12.27***). You will have noticed that 'United States [EXEMPLAR]' (like United States full name) is classified as derived, defined by its exemplar class–member tuple. If something occupies the member place of the United States full name exemplar class–member tuple, it must be an exemplar United States full name.

5.3 Re-engineering the second individual attribute sign

We have only re-engineered one attribute for the country full name attribute type. Following the rules, we need to re-engineer a second before we re-engineer the attribute type. We pick 'United Kingdom' from the partial listing in ***Table 12.1***. We follow the same re-engineering pattern as we used for the 'United States' full name attribute. This gives us the object schema shown in ***Figure 12.28***. Not surprisingly, this has the same pattern as the United States object schema in ***Figure 12.27***.

Figure 12.28: United Kingdom full names and full naming tuples object schema

The steps in the re-engineering process are not shown here because they are so similar to the ones we used for the 'United States' full name attribute. When modelling for real, it is tempting to miss out these steps and just re-construct the United States object schemas substituting the United Kingdom. This misses the point of the process, which is to check that the patterns are the same for both attributes before generalising them to the attribute type. Working through a similar process is often tedious; but it is necessary to confirm that the patterns are general across the attributes. I tend to find that the one time I do not work through a similar pattern, Murphy's law comes into operation. The pattern contains important variations that I have to go back later to unearth.

5.4 Re-engineering the attribute type sign

We can now re-engineer the country full name attribute type. This follows a similar pattern to the individual attribute sign, which was re-engineered into a number of related objects. In other words, the attribute signs' pattern is reflected at the attribute type level.

As with the full name attributes, we start by following the three steps in our systematic approach. We take the attribute type sign in the computer system. This is the sum of the full name fields of all the country records on the computer—including the United States full name field and the United Kingdom full name field.

5.4.1 An implicit relational attribute type sign

As with the attribute signs, the first step of re-engineering this attribute type sign (identifying the attribute type the sign refers to) is not straightforward. This is because the attribute type sign is implicitly related (like the attribute signs) to a country full name entity type sign (shown in ***Figure 12.29***). This is, as before, the result of trying to use the entity paradigm to describe patterns that do not fit into its constrained structure.

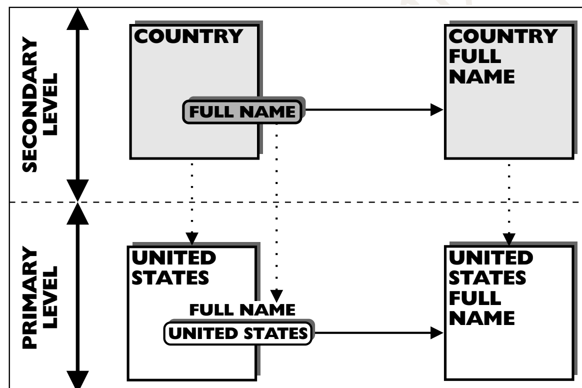


Figure 12.29: Implicit relational attribute type

5.4.2 Country full names class

Following the same pattern that we used at the attribute level, we start by re-engineering the implicit country full name entity sign. In the entity world, this refers to a country full name entity. We now find out what object it is re-engineered into. The classic re-engineering pattern for attribute types (which we looked at in ***Chapter 5***) has the attribute type transformed into the class of the objects that the individual attributes were re-engineered into. This pattern applies here. The country full name attribute type re-engineers into the country full names class. This is a class with the United States full names class and the United Kingdom full names class as members.

These members are classes; so, country full names is a class of classes (shown in ***Figure 12.30***). The entity paradigm cannot handle this pattern. It had to be distorted to fit into the entity framework. It also involves a new way of seeing, one unfamiliar to most people and one that treats the full names classes as objects and collects them into the country full names class—a class of classes.

© Copyright chrispartridge.net / BORO Centre.com

Figure 12.30: Country full names class

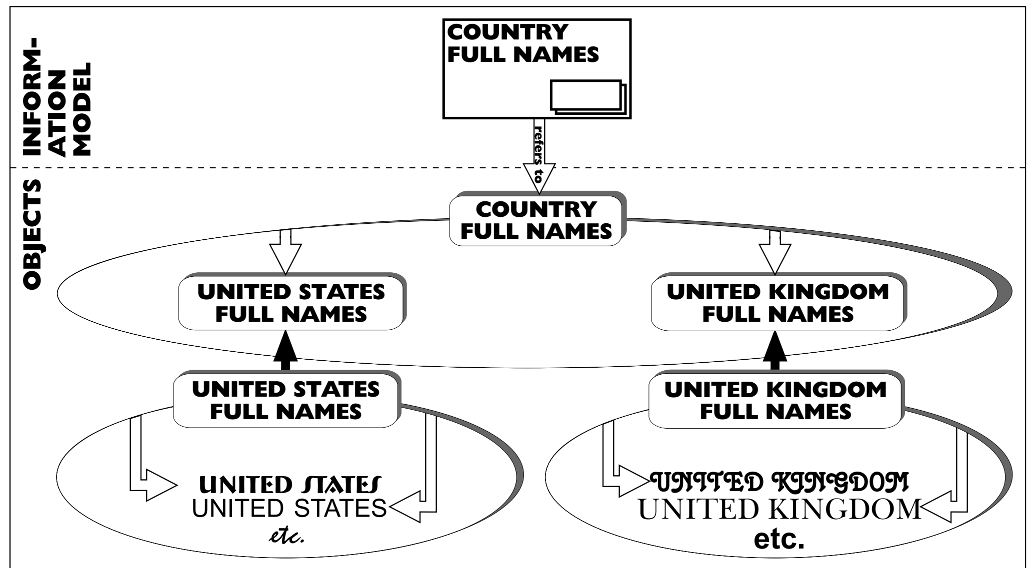
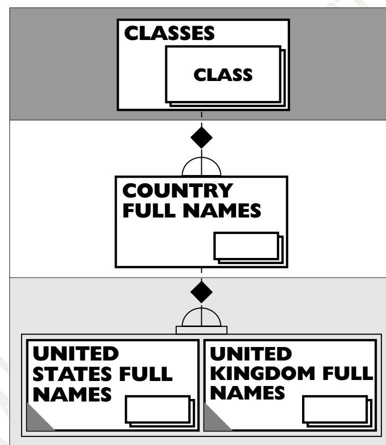


Figure 12.31: Country full names object schema



We increase coherence by recognising a structural pattern—in this case, a class–member tuple to the framework class, classes. We can now carry out the third step and construct the appropriate signs and include them in the object model, giving us the object schema in ***Figure 12.31***. The class–member sign to the partition box is partial because there are other full names for other countries (for instance, Germany full names).

5.4.3 Country full naming tuples classes

Continuing to follow the attribute level pattern, we now re-engineer the relational attribute type sign (shown in ***Figure 12.29***), which points to the country full name entity type sign. This refers to a country's full name relational attribute type that relates the country entity type to the country full name's entity type.

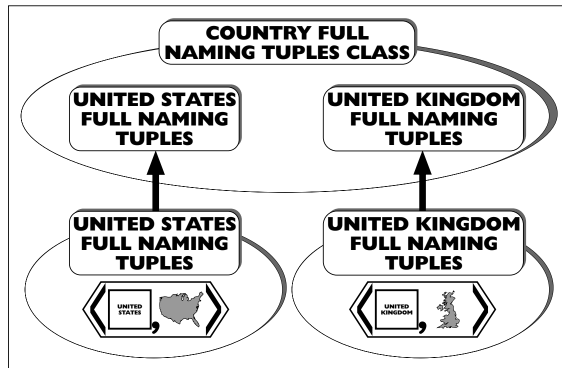


Figure 12.32: Country full naming tuples classes

We apply the classic re-engineering pattern for attribute types from ***Chapter 5*** again and transform the relational attribute type into the class of the objects that the attributes were re-engineered into. This is the class of all the individual country full naming tuples classes, such as the United States full naming tuples (shown in ***Figure 12.32***).

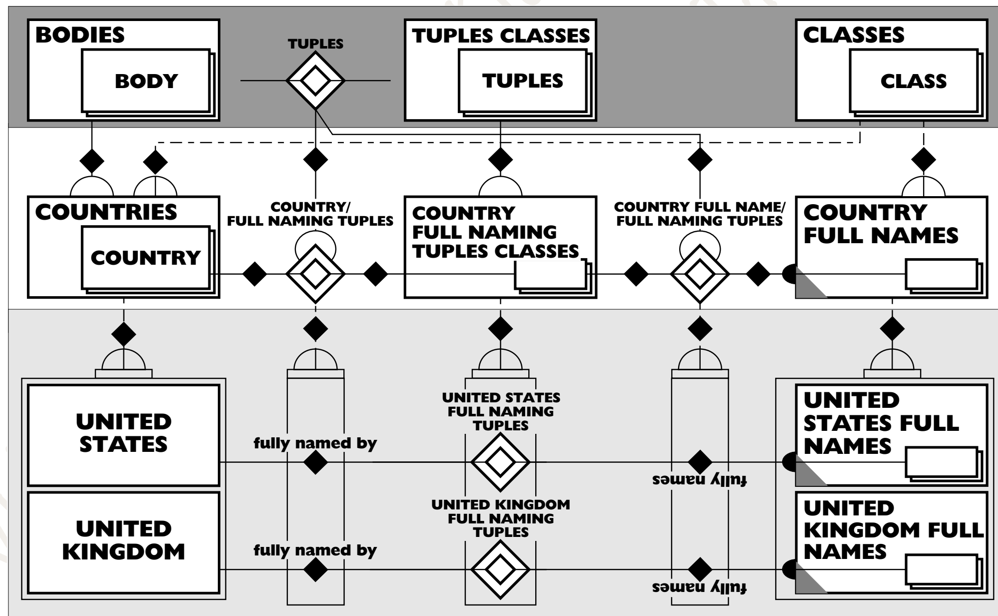


Figure 12.33: Country full naming tuples classes object schema

We identify its structural patterns. It has as members both the United States full naming tuples and the United Kingdom full naming tuples. We construct signs for the objects. This gives us the object schema in ***Figure 12.33***. You may notice that country full naming tuples classes is a class of tuples classes, (in other words, a class of classes) and so not a tuples class (which, by definition, has tuples as its members). This means that country full naming tuples classes is related to both countries and country full names by tuples classes of class places, such as country/full naming tuples. In addition, the country full names class (like its member classes) is classified as derived, defined by country full naming tuples classes.

5.4.4 *Exemplar country full names class*

We then follow the third and final leg of the attribute level pattern and re-engineer exemplar country full names. We re-engineer this into the class constructed by collecting together the individual exemplar full names—the class of exemplar country full names. We work out its structural patterns and construct the appropriate signs in the object model, giving us the object schema in ***Figure 12.34***. This captures the exemplar pattern at the class level.

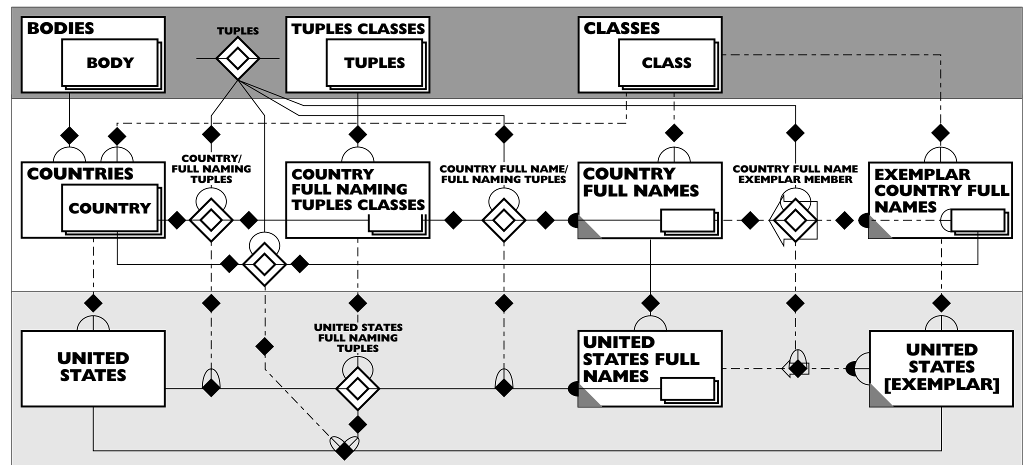


Figure 12.34: Exemplar country full names object schema

5.5 Re-engineering the country code attribute type sign

We have completed the re-engineering of the country full name attribute type. We now start re-engineering the second attribute type, country code. This is a name and uses the same basic naming pattern as country full names and so its re-engineering follows the same pattern. In a real exercise, it would be important to follow the pattern through step by step, no matter how tedious this felt; otherwise, important variations would be missed. However, for this worked example there is no need to do it.

The results of the re-engineering are the object schemas shown in ***Figure 12.35*** for country codes, ***Figure 12.36*** for country coding tuples classes and ***Figure 12.37*** for exemplar country codes. These follow the patterns for country full names shown in ***Figures 12.31***, ***12.33*** and ***12.34***. You will note that in ***Figure 12.36*** the country codes class is classified as derived (it is defined by the country coding tuples classes). Similarly the exemplar country codes, in ***Figure 12.37***, is classified as derived (defined by the country code exemplar member tuples class).

Figure 12.35: Country codes object schema

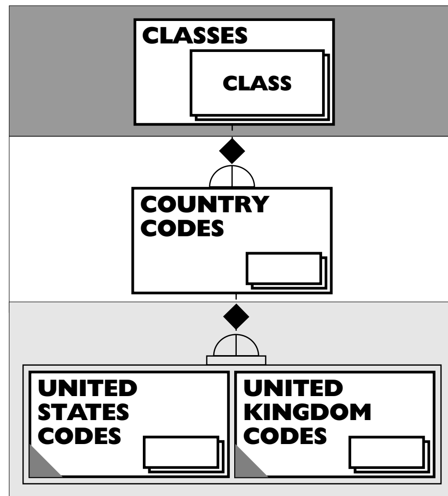
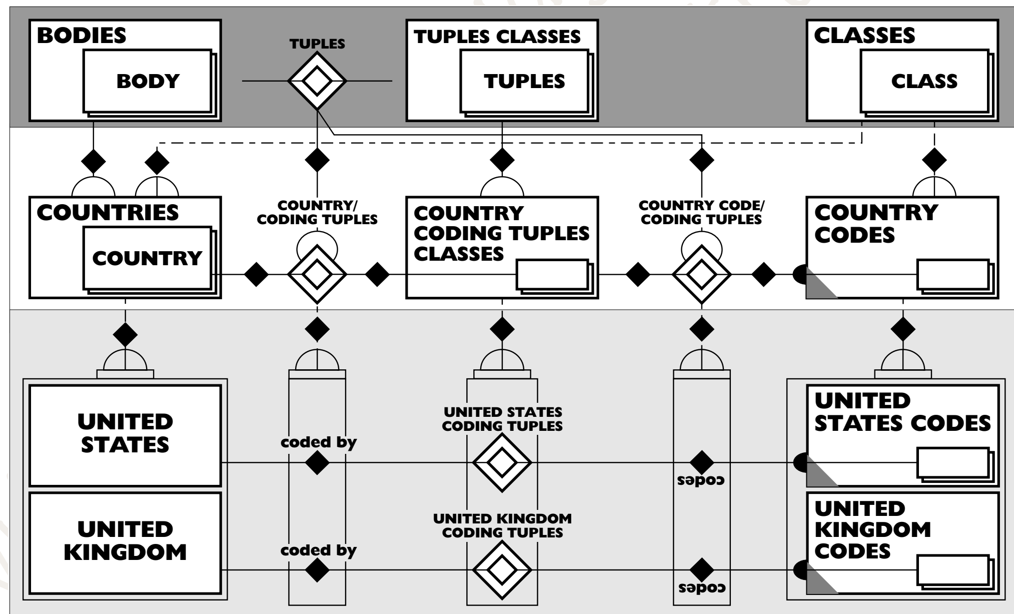


Figure 12.36: Country coding tuples classes object schema



5.5.1 Re-engineering the naming pattern

Re-engineering these name attribute types should have given us some idea of the distortions that the entity paradigm is forced to make to some patterns to fit them into its framework. It should also give some idea of how accurate we need to be to see how to unwind the distortions. Country has taken us from one end of the re-engineering spectrum to the other. Re-engineering the country entity type was simple and straightforward; whereas, re-engineering its name attribute types was far from straightforward. It needed careful analysis to reveal the underlying naming patterns.

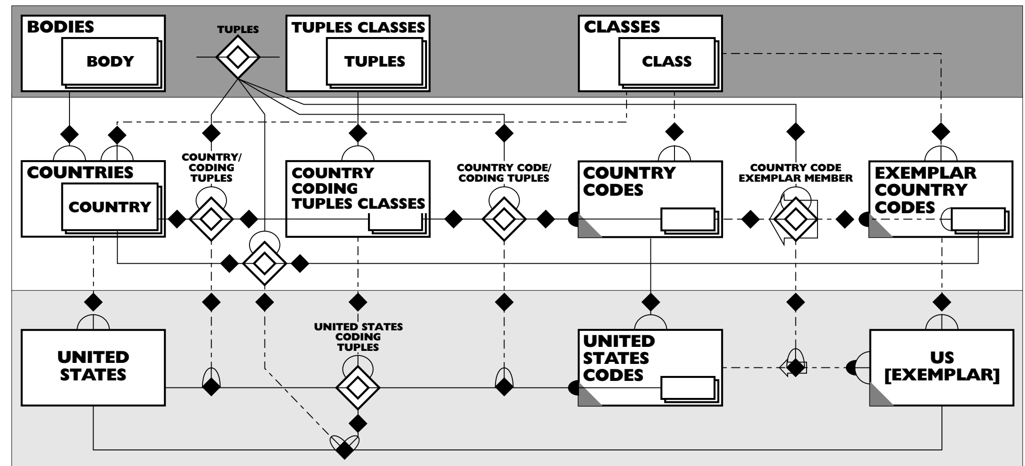


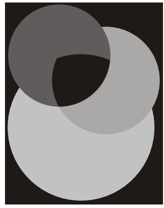
Figure 12.37: Exemplar country codes object schema

6 Basic elements of the re-engineering completed

We have now completed the direct re-engineering of the elements of the country entity format. We have constructed an object model that reflects the re-engineered objects. However, we have not completed the first stage yet. We still need to take advantage of any opportunities for generalising the objects. We do this in the following chapter.

© Copyright Chris Partridge
chris.partridge@BOROCentre

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 13

Generalising Country's Re-Used Patterns

1 Introduction
2 Generalising re-used patterns
3 First stage of the re-engineering completed

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction

In the previous chapter, we re-engineered the elements of the existing system's entity format directly into an object model. In this chapter, we complete the first stage of the re-engineering by generalising the patterns re-used in the modelling.

2 Generalising re-used patterns

In the previous chapter, we focused on the direct re-engineering of the entity format into objects. We now focus on a less direct form of re-engineering—generalisation. This is essential to any good re-engineering because it leads to the compacting that makes the model simpler and more powerful.

In practice, the direct re-engineering and generalisation would go on hand in hand, but I have kept them separate here because it is easier to examine them on their own.

We now work through how we generalise the example. Even this early on in the re-engineering, there are opportunities for generalisation. A good indication of an opportunity for generalisation is patterns of objects that have been re-used in the modelling. In this worked example, there are three obvious candidates:

1. The re-use of the country full names pattern in the re-engineering of country codes.
2. The re-use of the country full naming tuples classes pattern in the re-engineering of country coding tuples classes.
3. The re-use of the exemplar country full names pattern in the re-engineering of exemplar country codes.

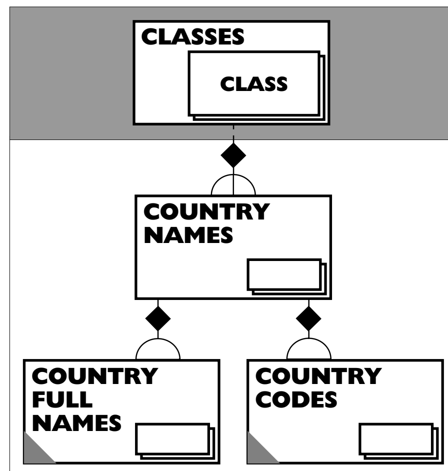
2.1 Generalising country full names and codes

We not only re-used the same pattern in the re-engineering of country full names and country codes; the two objects' connecting patterns are also very similar. This indicates

© Copyright Chris Partridge
chris.partridge@BORO.Centre.com

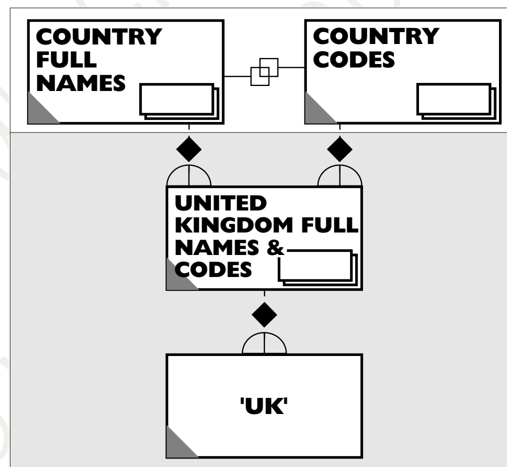
that they are ripe for generalisation. We now generalise them to the country names superclass, illustrated in ***Figure 13.1***.

Figure 13.1: Generalised country names object schema



It is tempting to put the country full names and country codes classes into a partition. But we should not because, in theory, they could overlap. For instance, we might decide to give a country the same full name and code. There is no law that says we cannot. We might decide to make 'UK' both the full name and code of the United Kingdom. Then United Kingdom names would be a member of both the country full names and country codes classes (shown in ***Figure 13.2***). There is a general tendency to be too quick in applying partitions. So it is worthwhile, as I illustrate here, to double-check that the classes are definitely distinct.

Figure 13.2: Identical United Kingdom names and codes object schema



2.2 Generalising country full naming and coding tuples classes

The country coding and full naming tuples classes also shared similar connecting patterns. So we generalise them to a country naming tuples classes super-class. To give a

good view of the generalised pattern, we put this and the country names generalisation onto one object schema, as shown in ***Figure 13.3***.

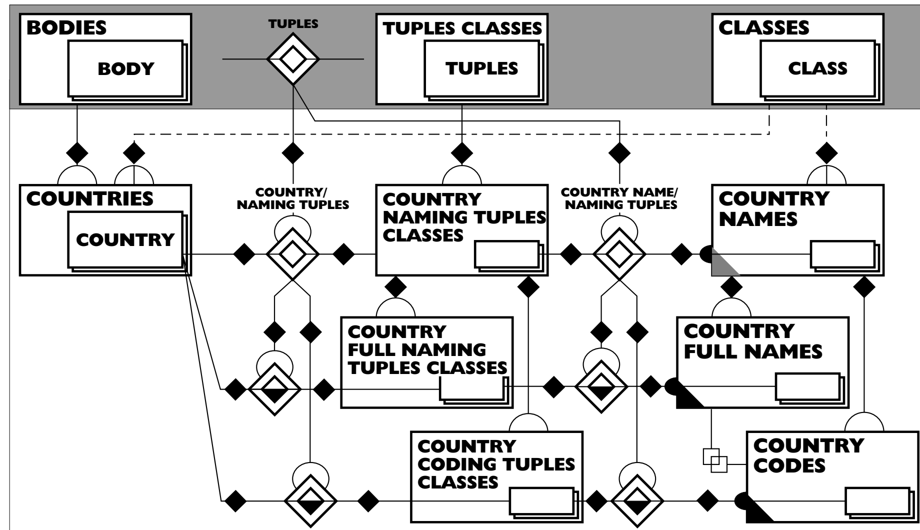


Figure 13.3: Integrated generalised country names/naming tuples classes object schema

Note that this generalisation has left the country full names and codes with no dependent connections. Therefore, we have made them redundant, as shown by the black triangular redundant icons. Furthermore, we have also classified the new country names as derived—defined by country naming tuples classes.

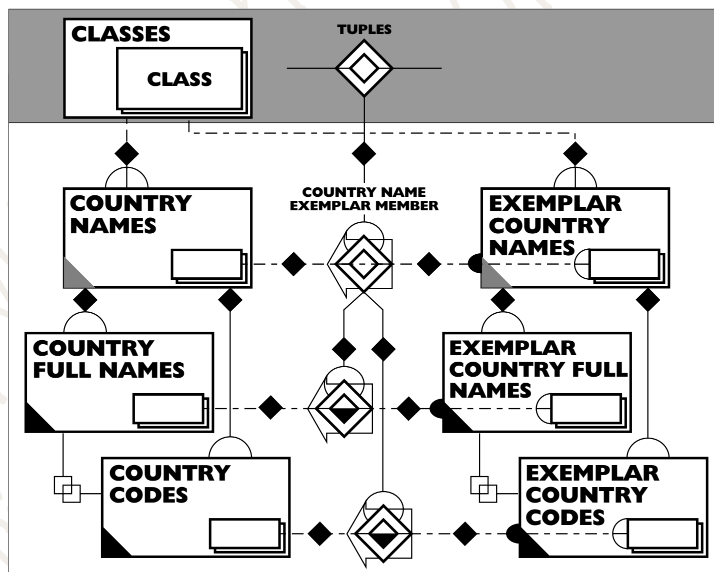


Figure 13.4: Generalised exemplar country names object schema

2.3 Generalising exemplar country full names and codes

The exemplar country full names and codes are in a similar situation. We generalise them to an exemplar country names super-class and get the object schema in ***Figure***

13.4. We sign the exemplar country full names and codes as overlapping for the same reason as the country full names and codes. They are also derived and now have no dependent connections. We therefore classify them and their connecting tuples as redundant, changing their grey derived icon to black. Furthermore, the new generalised exemplar country names is classified as derived.

2.4 First stage application level object model

Having done this generalisation, we can go one step further and compact the model. If we purge the model of redundant objects, then we can fit the whole application level model into the single object schema shown in ***Figure 13.5***. Sometimes people now see the possibility of generalising further to a general names class. There is such an object, but we leave it until a later example, where we have the patterns from two naming tuples classes to generalise from.

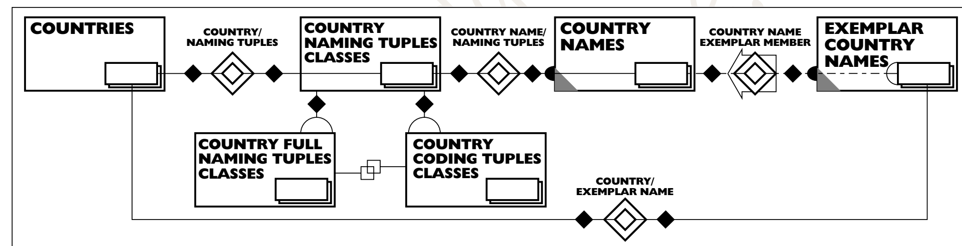


Figure 13.5: First stage application level object model

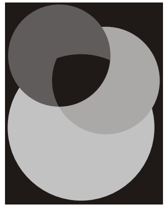
This shows how generalisation, along with re-use, are important factors in the compacting that leads to simpler, more powerful, object models. Without generalisation the more accurate object models would be bigger and so unwieldy.

3 First stage of the re-engineering completed

We have now completed the first stage of the REV-ENG re-engineering process. We have done quite a lot. From a re-engineering perspective, we have transformed the implicit spatial and naming patterns in the country entity format into an explicit object model. From a learning perspective, we have made ourselves reasonably familiar with the first stage of the approach. We now know how to deal with the application of the approach to both straightforward and difficult entity formats.

In the following chapter, we carry out the second and final stage of the re-engineering of this country example. The focus shifts from the signs in the existing system to the conceptual patterns for country in our heads. We look at how we re-engineer these into the object model.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 14

Re-Engineering Our Conceptual Patterns for Country

1 Introduction
2 Finding conceptual patterns
3 Character strings patterns
4 Nested countries pattern
5 More accurate nested countries patterns
6 Current countries
7 Summary

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction

This is the third of the three chapters that take us through the systematic process of re-engineering country patterns. This process has two stages (illustrated in ***Figure 14.1***). In the previous two chapters, we worked through the first stage of the process, re-engineering the country entity format into a business object model. In this chapter, we work our way through the second and final stage of the process, re-engineering our conceptual patterns for country into the object model.

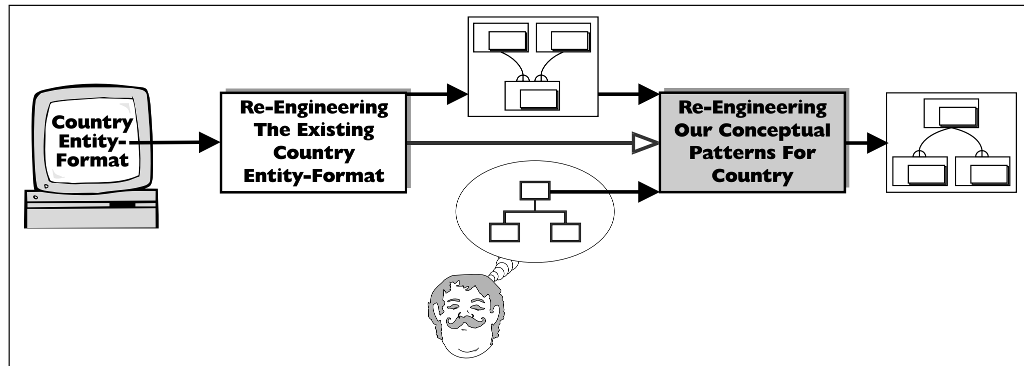


Figure 14.1: Second stage of the systematic re-engineering process

The entity paradigm (and so entity oriented computer systems) is too constrictive to hold many of the patterns for country. And it captures a distorted version of many of those that it does hold. The object paradigm does not have these restrictions; so, in the first stage, we re-engineer undistorted versions of the patterns into the object model.

In the second stage, we re-engineer the patterns in our conceptual systems (in other words, our brains). These are not as restricted as entity oriented computer systems and so hold a much richer store of patterns. This provides more of an opportunity for the power and flexibility of the object paradigm to come into play and so the construction of more accurate, simpler and functionally richer object models.

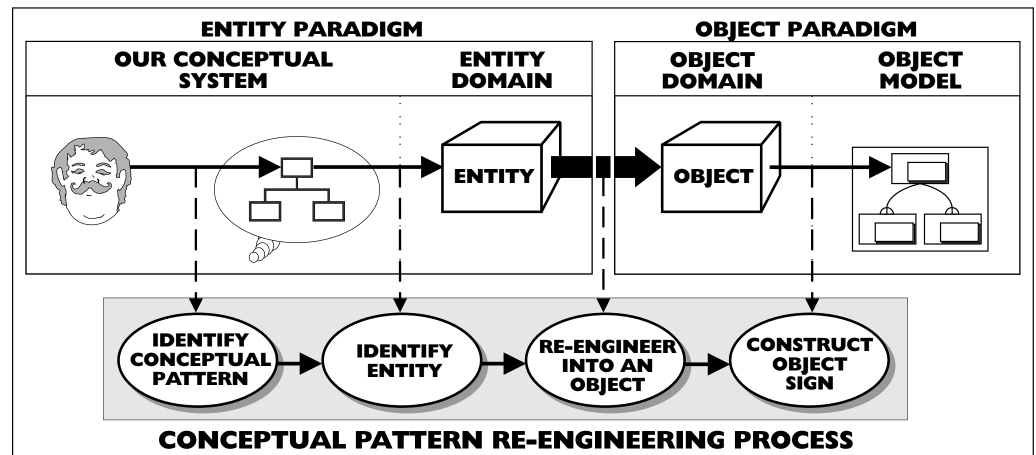
1.1 The three re-engineering steps

As outlined in ***Chapter*** , the re-engineering in the second stage follows similar steps to the first stage, with a different initial step. We do the following:

- Find a relevant conceptual country pattern,
- Identify the entity it refers to,
- Re-engineer it into an object pattern, and then
- Construct signs for the pattern in the model.

These steps are shown graphically in ***Figure 14.2***.

Figure 14.2: The four re-engineering steps in the second stage



Conceptual patterns, by their nature, are not very accessible. We cannot bring them up on a screen or print them out (as we can for entity formats of a computer system). So an important first step is finding relevant conceptual patterns. In this chapter, we look at some useful techniques to stimulate our thinking and tease them out.

Once we have found the conceptual patterns, we make them as formal as the entity formats and then re-engineer them into objects. Most people's conceptual patterns are only loosely based on substance semantics. This leads to a tendency to 'refer to' things informally; or, in other words, it is not always clear what the conceptual patterns refer to. So, even though our brains effortlessly use these patterns, trying to make them formal requires some thought. And even then we cannot be sure that what we have formalised is either complete or correct.

We illustrate this process of re-engineering our conceptual patterns with these four examples:

- Character strings patterns,
- Nested countries patterns,
- More accurate nested countries patterns, and
- Current countries.

2 Finding conceptual patterns

The first step in this second stage is finding the relevant conceptual patterns. This gives it a different flavour. The first stage is more analytic; whereas, the second stage is more investigative. For example, the entity formats of the existing system are reasonably formal and public—certainly when compared with our conceptual patterns. The first stage is largely a matter of analysing what the entity format refers to. By contrast, the second stage involves an open-ended search for relevant conceptual patterns.

© Copyright Chris Partridge
chris.partridge@BORO.Centre.com

2.1 Useful, relevant conceptual patterns

Before we look at the techniques for investigation, we need to clarify which patterns count as useful and relevant. In a new paradigm, the old ways of working are often turned on their head. What were sensible and solid ways of working become counter-productive. What were impractical ways of working become pragmatic and realistic. This means that when we shift to the new paradigm we have to unlearn the old ways of working and relearn new ones. This is true for the way in which we investigate our conceptual patterns.

Traditionally, system builders apply an informal cost–benefit analysis to what they include within the scope of their system and so what they are going to model. When they come across a complex area, they ask themselves whether the additional costs of automating it are justified by increased benefits. Typically, they apply a kind of parato rule. In most cases, automating 20 percent of the areas in the business delivers 80 percent of the benefits. More often than not, the 20 percent contains the simple everyday processes and the 80 percent contains the complex anomalous exceptions. In this situation, most system builders aim to maximise value for money. Their analysis shows that the cost of automating the complex areas is too high, given the benefits. So they focus on the 20 percent of the areas with a high payback and leave the complex areas to be handled manually.

Behind this traditional way of thinking lies the following two assumptions:

- A complex area is implemented with complex code, and
- The benefits of analysing it only arise in the implementation of that complex code.

In the entity oriented environment of traditional system building, this 'operational' way of thinking is sensible. However, in an object-oriented environment, we take an 'understanding' viewpoint and these two assumptions no longer hold. A complex area no longer has to be implemented in complex code. An area only looks complex because we have a complex conceptual pattern for it. Business object modelling can transform that complex conceptual pattern into a much simpler and easier to understand object pattern. The cost of implementing this new simple pattern is no more than for other simple areas.

Furthermore, in a re-engineering, the benefits of analysing a complex conceptual pattern are often no longer limited to the area in which it was found, but spread across the whole system. A complex pattern in an entity paradigm tends to be rich in implicit, re-usable patterns—that is what makes it complex. So when we re-engineer and generalise the pattern, it ends up not only simpler, but re-usable. It can be applied not only to entity formats in the existing system that are not part of the original complex area, but also to new patterns that are not in the existing model. This means that if we want to assess the benefits of analysing a pattern, we need to think in terms of the improvements its re-engineering will make to the overall model.

General re-usable patterns are what give an object model power. The re-vamped cost–benefit analysis now suggests that, when business object modelling, we should seek out the interesting complex areas that will give us these general re-usable patterns. This

© Copyright Chris Partridge chris.partridge@BORO Centre.com

represents a change in attitude to complex anomalous exception cases. Instead of excluding them from the analysis, we now welcome them as a source of useful patterns.

Another reason for welcoming these complex conceptual patterns is that they turn out to be difficult to find. In this chapter, we look at examples of techniques for finding them as well as how to re-engineer them into the model. In so doing, we grow our object model for spatial patterns.

3 Character strings patterns

We find our first missing conceptual pattern, character strings, using a simple technique and by conceptually reviewing the object model. This involves taking a careful look at the model and mentally comparing it with the conceptual patterns in our heads.

3.1 Conceptually reviewing the object model

Most people seem to find it quite easy to look at an object schema and naturally compare it with their conceptual patterns. This often starts happening during the first stage of the re-engineering. A modeller working on a schema will just look at it and see something wrong—the pattern of object signs do not fit together. This will be easy to see:

- First, because the object patters are described using visual patterns, something the human brain is very good at dealing with.
- Second, because the patterns in the object schemas are explicit and public.

When we review the object schemas, we might find that it does not have one of our conceptual patterns. Or, we might find that the model has the pattern, but it is different in important respects from our conceptual pattern.

When we analyse these, they usually provide us with a fruitful source of re-usable patterns. The differences may highlight inaccurate patterns in the model. This should not be surprising because, at this stage, the object model is a only a transformation of the existing system, and so still contains some of its faults. We can revise these inaccurate patterns, making them more re-usable. The missing conceptual patterns can often usefully be re-engineered, added to the object model, and generalised.

It is important to recognise that the review is a two way process. While we are enhancing the object model, it is also enhancing the way we see the business. This is particularly obvious when we review schemas produced by other people. At first glance, their find patterns may look wrong. (This must have already happened to some people with the schemas in earlier chapters.) Sometimes our intuition is trustworthy and the model's pattern need changing. Other times it is our intuition that is wrong and its patterns need changing. The object schema then 'teaches' us a new, more accurate, way of looking at the business. (This 'teaching' aspect of schemas can be particularly useful when the re-engineered business paradigm needs to be explained to people outside the re-engineering team.) By the end of the review process, our conceptual patterns and the object model's patterns should have converged. We and the object model should share a common way of seeing the business.

© Copyright Chris Partridge chris.partridge@BORO Centre.com

3.2 Finding the missing character strings concept

We now work through an example of reviewing an object schema, looking for a missing concept. Look at ***Figure 14.3***; it shows the object schema we re-engineered for United States full names (it is a copy of ***Figure 12.19***). When we worked through the re-engineering, I referred to United States full names as a 'class of character strings'. As I wrote the phrase 'character strings', it must be a concept in my conceptual system. As you read it, it must also be in your conceptual system. But the object 'character strings' does not appear anywhere in the object model. It is an example of a missing concept.

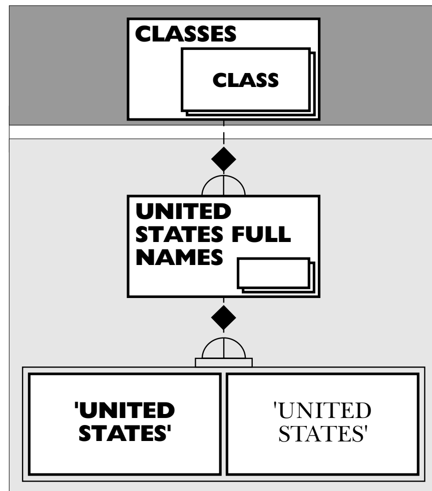


Figure 14.3: United States full names object schema

In this instance, the missing concept refers to a more general class object than the United States full names class. It is quite common for people to find that the first stage's object schemas are less general than their conceptual patterns. The question here is whether it is worthwhile including the more general concept of character strings in the model. In this case, not only is character strings a really re-usable general class, but it also captures an important part of our understanding of what a name is.

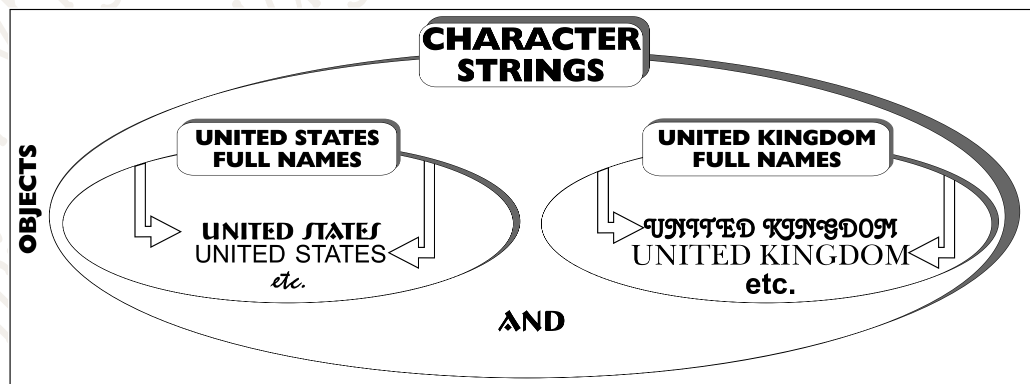


Figure 14.4: The character strings class

© Copyright chris.partridge@BOROCentre.com

3.3 Re-engineering the concept into an object

We now move onto the second step of the process—re-engineering the character strings conceptual pattern into an object pattern. This is relatively straightforward. Character strings is clearly a class and a super-class of United States full names. It, therefore, has all the members of the United States full names class as members. In fact, it has all character strings as members (illustrated in ***Figure 14.4***).

As well as re-engineering the missing concept, we need to find the re-engineered object's connecting patterns with other objects in the model. We want to maximise coherence and sense; so, we try and find all the relevant connecting patterns. This involves quite a few classes: United States full names object to start with, but also country names and exemplar country names as well as a new character string classes class.

3.4 Character strings and United States full names

We look for character strings' connecting patterns with the United States full names class first. The character strings class is, as we have already mentioned, a super-class of the United States full names class so we explicitly make the connection. Character strings is also a class, so it is a member of the framework class, classes. We make this connection as well. This gives us the schema in ***Figure 14.5***.

Character strings is the class of all character strings; so, it includes character strings other than United States full names, even those that are not country names or names at all. We can show this in the model by picking a character string at random, 'AND' say, and recognising it as a member of the character strings class. The object schema for this looks like ***Figure 14.5***.

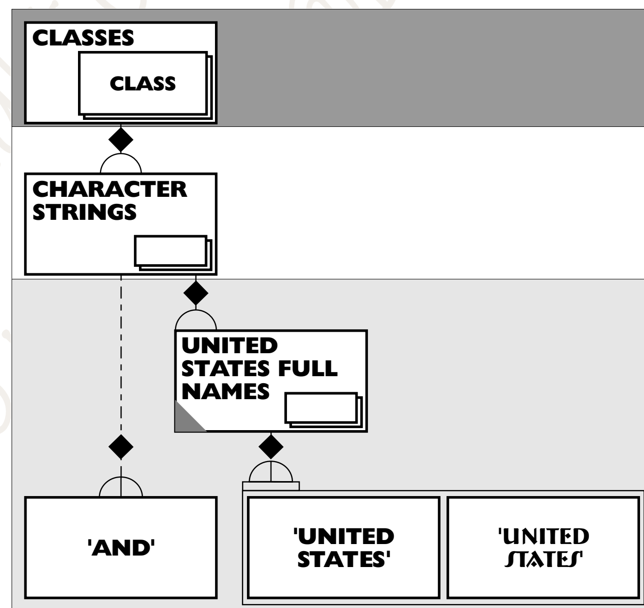


Figure 14.5: Character strings and United States full names

3.5 Character strings and country names class

Now we look for the connecting patterns between character strings and country names. There is a direct connection and a number of indirect connections. The direct connection is a super-sub-tuple-class. Members of the country names class, such as United States full names, are sub-classes of the character strings class. This direct connection is the super–sub-class of tuples shown in ***Figure 14.6***. The indirect connections are the tuple members of this class. Country names has as descendant members, classes such as United States full names and United Kingdom full names. These are, in turn, sub-classes of the character strings class. These examples of indirect connections are shown in ***Figure 14.6***.

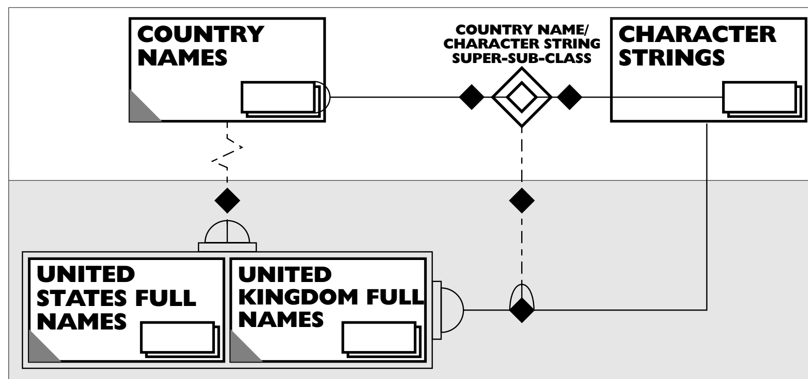


Figure 14.6: Character strings and country names

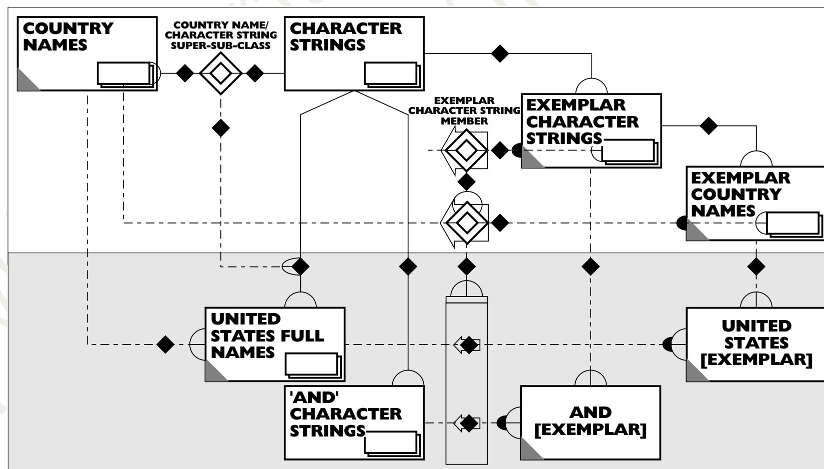


Figure 14.7: Character strings and exemplar character strings

3.6 Character strings and exemplar character strings

The character strings class suggests a generalisation of exemplar country names that provides us with another connecting pattern. Exemplar country names are character strings; for example, 'United States [EXEMPLAR]' full name is clearly a character string. So, if United States full names can be generalised to character strings, why can't exem-

plar country names also be generalised to exemplar character strings, and then up to character strings?

The answer is, it can, which we do now. We construct an exemplar character strings class that is a super-class of exemplar country names and a sub-class of character strings. The ‘United States [EXEMPLAR]’ full name and the ‘AND [EXEMPLAR]’ character string are both members of this class, though ‘United States [EXEMPLAR]’ full name is only a distant member.

These connecting patterns are shown in ***Figure 14.7***.

3.7 Character strings and character string classes

There is another connecting pattern between character strings and country names, one that involves a new class object—character string classes. This is an example of a useful type of class, the power class. A power class is the class of all sub-classes of a class. These classes are a common feature in mathematical set theory, where they are known as power sets and defined as the set of all sub-sets of a set.

The power class of the character strings class is the class of all sub-classes of the character strings class; or, in other words, the class of all classes of character strings. The connections between a class and its power class are strong. By definition, the members of the power class are sub-classes of the ‘powered’ class. From this, we can deduce that powered classes’ members are members of the power classes’ members. Signs for these connecting patterns are constructed for our example, giving us the object schema in ***Figure 14.8***. This same pattern is repeated for all power classes. You will have noticed that the character string classes class is derived from its power class tuples class. This is signed as a power class by the addition of a new power class component icon to the tuples class icon.

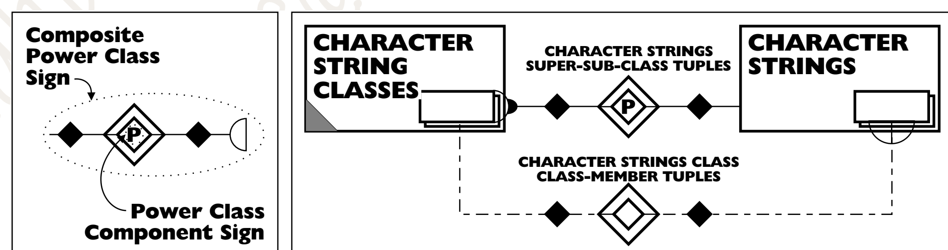


Figure 14.8: Character string classes object schema

Figure 14.8 shows a direct connecting pattern between this new character string classes and character strings. There is also an indirect connecting pattern by way of the country names class. The country names class has classes of character strings (for example, United States full names) as members. This makes it a class of classes of character strings. It, obviously, does not contain all the potential classes of character strings. This makes it a sub-class of the character string classes class. This gives us the indirect connecting pattern between character strings and character string classes shown in ***Figure 14.9***.

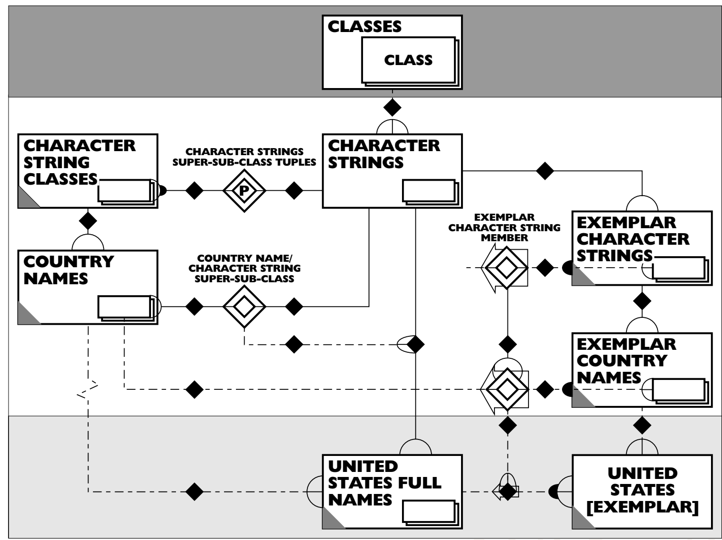


Figure 14.9: Character strings and character string classes

You should now begin to see how this process of generalising and tying together the objects with connecting patterns is starting to make the model webby. As the model grows and develops, it will become webbier and webbier.

4 Nested countries pattern

We now look at the next example of a missing conceptual pattern-nested countries. First, we look at how we might find that it is missing.

4.1 Looking at conceptual patterns in other 'data'

Reviewing the model's schemas is not the only way of finding missing conceptual patterns. We can also look at the patterns in 'raw data'. Lists are particularly useful sources of 'raw data' because they can be compared with the model methodically. Once you start looking, lists are reasonably easy to find. One useful source is international organisations, such as the International Standards Organisation (ISO). We could this comprehensive list of countries in this example.

Organisations often capture an aspect of their conceptual patterns in written descriptions. Different organisations with different purposes tend to describe different aspects of the conceptual patterns. So when we compare their views with the object model (which reflects the view of the re-engineered system), we usually throw up a relevant missing or inaccurate pattern. This is then re-engineered and included in the model.

4.1.1 Automating the checking

For some classes, the lists of data are too large for a visual line by line comparison with the model or each other. In these situations I have found it useful to load the lists onto a

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

computer (if they are not already there) and let the computer do the comparison. It can automatically and accurately spot any differences or missing items. A particularly useful way of doing this is to build a validation system based on the object model (described in a little more detail in ***Chapter 18***). Loading the data onto this system soon reveals any differences between the patterns of the data and the model.

I usually then use a technique that takes advantage of the human brain's ability to recognise spatial patterns. I generate a number of different versions of the lists sorted by various criteria. Patterns then begin to emerge from the page. Sometimes these are interesting. They may be genuinely new patterns or old ones we are so familiar with that we did not make them explicit and so did not include them in the model.

4.2 Missing nested countries pattern

A conceptual pattern is missing from our country model—nested countries. We can find it by comparing lists of countries with the countries class in the model. First, we note any differences and omissions and then try to determine whether they are relevant. This would, at some stage, identify some missing nested countries, such as those shaded in ***Table 14.1***.

Country Names	Country Codes
England	EG
Northern Ireland	NI
Scotland	SC
United Kingdom	UK
Wales	WL

Table 14.1: Selected partial country listing

England, Scotland, Northern Ireland and Wales are all countries. They are all nested countries because they are part of the United Kingdom. However, many computer systems, including the one that we have just re-engineered, do not recognise them as either nesting or countries. They assume that countries are mutually exclusive because they are used as a basis for summarising figures, such as sales or exposures.

We can see this by assuming that one of these systems contains records of all the countries listed in ***Table 14.1***. As it allocates each item to only one country, it is faced with a dilemma when summarising sales and exposures. If it allocates an English sale to the England figures, it would not appear in the United Kingdom summary figures. If, on the other hand, it allocates the sale to the United Kingdom figures, it would not appear in the England summary figures. In either case, one set of summarised figures would be wrong. There is no way out until the system can recognise that one country is part of another. Then the system would be able to allocate a sale to England and include England's totals in the United Kingdom's summary figures.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

We now re-engineer the conceptual pattern for nesting. It turns out to be an example of a familiar pattern—the whole–part pattern. The nesting pattern is re-engineered as couples of the whole and part countries belonging to the nested countries whole–part tuples class (illustrated in ***Figure 14.10***). The corresponding object schema is in ***Figure 14.11***. Notice that this shows the nested countries whole–part tuples class is redundant. It is a sub-class of the framework class, whole–part tuples and defined by the countries class.

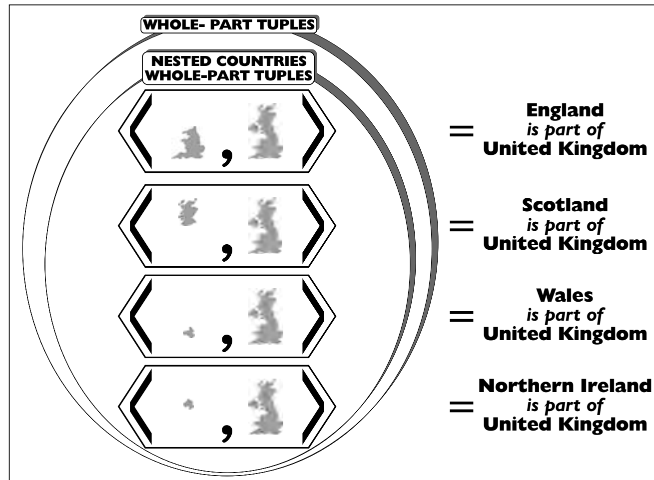


Figure 14.10: Nested countries whole–part tuples

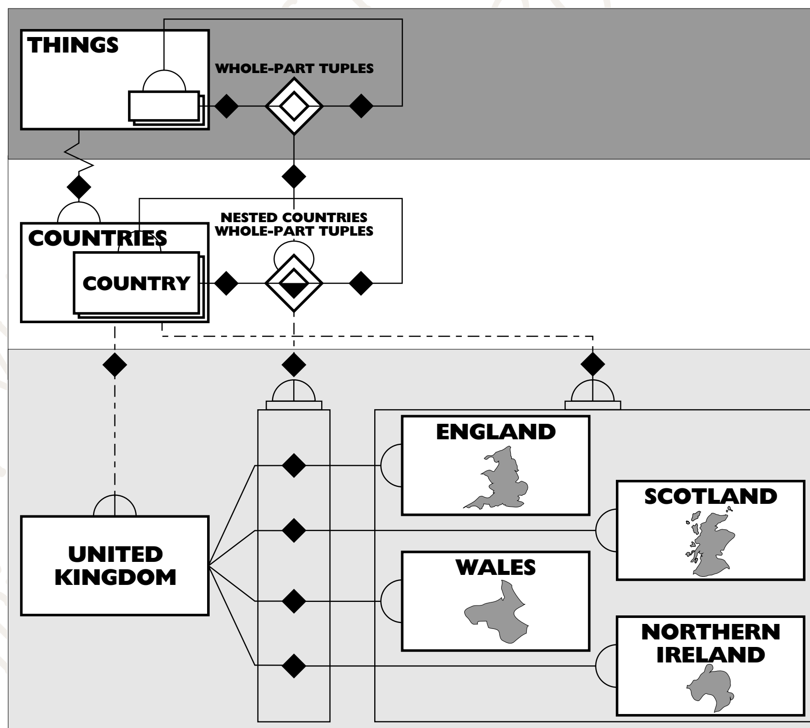


Figure 14.11: Nested countries whole–part tuples object schema

4.3 Direct benefits

Conceptual patterns, such as the countries nesting pattern, that have been omitted from entity systems in an attempt to keep them simple are often not only fertile sources of re-use but also bearers of direct benefits. This is the case here. One direct benefit was discussed earlier: once the pattern is implemented in the system, figures can be accurately summarised for nested countries. There are also other direct benefits. We look at two examples now:

- Bank of England reporting, and
- Bank holidays.

4.3.1 Bank of England reporting

Different institutions have different objectives and so look at countries in different ways. We expect them to have different views of countries. However, we often need a system with an accurate enough country pattern to handle these different views. Here is one example where the nested country pattern was needed.

Many years ago I worked on an international banking system where the list of ISO countries was used as a basis for country reporting to Head Office. When we started building a Bank of England (BOE) regulatory reporting module, we discovered a problem. While ISO regarded the UAE (United Arab Emirates) as a country, the Bank of England needed a more detailed analysis. It wanted its reporting for the UAE broken down into Abu Dhabi and Dubai (UAE's two main emirates)—without a summary at UAE level. The system only had a flat, simple, mutually exclusive, pattern for countries and so could not accommodate both views. In the end, despite all the attendant problems, we decided the easiest way out was to work around the problem and set up a separate BOE countries file. This meant countries had to be set up and maintained twice—once on the main countries file and again on the BOE countries file. If the system had had an accurate enough pattern, one that could handle nested countries, the problem would never have arisen.

4.3.2 Bank holidays

We sometimes only realise we have a problem when a way of solving it becomes available. The same sort of thing applies here. Once we are aware of what the pattern should be, we begin to see how not having it leads to small but irritating glitches in the design of systems. Consider this example.

Without the pattern for nesting countries, accurate recording of bank holidays (which we re-engineer in ***Chapter 17***) is not possible. For example, the United Kingdom, England, Scotland, Wales and Northern Ireland all have their own bank holidays. Recording all of these without the pattern for nested countries can lead to a similar problem to the sales figures summaries discussed earlier.

For example, a number of international banking systems (like the system in the example above) only have a flat, mutually exclusive pattern for countries. Most users set up the United Kingdom (and not England) as a country and record both United Kingdom and

© Copyright Chris Partridge
chris.partridge@brilliance.com

English bank holidays as United Kingdom bank holidays. Because most these of the banks have UK counterparties who operate from the City of London, they have no problems with this setup.

However, the system is less satisfactory for those few international operations based in Edinburgh, Scotland. They have counterparties in the City of London and so have to contend with both Scottish and English bank holidays. However they set up these holidays, they have problems. If they set up them up as bank holidays in the United Kingdom, then on English (non-Scottish) bank holidays the system would tell them that they were closed when they were open. On Scottish (non-English) bank holidays, it would tell them that their London-based counterparties were closed when they were open. If the system had had the nested countries pattern, this problem would probably never have arisen.

4.4 Potential for re-use

For us, a far more important consideration than these direct benefits is the opportunity for generalisation and re-use that the nested country pattern provides. A similar pattern occurs with nested regions, and both of these patterns seem to be merely parts of a larger nested area pattern (we re-engineer this more general pattern in ***Chapter 15***). This means that if we get the nesting pattern right here (or as right as we can), then, in the later examples, we can re-use it with little or no effort.

5 More accurate nested countries patterns

In the nested countries example, we re-engineered the pattern we found in the list directly into the model. It turns out that this pattern is not sufficiently accurate for our object model. We need spatial patterns that can be re-used, not just within this object model, but across a range of object models. For this to happen, the nested countries pattern needs to be more accurate. We now identify and re-engineer the nested country conceptual patterns that will give us a sufficiently accurate model.

5.1 Scottish Act of Union

One of the things that normally strikes people at some stage in the re-engineering of country is that the nested countries (or country whole–part) tuples class does not capture the nesting pattern accurately enough. It is easiest to spot the inaccuracy in a particular example—for instance, Scotland's Act of Union. In Britain, when talk about devolution for Scotland surfaces (which it does regularly), we are reminded that, before the 1707 Act of Union, Scotland was not part of Great Britain.

This is a good example of the problem with the nested countries tuples. At one time, Scotland was not part of Great Britain and at another time it was. Is Scotland a part of Great Britain or not? With our understanding of objects as four-dimensional objects persisting through time (and, in particular, the knowledge of the semantics of modelling temporal parts gained in ***Chapter 8***), we can see the answer. It has a similar pattern to

© Copyright Chris Partridge chris.partridge@BORO Centre.com

Chapter 's chairman example (illustrated in ***Figure 8.18***). The part-of connection is not between countries at all, but between a temporal part of a country—what we might call a country stage—and a country.

Seeing in terms of country stages, we say that the stage of Scotland before 1707 is not a part of Great Britain and the stage of Scotland after 1707 is a part of Great Britain; the stage is nested in the country. It's perhaps easier to see these objects in the space-time map in ***Figure 14.12***. From this we can construct the schema of re-engineered objects in ***Figure 14.13***.

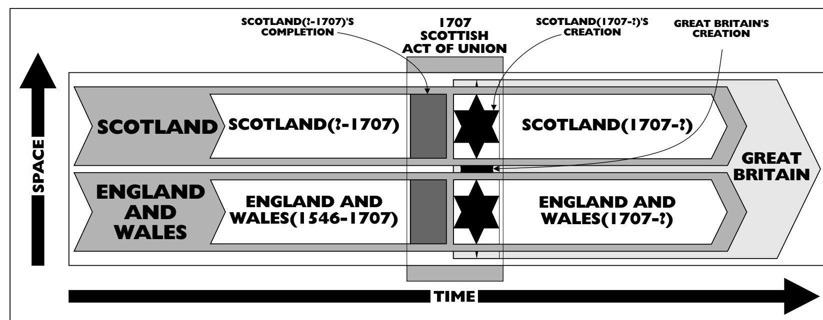


Figure 14.12: Space-time map for 1707 Scottish Act of Union

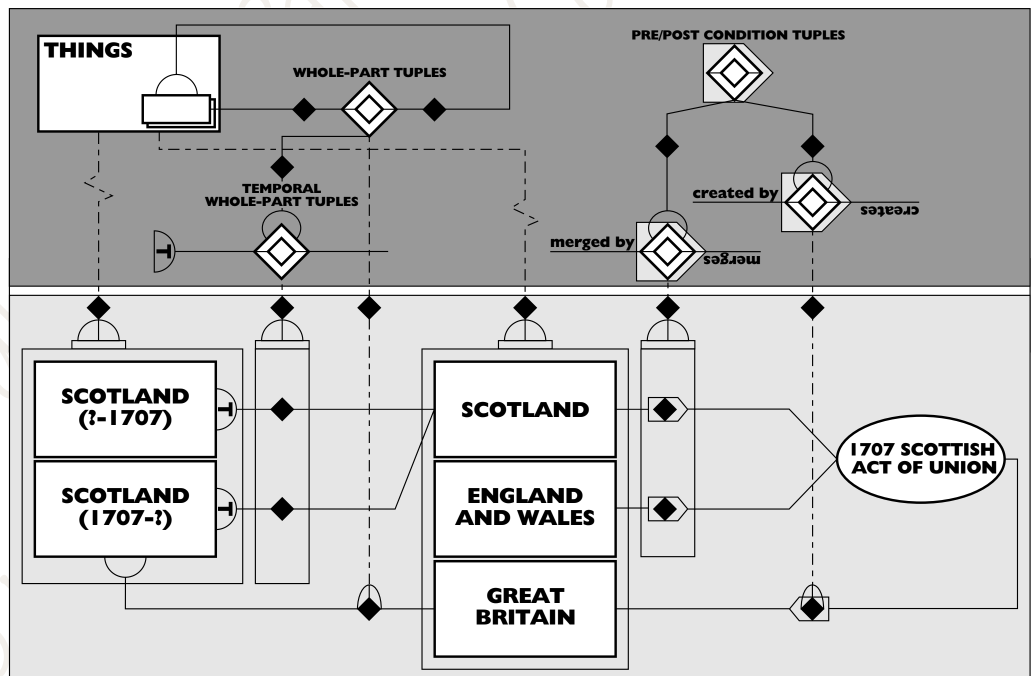


Figure 14.13: 1707 Scottish Act of Union object schema

It is tempting to add more to the schema in ***Figure 14.13***. There is obviously a 'next stage' connection between Scotland's two stages. Furthermore, it seems reasonable to connect these two stages to the 1707 Scottish Act of Union event with a before stage tuple and an after stage tuple. At a more general level, an event such as this needs an agent to initiate it. I have left all these out to keep the example simple.

However, the figures do reveal some cause and effect connections of the kind we looked at in ***Chapter 8***. In Aristotelian terms, both Scotland and England–and–Wales are material causes of the Act of Union event. They exist both before and after the event and contain a part of the event. Great Britain is a final cause. It starts with the Act of Union event and its creation event is part of the Act of Union event.

5.2 Welsh Act of Union

We need to re-engineer another example to check that we have captured the patterns correctly. We use Wales as the example. It went through a similar Act of Union from 1536 to 1543 (for simplicity sake we assume it is 1543). This created the country England–and–Wales, which, as ***Figure 14.12*** shows, was subsequently merged with Scotland to create Great Britain. ***Figure 14.14*** shows a space-time map for the Welsh Act of Union. It has the same pattern as the Scottish Act of Union. From this we can construct the object schema shown in ***Figure 14.15***; this is again similar to Scotland's.

Figure 14.14: Space-time map for 1543 Welsh Act of Union

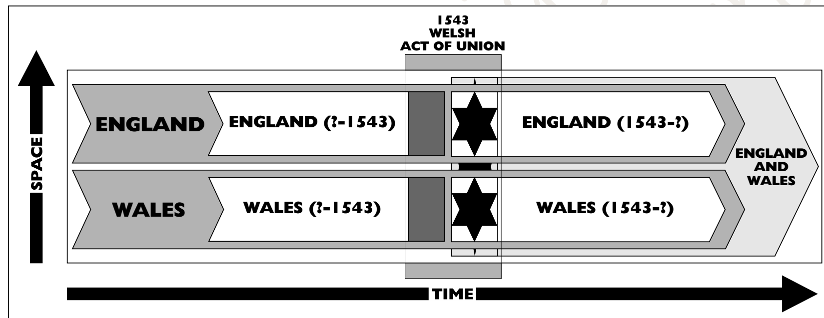
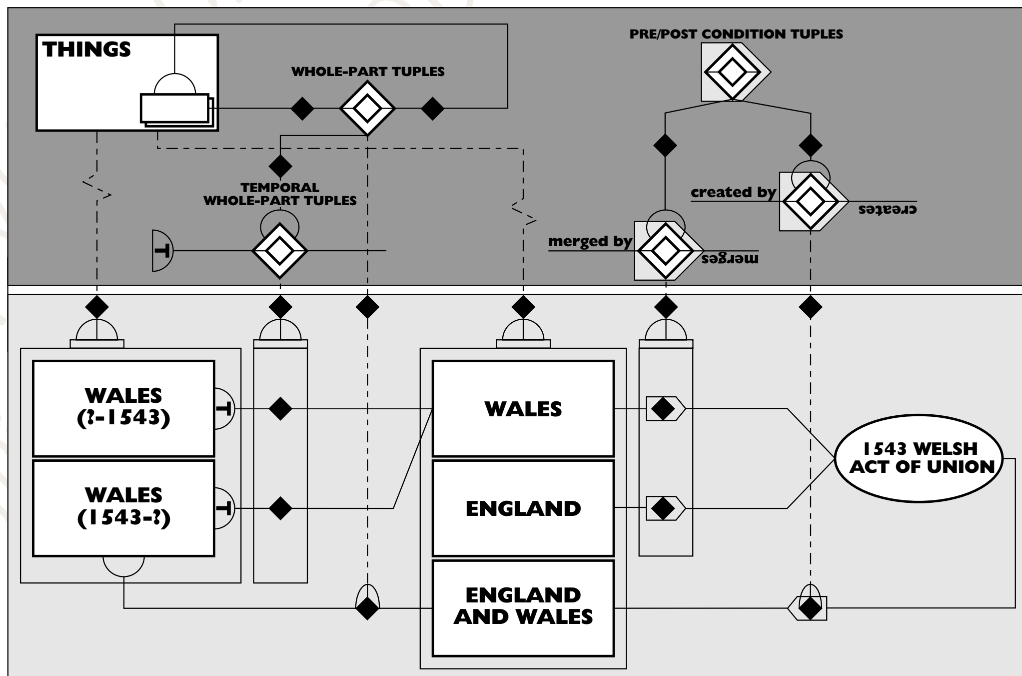


Figure 14.15: 1543 Welsh Act of Union object schema



The possibilities for re-use of this pattern should be becoming clearer. It looks as though it will apply to changes in other types of area, such as region. We shall see in ***Chapter 15*** how we re-use this pattern to model countries joining the European Community (EC).

5.3 Class level Acts of Union pattern

As we are now reasonably sure of the pattern, we can raise it to the class level. The nested country pattern at the class level involves both countries and country stages. So we combine the two into a single countries-and-stages class to act as one of the place classes for the whole–part tuples class, as shown in ***Figure 14.16***. As with ***Figure 14.11***, the sub-class of whole–part tuples (in this case, country/stage whole–part tuples) is redundant, defined by the countries and stages class. When it is purged, the individual whole–part tuples, such as <Scotland (1707–?), Scotland> will become nearest sub-classes of whole–part tuples. You will notice that two of the three individual country/ stage whole–part tuples are classified as temporal–whole–part tuples.

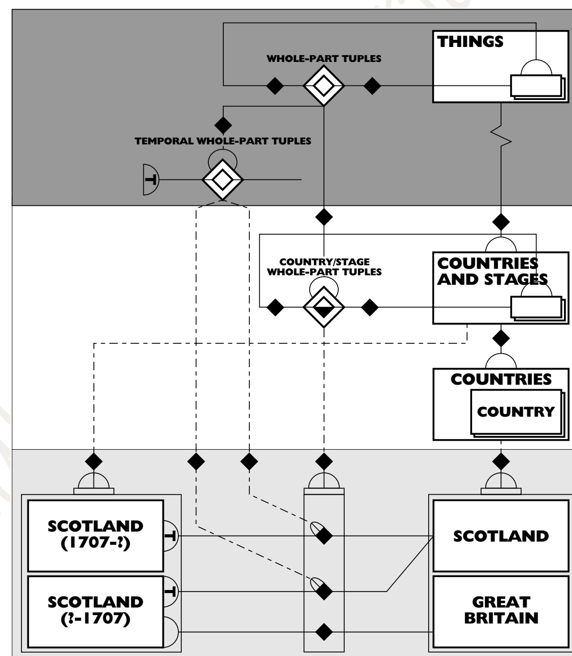


Figure 14.16: Countries and stages object schema

At the class level, the Act of Union pattern is between the Acts of Union's events class and the countries class, as shown in ***Figure 14.17***. In a real re-engineering, we would capture the cardinalities of the tuples to give us a more complete and accurate description of the pattern.

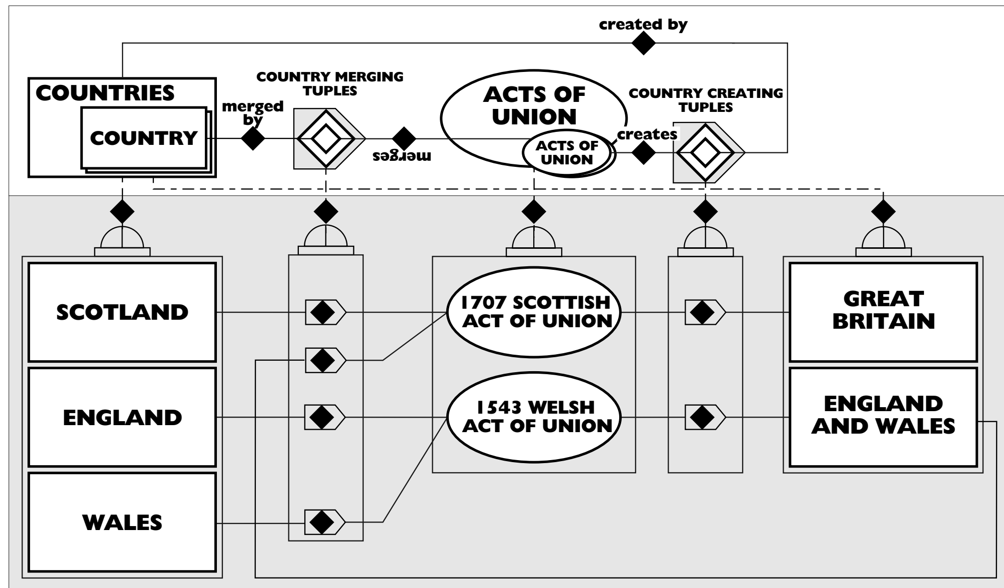


Figure 14.17: Acts of Union object schema

6 Current countries

This look at historic events may have suggested to some people an important pattern missing from the model. It does not yet describe which countries and country stages are current, so it cannot tell us whether a country is *now* part of another country. We re-engineer this pattern.

This involves using the dynamic now and current tuples classes that we examined in ***Chapter 8*** (illustrated in ***Figure 8.28***). The 'current' country or country stage occupies, along with the now class, the places in a current tuple. We capture this pattern using a sub-class of the current tuples class—the country/stage current tuples class—as shown in ***Figure 14.18***. Notice that this class is redundant—defined by the countries and stages class.

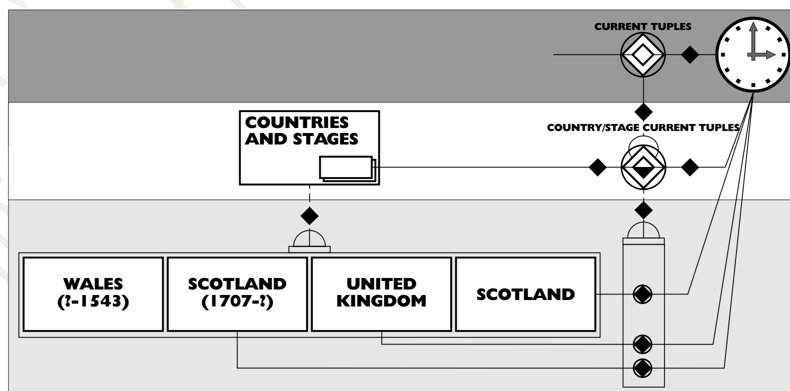


Figure 14.18: Country/stage current tuples object schema

This current pattern, along with the merging and dividing patterns in nested country stages, is a sophisticated way of modelling countries. It is also becoming more relevant in a world where, for example, USSR, Czechoslovakia and Yugoslavia have relatively recently divided into their constituent parts.

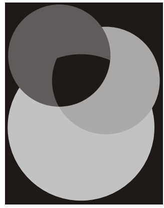
7 Summary

We have now completed all the tasks we are set out to do in this second stage of the re-engineering. The previous two chapters have given you a good feel for how patterns in the existing system are re-engineered into an object model. This chapter has given you a good feel for how the model is enhanced by re-engineering conceptual patterns that are missing from, or constrained in, the existing system. It has also given you a sense for the kind of accurate understanding of the patterns that is needed, and how this can be captured in an object model.

Between them, the three chapters have illustrated the systematic nature of the re-engineering process. It has been worked through in some detail. In the examples in later chapters, we shall forgo this detail to get a better overall view. In the next chapter we move onto the next worked example in this group of spatial patterns—region.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 15

Re-Engineering Region

1 Introduction
2 Re-engineering the region entity formats of the existing system
3 Re-engineering our conceptual patterns for region
4 Summary

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction

In the last three chapters, we re-engineered our first example of a spatial pattern, country. In this chapter, we re-engineer our second example, region (see ***Figure 15.1***). We currently have the beginnings of an object model for spatial patterns. Re-engineering region enhances this. In the next chapter, we re-engineer the third and final example, address, which completes the model.

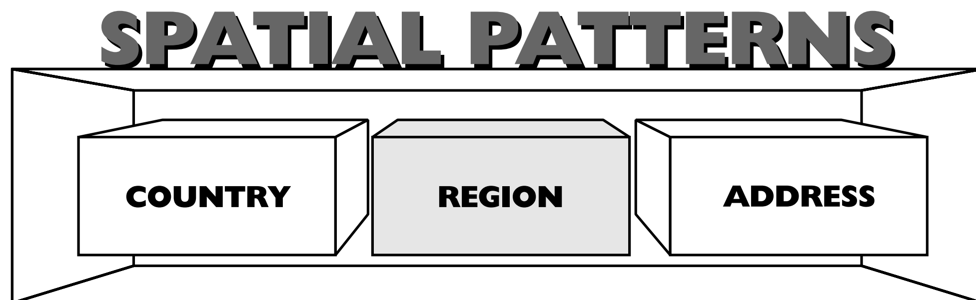


Figure 15.1: Region, second of three examples of spatial patterns

Because we are now familiar with the patterns for the systematic re-engineering process, in this and future examples, we do not work our way through every step. This not only helps us to take an overall view, it means we move quickly and avoid monotony. With region, we follow the systematic two-stage process for re-engineering. We:

- Re-engineer the region entity formats of the existing system, and
- Re-engineer our conceptual patterns for region.

2 Re-engineering the region entity formats of the existing system

Region is, in many respects, similar to country; so, we re-use its pattern of re-engineering. This means that there is potentially an opportunity to generalise across country and region, which we will look into later.

As with the country re-engineering, we follow the standard rules for re-engineering the entity formats:

- Re-engineer the individual entity and entity type signs before their associated individual attribute and attribute type signs.
- Re-engineer a couple of individual entity (attribute) signs and use the patterns to re-engineer their entity-(attribute)-type sign.

We start with the region entity signs, and then move onto their attribute signs: region full name and region code.

© Copyright Chris Partridge chris.partridge@BORO-Centre

2.1 Re-engineering the region entity type sign

Like before, we familiarise ourselves with the entity type by looking at a list of its individual entities, such as the partial listing of regions in ***Table 15.1***. As with country, we are only interested in the two name attributes shown in the listing, so we ignore region's other attributes. We can deduce from ***Table 15.1*** that the part of the region format we are interested in looks like ***Table 15.2***.

Region Name	Region Code
Europe	EU
Far East	FE
Middle East	ME
North America	NA

Table 15.1: Partial region listing

Entity type	Attribute type #1	Attribute type #2	Etc.
Region	Region full name	Region code	-

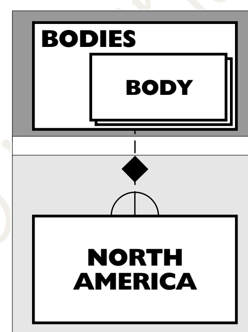
Table 15.2: Region entity format

© Copyright Chris Partridge chris.partridge@BOROCentre.com

2.2 Re-engineering region entity signs

We follow the rules and start by re-engineering a couple of entities for the entity type. We need to start with an entity; so, we select one from the partial listing of regions in ***Table 15.1***; we pick the North America entity. This re-engineering has the same pattern as the United States in the country example. Following this gives us the object schema in ***Figure 15.2***. Not surprisingly, this has the same shape as the United States schema in ***Figure 12.7***.

Figure 15.2: North America object schema



We now re-engineer another entity. We pick Europe from the partial listing of regions in ***Table 15.1***. We follow the same pattern that we used for United Kingdom in the country

example and this gives us the object schema in ***Figure 15.3***—with the same shape as ***Figure 12.9***.

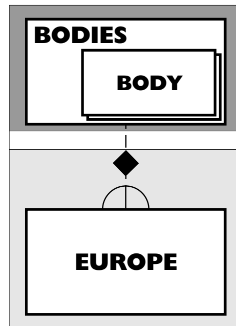


Figure 15.3: Europe object schema

Like the country example (see ***Figure 12.10***), we merge the two schemas and get ***Figure 15.4***. It should be becoming clear how re-using patterns can simplify the re-engineering process. Even though we need to retrace each of the steps to confirm that the patterns are the same shape, it still makes the whole process much simpler.

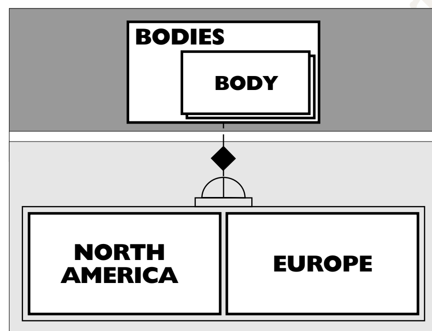


Figure 15.4: Merged North America and Europe object schema

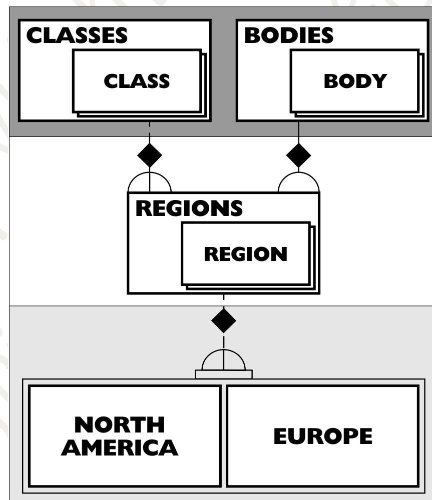


Figure 15.5: Regions object schema

2.3 Re-engineering the entity type sign

The re-engineering of the two individual regions reveals similar patterns; so, we use them (and the country entity type pattern) as a basis for re-engineering the region entity type. We get the object schema shown in ***Figure 15.5***; as you have by now come to expect, this is the same shape as the corresponding schema for the countries object in ***Figure 12.10***.

2.4 Re-engineering region full name attribute type sign

The re-engineering of the two region attribute type signs follows the same pattern as the country example. We follow the rules and start by re-engineering individual attributes and build up to the attribute type.

2.4.1 Re-engineering the attribute signs

We start with the entity North America's full name attribute, 'North America'. Following the country pattern, we get the object schema in ***Figure 15.6***. This is the same pattern as ***Figure 12.27***, a part from character string classes, which was re-engineered later. We included it in this schema to bind the objects more closely together.

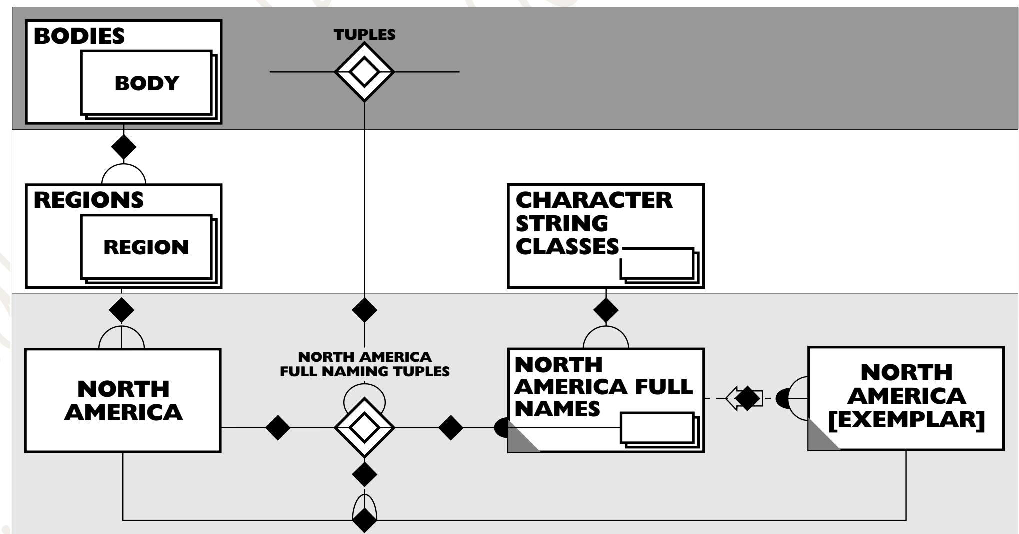


Figure 15.6: North America full naming tuples object schema

We check the patterns by re-engineering another attribute. We pick 'Europe— Europe's full name attribute. Again we follow the same country pattern, giving us the object schema in ***Figure 15.7***. As with ***Figure 15.6***, the only difference between this and its country equivalent is the inclusion of character string classes.

© Copyright chris partridge Partridge Centre.com

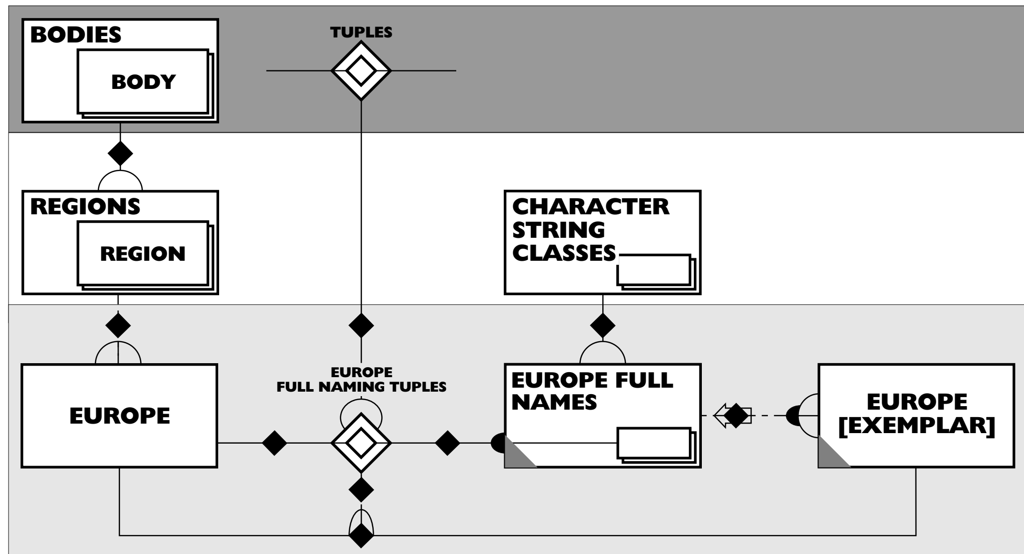


Figure 15.7: Europe full naming tuples object schema

2.4.2 *Re-engineering the attribute type sign*

Following the rules and the country pattern, we now re-engineer the region full name attribute type sign into the region full names class and the region full naming tuples classes. We end up with the object schema in ***Figure 15.8***, which shows the same pattern as ***Figure 12.33***.

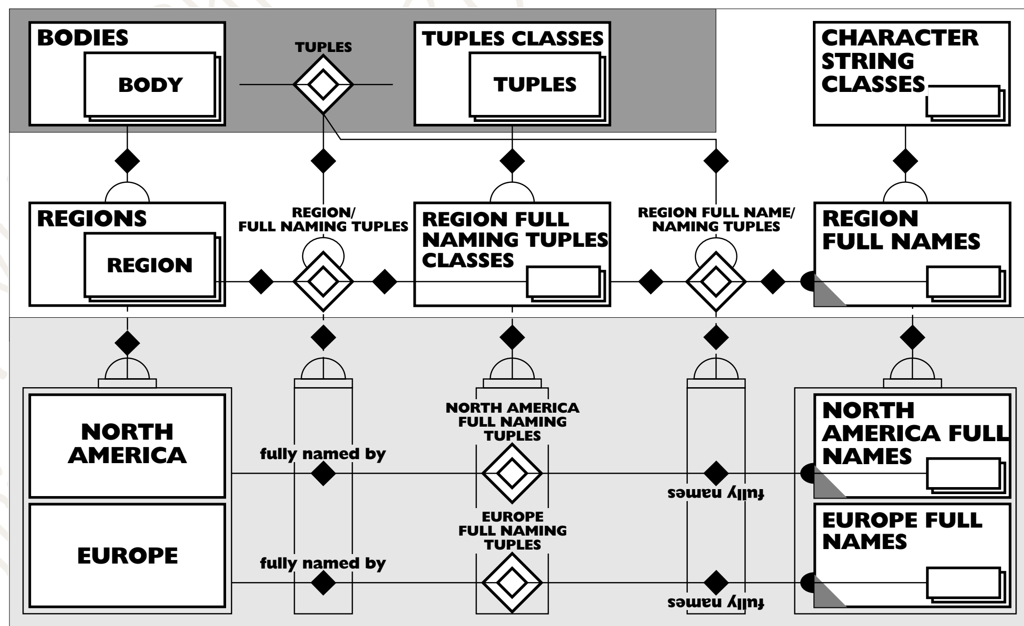


Figure 15.8: Region full naming tuples classes object schema

2.5 Re-engineering region code attribute type sign

The region code attribute type follows both the country pattern and the region full name pattern (in ***Figure 15.8***). We go straight to the final object schema (shown in ***Figure 15.9***).

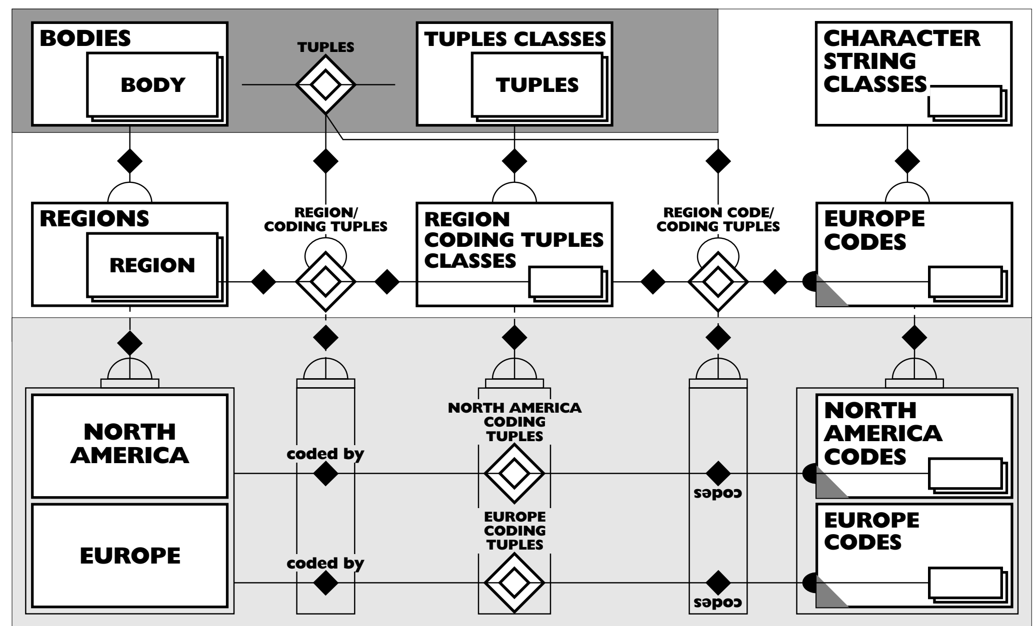


Figure 15.9: Region coding tuples classes object schema

2.6 Generalising within region

Now that we have re-engineered the entity formats, we exploit the opportunities for generalisation. These fall into two groups:

- Within region, and
- Across region and country.

The first group shares the same generalisation patterns as the country example. The second group, which ranges across both the region and country sections of the model, as based on the generalisation of region and country into geo-political area.

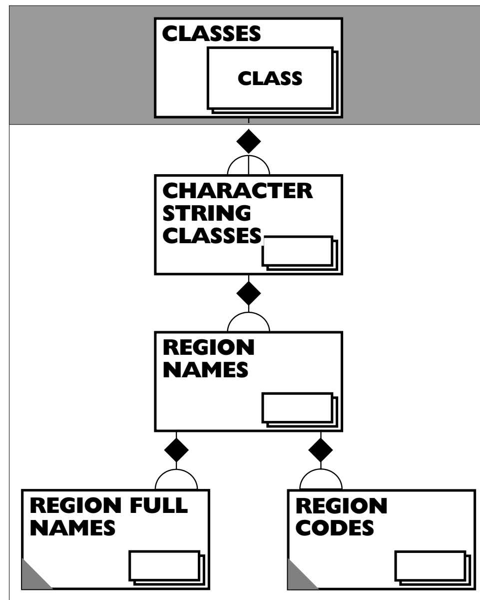
We now look at the first group. There are generalisation patterns for:

- Region names,
- Exemplar region names, and
- Region naming tuples classes.

2.6.1 Generalising to region names

Following the country re-engineering pattern, we generalise region full names and region codes into region names (shown in the schema in ***Figure 15.10***.)

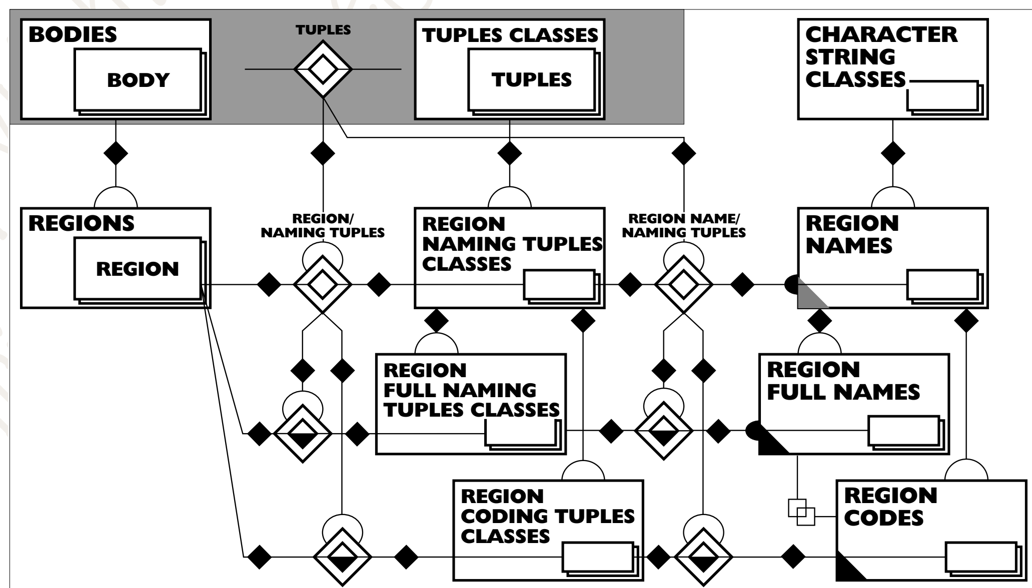
Figure 15.10: Generalised region names object schema



2.6.2 Generalising to region naming tuples classes

We follow the country pattern and generalise region full naming and coding tuples classes into region naming tuples classes (shown in the schema in *Figure 15.11*). Now that we have the general region naming tuples classes, we have also generalised the region full naming and coding tuples classes class places up the super–sub-class hierarchy to region name/naming tuples and region/naming tuples. We have also classified region full names and codes as redundant. We can now, if we wish, purge them, compacting the model.

Figure 15.11: Generalised region naming tuples classes object schema



© Copyright Chris Partridge chris@BOROCentre.com

2.6.3 Generalising to exemplar region names

We follow the pattern for exemplar country names (shown in ***Figure 13.4***) and generalise exemplar region full names and codes into exemplar region names. We also classify exemplar full names and codes as redundant. This gives us the schema in ***Figure 15.12***.

Figure 15.12: Generalised exemplar region names object schema

2.7 Generalising across region and country

Because we used the same patterns to re-engineer the entity formats of both regions and countries, an opportunity potentially exists to generalise across country and region.

2.7.1 Generalising to geo-political areas

The opportunity is real. Country and region can be generalised to a super-class, geo-political area. To help us see the generalisation more clearly we look at the schema before and after generalisation (shown in ***Figures 15.13*** and ***15.14***).

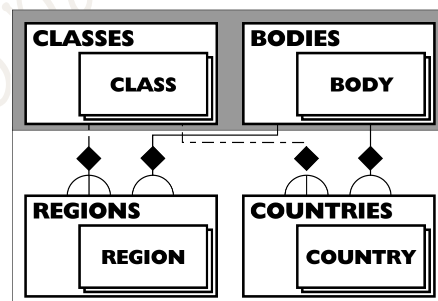


Figure 15.13: Before generalised regions and countries object schema

Before the generalisation, countries and regions are both separately connected to the framework level objects, classes and bodies. After the generalisation, geo-political areas is connected to these framework level objects, with regions and countries connected to geo-political areas.

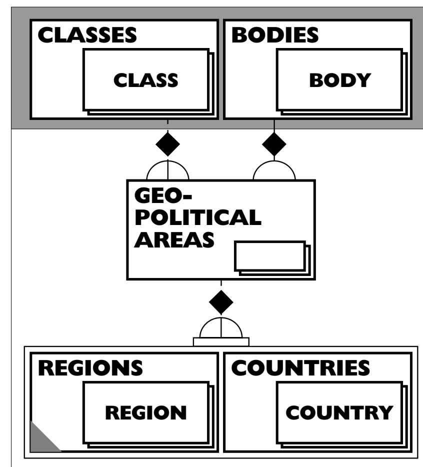


Figure 15.14: After generalised regions and countries object schema

There is one final bit of re-engineering with regard to what a region is. It is really no more than a geo-political area that is not a country. This means it is derived from the country class. We reflect this in the object schema in ***Figure 15.14***. Regions cannot be classified as redundant because it still has a number of connecting patterns depending on it, such as region naming tuples classes and region names. When we have generalised these, we will classify it as redundant.

2.7.2 Generalising to geo-political area names

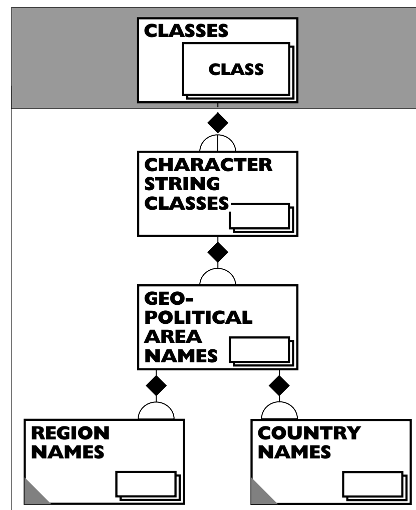
This generalisation of countries and regions to geo-political areas creates further opportunities for generalising their connecting patterns. Where countries and regions have similar connecting patterns, these can now be generalised into a single higher level connecting pattern for geo-political areas. We do this for the following connecting patterns:

- Geo-political area names,
- Geo-political area full naming and coding tuples classes,
- Exemplar geo-political area names, and
- Geo-political area naming tuples classes.

We start by generalising region and country names into geo-political area names. The result is shown in ***Figure 15.15***. This generalisation pattern is not new. It is exactly the same shape as the generalisation of country full names and codes to country names in ***Figure 13.1*** (and so the generalisation of region full names and codes to region names in ***Figure 15.10***). The region and country names are classified as derived. Later on, when we re-engineer their naming tuples classes, removing their dependent connections, we will classify them as redundant.

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

Figure 15.15: Generalised geo-political area names object schema

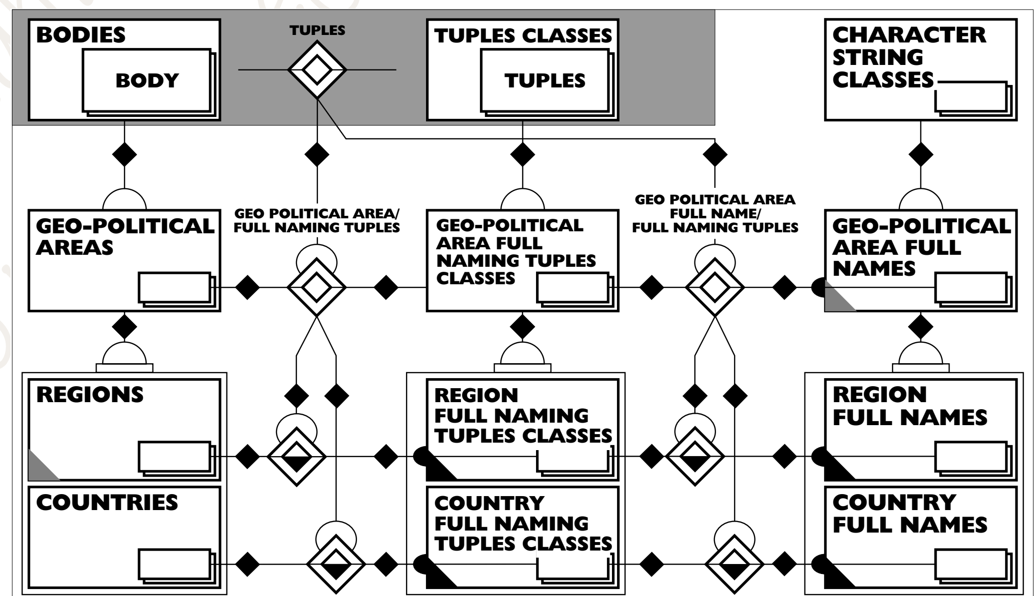


2.7.3 Generalising geo-political area full naming and coding tuples classes

As the object model now stands, we still use separate full naming and coding tuples classes for country and region. It makes sense for us to generalise each of these up to a single class at the geo-political area level. This is the next step in moving the patterns for 'naming' from countries and regions up to the geo-political area level.

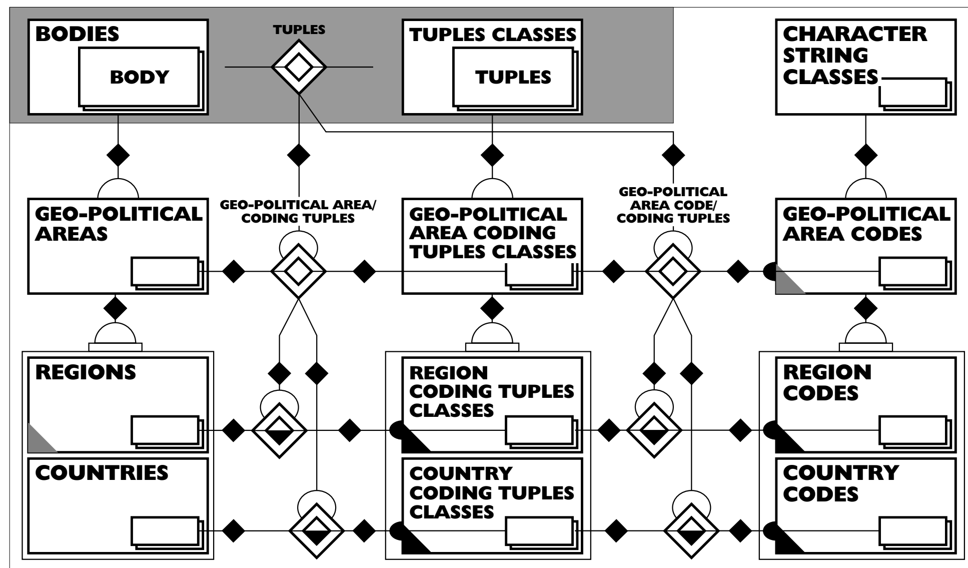
We generalise the region and country full naming tuples classes first. The result is shown in ***Figure 15.16***. Then we follow the same pattern and generalise the region and country coding tuples classes. The result is shown in ***Figure 15.17***. As you can see, the generalisation of the country and region level tuples classes to geo-political area level has enabled us to classify them as redundant.

Figure 15.16: Generalised geo-political area full naming tuples classes object schema



© Copyright Chris Partridge chris@BORO Centre.com

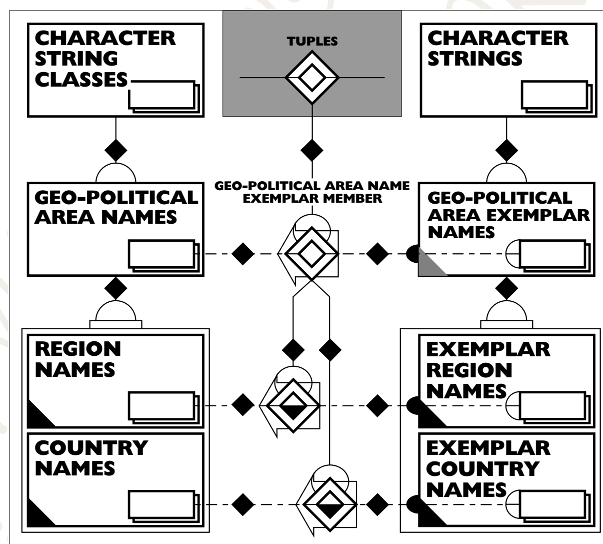
Figure 15.17: Generalised geo-political area coding tuples classes object schema



2.7.4 *Generalising to exemplar geo-political area names*

We can generalise exemplar names to the geo-political area level, as shown in ***Figure 15.18***

Figure 15.18: Generalised exemplar geo-political names object schema

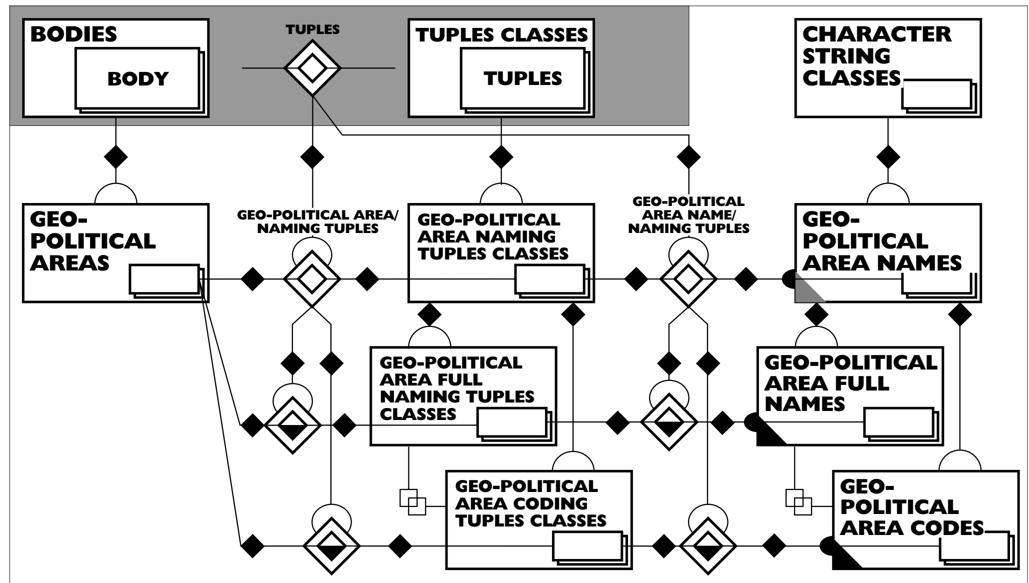


2.7.5 *Generalising to geo-political area naming tuples classes*

Finally we generalise to geo-political area, naming tuples classes from the geo-political area full naming tuples classes and geo-political area coding tuples classes (shown in ***Figure 15.19***). This has a similar pattern to the naming tuples classes generalisation

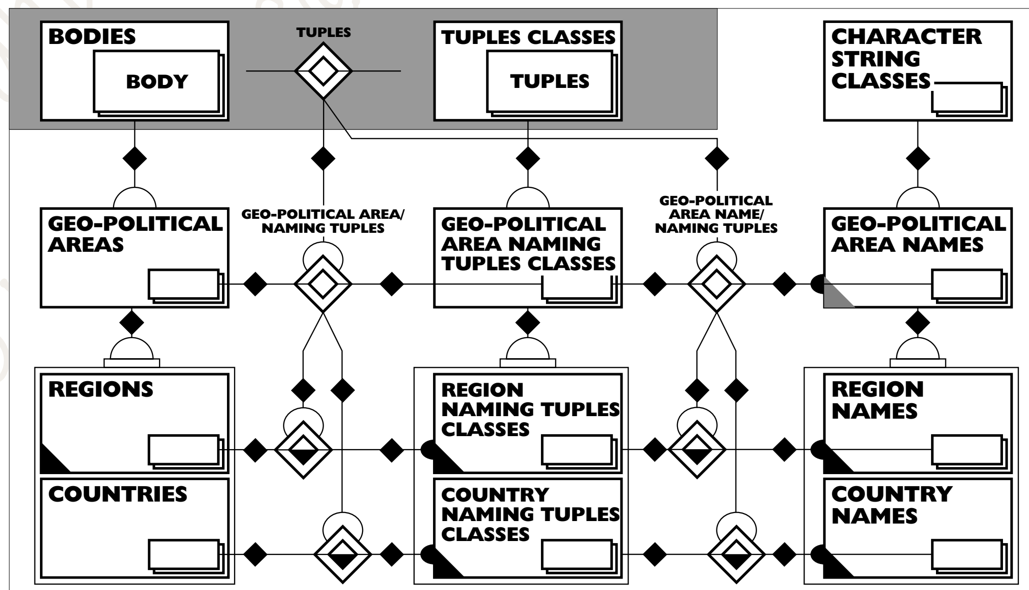
schemas for country and region. Following the region pattern (shown in ***Figure 15.11***), we classify the codes and full names objects redundant.

Figure 15.19: Generalised geo-political area naming tuples classes object schema—1



We can also follow a different generalisation route to the same geo-political area naming tuples classes, by generalising the country and region naming tuples classes. This gives us the object schema in ***Figure 15.20***. With the generalisation to geo-political area names, we classify the lower level region and country names redundant. Our classification of the region naming tuples classes object as redundant means that the regions object no longer has any dependent connections; so, we have also classified it as redundant.

Figure 15.20: Generalised geo-political area naming tuples classes object schema—2



The generalisation has significantly compacted the model. The application level now has only eleven non-redundant objects. These can all be fitted onto one schema—***Figure 15.21***. You may notice that the character strings patterns are not included in this schema; this is because they are not part of the first stage re-engineering. They were initially constructed as part of the second stage of the country re-engineering.

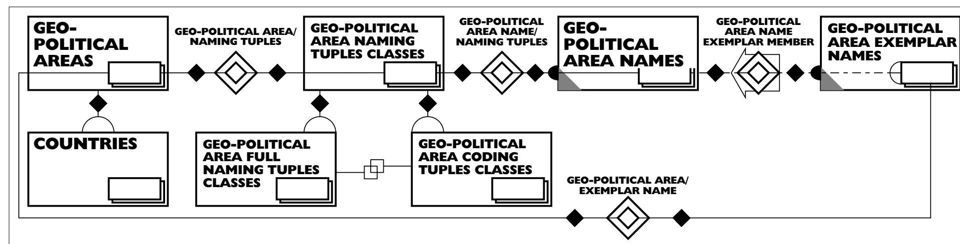


Figure 15.21: First stage application level object schema

2.8 Re-engineering the region entity formats

This completes the first stage—re-engineering the existing system's region entity formats. The generalisation of the names and region patterns to the geo-political level is a good illustration of how patterns from previous re-engineerings are re-used to both simplify the re-engineering process and provide opportunities for generalisation.

The example has generated high levels of generalisation, re-use and compacting. What is also interesting is seeing the large number of patterns that have been re-used from the re-engineering of country.

It also shows quite clearly that the re-engineering of the entity formats is not just a translation of the existing entity patterns into an object semantics. For instance, you will not find the patterns for the comprehensive view of countries and regions as geo-political areas, and the general view of names across all three, in the existing entity system. The systematic re-engineering process takes patterns from an existing entity system and produces a more general object model.

3 Re-engineering our conceptual patterns for region

We now move onto the second stage of the re-engineering—capturing our conceptual patterns for region. The previous chapter's country example provided us with an illustration of the process for investigating our conceptual patterns. It showed us the steps that we need to go through to identify interesting ones and the process of re-engineering them into the object model.

If this was a 'real' re-engineering, and not an exercise, we would go through a full investigation of all region's conceptual patterns. Instead, we are just going to look at how the nesting and stages patterns re-engineered in the country example can be re-used on region.

We apply country's two patterns to region. Then (as with the patterns in the first stage of region's re-engineering), we generalise them and their corresponding country patterns up to the geo-political areas level. It turns out that all these patterns can be generalised into a single geo-political areas pattern.

3.1 Nested regions

When we re-engineered country's nesting pattern in ***Chapter 14*** (illustrated in ***Figure 14.11***, I noted that it looked as though it could be re-used for other types of areas. It can be; we now look at two examples:

- Region nested within region, and
- Country nested in region.

We start off, as we did in the country example, by assuming that the individual nesting patterns are between individual regions and countries—not stages. We do not have region's stages pattern until we re-engineer it in the next section.

3.1.1 Region nested in region

We start with the region nested in region pattern. The original sample of regions we listed in ***Table 15.1*** are mutually exclusive; they do not nest. In fact, most computer system's regions pattern assumes that regions do not nest. We came across a similar assumption when we re-engineered country's nesting pattern. Both assumptions are made for the same reason; the systems use both countries and regions as hooks to summarise figures. Typically, countries provide the first level of summarisation, regions the second.

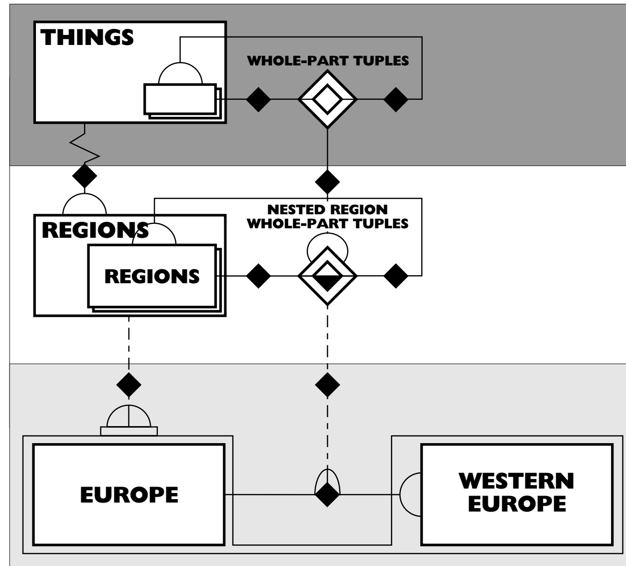
However in 'reality' we can and do have nested regions, just as we can and do have nested countries. For the re-engineering, we take a revised sample of regions, one that provides us with examples of nesting. This is given in ***Table 15.3***. The new regions are shaded.

Region Name	Region Code	Nesting in Region
Europe	EU	
European Community	EC	
Far East	FE	
Middle East	ME	
North America	NA	
South America	SA	
Western Europe	WE	Europe

Table 15.3: Revised partial region listing

When we re-engineer the Western Europe nesting in Europe pattern in ***Table 15.3***, we get the object schema in ***Figure 15.22***. This has the same pattern as nested country whole–part tuples (shown in ***Figure 14.11***). As in the nested country pattern, the nested region whole–part tuples is redundant, derived from the regions class.

Figure 15.22: Region nesting in region



3.1.2 *Country nested in region*

We now turn our attention to the nesting pattern from country to region. This pattern is likely to be an existing system pattern. Many computer systems, certainly in my experience, connect countries to regions. Normally their country entity format has a 'belonging to region' attribute type that relates each country to one region—like the one shown in ***Table 15.4***.

Entity Type	Attribute Type #1	Attribute Type #2	Attribute Type #3	Etc.
Country	Country name	Country code	Belonging to region	-

Table 15.4: Revised country entity format

If we update the original Partial Country Listing (***Table 12.1***) with the 'new' attribute type, we get the extended partial country listing in ***Table 15.5***.

If we re-engineer this, we get the, by now familiar, nesting tuples pattern, this time between the countries and regions classes (shown in the schema in ***Figure 15.23***). You may have noticed that the country nested in region whole–part tuples class is shown as derived—rather than redundant as in the earlier versions of the nesting pattern (for example, ***Figure 15.22***). This is because the tuple class has additional information in its cardinality faithfully reflecting the constraints in the entity format in ***Table 15.4***. The cardinality linking it to countries is constrained to optional-to-one; in other words, each country can only 'belong to' one region. This is because (as we discussed in some

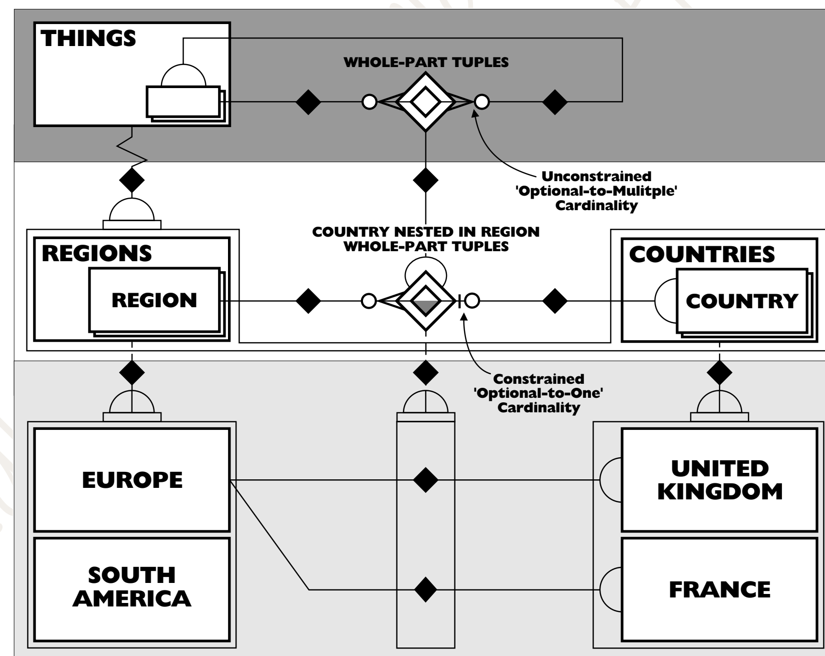
© Copyright Chris Partridge chris.partridge@BORO Centre.com

detail in ***Chapter 3***) attributes, such as ‘belonging to region’ cannot reflect ‘many-to-many’ connections.

Country Names	Country Codes	Belonging to Regions
Germany	DM	EU (Europe)
Italy	IT	EU (Europe)
Japan	JP	FE (Far East)
Turkey	TK	ME (Middle East)
United Kingdom	UK	EU (Europe)
United States	US	NA (North America)

Table 15.5: Extended partial country listing

Figure 15.23: Country nested in region object schema



If you look carefully at the country listing in ***Table 15.5*** and the region listing in ***Table 15.3***, you should be able to spot how the entity format constraint has made the listing incomplete. Something is missing for Germany, Italy and the United Kingdom. They are not only part of Europe but also part of the European Community.

One simple workaround that may spring to mind is to make the European Community (EC) part of Europe (this gives a tree structure). But the example has been constructed so that this does not work. Turkey, a potential member of the EC (it has applied), is not part of Europe. In other words, Europe does not necessarily contain all the European

© Copyright chrispartridge.com

Community (it will not when Turkey joins the EC). And vice versa, the European Community does not yet contain all of Europe.

So an accurate model needs to be able to reflect European EC members, such as Germany, as nested in at least two regions: Europe and the EC (this gives a lattice structure). We revise the extended partial country listing in ***Table 15.5*** to reflect this. The result is shown in ***Table 15.6***.

Country names	Country Codes	Belonging to Regions
Germany	DE	EU (Europe) WE (Western Europe) EC (European Community)
Italy	IT	EU (Europe) WE (Western Europe) EC (European Community)
Turkey	TK	ME (Middle East) EC (European Community)
Japan	JP	FE (Far East)
United Kingdom	GB	EU (Europe) EC (European Community)
United States	US	NA (North America)

Table 15.6: Revised extended partial country listing

We reflect this insight in the object model by relaxing the cardinality from optional-to-one to optional-to-multiple (shown in ***Figure 15.24***). If you think about it, you will realise that this less constrained cardinality applies to the other nesting patterns (country nested in country and region nested in region) as well. You may have noticed that the relaxation of the cardinality has another effect. The country nested in region whole–part tuples class is now classified redundant—like the other nesting tuples classes. This is because its cardinalities no longer contain any additional information.

3.2 Region stages

We now extend the nesting pattern to include region stages, following the country pattern. We start by re-engineering one of the events that cause these stages, country joining region. This gives us the region stages pattern, which we use to generalise the nested pattern to the geo-political areas level.

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

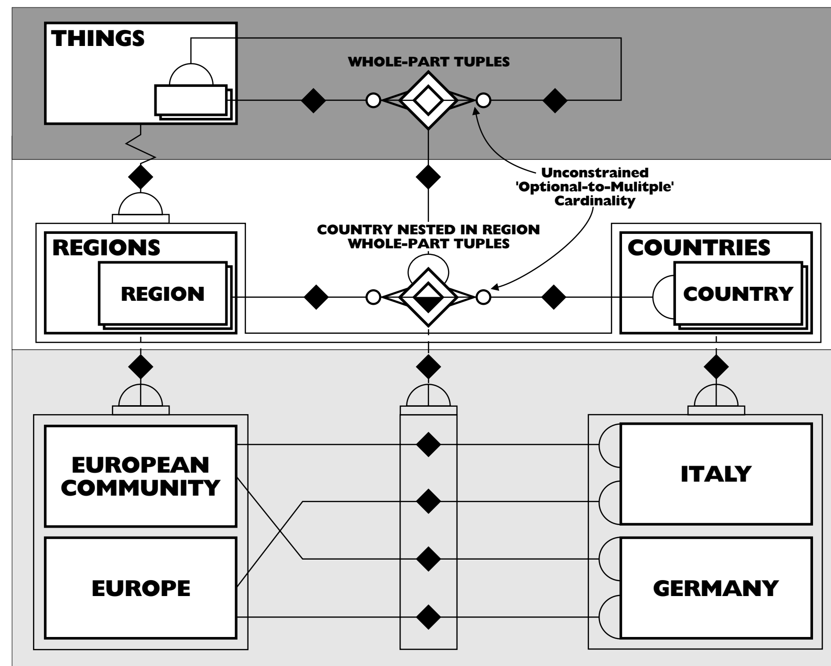


Figure 15.24: Unconstrained country nesting in region object schema

3.2.1 Re-engineering the country joining region event pattern

We start with an example; the United Kingdom joining the European Community in 1973. The re-engineering gives us the space-time map shown in ***Figure 15.25*** and the object schema shown in ***Figure 15.26***. These have similar shapes to the country re-engineering example; the 1707 Act of Union event joining Scotland to the United Kingdom (shown in ***Figures 14.12*** and ***14.13***). This bodes well for the generalisation.

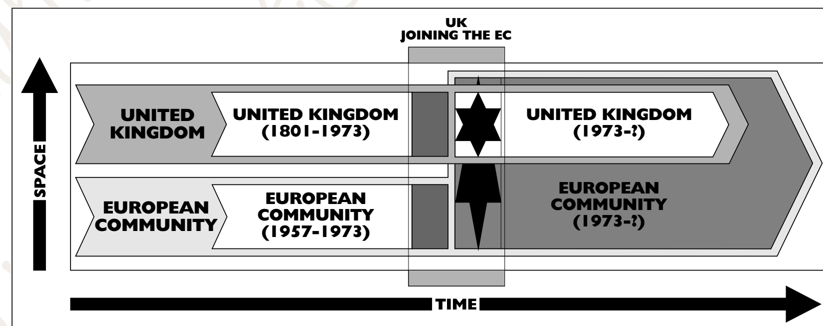


Figure 15.25: United Kingdom joining the European Community space-time map

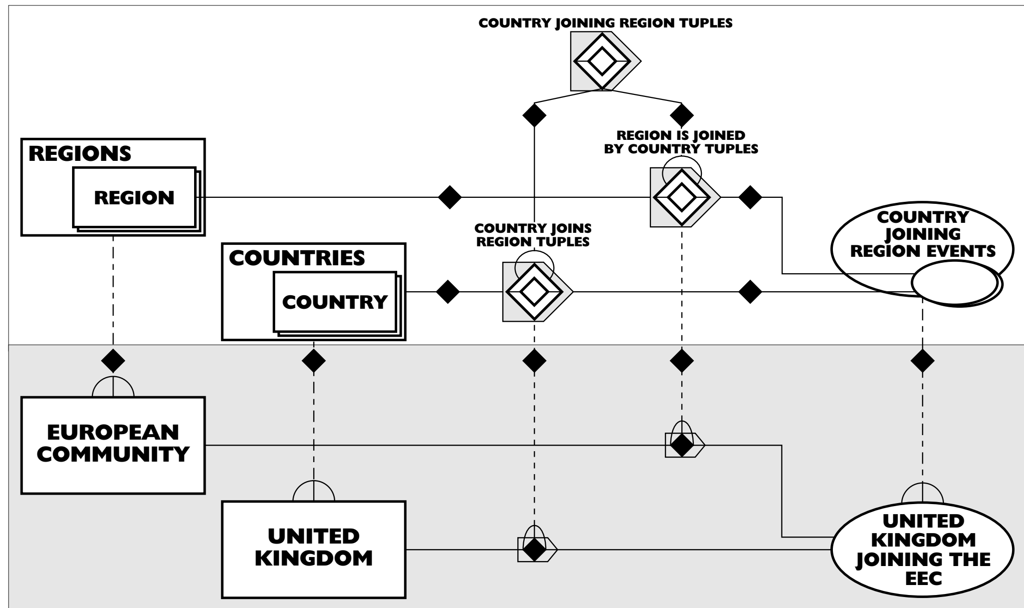


Figure 15.26: Country joining region events object schema

We then take the pattern up to class level. First for the country joining region events, giving us the schema in ***Figure 15.26*** And then for the country and region stages, giving us the schema in ***Figure 15.27***. Note that this combines the nesting and stages patterns and generalises them to the geo-political areas level.

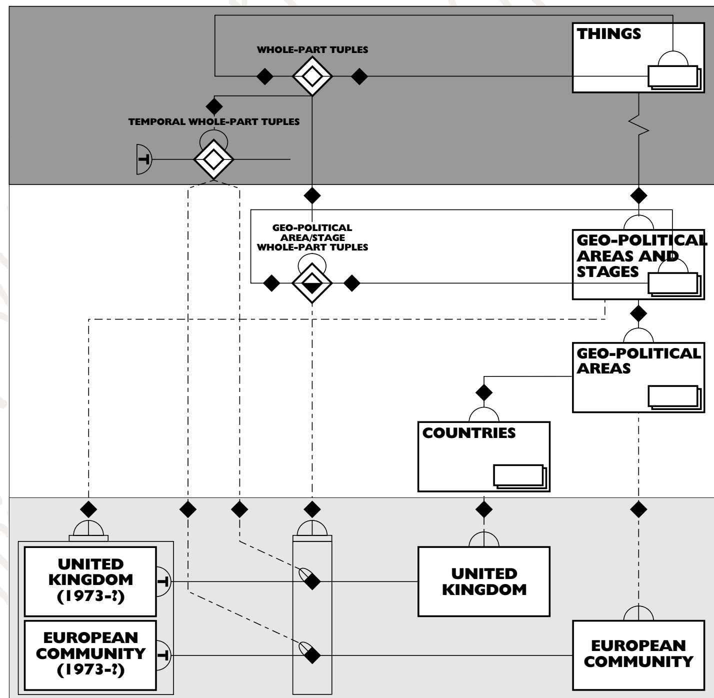


Figure 15.27: Generalised geo-political areas & stages

3.2.2 Generalising to geo-political areas joining event

Our concern here is not with joining events. We only use them to provide us with an example of the region stages pattern. However, it should be clear that the country joining region event pattern is one of the geo-political area joining event patterns. And that country's Acts of Union pattern (see ***Figure 14.17***) is also one, although they have slightly different shapes. Unlike the country joining region event pattern, the joining in the Acts of Union pattern constructs a new country. In a 'real' re-engineering, we would re-engineer all the different patterns and then generalise them to the geo-political area joining event, raising the pattern up to the geo-political areas level. But we have done all we need to do for this worked example.

3.2.3 Geo-political area nesting/stage pattern re-use

If we consider the re-engineering so far, three main patterns have emerged. These are:

- the naming pattern,
- the geo-political area joining pattern, and
- the combined geo-political area nesting/stage pattern.

All of these provide us with good examples of the power of re-engineering. Take the last pattern—geo-political area nesting/stage. The application level (the one that matters for system building) of the object model in Figure 15.27 is both more general and functionally richer than the system from which it was re-engineered. It operates at the geo-political area level, and unlike the entity format, it can reflect:

- countries nested in countries,
- regions nested in regions, and
- countries nested in multiple regions.

In the next chapter, this pattern will be re-used again, generating more compacting.

4 Summary

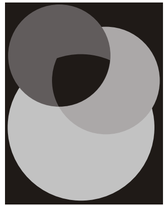
Our object model for spatial patterns is now more general and more powerful after the generalising of country's conceptual patterns across regions and up to geo-political areas.

This example has shown, again, how much potential there is for re-use and generalisation. The way in which country and region share patterns, and the ease with which these patterns combine, is uncanny. However, this can be explained in terms of our strong unconscious conceptual patterns for space. Uncovering these and making them explicit and accurate in an object model is the purpose of the re-engineering.

The re-engineering of spatial patterns is not yet complete. In the next chapter, we re-engineer our final example of a spatial pattern—address. However, this will demonstrate the re-usability of our general, object model, rather than enhance it.

© Copyright Chris Partridge, chris.partridge@BORO-Centre.com

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 16

Re-Engineering Bank Address

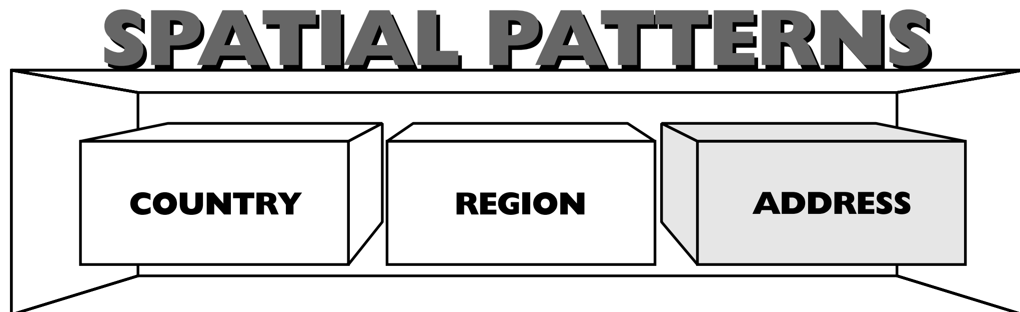
© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction
2 Familiarising ourselves with bank address entity formats
3 Re-Engineering bank
4 Re-Engineering address
5 Nested address lines
6 Address joining events
7 Generalising name
8 An aspect of a pattern
9 Summary

1 Introduction

We have completed the re-engineering of the first two examples of spatial patterns; country and region. We now move onto the third and final example, address (see ***Figure 16.1***). This example illustrates the power of the basic object model for spatial patterns we have constructed. It shows how it can be re-used to capture the patterns for address, without any changes. It also provides us with an opportunity to construct a truly general model for the naming pattern.

Figure 16.1: Third and final of three examples of spatial patterns



As you are now quite familiar with the systematic re-engineering approach, we take a high level view. Even though address's entity format is quite different from the earlier formats, its re-engineering follows the same basic pattern. Like before, we re-use patterns that we have already re-engineered; and, as you have now come to expect, this means we generalise them.

2 Familiarising ourselves with bank address entity formats

We start off, as always, by familiarising ourselves with specific examples of the entity format we are going to re-engineer.

Bank	001	BarcWest Bank	003	Chase Hanover Bank
Address		Black Horse House Moorgate London United Kingdom		BarcWest Tower Old Broad Street London United Kingdom
Bank	002	NatLand Bank	004	Banco di Guernsey
Address		Listening Buildings 21 Moorgate London England		54th Floor BarcWest Tower Old Broad Street London England

Table 16.1: Partial address listing

Table 16.1 has a partial address listing that we use to do this. This is really a partial view of the address fields on the bank file—in other words, the attributes for the bank entity type. We call it bank address, because we are only interested in bank address's spatial patterns. However, we must re-engineer bank to get to bank address.

From ***Table 16.1***, we can work out that the entity format for bank address (bank) that is shown in ***Table 16.2***

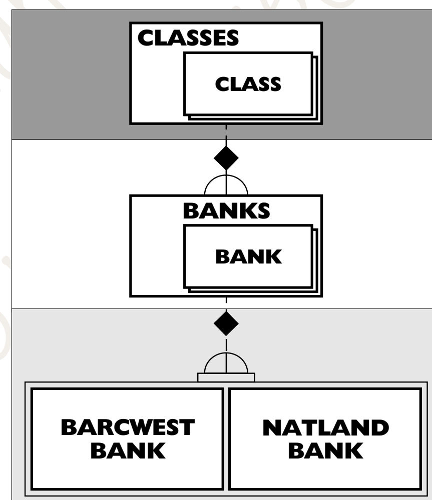
Entity type	Bank
Attribute type #1	Bank code
Attribute type #2	Bank full name
Attribute type #3	Address line 1
Attribute type #4	Address line 2
Attribute type #5	Address line 3
Attribute type #6	Address line 4
Attribute type #7	Address line 5

Table 16.2: Bank address entity format

3 Re-Engineering bank

We start by re-engineering, in outline, the bank elements of the entity format. In this example, we look at the bank entity type sign and its attribute types, bank full name and code.

Figure 16.2: Banks object schema



© Copyright chris.partridge@BOROCentre.com

3.1 Re-engineering the bank entity type sign

We follow the rules and start by re-engineering the entity type sign before the attribute type signs. We are not really interested in bank because it is not a spatial or naming pattern; it is just a hook to hang address's spatial patterns on; so, we move through the re-engineering quickly. We re-engineer two individual banks and use these to construct the banks class. We construct signs for these classes and get the object schema in ***Figure 16.2***.

3.2 Re-engineering bank's full name and code attribute type signs

The bank full name and code attribute types are based on naming not spatial patterns, and they can be used to generalise the naming patterns in the model. We start by re-using the geo-political naming pattern (illustrated in ***Figure 15.21***) as a template to construct the bank naming pattern. This gives us ***Figure 16.3***. We will return to this schema later in the chapter to generalise the naming patterns.

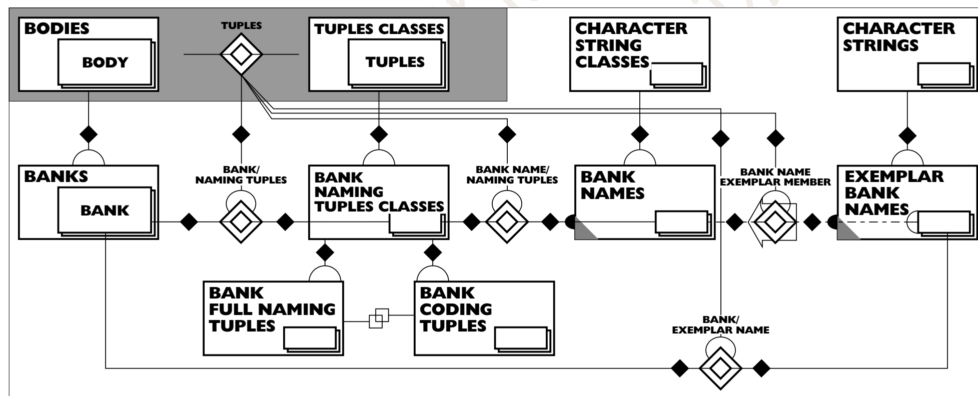


Figure 16.3: Bank naming tuples classes object schema

4 Re-Engineering address

We now move from bank to address and start to re-engineer the five address lines. When most people look at the lines of an address, they see a series of simple attribute signs. When we start analysing these, we will see a very different pattern—one that has been severely distorted to fit into the attribute structure.

4.1 Re-engineering address line one attribute type sign

We follow the rules. We have already re-engineered the entity type sign; so, we can start re-engineering the attribute type signs. We start with the address line one sign.

Table 16.3 has a partial listing of the individual attributes that we use to start the re-engineering.

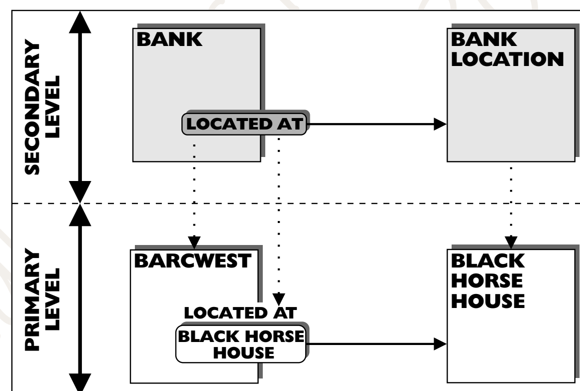
Bank	Address line one
BarcWest Bank	Black Horse House
NatLand Bank	Listening Buildings
Chase Hanover Bank	BarcWest Tower
Banco di Guernsey	54th Floor

Table 16.3: Partial address line one listing

4.1.1 Re-engineering the individual attribute signs

Following the rules, we re-engineer some individual address line one attribute signs before we re-engineer the attribute type sign. We pick BarcWest's—Black Horse House—from *Table 16.3*. What does this sign refer to? Within the entity paradigm, it not only refers to a 'located at' attribute of BarcWest but also implicitly to Black Horse House, where BarcWest is located. It is the implicit relational attribute sign described by *Figure 16.4*.

Figure 16.4: Bank 'located at' location attribute



We start the re-engineering with the Black Horse House entity. This is a building on Moorgate, one of the streets in the City of London. In the object paradigm it persists through time and has a continuous four-dimensional extension; so, it is a physical body object; in other words, a sub-class of the framework class, bodies.

Then we re-engineer the 'located at' connection it has with BarcWest Bank. This is transformed into the couple <BarcWest Bank, Black Horse House> belonging to a bank 'located at' tuples class.

We pick another attribute sign from *Table 16.3*—NatLand Bank's Listening Buildings—and re-engineer it. It follows the same pattern. The sign also refers to a building on Moorgate, one where NatLand Bank is located. We construct signs for both these

objects and their sub-class connection to bodies. This gives the object schema in ***Figure 16.5***.

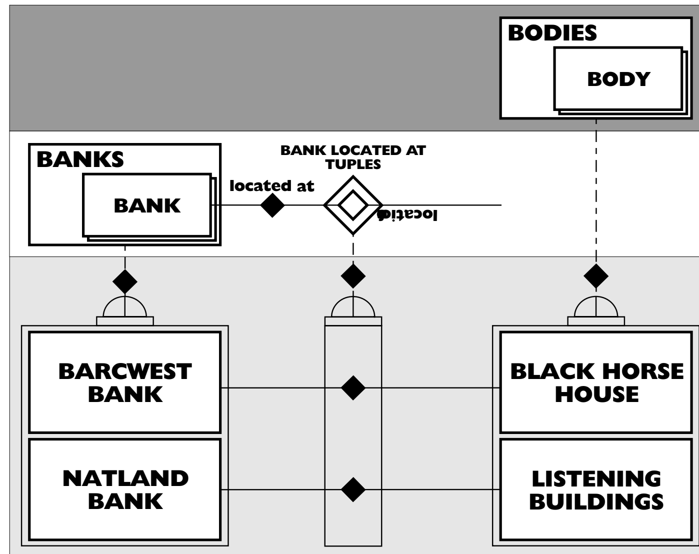


Figure 16.5: Bank 'located at' tuples object schema

4.1.2 Re-engineering the attribute type sign

We now re-engineer the address line one attribute type sign. We have done part of the work with the bank 'located at' tuples class. We now need to construct a class for Black Horse House and Listening Buildings. Address line one is not really a very informative name for the class, so we use 'bank locations'. We construct a sign for it and get the object schema in ***Figure 16.6***. You will notice that bank locations is derived from the bank 'located at' tuples class.

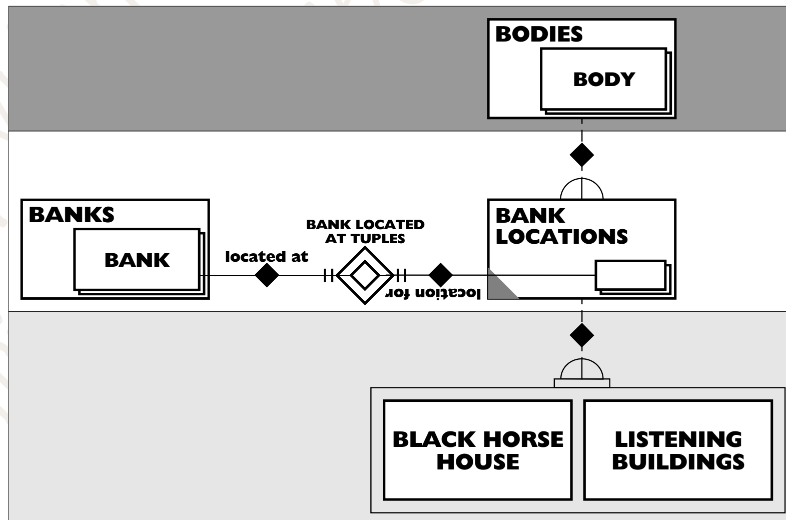


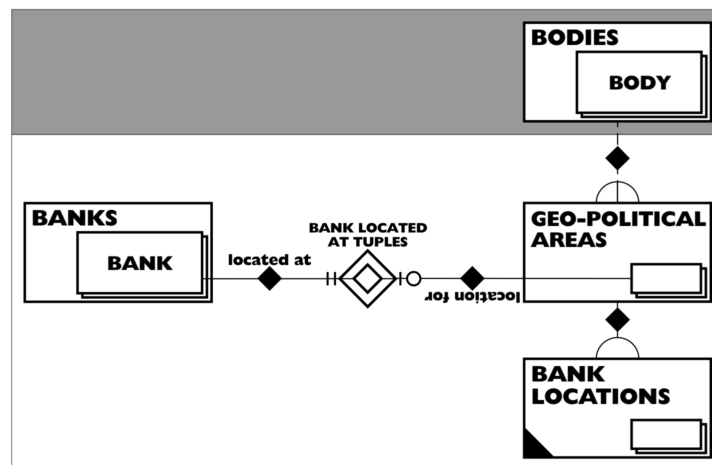
Figure 16.6: Bank locations object schema

© Copyright chris.partridge@BORO Centre.com

If we think carefully, maybe stretching our imaginations a little, we can see that bank locations are a type of a geo-political area. They are not as big as countries or regions, but they work in the same way, occupying a defined space. Once we recognise this, we see that bank locations is a sub-class of geo-political areas. This gives us an opportunity to generalise and compact.

We can generalise the 'bank location for' class place up the super–sub-class hierarchy to geo-political areas following the compacting pattern we have used in earlier examples and discussed in ***Chapter 10*** (illustrated in ***Figure 10.46***). This makes bank locations redundant; it is just those geo-political areas at which a bank is located (shown in the object schema in ***Figure 16.7***).

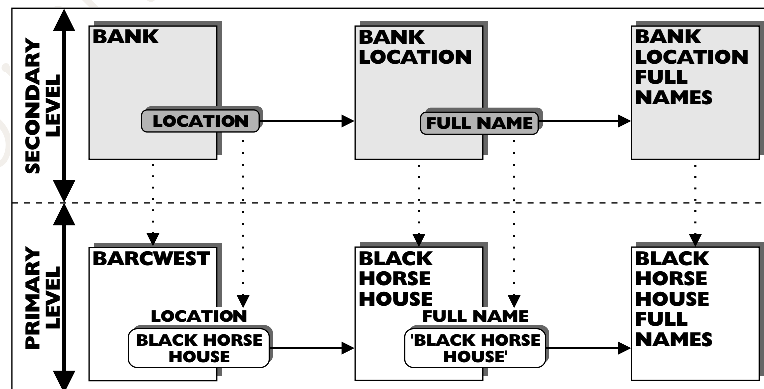
Figure 16.7: Compacted bank 'located at' tuples object schema



4.1.3 Address line one's naming pattern

The address line one (or 'located at') attribute sign needs one last important bit of analysis. Look again at BarcWest's attribute sign. It contains the character string 'Black Horse House'. The particular characters in the string are important. They make up the full name of the location. This means we need to revise the entity model of the attribute in ***Figure 16.4***; we have to add in the bank full names entity as shown in ***Figure 16.8***. We now need to re-engineer this revised entity model.

Figure 16.8: Bank full name and location attributes



©Copyright Chris Partridge chris.partridge@BORO...com

However, if we look carefully at the patterns in our lexicon, we will see that the naming pattern has already been re-engineered. In the region example, we generalised the naming pattern, including the full naming pattern, up to geo-political area level. Because the bank location is a sub-class of geo-political area, it inherits its full naming patterns. Bank location—as a geo-political area—already has a full name. This is illustrated in ***Figure 16.9***. The re-engineering does not need to construct any new objects.

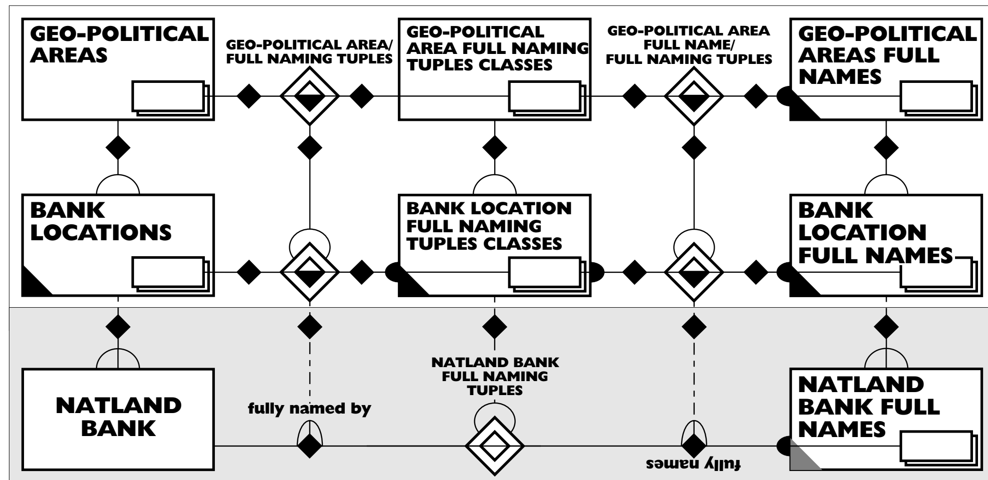


Figure 16.9: Bank locations full name object schema

However, it does introduce a new re-engineering pattern for attribute type signs, such as bank location. We are familiar with the re-engineering pattern for relational attribute type signs, which gives us a tuples class and its place class. We are also familiar with the re-engineering pattern for naming attribute type signs, which gives us a naming tuples class and its name class. Here, we have a re-engineering that combines the two patterns—a sort of naming relational pattern. This is a useful pattern to have when re-engineering. Naming relational attribute type signs are common in entity based systems.

This also is a good illustration of the semantic accuracy required when re-engineering. If we had not been careful, we could have easily stopped the analysis at bank location. Then we would not have recognised that there was a full name pattern to re-engineer—even though it now stares us in the face.

4.2 Re-engineering the other address lines attribute type signs

Because the re-engineering of the other address lines follows a similar pattern, we do it here in bulk. This also helps us to see a connecting pattern between the lines.

Like bank locations, with a little stretching of the imagination, we can see the other address lines as full names for geo-political areas. BarcWest's 'Old Broad Street' and 'London' are larger than bank locations and smaller than countries or regions, but still the same type of object. Address line four, 'United Kingdom', is a country—so, by our previous re-engineering, already a geo-political area. This has other implications that we will examine later.

These other address lines do not have exactly the same pattern as address line one. Unlike it, they do not have a 'located at' connection with a bank, and so are not bank locations. Otherwise, their patterns are the same. At the application level, the other address lines are geo-political areas, and so, like address line one, inherit its full naming patterns. This means we do not need to re-engineer any new application level objects for them (illustrated by the schema for BarcWest's address in ***Figure 16.10***).

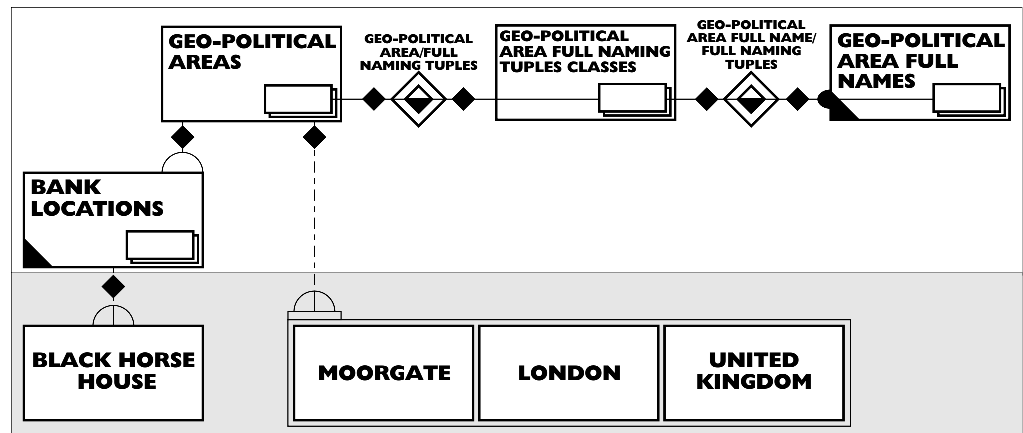


Figure 16.10: BarcWest's address object schema

5 Nested address lines

We have re-engineered all the explicit patterns for these bank address lines, but there is still one vital implicit pattern that our accurate semantic analysis needs to pick up.

If we look at the listing of addresses in ***Table 16.1*** again, it appears that there cannot be anything special about which line the objects appear on. For example, one of them, 'BarcWest Tower', appears as address line one in Chase Hanover's address and as address line two in Banco di Guernsey's address. It cannot be intrinsically both a line one and a line two object.

However, the order of the lines is important. This is easy to show; look at the addresses in ***Table 16.4***. They are clearly not valid addresses; yet, all I have done is move the position of the lines. The reason they are not valid addresses is the positioning of the lines reflects an implicit pattern, and changing the order of the lines breaks it. We are so

© Copyright Chris Partridge chris.partridge@BORO...

familiar with this implicit pattern that we automatically read it in. We only notice that it existed when it is not there.

Bank	001	BarcWest Bank	003	Chase Hanover Bank
Address		United Kingdom		London
		Moorgate		BarcWest Tower
		Black Horse House		United Kingdom
		London		Old Broad Street

Table 16.4: Altered address lines

5.1 Implicit whole–part tuples

A little thought shows that the order of the address lines implies a whole–part connection between them. For example, because London is on the line before United Kingdom, this implies it is part of United Kingdom. We broke the order of this whole–part pattern when we moved the lines around. ***Figure 16.11*** shows an entity view of the whole–part pattern for BarcWest's address.

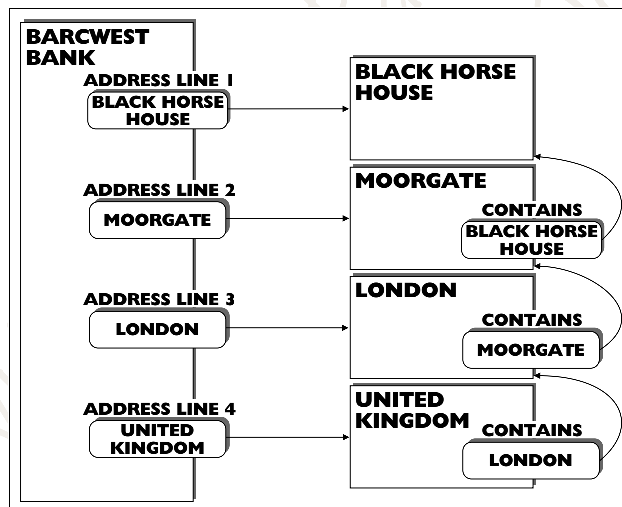


Figure 16.11: BarcWest's address's implicit whole–part patterns

This is not how Aristotle's entity paradigm structure was originally meant to work. It has been 'bent'. We are now so familiar with this 'bending' that we automatically supply the implicit whole–part patterns. However, we do not need to do any bending in the object paradigm. In fact, the strong reference principle demands that we map the whole–part connections directly and explicitly into the model. ***Figure 16.12*** illustrates the result.

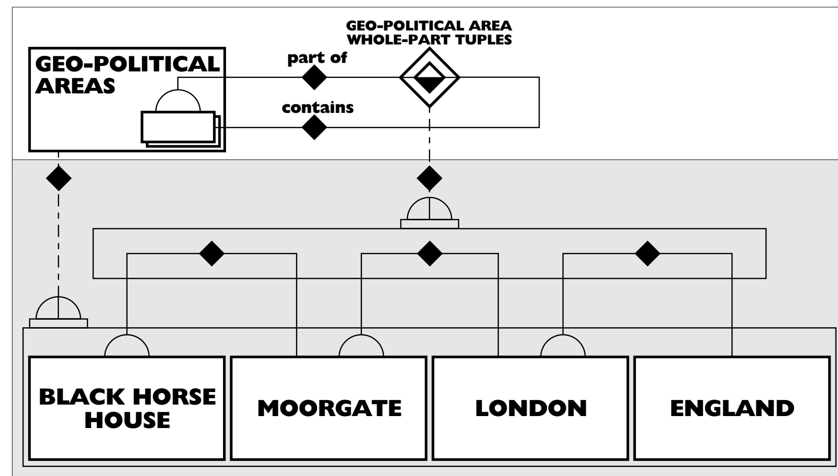


Figure 16.12: BarcWest's address's whole–part tuples object schema

5.2 More accurate whole–part pattern

You have probably recognised that this modelling of the whole–part pattern is not as accurate as it should be. In ***Chapter 14***, when we re-engineered country's nested whole–part patterns, we saw how Scotland was only 'part of' the Great Britain for part of its life. In other words, a stage of Scotland—and not Scotland—was part of the United Kingdom. This is illustrated in ***Figure 14.12***.

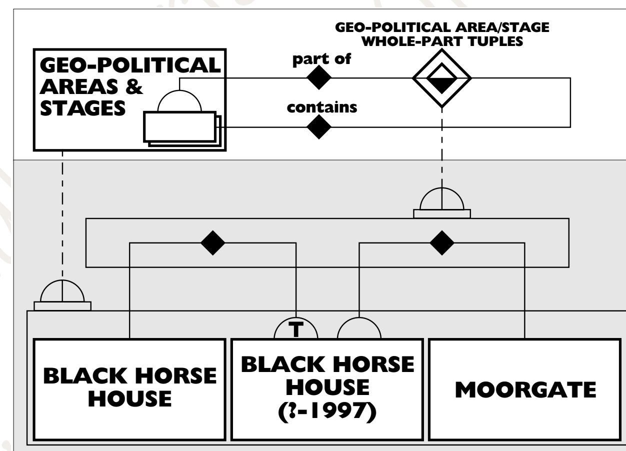


Figure 16.13: More accurate whole–part patterns

The address lines are geo-political areas and so could have a similar pattern to Scotland. Though it is unlikely, there is no reason why Black Horse House should not be bought by a foreign billionaire and transported to another country. In this case, Black Horse House would have a similar pattern to Scotland. It would only be part of Moorgate for part of its life. In other words, a stage (state) of Black Horse House would be part of Moorgate. We do not need a new pattern to capture this. We can re-use the geo-political areas and stages nesting pattern. ***Figure 16.13*** illustrates what the more accurate model looks like.

© Copyright chris partridge BORO centre

5.3 Re-engineering the same object twice

When we were re-engineering the other address lines, we noted that one of the address lines—BarcWest's address line four—was the full name for a country—'United Kingdom'. This raises an important point. It is impractical to apply the strong reference principle consistently in the constrained entity environment. So, sometimes, there is more than one sign for an entity. This is a case in point. United Kingdom on the country file and United Kingdom in BarcWest's address both refer to the same entity.

Under the strong reference principle, we should re-engineer these two entity signs into one sign in the object model. Where there is duplication in the existing system, we *do not* replicate it in the model. So we do not re-engineer a new sign in the model for the United Kingdom, but re-use the existing one. ***Figure 16.14*** shows the re-engineering pattern.

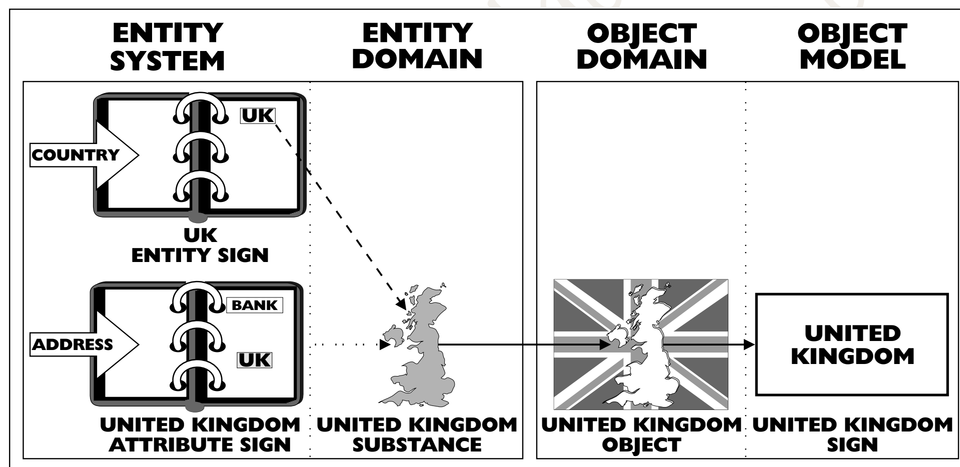


Figure 16.14: United Kingdom re-engineering pattern

Duplication of signs occurs with a vengeance within address. Consider the list of address line attributes from ***Table 16.1*** in ***Table 16.5***. It is plain that the four Londons (and three Englands) in the various addresses are full names for the same entities. ***Figure 16.15*** shows what the re-engineered undistorted de-duplicated model looks like for two of the addresses.

This de-duplication leads to compacting. Transforming two (or more) signs in the entity model into one object sign makes the model smaller and leaner. It also makes maintenance of the implemented system easier. In the existing system, when an address entity changes its full name then this change has to be applied to all its full name signs. Let's say London changed its name to Londres; then, every bank address line that contains 'London' would need to be changed. There are four of these in this small example; there could be hundreds, or even thousands, more in a working entity oriented system. How-

ever, in a system based on the object model, there would only be one and so the change would only need to be done once.

Bank	Address Line No.	Location
BarcWest Bank	3	London
NatLand Bank	3	London
	4	England
Chase Hanover Bank	3	London
Banco di Guernsey	4	London
	5	England

Table 16.5: Selected address line attributes listing

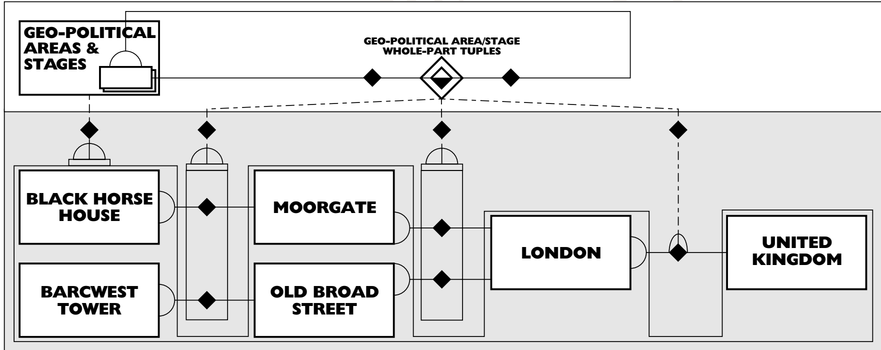


Figure 16.15: BarcWest and Chase Hanover Bank addresses

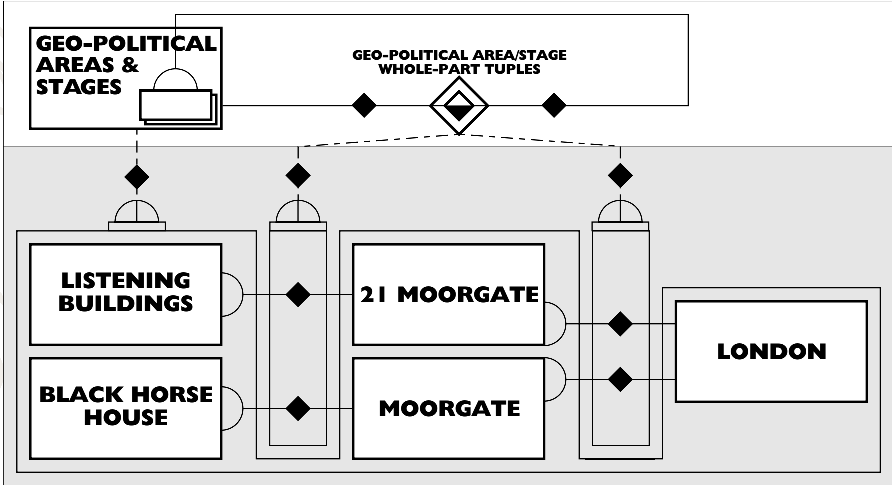


Figure 16.16: Moorgate object schema

© Copyright chris.partridge@bore.com

5.4 One name referring to two objects

Addresses contain a number of implicit patterns and it can take some time to uncover. For example, one of the addresses in ***Table 16.1*** contains another implicit pattern. Consider two address lines; BarcWest Bank's address line two—'Moorgate'—and NatLand Bank's address line two—'21 Moorgate'. We automatically recognise that BarcWest and NatLand Banks are both located on the same street—Moorgate. However, this recognition has not been captured by the re-engineering (shown in their model in ***Figure 16.16***). Listening Buildings is not signed as a part of Moorgate.

To capture this, we have to divide the name '21 Moorgate' into two names—'21' and 'Moorgate'. The name '21' is another way of indicating the NatLand Bank's location, Listening Buildings. It is not a full name, so we treat it as a code. 'Moorgate' is a duplication of BarcWest Bank's address line two—Moorgate—the full name for a street. The new model is shown in ***Figure 16.17***. This clearly shows that the two buildings are both located in Moorgate.

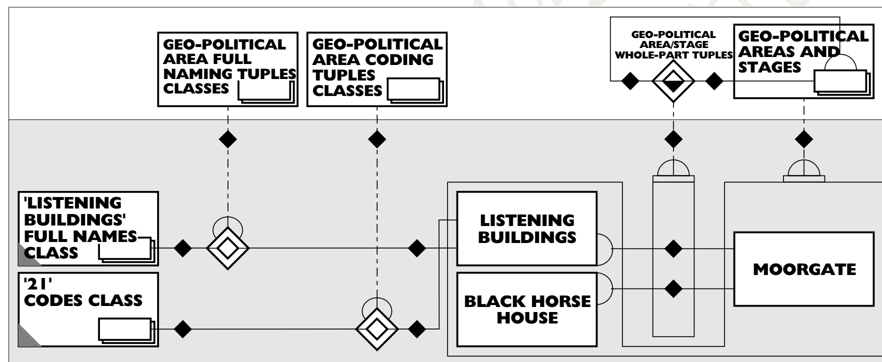


Figure 16.17: New Moorgate object schema

6 Address joining events

When we re-engineered country and region, the nesting pattern went hand in hand with joining events. For example, the stage of Scotland that is part of Great Britain was created by the 1707 Scottish Act of Union Joining Event. When we re-engineered region, we generalised the patterns to the geo-political area level. Addresses, as geo-political areas, inherit the joining event patterns.

These patterns are useful for addresses. When counties' boundaries are moved and towns end up in new counties—a reasonably regular occurrence nowadays—this is a joining event. It can be modelled re-using the geo-political area joining patterns.

7 Generalising name

We now turn our attention from spatial to naming patterns. At the beginning of this chapter, when we were re-engineering the bank full name and code attribute type signs, I

noted that we would use their object model to generalise the naming pattern. We do this now; we generalise naming above and away from geo-political areas and bank. This gives us a separate re-usable web of completely general naming patterns.

7.1 A general model for naming patterns

The bank and geo-political area naming models are the same shape. It seems likely that all naming models share the same shaped pattern. So there is an opportunity to generalise to a naming pattern. We take this now. We start with the general naming tuple and build up its connecting patterns. We derive from it the class of named objects—those objects that are connected by the named object/naming tuple class place. In theory, any object can be named; so, this named object class is a sub-class of the framework class, objects. We also derive the names class—those objects that are connected by a class place link to the name/naming tuples class. This names class is a sub-class of character strings.

This gives us the general model for the naming pattern shown in ***Figure 16.18***. As you can see, it is independent of both geo-political areas and banks. Furthermore, it has 15 objects of which only 10 are specific to names. This pattern is ubiquitous; its model will be re-used a large number of times in almost every re-engineering. Therefore, it makes sense to put its model into the general lexicon (which we discussed in ***Chapter 11***) by classifying its objects as framework level. As you can see, this has been done in ***Figure 16.18***.

Figure 16.18: Generalised naming tuples object schema

Undoubtedly, when the naming model is re-used, there will be opportunities to enhance it. For example, currency symbols, such as '£' and '$', which are not currently catered for, might be found as attributes of the currency entities. These would be generalised into the naming pattern. They would probably be generalised to a new sub-class of the naming tuples class—the symbol naming tuples classes. In this way, the model naturally grows as we expand our explicit understanding of the naming pattern.

7.2 Compacting with the general naming model

This general naming model leads to substantial compacting. Most of the systems that I have worked on have had well over 100 entity types (files). Almost all of these had

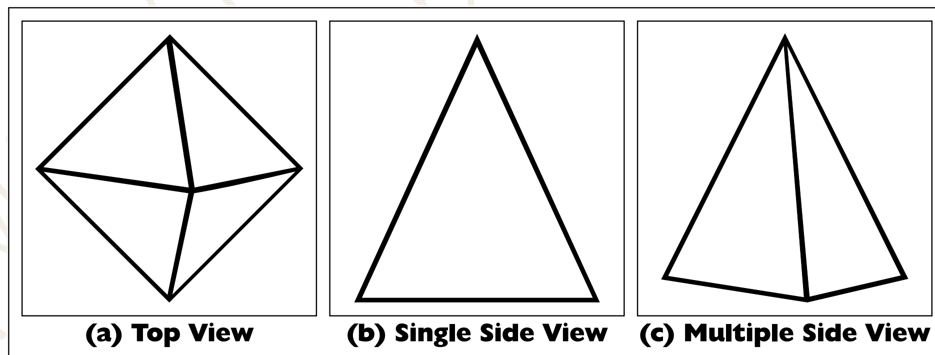
name attribute type;, many have had two (a full name and code), some have had more. So there would usually be well over 200 name attribute types in a system. We re-engineered these by transforming their entity types into classes that were sub-classes of the named objects class. Because they then inherited the general naming pattern, no other re-engineering was necessary. As well as simplifying the re-engineering process, this compacts the model. The original 200 name attribute types are compacted into the naming pattern's 10 objects. Furthermore, adding additional name attribute types would not increase the size of the model.

8 An aspect of a pattern

The address format was originally designed for paper and ink technology; it fits naturally onto the cover of an envelope. It also transfers effortlessly into the entity paradigm's framework. The re-engineering of address gives us a good insight into how much some patterns have had to be distorted to fit onto paper and ink technology and into the entity paradigm. The patterns we end up with in the object model are very different from the patterns we start with in the existing system.

These radically changing patterns can seem odd. I find it useful to think of them in terms of this analogy. Consider a three-dimensional pyramid, but assume that we can only see it from one aspect at a time. If we look from the top—***Figure 16.19*** (a)—it has a diamond outline. If we look at if from the ground, directly facing one side—Figure 16.19 (b)—we see a triangle. If we move so that we are not directly facing a side – Figure 16.19 (c)—we see an odd shaped four-sided figure. Because we can only see the pyramid in two dimensions, we only see one aspect of it at one time.

Figure 16.19: Aspects of a three-dimensional pyramid



Assume now we were to try and build a model of the pyramid. We would not build three models, one for each of the aspects; we would build one model of the complete pyramid. We could re-create the aspects we looked at earlier by taking a particular view of our three-dimensional model. The radical differences between the entity patterns and the object patterns are like the radical differences between any one of the two-dimensional views and the three-dimensional model. In the same way that only looking at the pyramid in two-dimensions views gives us an aspect, so the two-dimensional constraints of the entity paradigm produce partial views of business objects.

9 Summary

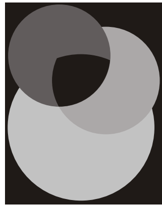
This re-engineering of address has completed the basic object model for spatial patterns; it is ready for re-use in future re-engineering projects. More importantly, this and previous chapters have given us a feel for how the systematic re-engineering process works. We now know the standard steps in the process and are familiar with a number of common re-engineering patterns.

As discussed in the previous section, address has also given us a useful example of how using the entity paradigm (the paradigm behind address) can distort the overall shape of business objects. We have seen how fitting address into the entity paradigm's constraints fixes one particular view. We have also seen how re-engineering frees it from that constraint, giving us a full rounded view of the business objects. Furthermore, we have seen how general objects that enable substantial re-use, such as the naming objects, are constructed.

In the next chapter, we re-engineer some of the entity paradigm's temporal patterns into an object model for time. Like space, time is fundamental to most business paradigms and so the model is very re-usable.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 17

Re-Engineering Time

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction
2 Re-engineering an existing system's bank holiday entity format
3 Re-engineering day
4 Re-engineering bank holiday
5 Re-engineering an existing system's weekend entity formats
6 Re-engineering our conceptual patterns for bank holiday and weekend
7 The object model for temporal patterns
8 Summary

1 Introduction

In the last five chapters, we constructed an object model for spatial patterns by re-engineering low-level three entity formats. We now turn to the similar task of constructing a generalised object model for temporal patterns. We do this by re-engineering the two low-level entity formats; bank holiday and weekend (***Figure 17.1***). Their re-engineering reveals temporal patterns that are very general and will be found in most systems. Now, that we are familiar with the steps in the re-engineering process, we focus on what is being re-engineered.

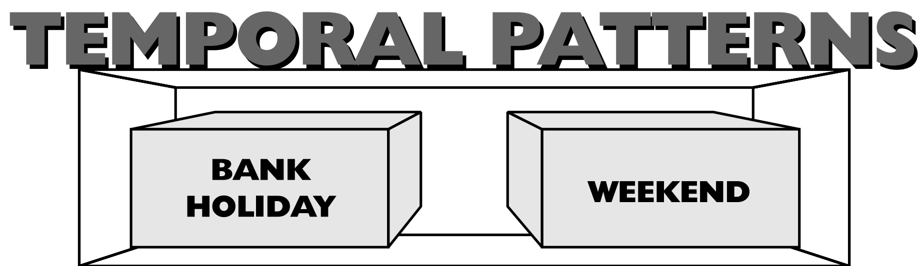


Figure 17.1: Two examples of temporal patterns

The spatial patterns that we re-engineered in the previous examples were reasonably intuitive. The analysis can be seen as, in some ways, a clarification of our common-sense views of space. The re-engineering of time is different. The object paradigm offers such a radically different view of time and temporal patterns that our intuitions cannot give us any sensible guidance (as we saw when we examined its semantics in ***Chapter 8***). Even though the temporal model we re-engineer in this chapter provides a good explanation of our everyday uses of temporal terms, the patterns themselves initially seem odd. This is a good sign; it is to be expected from a radical re-engineering.

2 Re-engineering an existing system's bank holiday entity format

We start by re-engineering the bank holiday entity format. As usual we start by looking at some of the existing entities. ***Table 17.1*** gives us a Partial Bank Holiday Listing.

Country Code	Date
UK	09-Apr-93
UK	03-May-93
US	18-Jan-93
US	16-Feb-93
US	31-May-93

Table 17.1: Partial bank holiday listing

From this, we can deduce the entity format shown in ***Table 17.2***.

Entity Type	Attribute Type #1	Attribute Type #2
Bank holidays	Country code	Date

Table 17.2: Bank holidays entity format

2.1 Bank holiday as a relational entity

Bank holiday is a relational entity type. It is derived from the entity types, country and day. It is not a pseudo entity type, such as the employee–project we discussed in ***Chapter 3*** (see ***Figure 3.24***). Those were a workaround solution to the problem of handling many-to-many relational attribute types in the entity paradigm. Unlike them, bank holiday is real. We will confirm this when we see the object that it is transformed into. But, like the pseudo entity types, bank holiday's two attribute types link it to the entity types from which it is derived (illustrated in ***Figure 17.2***).

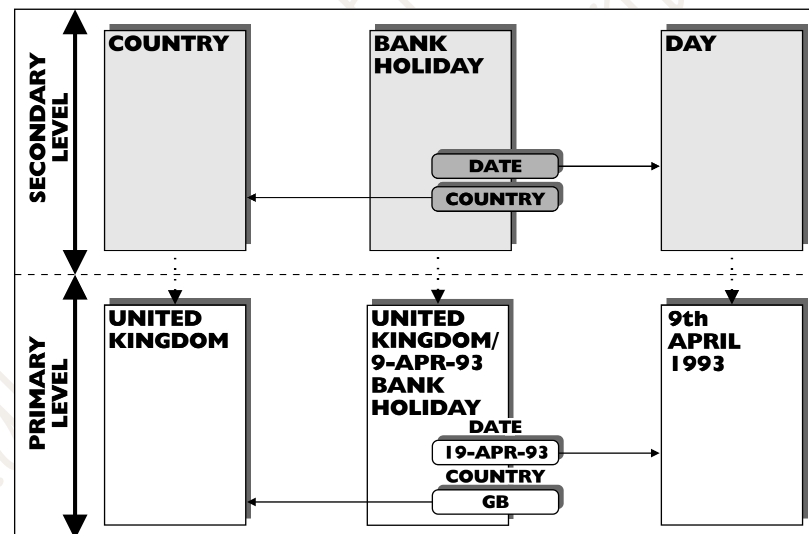


Figure 17.2: Bank holiday relational entity

When re-engineering a real or pseudo relational entity type, we follow a similar rule to re-engineering a relational attribute. We re-engineer the related entity types before we re-engineer the relational entity type. So here we re-engineer country and day before we re-engineer bank holiday. Country has already been re-engineered in a previous example; so, we only need to re-engineer day.

3 Re-engineering day

Most computer systems do not have a day file (corresponding to the entity type shown in ***Figure 17.2***) with a record (entity) for each day. In other words, the day (or date) entity is not usually found explicitly implemented in most computer systems. However, it

is there, but implemented implicitly. The individual day entities are deduced from the day code attributes of other entities. It can be thought of as an entity implemented as a process (as we discussed in ***Chapter 2***).

We need to re-engineer this implicit entity. The easiest way to do this is to construct an explicit entity format for it and subject this to the standard re-engineering procedure. Once the format is constructed, we can continue as usual. We familiarise ourselves with the entity format by looking at some examples of what we are going to re-engineer. Table ***Table 17.3*** gives us a partial day listing.

Day Code
09-Apr-93
10-Apr-93
11-Apr-93
12-Apr-93
13-Apr-93
14-Apr-93

Table 17.3: Partial day listing

From this, we deduce the entity format shown in ***Table 17.4***.

Entity Type	Attribute Type #1
Day	Day Code

Table 17.4: Day entity format

3.1 Re-engineering the day entity type sign

In this example, we only re-engineer the day entity type sign. We do not need to re-engineer its associated day code attribute type sign. If we were to re-engineer it, then it would fall under the naming pattern re-engineered in the previous chapters. For the day entity type sign, we follow the second rule and re-engineer a couple of day entities before the day entity type.

We start with an individual day selected from ***Table 17.3***—9th April 1993. From an entity point of view, the sign refers to a period of time—a day. It starts just after midnight of the 8th April 1993 and continues until midnight 9th April 1993.

Our examination of the object semantics of temporal patterns, in ***Chapter 8***, tells us what object this entity is transformed into. It is a temporal stage of the whole of space-time. It is every part of space-time for the period that starts just after midnight of the 8th April 1993 and continues until midnight 9th April 1993. Because it persists through time, it is a physical body. A space-time map of the object is given in ***Figure***

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

17.3, an object schema in ***Figure 17.4***. Other days can be re-engineered into similar patterns.

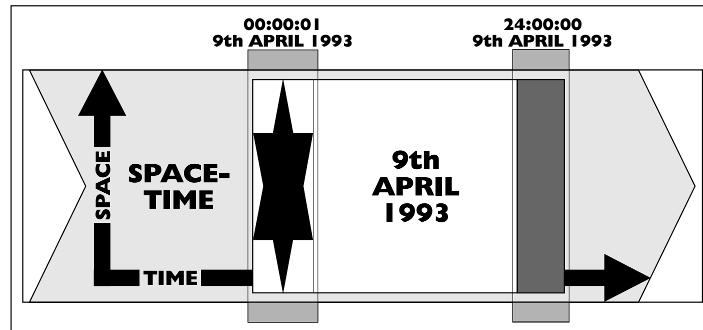


Figure 17.3: Space-time map for 9th April 1993

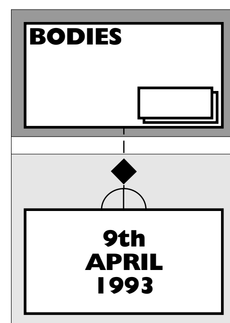


Figure 17.4: 9th April 1993 object schema

This is a very different way of seeing time, but we can translate the way we talk into it. We normally say 'today is the 9th April'. This can be understood as 'now is a temporal part of the four-dimensional object 9th April'. This is a mouthful and so not a practical alternative for everyday conversation.

3.1.1 Re-engineering the entity type sign

We follow the normal re-engineering pattern. The entity type sign is re-engineered into the class of objects that the entity signs were re-engineered into. This gives us the class days. The members of this days class divide the four-dimensional space-time 'sausage' into regular day long slices. This is illustrated in the space-time map in ***Figure 17.5*** and modelled in ***Figure 17.6***.

3.2 Re-engineering other time periods

We can extrapolate this pattern to give an object-oriented view of other time periods, such as months and years. These divide the space-time 'sausage' into larger month and year long temporal slices (shown in the space-time map in ***Figure 17.7***). The smaller slices are temporal parts of the larger slices. So, in the object paradigm, a day is part of a month and a month part of a year in the same way as my hand is a part of my arm.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

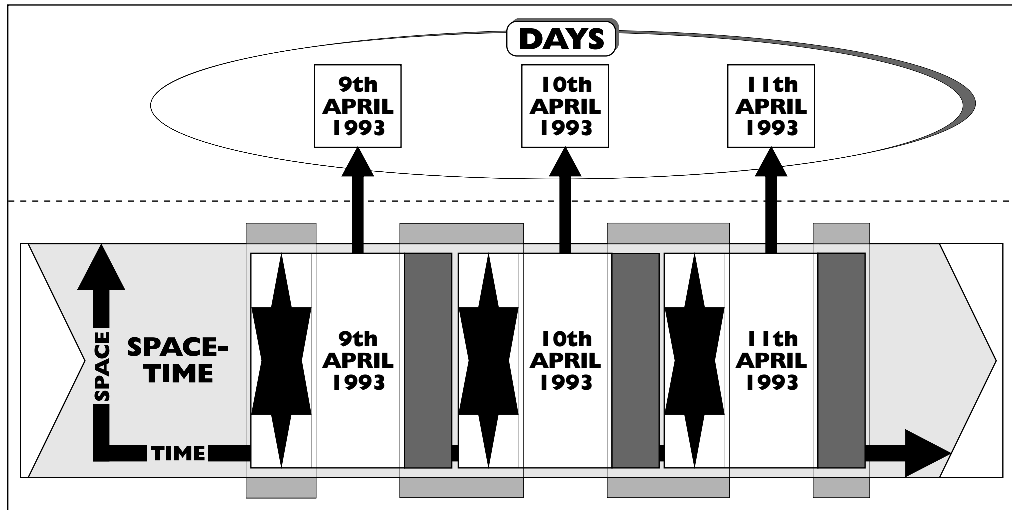


Figure 17.5: The days class

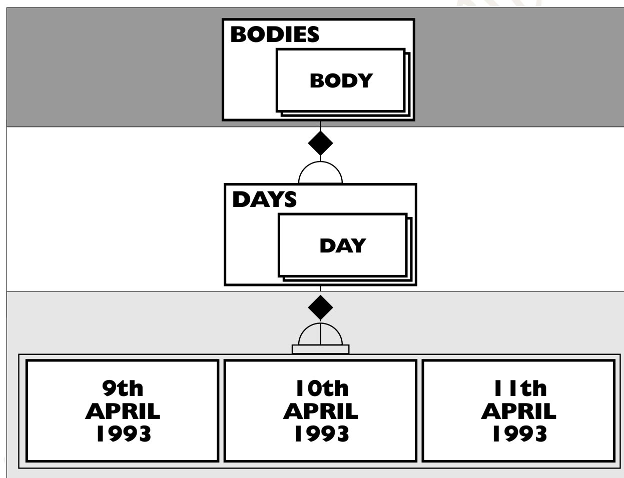


Figure 17.6: Days object schema

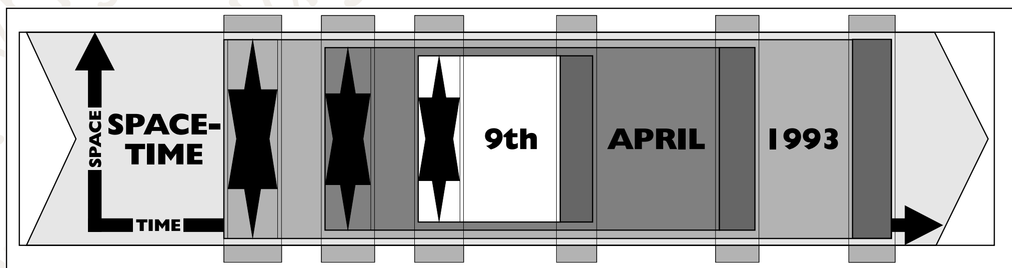


Figure 17.7: Year, month and day space-time Map

All three classes can be regarded as members of a calendar periods class (shown in the object schema in *Figure 17.8*). This gives us one of the groups of basic temporal patterns we need for our temporal object model. The remainder of the re-engineering will give us the rest.

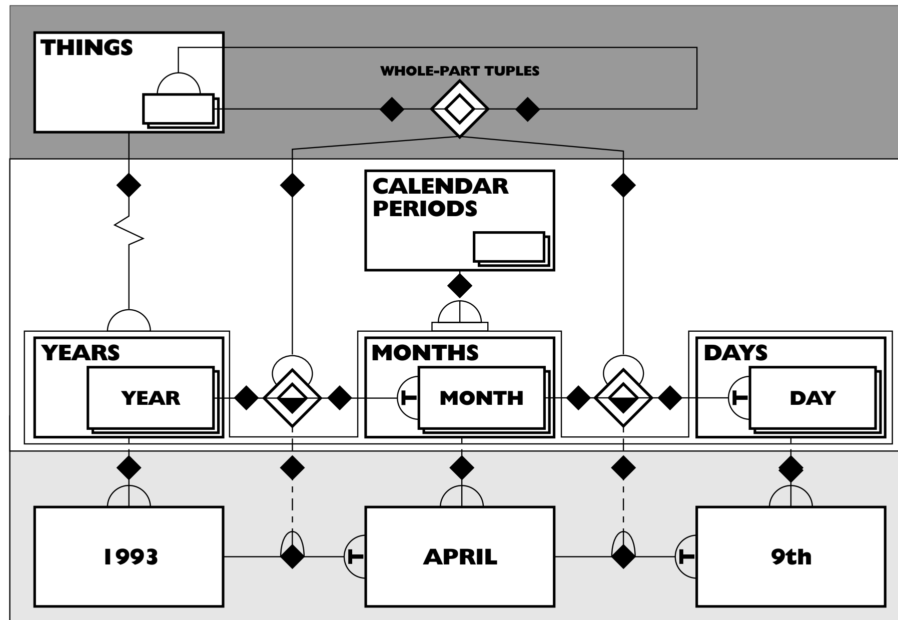


Figure 17.8: Calendar periods object schema

4 Re-engineering bank holiday

Now that we have re-engineered country and day, we can re-engineer the derived relational entity, bank holidays.

4.1 Re-engineering the bank holiday entity type sign

We follow our rules and start with an entity, an individual bank holiday. Let's start with the 9th April 1993 in the United Kingdom, the first entry in ***Table 17.1***. What does this refer to? From an entity point of view, it is a period of time—a day— that is a bank holiday in the United Kingdom.

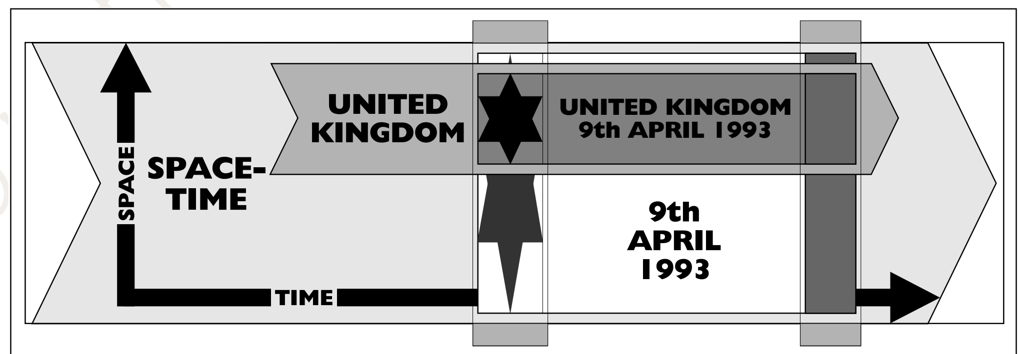


Figure 17.9: United Kingdom's 9th April bank holiday space-time Map

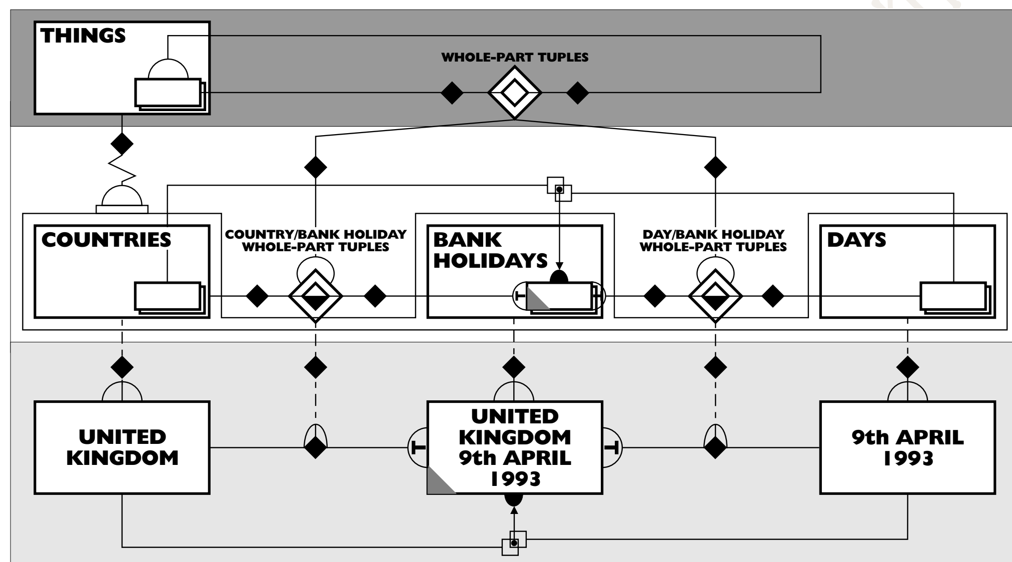
We know what this bank holiday's related objects, United Kingdom and 9th April 1993, are. These give us a clue to what the bank holiday object is. It is the intersection (over-

©Copyright Chris Partridge chris.partridge@BORO Centre

lap) of the two objects the United Kingdom and the 9th April—in other words, that bit of space-time that is part of both objects. This is illustrated in the space-time map in ***Figure 17.9***. This shows that the intersected bank holiday object is constructed from the other two objects.

We follow the standard pattern of re-engineering for the entity type sign. It re-engineers into the class of objects that its entity signs were re-engineered into; this is the bank holidays class. All the members of this class are logically dependent on the whole–part tuples that they have with members of the countries and days classes. This makes the members derived, but not the class itself. This is shown in the object schema in ***Figure 17.10***.

Figure 17.10: Bank holidays object schema



Notice that the bank holiday's connections to the intersecting objects are whole–part; they are structural connections between extensions. These define the individual bank holiday object and so we classify it as derived. The re-engineering of the other bank holidays follows the same pattern.

It may seem odd that a bank holiday is a physical object. But it gives a good explanation of the way we talk. If we are travelling through a country on a bank holiday, we say, 'it is a bank holiday here'. In object-oriented English, this translates into 'my here object is part of the bank holiday object'. If we were to travel straight onto the next country, we would say, 'it is not a bank holiday here'. This translates as 'my here object is not part of a bank holiday object'.

4.2 State of the object model for temporal patterns

This completes the re-engineering of the bank holiday entity formats. It has given us a reasonable foundation for the object model of temporal patterns. The calendar periods are useful general patterns that can be re-used in most re-engineerings. The bank holiday pattern is lower level, but still reasonably useful. It also links us in to the spatial pat-

terns' object model by way of countries. Later on in this chapter, we will see how we can generalise this link up to geo-political areas.

5 Re-engineering an existing system's weekend entity formats

We now turn to the re-engineering of the second example—the weekend entity format. We start, as usual, by looking at the existing entities; a partial list is given in ***Table 17.5***.

Country	Indicators						
Code	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
UK	No	No	No	No	No	Yes	Yes
US	No	No	No	No	No	Yes	Yes

Table 17.5: Partial weekend listing

From this, we get the entity format in ***Table 17.6***.

Entity Type	Weekend
Attribute Type #1	Country code
Attribute Type #2	Monday indicator
Attribute Type #3	Tuesday indicator
Attribute Type #4	Wednesday indicator
Attribute Type #5	Thursday indicator
Attribute Type #6	Friday indicator
Attribute Type #7	Saturday indicator
Attribute Type #8	Sunday indicator

Table 17.6: Weekend entity format

5.1 Re-engineering the weekend entity type sign

We follow the rules and pick an individual weekend entity sign to re-engineer—the United Kingdom (coded as UK) from the partial listing in ***Table 17.5***. This indicates that the United Kingdom's weekend is Saturday and Sunday. Even though Saturday and Sunday appear as individual entities in this system, they are entity types. Their entities are individual Saturdays and Sundays—such as the 6th and 7th May 1995. This means that the United Kingdom entry we selected from ***Table 17.5*** is also an entity type with entities.

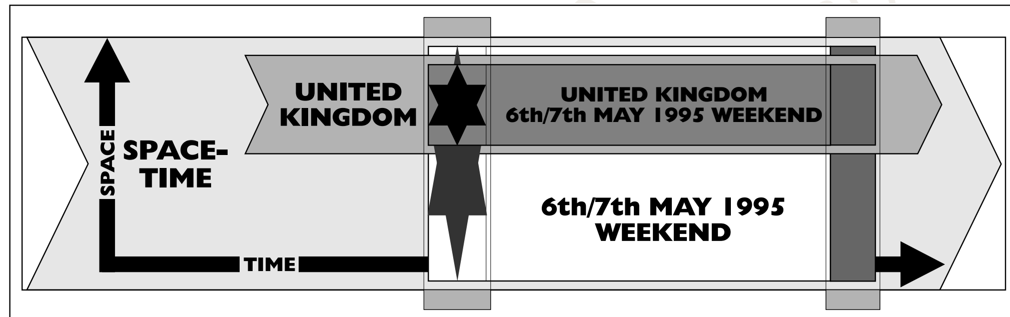
©Copyright chris.partridge@BORO Centre.com

A partial listing of these entities—particular United Kingdom weekends—is given in ***Table 17.7***. We start re-engineering at this level and work our way up to the selected entity sign. We select an example weekend—the 6th/7th May 1995—to re-engineer.

Weekends	Dates	Days
#101	6th/7th May 1995	Saturday/Sunday
#102	13th/14th May 1995	Saturday/Sunday
#103	20th/21st May 1995	Saturday/Sunday

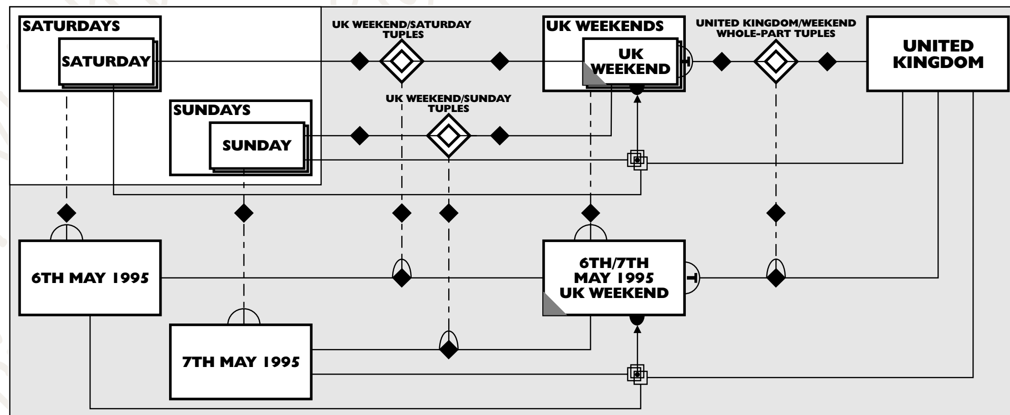
Table 17.7: Particular United Kingdom weekends partial listing

Figure 17.11: United Kingdom's 6th/7th May weekend space-time map



The particular weekends shown in ***Table 17.7*** have a similar pattern to bank holidays. They are a particular period of two days in a particular country—the United Kingdom. They each contain a particular Saturday and a particular Sunday. This pattern of whole–part connections is shown in the space-time map in ***Figure 17.11***. All of the United Kingdom's weekends share the same pattern.

Figure 17.12: United Kingdom's weekends object schema

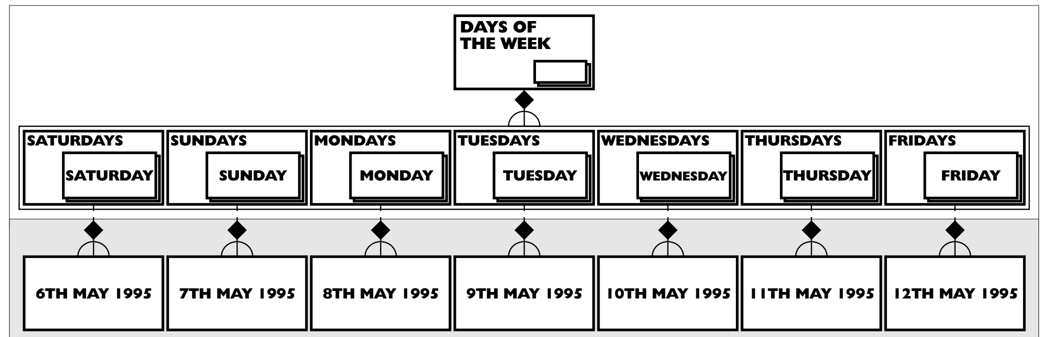


If we generalise the pattern up to class level, we get the object schema in ***Figure 17.12***. Notice how all the tuples have a structural extension element.

The particular Saturday—the 6th May 1995—belongs to the Saturdays class: the particular Sunday—the 7th May 1995—belongs to the Sundays class. We can generalise this

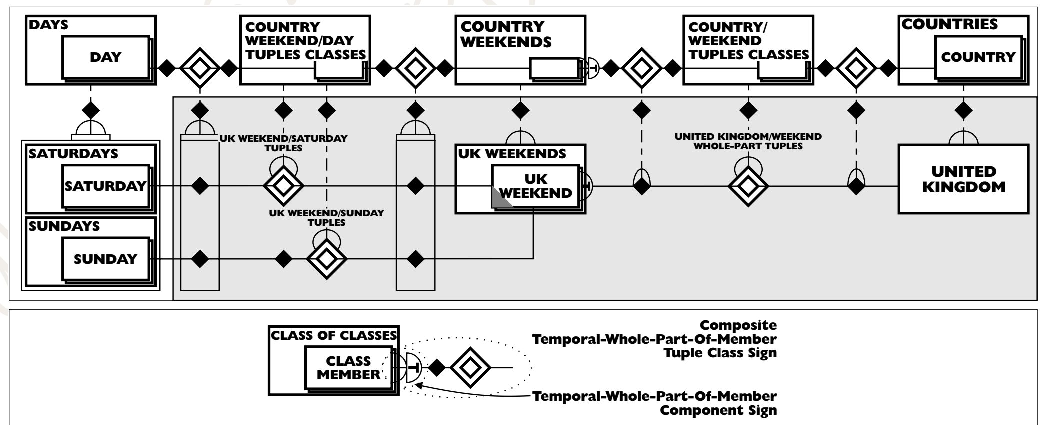
pattern across all the days of the week. This gives us the object schema in ***Figure 17.13***. This is a useful addition to the temporal object model—this general group of objects is common in systems. If we want to, we can also add the sign for a weeks class to the model, where each week is the fusion of seven consecutive days, starting with a Monday.

Figure 17.13:
Days of the week



We are now ready to re-engineer the entity type sign. It is transformed into a class of classes, the country's weekends class, which has, for instance, the U(nited) K(ingdom)'s weekends class as a member. The UK weekends class's tuples are generalised into tuples classes (shown in ***Figure 17.14***). This is similar to the naming tuples' generalisation into naming tuples classes shown in ***Figure 12.33***.

Figure 17.14:
Country weekends class object schema



The generalisation of UK weekends' temporal–whole–part tuples raises a notation issue. Temporal–whole–part tuples link individual objects—not classes. At the UK weekends level, this does not pose a problem because its members are at the individual object level. However, it does pose a notation problem at the level of country weekends, which is a class of classes. Here, there is no individual object to be a whole–part of. This is resolved by using a combination of existing components to construct a new composite sign, the temporal–whole–part-member-of sign (shown in ***Figure 17.14***).

5.2 The status of the object model for temporal patterns

This completes the re-engineering of the weekend entity format. It has added two important new groups of patterns to the object model—days and country weekends. These are, like calendar periods, useful general patterns that are found in many systems. Even though the country weekend pattern is lower level, and so less common, it is still useful. It also usefully illustrates how patterns are generalised up the class–member hierarchy as well as the super–sub-class hierarchy.

It is worth noting that a number of the temporal patterns captured in the model, such as particular UK weekends and days of the week, are traditionally implemented as date calculation processes. This is an example of the point first raised in ***Chapter 2*** ; that is, data and process in the information system are not a direct reflection of bodies and events in the real world. Like this case, individual bodies can be implemented as processes.

6 Re-engineering our conceptual patterns for bank holiday and weekend

So far we have constructed the temporal object model by re-engineering entity formats from existing systems. We now look at some of the more complex conceptual patterns that we can re-engineer to make the model substantially more accurate. We look at two patterns:

- Non-country/day holidays, and
- Time zones

All these patterns can be found in manuals listing information on holidays. For example, in the financial sector, SWIFT produces a manual for its customers whose lists contain these patterns (I have taken the following examples from an old copy).

6.1 Non-country/day holidays

Not all holidays fit into the simple country and day intersection pattern that we have just re-engineered. There are a number of non-country/day holiday patterns. We look at two of them:

- Non-country holidays, and
- Half-day holidays.

Many examples of both of these patterns can be found in the Financial Institution Holidays section of the SWIFT manual. We use some of them to generalise the model.

6.1.1 Non-country holidays

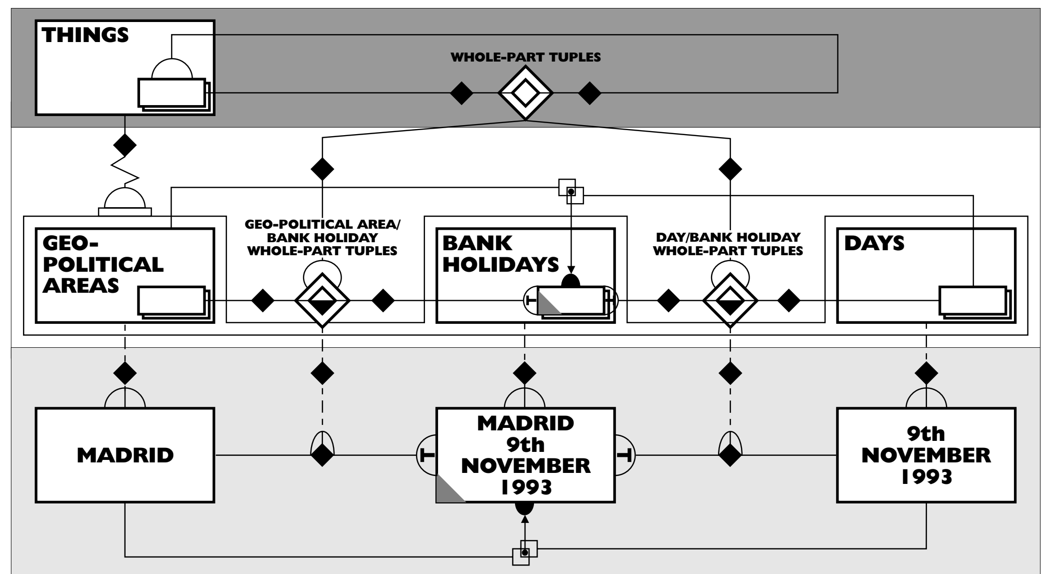
Some holidays apply to only a part of the country. For example, the 9th November 1993 was only a holiday in Madrid—not in the rest of Spain. This gives us an opportunity to

© Copyright Chris Partridge chris.partridge@BORO Centre.com

generalise the link between countries and bank holidays re-engineered earlier in this chapter (illustrated in ***Figure 17.10***).

Madrid is a city and so a geo-political area. We capture the pattern for its 9th November 1993 holiday by generalising the country/bank holiday whole–part tuples link from country to geo-political area. This gives us geo-political area/bank holiday whole–part tuples (shown in ***Figure 17.15***).

Figure 17.15: Geopolitical area bank holidays



6.1.2 *Half-day holidays*

The SWIFT manual has many examples of holidays that last for only part of a day. For example, Korea only has Saturday afternoon as a holiday; people work in the morning. Similarly, Italy, among others, has the afternoons of the 24th and 31st December 1993 as holidays. To re-engineer this, we need objects for mornings and afternoons as both calendar and weekly periods. I leave this as an exercise for you. The object model you produce should generalise bank holidays' connection to day up to a more general calendar period object.

6.2 Time zones

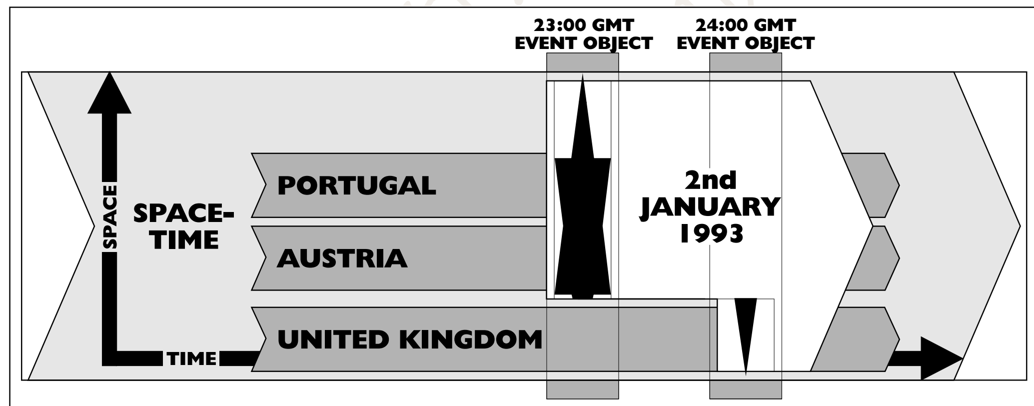
There is one important conceptual pattern we have not touched on yet and that is time zones. We all know that the world is divided into time zones. At a certain time of the year, when it is 2 pm in England, it is 9 am in New York. Different parts of the globe at the 'same' moment in time have different 'times'. Some big countries, such as the United States and Australia, have more than one time zone and so more than one time.

The SWIFT manual has a section that describes these called delta time zones. ***Table 17.8*** shows a selection of its information.

Country	Effective Dates	Gmt Differences
Austria	01 Jan 1993 to 27 Mar 1993 28 Mar 1993 to 25 Sep 1993 26 Sep 1993 to 31 Dec 1993	+1:00 +2:00 +1:00
Portugal	01 Jan 1993 to 27 Mar 1993 28 Mar 1993 to 25 Sep 1993 26 Sep 1993 to 31 Dec 1993	+1:00 +2:00 +1:00
United Kingdom	01 Jan 1993 to 27 Mar 1993 28 Mar 1993 to 23 Oct 1993 24 Oct 1993 to 31 Dec 1993	+0:00 +1:00 +0:00

Table 17.8: Selected time zone information

Figure 17.16: Local day object—a zigzag pattern space-time map



An object-oriented view of ***Table 17.8*** sees GMT and local times as objects. The GMT objects are temporal slices straight through the overall space-time object. The local times objects, however, make zigzag slices through space-time, 'overlapping' with different GMT time objects in different countries. For example, the local 2nd January 1993 day starts with the 23:00 GMT event object in Austria and Portugal, but starts with the 24:00 GMT event object in the United Kingdom. This particular zigzag pattern is illustrated in ***Figure 17.16***.

We do not necessarily need to change the model as a result of this analysis. If we are happy with a locally consistent model, then we can say the days class (shown in ***Figure 17.13***) has our local days as members. If, however, we want a globally consistent model, we need to make some changes. We must model both the standard GMT time objects and the various time zones' local times objects. This involves explicitly modelling time zones.

© Copyright chris partridge bridge centre.com

7 The object model for temporal patterns

We have now completed all the work on the temporal patterns object model. The re-engineering of the entity formats for bank holidays and weekends gave us a general model for some simple and basic temporal patterns—calendar periods and days of the week. The brief look at our conceptual patterns has made us aware that existing systems only really capture the simple patterns. There is a range of more sophisticated patterns that we need to model and generalise. However, the model, as it stands, is a good basis for going forward. It is a strong foundation that can be built on and enhanced as more re-engineering is done.

This example also illustrates an important aspect of re-engineering. It shows how the object paradigm's amalgamation of space and time to space-time reveals previously common but mysterious temporal patterns as structural whole–part patterns. Bank holidays are revealed as the intersection of countries and days objects. Country weekends are also revealed to be classes of temporal parts of countries. These examples show how the reference of temporal patterns becomes much clearer when it is revealed as four-dimensional extension. This also clarifies the sense of the patterns; many of the connections between objects are revealed as structural extension-based patterns.

8 Summary

This is the last of the examples. Working through them has achieved a lot. Together they have:

- Given us a feel for what re-engineering is like and the sort of models it produces,
- Shown us the details of a systematic approach to re-engineering the entity formats in the existing system,
- Given us a feel for what analysing object semantics means in practice, and
- Illustrated how the object syntax and notation work on business examples.

In addition they have provided us with three object models:

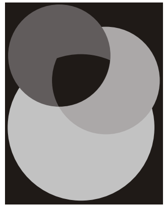
- Spatial patterns,
- Temporal patterns, and
- Naming patterns.

These contain fundamental and general objects that can be re-used in our re-engineerings.

This is not the final chapter; one more is left. In it, we briefly examine some of the challenges you will face when you set up a re-engineering project using this approach.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Chapter 18

Starting a Re-Engineering Project

1 Introduction
2 Take a re-engineering approach
3 Establish priorities for the construction of fruitful, general, and so re-usable patterns
4 Taking care to manage large projects in a generalisation-friendly way
5 Produce a validated understanding of the business
6 Object model the migration of business patterns
7 Summary

© Copyright: Chris Partridge
chris.partridge@BOROCentre.com

1Introduction

Parts One to Five gave you an understanding of what business objects are. The previous chapters of this part of the book (Part Six) showed you how to apply this understanding, describing a systematic method for re-engineering an entity oriented system into a business object model. Some of you will now be contemplating setting up a project to re-engineer your entity oriented systems. You want the project to be a success. This involves producing a good business object model and accurately embedding it in an implemented system.

Inevitably a re-engineering project works in a different way from traditional system building projects. Its success depends, to quite a large extent, on managing it in a way that recognises these differences. This chapter outlines five management tactics that I have found help to make re-engineering projects a success. These are:

- Taking a re-engineering approach,
- Establishing priorities for constructing fruitful, general and so re-usable patterns,
- Taking care to manage large projects in a generalisation-friendly way,
- Producing a validated understanding of the business, and
- Object modelling the ‘data’ translation/migration.

2 Take a re-engineering approach

It is important to approach the project as a re-engineering exercise. People sometimes succumb to the temptation of thinking that they can only construct a radically new system by ‘starting with a blank sheet of paper’. They want to ignore the existing system completely, so as not to taint the new system with its mistakes.

While this tactic may have its merits when working within a paradigm, it is not re-engineering. In fact, it is the wrong way of shifting to a new and better business paradigm. Apart from the actual difficulty of ‘blanking out’ all knowledge of the existing system’s patterns, this approach ignores the nature of re-engineering and evidence of how successful re-engineerings have worked.

2.1 Salvaging investment in business patterns

A core feature of re-engineering is that it salvages the business patterns embedded in the existing system. The great 17th century physicist Isaac Newton used a striking image for this. He commented in *Mathematical Principles* (describing his re-engineered physics) that his work was only possible because ‘he stood on the shoulders of giants’. A new paradigm may be radically different, but it normally takes full advantage of the investment in the patterns of the old paradigm. Building patterns from scratch takes a substantial investment of time and effort and is just not practical in most situations.

©Copyright Chris Partridge chris.partridge@BOROCentre.com

This is why a re-engineering approach actively seeks out and re-engineers the business patterns in the existing system, salvaging the investment made in them. We saw how this worked in the examples in previous chapters. The re-engineering of the existing system's entity formats yielded radically different, useful, general patterns that formed a foundation for the business object models.

From an economic point of view, the big benefit of the salvage approach is that it needs much less investment than most other approaches. A systematic re-engineering approach, such as that described in the previous chapters, salvages the existing system's investment in business patterns. The re-engineering project effectively starts with this investment in hand.

2.2 A well-defined scope

Basing the project on the re-engineering of the existing system's entity formats also provides a simple and effective way of scoping the project. From a management control point of view, it is important to have a reasonably clear idea of the boundaries of the project. The existing system's business patterns provide just that. We can define the scope in terms of the entity formats in the existing system that hold these patterns (things such as files and records). It is a simple matter for us to list these, giving the project a clear-cut boundary.

If we did not have a clear-cut boundary, there could be problems. Objects tend to be closely linked to a number of other objects. Their webby nature means that they have few, if any, natural boundaries. If we were to keep on analysing them until we found a natural boundary, we would end up constructing a model of everything.

You may want the project to include in its scope some business patterns that are not embedded in the existing system (in other words, new requirements). These can be associated with a related pattern that is embedded, and the patterns re-engineered together. Time zones (from the re-engineering of temporal patterns in ***Chapter 17***) could be considered an example. If they were within the scope, but not embedded in the existing system, then we could associate them with either the bank holiday or weekend patterns, which are. The two sets of patterns could then be re-engineered together.

However, it makes sense to try and schedule the development of specific new requirements after the re-engineering project. Re-engineering delivers significant amounts of extra functionality as well as a new way of seeing the business. When this takes shape, the original 'new' requirements may already be satisfied or have taken on a completely new meaning.

3 Establish priorities for the construction of fruitful, general, and so re-usable patterns

An important benefit of the object paradigm that it enables us to construct fruitful, general and so re-usable patterns. However, to get these patterns, we have to actively

exploit the paradigm. This means establishing priorities for the construction of re-usable patterns.

3.1 Establishing priorities for the construction of general, and so re-usable patterns

The worked examples in the previous chapters illustrated how vital generalising is to successful business object modelling. The more general a pattern, the more potentially re-usable it is and so more useful.

3.1.1 Generalisation produces a compact system

Generalisation's usefulness comes, in part, from its ability to compact. It enables a large number of base patterns to be re-engineered into a much smaller number of simpler more general patterns. In essence, it fits more information into a smaller space (illustrated graphically in ***Figure 18.1***).

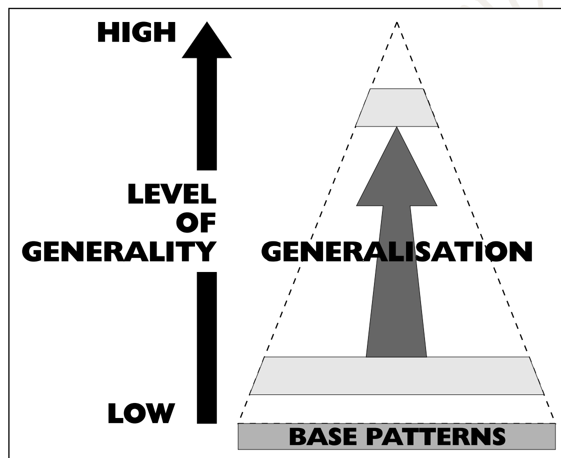


Figure 18.1: Generalisation fits more information into a smaller space

3.1.1.1 Generalisation leads to less costly components

In computing terms, generalisation's compacting means fewer, simpler components. Compacting works its way through the development life cycle. Compacting during business object modelling leads to fewer components in the business model, and this translates into fewer components at all the later stages of system building. There are fewer and simpler components to specify during systems analysis and design and so fewer and simpler components to code and test. This in turn means fewer and simpler components to maintain and fix. Furthermore, when people start to learn the system, there are fewer and simpler components to master. Overall, the effort required to build and maintain the system is reduced. This translates into a reduction in time and cost. So systems with generalised components are quicker and cheaper to both build and maintain.

© Copyright chrispartridge.com / BORO Centre.com

3.1.1.2 Generalisation means no inherent complexity

An important consequence of using generalisation as a core tool in system building is that we can no longer think of a pattern having an inherent complexity. We are used to thinking that a complex set of business patterns needs a computer system of equivalent complexity. With generalisation, this rule of thumb no longer works. The worked examples have shown us how, using a combination of re-engineering and generalisation, we can transform complex patterns into simpler, more powerful, general patterns. This may initially seem slightly counter-intuitive, but it is a natural way of working. It is, for example, the way in which science accumulates knowledge. If we look at its history, we see again and again a complex theory being superseded by a simpler, more powerful, theory.

Part of what is going on is that as we generalise the patterns, their scope increases. This is shown schematically in ***Figure 18.2***. Here, the patterns, #1 and #2, are generalised into pattern #A, which covers the same scope as #1 and #2.

The number of patterns needed for the full scope has halved from two to one. However, this is only part of the story. In actual re-engineerings, the generalised pattern often turns out to be simpler. We saw this in the model for spatial patterns. After re-engineering the first group of entity formats, the final model not only has fewer objects (at the application level), but is also simpler. The power of the spatial model was revealed in re-engineering of bank address in ***Chapter 16***, where no new business objects were needed.

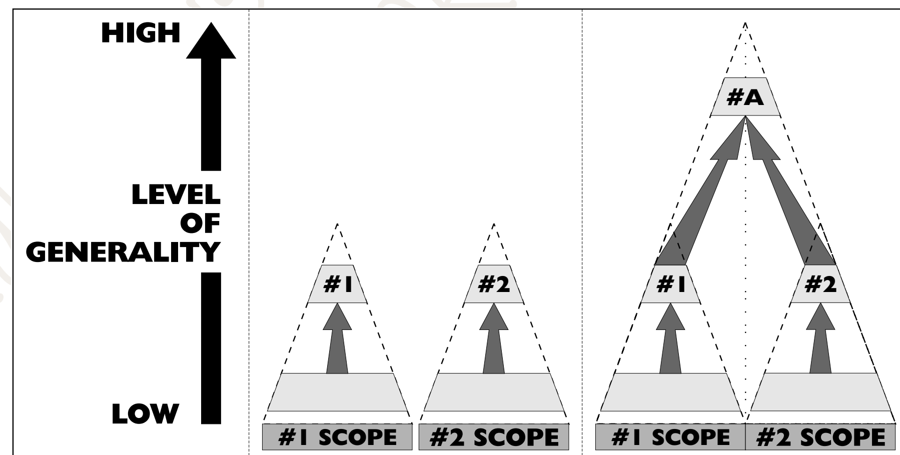


Figure 18.2: Generalisation creates patterns with increased scope

3.1.1.3 The impact on estimating

This lack of inherent complexity has a serious impact on the way in which we estimate re-engineering projects. In traditional estimating, a sensible rule of thumb is that there is a reasonable correlation between the complexity of the requirements and the effort (and so cost) of building the system. This assumes that complexity is unaffected by the system building process. The complex pattern that goes into the process inevitably leads to

© Copyright chris.partridge@BORO Centre.com

complex code. This made estimating relatively easy. The resources needed to build a system could be calculated from the complexity of its requirements.

With generalisation, this is no longer true. The business modelling process can significantly reduce the complexity of a pattern. Two patterns, one complex and the other simple, may both be re-engineered into the same simple pattern. This happened in the worked examples with the simple country pattern (in ***Chapters 12***, ***13*** and ***14***) and the more complex address pattern (in ***Chapter 16***). We re-engineered both entity formats into the same general pattern – which will end up as the same computer code. The complexity of the requirements no longer correlates well with the resources needed to build the system.

It might seem that the correlation still applies within the business object modelling stage. It is certainly true that a complex pattern can take longer to model than a simpler one, other things being equal. However, other things are not often equal. For example, the re-engineering of the simple country pattern took much longer than the re-engineering of the more complex address pattern, because we could match the address patterns with the re-engineered country patterns. The estimation of effort can no longer be based simply on the complexity of a requirement.

With experience, one acquires a rough feel for how much compacting to expect from a group of patterns, and can translate this into a rough estimate. I suspect it will be some years before there is enough hard data to work out a formula. This makes accurate estimating difficult. The bright side is that as generalisation compacts and simplifies, so building a system from a generalised business object model takes less time and effort than building it from an ungeneralised model.

3.1.2 A generalisation friendly environment

Generalisation brings benefits, but how do we encourage generalisation? People naturally and unconsciously generalise patterns. However, without a framework to help them, this instinctive tendency does not normally lead to very general patterns. The object paradigm remedies this by providing a generalisation friendly environment, within which a systematic approach can encourage more general patterns.

Traditional methods of system building do not have access to the compacting power of generalisation. They do not provide an environment that is conducive to producing general objects and so have no reason to make generalisation a priority. A good, and widespread, example of this is common subroutines. Analysts and programmers naturally recognise their potential. They naturally construct reasonably general subroutines during system building. However, traditional approaches to modelling processes, such as functional decomposition, hinder rather than help this natural process. Analysts have to rely on their instinct and initiative and ignore the approach.

One particular experience of this sticks clearly in my mind. Many years ago one of the projects I was managing was a large development, using the popular SSADM method. Towards the end of the analysis stage, a lead analyst told me he was going to start identifying common subroutines. After some discussion, it became clear that he was in uncharted (though familiar) territory. This task was not identified by the method; so, he

had not put it in his plan. Yet, he realised it needed to be done. The method had no systematic techniques to help him, so he had to rely on his intuition.

When he discovered common subroutines, he found that they could not be described either in the method's (data flow diagram) functional decomposition structure or in the CASE tool he was using. The root of the problem was that neither of them could model the way two processes use the same common subroutine—what could be called re-composition. They could handle de-composition's tree structure, but not re-composition's lattice structure. Given that common subroutines depend on re-composition, this meant functional decomposition actually hindered this type of generalisation.

By contrast, a re-engineering project not only supports generalisation, but also has a systematic approach that actively encourages it. In this environment, it makes sense for project managers to make generalisation one of the top priorities. However, old habits die hard. Modellers used to a traditional environment do not naturally push generalisation as far as it should go. Project leaders can help ensure successful generalisation by actively checking how much of it is going on and persuading modellers to do more when it is needed.

3.1.3 *Introducing generalisation during business modelling*

A general theme running through many approaches to building computer systems is that the earlier in the system development life-cycle we introduce a good technique, the greater the benefits. This not only feels intuitively correct for generalisation, it is correct—the best time to generalise is during business modelling. However, as the tale of the lead analyst identifying common subroutines above illustrates, system builders often generalise later in the life-cycle.

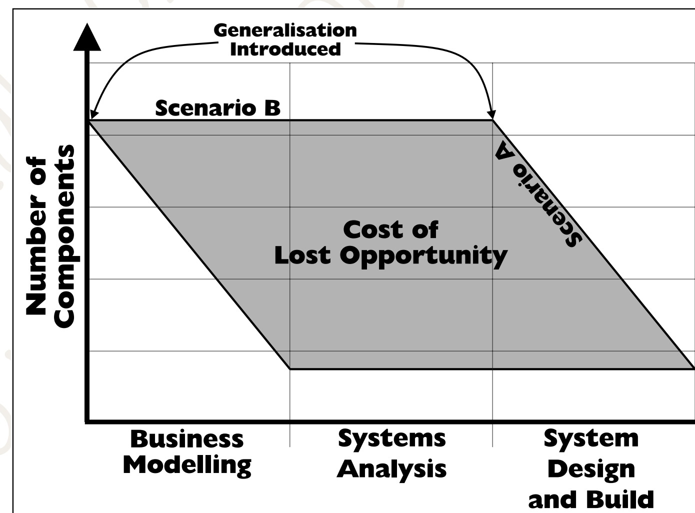


Figure 18.3: Introducing generalisation at different life-cycle stages

© Copyright Chris Partridge chris@BORO Centre.com

3.1.3.1 Economically sensible to generalise business patterns early

It is reasonably obvious that the earlier generalisation is introduced into the development life-cycle, the greater the reduction in overall effort. We can visualise this by thinking in terms of the reduction in the number of system components. ***Figure 18.3*** shows a simplified schema of this.

If generalisation is introduced at the business modelling stage (shown as scenario A in the schema), the benefit of compacting is delivered from the beginning of the life-cycle. The compacted business components are used in system analysis and on into system design and build.

If, however, generalisation is introduced at the system design and build stage (shown as scenario B in the schema), then the reduction in components only occurs then. (For simplicity's sake, it is assumed that the final reduction in scenario B is equivalent to scenario A.) The reduction in costs associated with compacting only starts appearing then; it does not happen during business modelling or systems analysis. The cost of this lost opportunity is shown in the schema by the shading between scenarios A and B. The earlier we compact the components, the greater the reduction in costs. The later we compact the components, the greater the cost of the lost opportunity.

3.1.3.2 The natural stage to generalise business patterns

As well as the 'economic' cost reasons for generalising business patterns during business modelling, there is also a sound practical reason for generalising them at this stage—it is the natural stage to do it. We use similarity to generalise business patterns. Finding this similarity is more naturally done at the business modelling stage, while we are looking at the objects in the business (illustrated in ***Figure 18.4***). In the later stages, when we turn our attention away from the business and towards the system, the business patterns are not so visible.

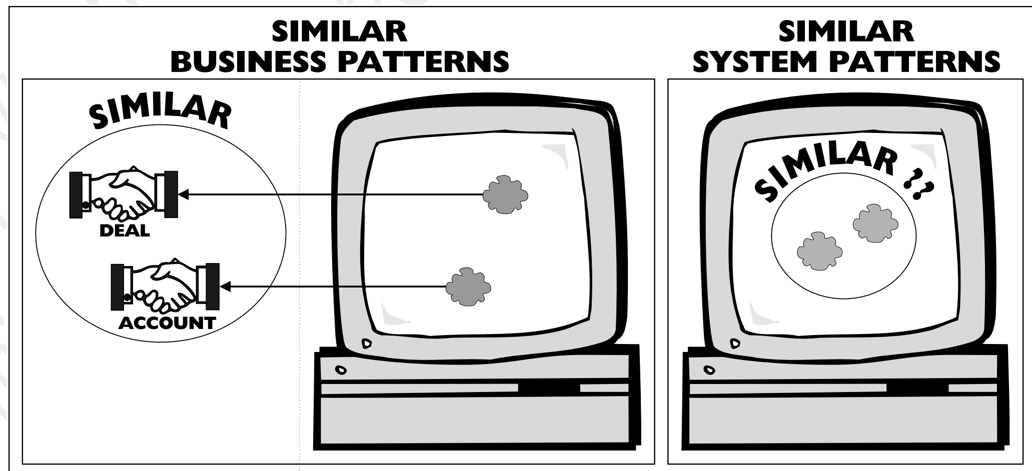


Figure 18.4: Similar business patterns

© Copyright chris partridge chris@BORO Centre.com

3.2 Establishing priorities for the construction of fruitful patterns

Generalisation and fruitfulness can be seen as two sides of the same coin. A fruitful pattern can either be sufficiently general to be re-used frequently or sufficiently similar to many other patterns to be generalised into a common pattern. While we can systematically generalise, fruitfulness is more elusive. It is a kind of potential for generalisation—an ability to deal with future patterns that have not been modelled as yet.

3.2.1 Fruitfulness during and beyond a project

This fruitfulness shows itself in two ways—within the scope of the project and outside it. While a project is in progress, the team can see how a fruitful pattern leads to high levels of generalisation. Its potential fruitfulness becomes actual before their eyes as the re-engineering reveals generalisations. But does this potential only extend as far as the agreed scope? If the scope were widened, would the pattern's fruitfulness suddenly dry up?

Our experience is that it does not. Where a pattern has been fruitful within a re-engineering project, its fruitfulness always seems to extend well beyond the scope of the project. One way this reveals itself was mentioned in the ***Preface***. There, I described my experience of users finding that their re-engineered system could handle situations not in the scope, including situations they did not even envisage when the system was developed.

The naming and spatio-temporal patterns in the previous chapters provide another example. They were fruitful outside their initial scope. For example, the fruitfulness of the naming and the nesting geo-political area patterns was clearly shown in the re-engineering of address. They effectively matched all the address patterns. In my experience, the naming and spatio-temporal patterns are also fruitful outside the scope of the worked examples. I have found their patterns re-appearing in many re-engineerings. (You might remember it re-appeared in the bank holiday example in the last chapter.) This often means that future business requirements are either already catered for by the system or can be relatively easily dealt with.

3.2.2 Building fruitful patterns from complex entity formats

Seeking out fruitful business patterns is a sensible goal for a re-engineering. However, taking this as a goal, overturns a rule of thumb in traditional system building. We touched on this point when capturing the conceptual patterns for country in ***Chapter 14***. There we noted a tendency to favour dropping complex patterns when setting the scope of a project,. In a traditional environment, this makes sense because they take more of an effort to build. In an object-oriented environment, it does not. It is not that complex patterns no longer necessarily take more resources to build. It is that complex patterns are more likely to have fruitful patterns embedded in them. These fruitful patterns are the Holy Grails of business modellers; the more that can be found the better.

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

3.2.3 More accurate patterns are more fruitful

There is another way to increase the fruitfulness of the business patterns, and that is to make them more accurate. In ***Chapter 1***, we looked at how increased physical accuracy was essential to the introduction of interchangeable parts in manufacturing. We observed that a similar revolution in accuracy—this time, accurately reflecting the world—was necessary for the introduction of interchangeable parts in business modelling.

In later chapters, we saw how this accuracy has increased as information paradigms evolved. We saw, for example, that modern literate western culture has more accurate notions of sameness and signs than the Huichol Indians, who see corn and deer as the same (we discussed this in ***Chapter 4***. We discussed how western culture is now developing an understanding of the logical paradigm's more accurate distinction between the whole–part, super–sub-class and class–member patterns. And how it has started to absorb the object paradigm's notion of sameness for four-dimensional objects. It is in the process of providing an accurate explanation of how something now is the same as it was yesterday; it no longer has to be both the same and different, much like the Huichol's corn and deer.

When we business object model, it is important that we take advantage of the object paradigm's ability to produce more accurate patterns by using them to construct more general and reusable patterns. Just as physical accuracy enabled interchangeable reusable parts, so referential accuracy encourages the information paradigm's counterpart—generalisation and re-use. Increased accuracy reveals the patterns more explicitly, taking the guesswork out of whether patterns are similar or not. The general patterns constructed from more accurate lower level patterns inherit their accuracy and so are able to operate at higher levels of generality.

So it makes sense for the manager of a re-engineering project to try and determine whether his modellers are being referentially accurate. And if they are not, to take remedial action. This should help to ensure the fruitfulness of the patterns.

3.2.3.1 Object model's lower granularity

Increased referential accuracy also leads to a lower granularity in the descriptions of patterns—which initially means more objects. If you look at the early version of the spatial model in ***Chapters 13*** and ***14***, you can see that the re-engineering increased the number of operational items. From this, it might appear that we have to weigh the benefits of increased accuracy against the cost of handling a larger model. But, it turns out we do not. The increase in accuracy leads to a corresponding increase in generalisation, which reduces the number of application objects significantly.

In the region example in ***Chapter 14***, the generalised patterns made almost all the patterns from the re-engineering of the country example redundant; this was only the second file re-engineered. In the first stage of the address example in ***Chapter 15***, no new application objects were re-engineered. It is these application objects that the system builders construct. Once the re-engineering gets beyond a few entity formats, at the application level, the compacting effects of generalising more accurate patterns more than outweighs the expanding effects of their lower granularity.

3.2.3.2 Learning to see accurately

Learning to see with the referential accuracy demanded by the object paradigm is one of the most challenging aspects of business object modelling. It is important for the success of the project that the people undertaking the business modelling have mastered the challenge and learnt to see in this new way. It also helps if the people carrying out the systems analysis have at least a broad understanding of business objects.

Managers should ensure that the people working on their project have the right level of understanding. For the people that need training, this book is one way of providing a useful grounding in business object modelling. It can be usefully supplemented by formal training courses. However, once people have mastered the foundations, there is no real substitute for experience on a real project. At this stage, the easiest way to progress is by working with expert practitioners, learning by example.

This generally means that IT departments starting out on a project have to either buy in or grow experts. If the decision is to grow, then it is only sensible for inexperienced people to cut their teeth on a trial project in a non-critical area of the business. Their learning can be speeded up considerably if the team is beefed up with an expert mentor.

4 Taking care to manage large projects in a generalisation-friendly way

To take full advantage of the benefits generalisation brings, we need to manage it. In large projects, it is particularly easy to stifle the potential for high levels of generalisation. To create a stable generalisation-friendly environment, we need to ensure the careful management of the balance between the increased opportunity for generalisation that comes with widening the scope and the increased risks associated with large projects.

4.1 Widening the scope increases the opportunities for generalisation

Each new business pattern added to the scope of a re-engineering project brings an opportunity for generalisation. We saw an example of this in ***Chapter 15's*** re-engineering of region. Extending the scope of the re-engineering from country to region enabled us to generalise both patterns to geo-political area. Though it may seem counter-intuitive, widening the scope led to a smaller, more general and powerful model rather than a bigger one.

The more conceptually powerful the model, the more pronounced this effect. The model does not have to grow in size or complexity as we add patterns, we can generalise and make it smaller and simpler instead. When we introduce new patterns to a generalised model, the likelihood of us being able to generalise them is higher. This is because the model's general patterns are more likely to match with the new patterns.

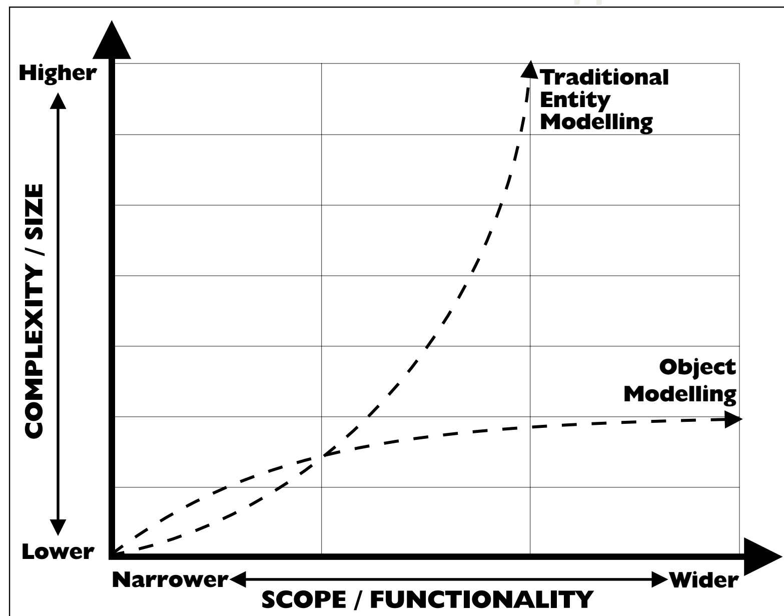
This also means that the more general the model, the smaller the cost of re-engineering each new pattern is likely to be. We saw an example of this in address's re-engineering

© Copyright Chris Partridge chris.partridge@BORO Centre.com

in ***Chapter 16***. The spatial model was sufficiently general that the new business patterns introduced in the address entity formats all matched with existing patterns. There were no new patterns; so, the cost of building new computer code for the address business patterns would be nil!

This is the complete opposite of what happens in traditional system building, where generalisation is not properly supported. There we have to harmonise each additional pattern with the existing patterns, adding to the complexity of the system. As the number of patterns in the system increases, the task of harmonisation gets more onerous. The traditional rule of thumb is that the more patterns there are, the greater the cost of handling each new pattern. The difference between traditional system building and system building using business object modelling is shown graphically in ***Figure 18.5*** (this is a reproduction of ***Figure P.5***.)

Figure 18.5: Correlation between scope and complexity



4.2 Balancing the economies of scope against the problems of size

This would seem to imply, at least in theory, that it is better to have as wide a scope as possible in a re-engineering project, because this will keep the costs of building the system down. To an extent this is correct, because the wider the scope the greater the opportunity for generalisation. But as the scope increases, so does the size of the project. And a large project brings increased risks.

The scope of re-engineering projects is defined in terms of the entity formats of an existing system. If the system is small, it has few entity formats and so can be re-engineered in one go. Doing it piecemeal would just take longer and demand more effort. When a large system is re-engineered the situation changes. The scope includes many more entity formats. Re-engineering them all in a single project usually takes either many years, a large team or both.

However, the longer a project lasts and the larger the number of people involved, the more difficult it is to manage. If it gets very large it is much more likely to end up out of control. Furthermore, you cannot guarantee that, after all the man years of effort, the system will actually work. So, when re-engineering large systems, we need to balance the benefits of a wide scope against the inherent risks in a large project.

4.3 Chunking the existing system

In practice, people tend to redevelop large systems bit by bit. It is like the child's riddle—'How do you eat an elephant?' The answer is 'in bite-sized chunks'. By dividing the system into manageable (digestible) chunks, we can keep things under control. As we implement chunks at regular intervals, we are giving tangible evidence of progress. Management can see the results of their investment reasonably soon after they make it—instead of waiting until the end of the overall project.

One problem has to be overcome in this approach. A system, by its very nature, is an interconnected coherent whole. When we redevelop a chunk, we have to fit it in with the existing system, if it and the system are to work together. However, if we design the new chunk to work with the old system, it will probably inherit some of the structure of the old system.

The standard tactic in traditional system re-development for dealing with this problem is to design the new chunk to work unencumbered by the existing system. Then, to get the two to work together, a temporary interface is built that handles the connections between the old and the new. As each new chunk is redeveloped, it permanently links up with the other new chunks and connects to the old system through a new temporary interface. When all the chunks are redeveloped there is no longer a need for a temporary interface. This is shown schematically in ***Figure 18.6***.

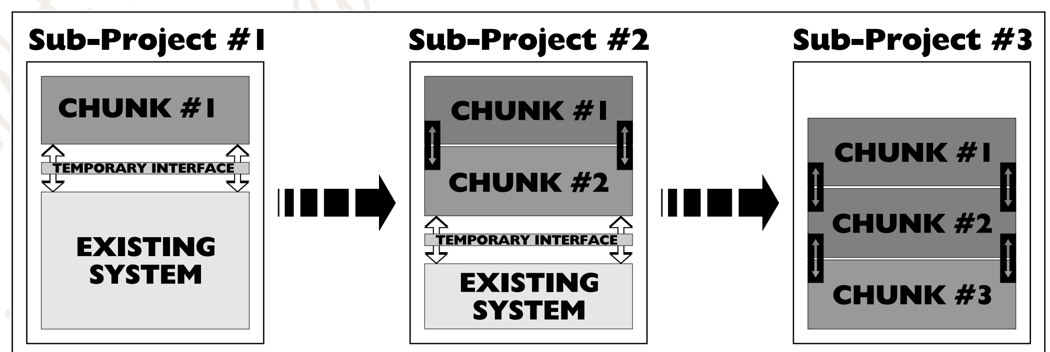


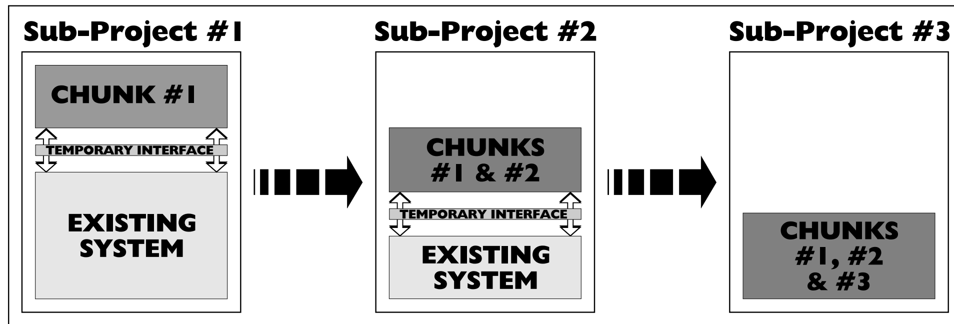
Figure 18.6: Eating the problem in bite-size chunks

4.4 The object re-engineering approach to chunking

When I have been involved in the re-engineering of large systems, the team has followed the standard practice of chunking. We divided the overall project into a number of subprojects, each dealing with a chunk of the existing system. But we did not re-engineer each chunk in isolation. If we had, this would have restricted the scope of each re-engineering to its chunk, losing the potential for generalising patterns across chunks.

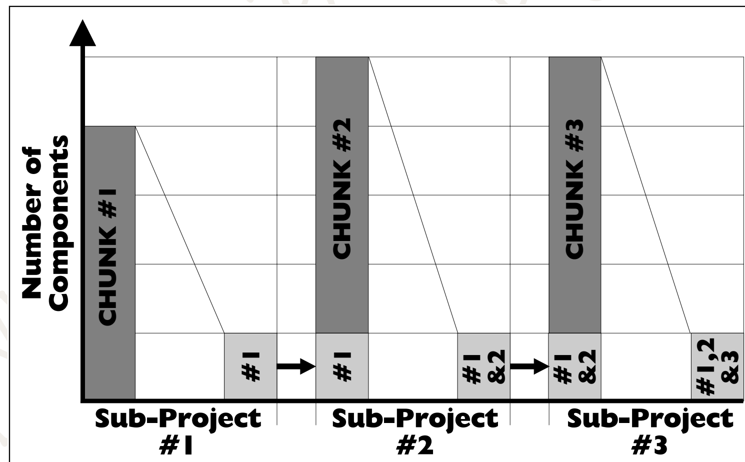
Instead, we adopted a different approach. As we re-engineered the chunks in turn, we included the scope of all the previous chunks. This meant that by the time we came to the last chunk, the scope of the re-engineering was the whole existing system (illustrated in ***Figure 18.7***). In this way, we took full advantage of the power of generalisation across a wide scope, without the risks associated with a large project.

Figure 18.7: Re-engineering combined chunks



In a traditional environment where there is little generalisation, this approach would make little sense. If, when the first chunk was re-developed, its requirements were included in the scope of the second chunk, then this would effectively double the size of the chunk. Adding in more chunks as they were re-developed to subsequent chunks would treble, quadruple, and so on, their size. This would lead to larger and larger projects, defeating the whole purpose of chunking.

Figure 18.8: Compacting combined chunks



However, in an object-oriented environment, the approach is sound. When we re-engineer a chunk, we generalise and compact its business patterns. So, when we widen the scope to include the re-engineered patterns from the earlier chunks, this does not lead to a substantial increase in the actual number of patterns. The earlier chunks' patterns are general; so, it leads to a substantial increase in the opportunities for generalisation. As a result, adding in earlier chunks does not substantially increase the size of individual chunks. In some cases it can actually substantially reduce the size of the re-engineered chunk (illustrated schematically in ***Figure 18.8***). By the time we get to the last

chunk, the scope has widened to the whole system without any of the sub-projects being any larger than they would have been in a traditional system building project.

4.5 The benefits of chunking

One of the big benefits of this kind of chunking is that there are multiple opportunities to get a pattern right. After each combined chunk is implemented, the modellers get a chance to see how their patterns are performing in a live system. This suggests improvements that they can incorporate into the re-engineering of the next combined chunk. Each redeveloped chunk, except the final one, can be treated as a prototype for the next chunk of redevelopment.

This encourages the development of fruitful patterns. People are unlikely to find the most fruitful general patterns at the first attempt. To some extent, they have to go through a process of trial and error. And chunking offers a controlled environment for trying out the patterns and finding any errors. The opportunity to have a second, third and even fourth chance to construct the right pattern, and to see each attempt in live operation, significantly increases the chance of constructing a fruitful general pattern.

This goes against the grain of the mind-set associated with traditional approaches. Because these normally only allow one attempt at constructing the right pattern, great store is set on finding a strong rigid fixed pattern that lasts the whole of the redevelopment and beyond. The object approach turns this value judgement on its head; in a re-engineering, fixed patterns are bad. If a pattern remains fixed, then this is probably because it is not being generalised. Under the object approach, the goal becomes generalising patterns rather than finding fixed ones.

4.6 Choosing chunks

One awkward management decision is deciding how to chunk up the existing system. There are many factors to weigh up, and these vary from system to system. One important factor is the type of information passing between the candidate chunks. If we choose chunks that have only a few types of information passing between them and the rest of the system, then we keep the 'complexity' of the temporary interface low.

Another factor is encouraging generalisation by putting similar patterns together in the same chunk. For example, most modellers familiar with the financial sector would be able to guess that the securities and currencies patterns are similar. If we were to allocate these patterns to the same chunk, then this would encourage their generalisation to the financial asset pattern (illustrated in ***Figure E.6***).

Most systems naturally fall into a number of modules; often, these are a reasonable basis for chunking. This still leaves open decisions on whether to have each module as a small chunk or group modules together into bigger chunks. However, someone with a working knowledge of the system should be able to have a good stab at chunking, once they understand the principles of the re-engineering approach.

© Copyright Chris Partridge chris.partridge@BORO.com

4.7 Scheduling the sub-projects with the overall project

One management task is planning how the schedule for the chunked sub-projects will fit into the overall project. The simplest schedule has each chunk completely redeveloped and implemented before the next (combined) chunk is started (shown in ***Figure 18.9***).

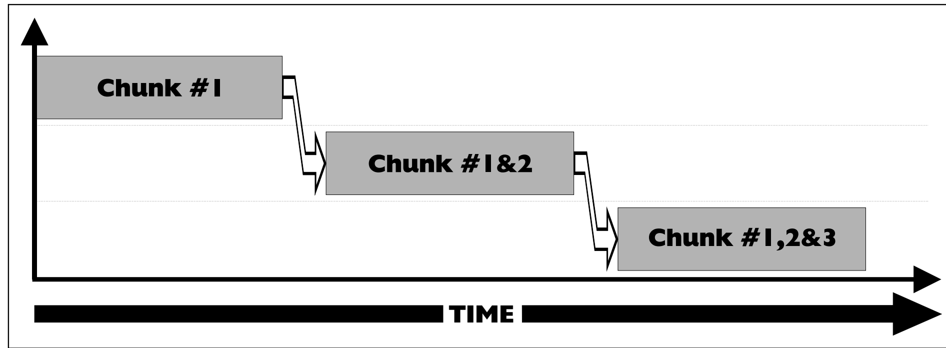


Figure 18.9: A simple sequential pattern for the overall structure

If there are tight time constraints on the overall project, then this is probably not the best schedule. In this situation, it is sensible to overlap the chunked sub-projects. One solution is to overlap them within a life-cycle stage (shown in ***Figure 18.10***). When the business modelling stage is complete for the first chunk, the systems analysis stage is started. At the same time, business modelling starts on the second chunk, including the compacted business model from the first chunk.

The perceived benefit of overlapping the sub-projects is that the overall project takes less time than if the sub-projects were to follow one after another in sequence. However, when the sub-projects are overlapped, the experience from implementing one sub-project no longer feeds back into the business modelling of the next. Project managers need to weigh up the relative benefits of sequencing or overlapping the sub-projects for their particular overall project.

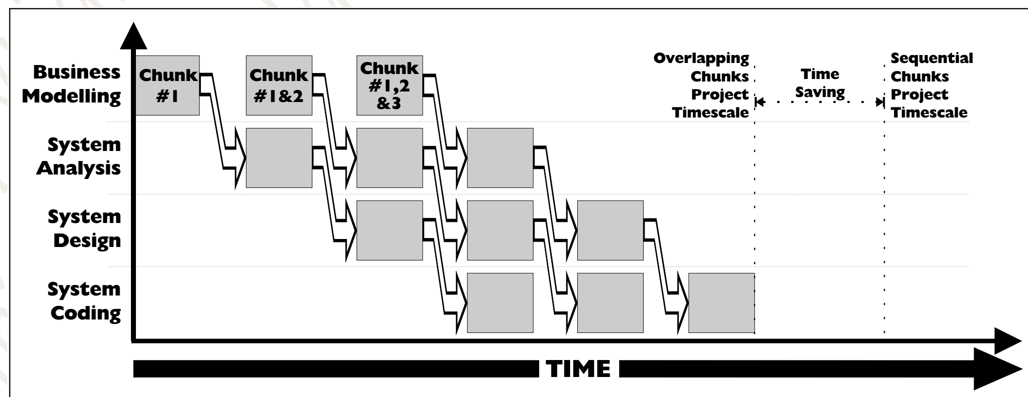


Figure 18.10: An overlapping pattern for the overall structure

4.8 How to order the individual chunk sub-projects

It is important to consider the order of the individual chunk sub-projects as well as their overall structure. To some extent, this is dictated by the demands of the business. If a particular chunk contains new functionality that is critical to the business, naturally it is given a high priority.

However, the dependencies between business patterns also dictate, to some extent, the order of re-engineering for chunks and entity formats within chunks. For instance, transaction entity formats should, in general, be re-engineered after 'static data' entity formats. This is because the business patterns in transactions tend to depend on the patterns in 'static data'.

We shall see an example of this in the re-engineering of an accounting transaction in the ***Epilogue***. Its patterns depend on the 'static data' person and asset patterns, but not vice versa. In a 'real' re-engineering, it would make sense to re-engineer this static data before the accounting transaction.

When planning the order of the individual sub-projects (and the order of the entity formats within the sub-project), it makes sense to take account of the dependencies between the business patterns and to plan to re-engineer the dependent patterns after the patterns they depend on.

These dependencies between patterns are not just a feature of object systems. They are reasonably well-known in larger traditional systems. I have come across a number of package systems that explains the dependencies between data in their start-up documentation, saying, for instance, that company data depends on the correct country data being available. These dependencies are then used to suggest a schedule for setting up data in the system.

4.9 Ephemeral documentation

Because this approach treats all chunks, except the final one, as prototypes for the next chunk, this raises a tricky issue for system documentation. Unlike traditional approaches, the re-engineering approach expects the early (prototype) chunks to change significantly as their patterns are generalised. This means that the documentation for these chunks is ephemeral, going out-of-date when the next chunk is re-engineered. So, producing full documentation for each chunk seems like a costly waste of time. But this has to be set against the problems of running the implemented chunks in a live system without all the documentation. And the problem applies to all the types of documentation; both business model and system.

It is sensible when planning the project to specify the types of ephemeral documentation that will be produced during the re-engineering of the prototype chunks, balancing the cost of producing it against the benefit it brings. Also, the plan for the final chunk needs to contain the task of producing the full set of documentation. Then, everyone can be clear about what documentation has to be produced when.

© Copyright Chris Partridge chris.partridge@BORO Centre.com

When assessing whether particular types of ephemeral documentation should be produced, we need to consider its uses, both in live operation and the system building process. It makes sense to produce documentation that is key to either of these. For example, it could be argued that producing tidied up versions of the object schemas is not particularly important to the live operation of the system. However, I have found that trying to produce presentable versions often brings to the surface useful insights, improving the quality of the business model. For this reason, I usually suggest that they are produced.

5 Produce a validated understanding of the business

Industry studies seem to show that a large number of the errors found when systems are implemented are because of misunderstandings about what the business required rather than errors in coding. These types of errors are, for the most part, avoidable in a re-engineering project. The re-engineering should produce a business model that reflects the business patterns accurately. And, in theory at least, these should not need to be changed during the system building process. Nor should they lead to errors in the implemented system.

However, I have found that object schemas on their own do not provide a complete enough check on the accuracy of the patterns in the business model. They are a potent tool for making visible the patterns we use to understand the world. They take advantage of the human brain's ability to spot any out-of-place shapes in the patterns. But, despite all this, they do not provide as complete a check on the accuracy of the patterns as required. I find that I need to build a validation system to give myself a reasonable confidence that the reflections are accurate.

5.1 Building a validation system

The validation system is the business model translated into a database and populated with a representative sample of operational objects. I use the existing system as my primary source for the operational objects, migrating its data onto the validation system. For small files, I migrate all the operational data; but, with the larger files, I usually only migrate a representative sample. Where possible, I migrate the data automatically. I also load up any new operational 'data' found during the analysis of conceptual patterns (an example would be England and the other nested countries in ***Chapter 14's*** country re-engineering).

The validation system does not require sophisticated technology and should not involve much effort. It can be built within a CASE tool (if one is being used) or constructed on a simple computer database. (I have found that non-object-oriented PC databases are a cheap and effective solution.)

I normally construct the validation system as I am doing the modelling, translating and migrating the data from the existing system as I re-engineer its entity formats. This usually brings up issues, which I can resolve there and then. When sufficient data has been migrated, I produce reports and enquiries from the validation system. This enables me

© Copyright Chris Partridge chris.partridge@BORO Centre.com

to touch and feel each bit of the model as it grows; there is no real substitute for this. I can often see immediately whether something works and change it if it does not. In addition, most users have found these reports and enquiries a more accessible way of checking the model than object schemas.

The key benefit from constructing the validation system is that inaccurate reflections of business patterns are found and fixed before any time has been spent building them into the system. This helps avoid the frustrating experience common in traditional system building, that is, finding resources have been wasted building inaccurate business patterns. Using a validation system significantly reduces the level of these errors, minimising the wasted resources.

6 Object model the migration of business patterns

A re-engineering project, by its nature, involves the migration of business patterns from the existing system to the new object system. These will be application level patterns, such as countries, and operational level patterns, such as United States. If the business paradigm embedded in the final system does not accurately reflect the business model (and so the real world), then much of the model's power can be lost. One way of ensuring that this does not happen is having a sufficiently formal and accurate specification of how the business patterns are migrated. I do this by constructing an object model of the migration.

6.1 Tracing the migration of application level business patterns

The business model produced by the re-engineering process contains an accurate reflection of the business. It should be embedded in the final system. However, I have found that a common problem with the embedding is that some system analysts and designers treat the business model as a proposal rather than a formally defined input into the process. They assume that they can exercise their judgement to pick and choose what to embed and amend as they see fit.

This is, in my experience, a general problem for all business models—not just object models. But with an object model, it is a sure-fire way of losing the benefits of business object modelling. The object model is a tightly connected system. Fiddling with bits of it, particularly by someone who does not understand the business patterns, is almost certain to have a deleterious effect on the whole structure.

This is not to say that systems analysts and designers should be discouraged from finding inaccuracies in the business model—quite the opposite. But if they do find what they consider to be an inaccuracy, business modellers should check it. If it is a real inaccuracy, the more accurate pattern should be applied to the business model and it should work its way through normal channels to the systems analyst and not unilaterally applied to the system specification.

I have found that the simplest way to ensure that the business model's accurate patterns are embedded unchanged in the final system is to provide a system for confirming

© Copyright Chris Partridge
chris.partridge@BORO-ltd.com

that this has been done. I model the business patterns' translation into the system model and onwards into the implemented system. This translation model provides traceability. We can trace the migration of the patterns from the business object model to the implemented system. Any unauthorised changes are brought to light. The formal nature of the translation model also means that the checking can be automated.

6.2 Modelling the migration of application level business patterns

The simplest way to model the migration of the application level business patterns is to extend the business object model to include the translation of the patterns into the implemented system. The first step is to extend the meta-model. Then as a second step, these two extensions are populated.

We need two extensions to the meta-model. The first is a system object model for the paradigm used by the implemented system. The second is a general translation tuple that has as members the tuples connecting the objects in the business model and the objects in the implemented system model. The result is illustrated in ***Figure 18.11***.

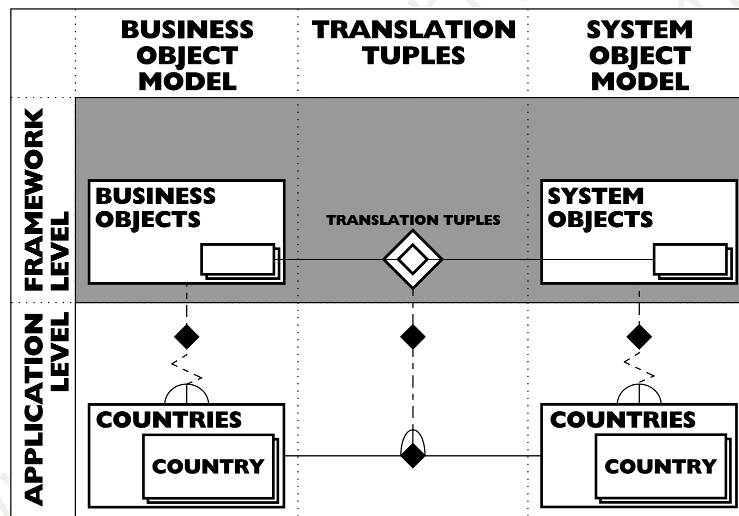


Figure 18.11: Extended application level migration model

It is worth bearing in mind that the business object model is technology independent. This means, among other things, that it can be implemented on any technology. It can be implemented into an object database, a relational database or even simple flat files. It can be implemented in an object-oriented programming language, such as C++ or Smalltalk, or it can be implemented in a traditional language, such as COBOL. However, each of these implementations requires its own system meta-model in the migration model.

6.3 Modelling the migration of operational level business patterns

In a re-engineering project, we need to migrate the operational level business patterns as well as the application level patterns. I normally do this twice. I do this once during

business modelling, when I populate the validation system with operational objects (described earlier); and then a second time at implementation, when I populate the implemented system with relevant (operational) static data (items such as currency and clients). These operational objects come, for the most part, from the existing system.

We normally re-engineer a few representative operational objects from the existing system to provide us with the basic patterns from which we generalise the application level patterns. For example, in ***Chapter 12*** we re-engineered the individual objects, the United States and the United Kingdom, and generalised them into the countries class. However, the validation and implemented systems need many more operational objects than the re-engineering.

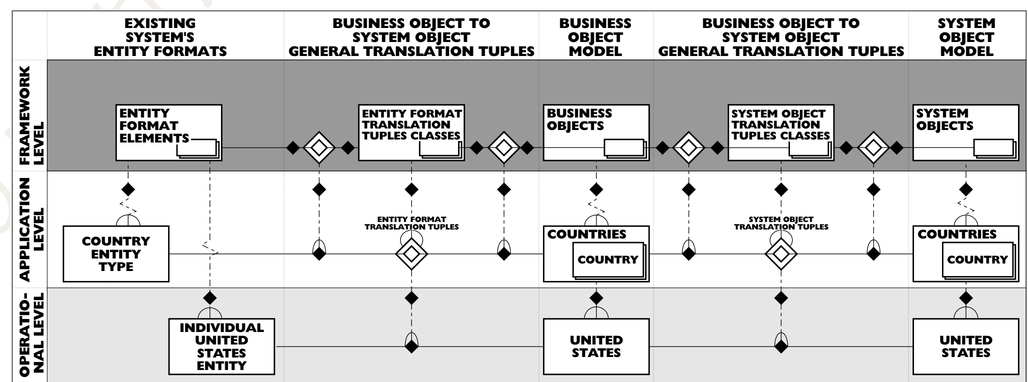
I have found that it helps me to manage the migration of this large number of operational objects, if I model its general patterns. I do not describe each individual operational object's migration pattern; instead, I use a representative sample to construct general 'application level' migration patterns. These then constitute a migration specification I can use for all the operational objects.

My first step is to extend the business object meta-model. I extended it for the target system (either the validation system or the implemented system), when I set up the 'system' for checking whether business patterns were properly embedded in the implemented system. (This was described in the previous section and illustrated in ***Figure 18.11***.)

So I now extend the meta-model to include the existing system, the prime source for operational objects. I also include a general translation tuple linking the existing system's entities to the business objects. ***Figure 18.12*** provides an idea of what the extended meta-model would look like.

The second step is to model the migration of some representative operational objects and to discover the basic patterns for application level migration. These are migration 'rules'. I use them to specify how the operational entities from the existing system can be correctly embedded in the target system. I sometimes use them as a model for an automated migration process. Either way, they greatly simplify the migration of data from the existing system to the new system.

Figure 18.12: Extended operational level migration model

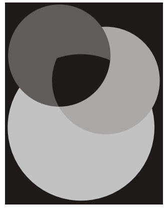


7 Summary

This is the end of the main part of the book. This should have given you a good understanding of what business objects are and how you can use them to help you re-engineer your legacy entity oriented systems. (In the ***Epilogue*** we focus on a different topic, using business objects to re-engineer the business.)

You now appreciate the benefits of re-engineering general, fruitful and thus re-usable business objects. This final chapter stresses the importance of embedding them properly in the final system, if you want to harvest these benefits. To those of you about to use what you have learnt here on a re-engineering project—good luck!

© Copyright Chris Partridge
chris.partridge@BOROCentre.com



Epilogue

Using Objects to Re-Engineer the Business

1 Introduction
2 The accounting paradigm's debit and credit pattern
3 Accounting's ledger hierarchy
4 Developing a new object-oriented accounting paradigm
5 Industrialising information
6 21st century information industries

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

1 Introduction

The main body of the book, particularly Part Six, deals with how business objects are starting to revolutionise the way computer systems are built. We saw that business objects are not only making systems simpler and functionally richer and so cheaper to build and maintain. This inevitably leads to big changes, such as substantially increased levels of automation. However, it would be a serious mistake for us to think that business objects will only change computer systems. We would be missing their far more exciting potential for re-engineering the business.

1.1 Missing the wider potential

It often happens that the wider potential of a radical innovation is missed. History is littered with examples. For instance, Western Union, the telegraph company, turned down the chance to buy Alexander Graham Bell's 1876 telephone patent for a small sum. It thought that it was thinking strategically when it offered to stay out of telephones if Bell stayed out of telegraphy. (Bell also missed the point: he entitled his patent *Improvements in Telegraphy.*)

More recently, the inventor of the transistor, one of the 20th century's most important innovations, thought it might be used to make better hearing aids. Even more recently, when the laser was invented at Bell Labs, its lawyers were initially unwilling even to apply for a patent on the invention, believing it had no possible relevance to the telephone industry.

In the computing industry, the founder of IBM, Thomas J. Watson, Senior, originally declared in 1948 that as many as 12 companies might some day have their own computers (a few years later he revised this figure to 50). He anticipated that scientists and engineers would use them as improved calculating machines—replacements for their log tables and slide rules. He had no idea that business people and accountants might be a market.

1.2 Business objects' wider application

We would be making a similar mistake to Thomas J. Watson, if we expect business objects to be only used to build better computer systems that automate more of the business. Like other radical innovations, they have a wider potential than their obvious application. Surprising as it may seem, I expect that their most significant impact will not be on computer systems, but on the businesses underlying those systems. They will play an important part in the industrialisation of business's information. This will have its biggest impact on those businesses (or parts of the business) that work with information, for example:

- Information industries—such as banking and insurance, and
- Information professions—such as accounting and law.

© Copyright Chris Partridge
chris.partridge@BORO-Centre.com

2 The accounting paradigm's debit and credit pattern

We can get some idea of how far reaching the effects of this industrialisation will be by looking at an example of how business objects are going to change an area of the business. The spatial, temporal and naming patterns that we re-engineered in Part Six are too general for this. We use as our example a paradigm that is central to the management of most businesses—the accounting paradigm. Re-engineering this is a substantial task; all we do here is outline how a core pattern—the accounting transaction's debit and credit pattern—can be re-engineered.

The accounting paradigm is a pertinent example. Its current framework is the accounting ledger, whose columns and rows are designed for paper's two-dimensional surface. The worked examples in Part Six showed us the extent to which the paper-bound entity paradigm constrains and distorts patterns. The example of an accounting pattern that we are going to look at—accounting transaction – has been distorted in a similar way to fit into the constraints of paper's two-dimensional surface. The re-engineering will free it from those constraints.

2.1 From journal transaction to debit and credit movements

Accounting transaction's debit and credit movement pattern is well over five hundred years old, but was effectively standardised in the 15th century. This happened when the invention of printing lead to the publication and wide distribution of a number of books describing the process of bookkeeping.

The first, and most famous, book was by a Franciscan monk, Fra Luca Pacioli. In 1494 he published a book on mathematics (*Summa de Arithmetica, Geometria, Proportioni et Proportionalita*), which contained a treatise on bookkeeping (*Particularis de Computis et Scripturis*, which translates as 'Details of Accounting and Recording').

In his treatise, Pacioli described the book-keeping method used by the merchants of Venice (which was then one of the most powerful city states in Europe); hence, he called it the Method of Venice. The method was not new; the merchants of Venice had been using it for centuries. However, once Pacioli's book was published, bookkeepers across Europe started to standardise on it. The Method of Venice has proved to be extremely durable; accountants and bookkeepers still use something similar today.

A central feature of the Method of Venice is double entry bookkeeping. It is called double entry because a transaction is, in general, entered twice, firstly into a journal and secondly into a ledger. It is entered into the journal in the format of an accounting transaction. It is then divided into a debit and a credit movement and these are entered into different parts of the ledger.

This is an example of how resorting and reformatting is done within the constraints of paper and ink technology. Each 'book' contains the same information, but in a different format and order; each gives us a view of the business. Paper and ink technology sets a

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

limit on the number of possible views. For example, taking any more than these two book-keeping views would involve significant extra effort.

Nowadays, most computing systems automate this manual resorting and re-formatting. When a transaction is entered, they first store it on a transaction file. Then, they automatically re-format it as a debit and credit movement and 'post' it to an account movements file, updating the relevant ledger balances.

The way in which the book-keeping process divides the transaction into a debit and credit movement for the ledger view, suggests that it sees the transaction as having the two movements as components—as illustrated schematically in ***Figure E.1***.

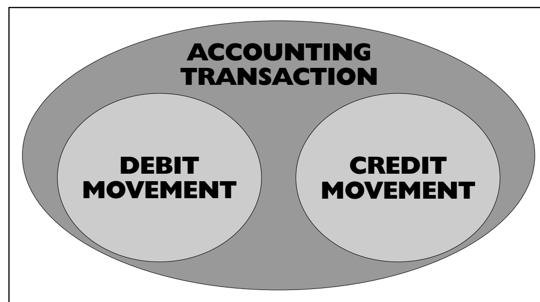


Figure E.1: An accounting transaction and its component movements

2.2 The accounting transaction and movements entity formats

We now re-engineer the accounting transaction and its two components. This should unwind any distortions imposed by pen and paper technology—revealing the objects that the transaction refers to.

Transaction Code	Transaction Date	From Account	To Account	Amount
#101	25-Apr-94	Joe Bloggs	Me	£10,000

Table E.1: Partial accounting transactions listing

Entity type	Accounting Transaction
Attribute type #1	Transaction code
Attribute type #2	Transaction date
Attribute type #3	From Account
Attribute type #4	To Account
Attribute type #5	Amount

Table E.2: Accounting transaction entity format

We follow the process used in the worked examples. We look at a listing of the entities and then their entity formats. The entities are shown in ***Tables E.1*** and ***E.3***, their for-

© Copyright Chris Partridge chris.partridge@BORO Centre.com

mats in ***Tables E.2*** and ***E.4***. You can see how neatly accounting's paper-based rows and columns map into the similarly paper-based entity formats

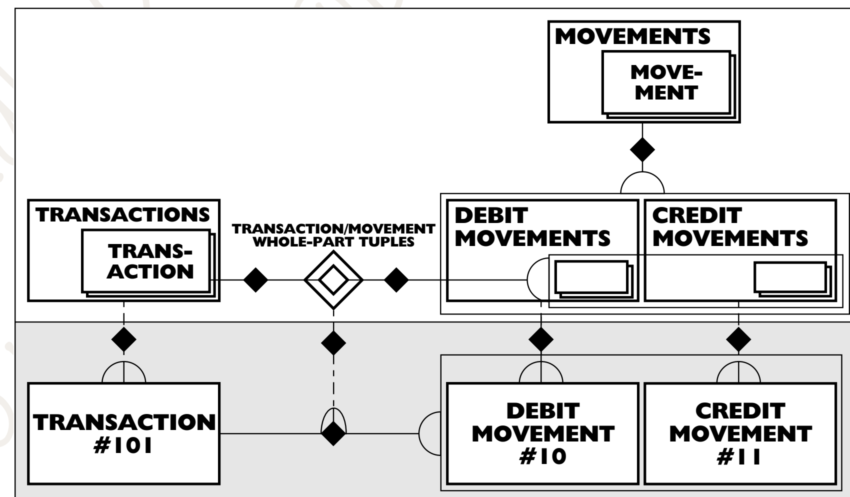
Entry Code	Transaction Code	Transaction Date	Account	Debit/credit Indicator	Amount
#10	#101	25-Apr-94	Joe Bloggs	Debit	£10,000
#11	#101	25-Apr-94	Me	Credit	£10,000

Table E.3: Partial accounting movements listing

Entity type	Accounting movement
Attribute type #1	Entry code
Attribute type #2	Transaction code
Attribute type #3	Transaction date
Attribute type #4	Account
Attribute type #5	Debit/credit indicator
Attribute type #6	Amount

Table E.4: Accounting movement entity format

Figure E.2: Accounting transaction model



© Copyright Chris Partridge chris@BORO Centre.com

2.3 Re-engineering the accounting transaction pattern

If we assume that the movements are components of the transaction, we might re-engineer the two entity formats into the model in ***Figure E.2***.

2.4 Re-engineering a transaction event

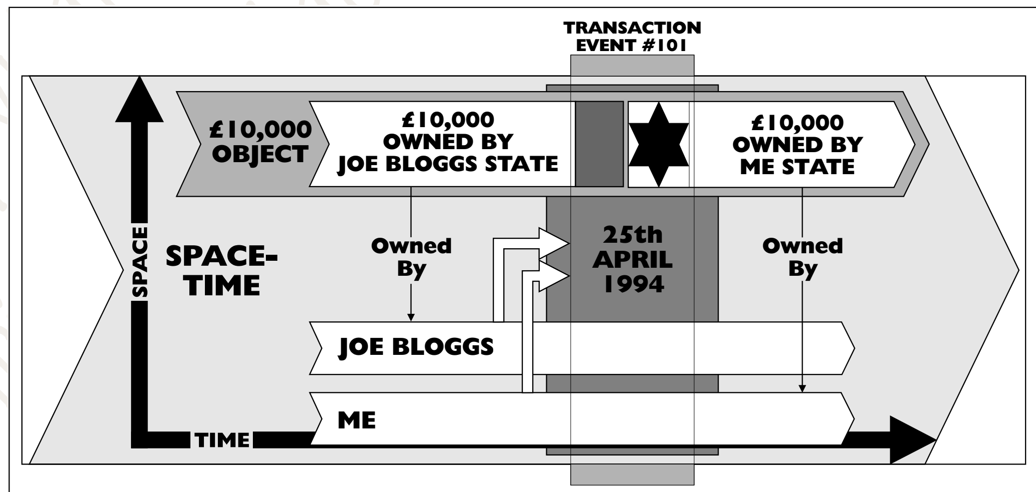
We begin to realise that ***Figure E.2*** is not accurate, when we ask, from an object point of view, what the accounting entities refer to. We realise that the model is describing patterns in the information rather than in what the information refers to—the business.

A big clue that this is happening is the type of sign used for the transactions and movements. They are modelled with physical body signs, suggesting that they are physical bodies. But in the business neither the transaction nor the movements persist through time. This makes them events, not physical bodies. (We looked at this type of mis-classification for accounting movements in ***Chapter 2***, when we considered the data–process and things–changes distinctions.)

If we now look at the actual transaction event in the business, we get a very different pattern from ***Figures E.1*** and ***E.2***. We start by asking what the event happens to. The answer is the £10,000—it changes owner. We came across this pattern in ***Chapter 8***, where we looked at the sale of a car (illustrated in ***Figures 8.1*** and ***8.2***). In that pattern, the car moved from an 'owned by garage' state into an 'owned by Ms Brown' state. We re-use the pattern here on the £10,000. We see it 'moving' from an 'owned by Joe Bloggs' state into an 'owned by me' state.

The transaction event is revealed as a 'change' in the £10,000's states (described in the space-time map in ***Figures E.3***). You may have noticed that in this revised view there are no debit #10 and credit #11 movement objects (illustrated in ***Figures E.2***). Debits and credits are ways of looking at the transaction event, not objects.

Figure E.3: £10,000 state change event space-time map

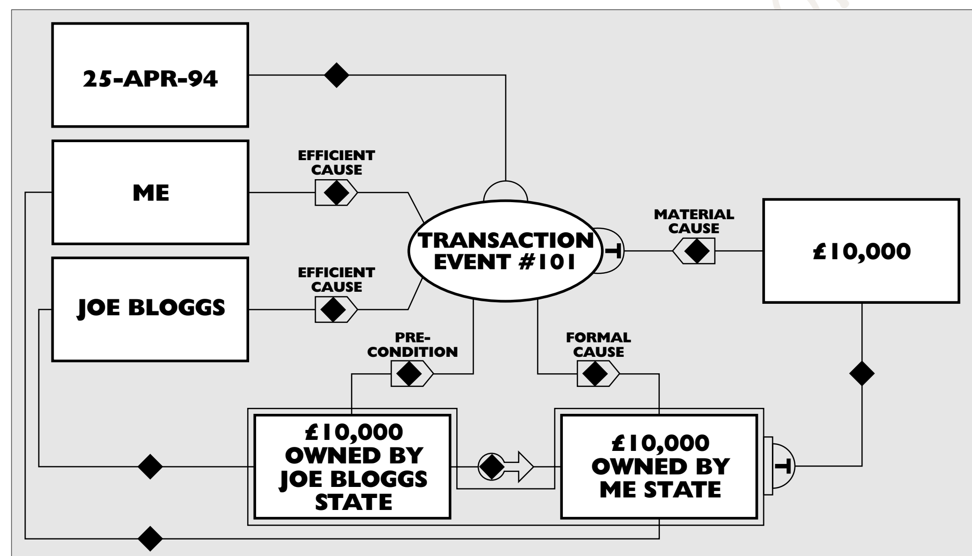


This space-time map helps us see the event's causal connections (as explained in ***Chapter 8*** and illustrated by ***Figure 8.27***). For example, the two parties to the transaction (Joe Bloggs and Me) are, in Aristotelian terms, the efficient causes of the event (things that make the change happen). In addition:

- The £10,000 is a material cause (what the change happens to), and
- The '£10,000 owned by me' state is a formal cause (what the change results in).

In addition, the '£10,000 owned by Joe Bloggs' state is a pre-condition All these causal connections are modelled in the schema in ***Figure E.4***. It also describes the structural nature of the transaction event's connection with the 25-Apr-94 day (date) object, which is whole–part.

Figure E.4: £10,000 transaction event #101 causes object schema



This is a very different pattern from that in ***Figure E.2***. Its pattern was moulded by the constraints of paper and ink technology—particularly its re-sorting and re-formatting process. It gave a reflection of how transactions are re-formatted into movements, not a reflection of the business.

2.5 Re-engineering the overall transaction event

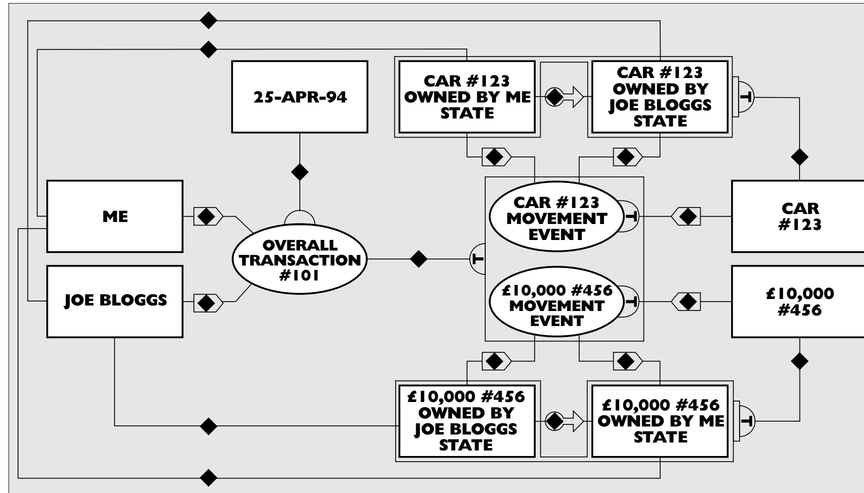
The re-engineered transaction event in ***Figure E.4*** only covers half the transaction. Joe Bloggs paid £10,000 for something, which does not appear in the accounting transaction. This is because accounting transactions only record 'movements' of money. They ignore the non-money element. Once we recognise this non-money element, we can see that the two elements combine to form an overall transaction.

When we analyse the non-money element of the transaction, we see it has the same pattern as the money element. Assume that Joe Bloggs bought car #123 with his £10,000. This car has an owned by me state ending in a movement event followed by an owned by Joe Bloggs state. The car and £10,000 movement events are the encap-

© Copyright chris partridge .com

sulated parts of an overall transaction. Once we recognise this, we can see that Joe Bloggs and I are parties to the overall transaction rather than the individual movement events. We capture this insight in ***Figure E.5***.

Figure E.5: Overall transaction #101's object schema

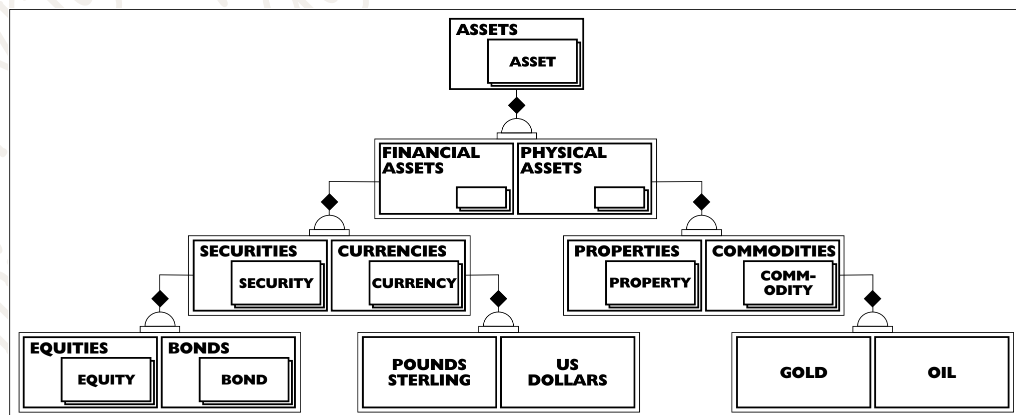


2.6 Generalising to the assets level

In ***Figure 5*** it is clear that the pattern for the money and non-money movement events are similar. When we generalise this individual transaction pattern to class level, we construct one class-level pattern that has both money and non-money elements as its members. The accounting transaction pattern cannot recognise this generalisation because it is artificially restricted to the money elements only—a result of its origins in paper technology's rows and columns.

Money—or currency—is merely one type of asset. It is not even a major type of asset as the schema of generalised assets in ***Figure E.6*** shows.

Figure E.6: Generalised assets



The asset super–sub-class hierarchy is rich. The re-engineerings that I have been involved in have revealed a variety of asset sub-classes. Things such as dividend enti-

tlement coupons and tax credit vouchers turn out to share in the overall pattern of the asset family. However, for our current purposes, the key aspect of this asset hierarchy is that it shows the types of asset our re-engineered transaction can model.

2.7 Generalising transactions to orders/exchanges

There is another direction in which the transaction pattern can be generalised. Transactions are composed of two general patterns—order and exchange. These can be combined in a different way to construct another core pattern—order then exchange.

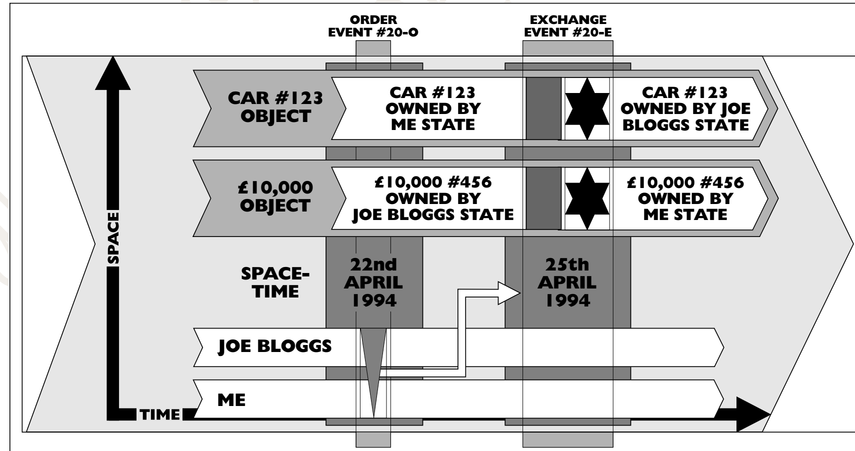
We now assume that Joe Bloggs called up and ordered his car a few days before he came in and exchanged his £10,000 for it. The order then exchange would be recorded as in ***Table E.5***.

Order Code	Ordered By	On	For	Item Type	Number Of Items	Item Cost	Total Cost
#20	Joe Bloggs	22-Apr-94	25-Apr-94	Car	1	£10,000	£10,000

Table E.5: Partial simplified orders then exchanges listing

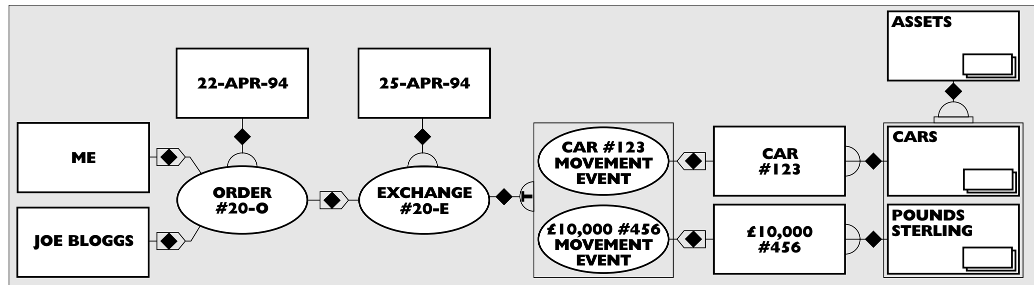
If we re-engineer this order, we find it refers to the objects described by the space-time map in ***Figure E.7***. Notice that the order and exchange elements look as if they are the transaction illustrated in ***Figure E.3*** divided in half.

Figure E.7: Order space-time map



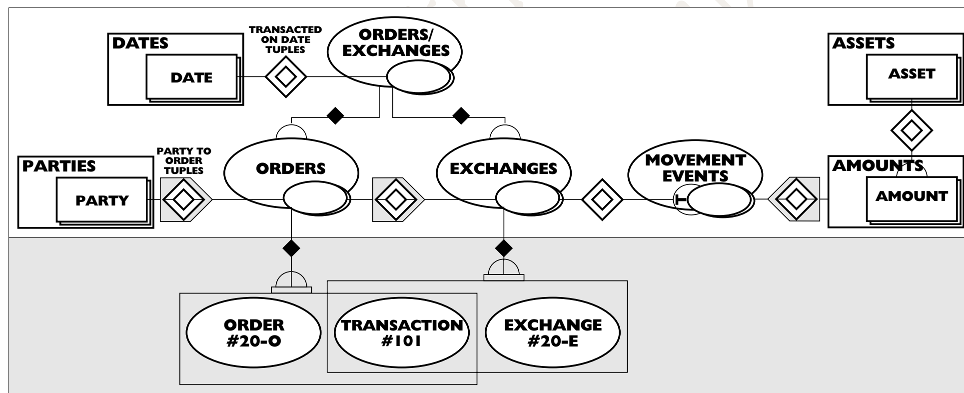
We intuitively understand the order as contracting for the future exchange. This is reflected in the object schema in ***Figure E.8***. The order event gets its meaning (in Fregean terms, its sense) from its connecting pattern with its exchange. Notice that now we have generalised assets; we show the two amounts as belonging to the asset subclasses—cars and sterling. Notice also that the underlying pattern has been made clearer by the omission from the schema of the before and after states of the amounts.

Figure E.8: Order object schema



It is interesting to compare this schema with the schema in ***Figure E.5***. The connecting patterns that 'Me' and Joe Bloggs had with transaction #101 have moved along to order #20-O. It is as if the standalone transaction in ***Figure E.5*** has been divided into two—which it indeed has. Order #20-O has separated the parties contracting to the exchange from exchange #20-E. The individual level order, exchange and transaction patterns generalise into the class level pattern shown in the simplified object schema in ***Figure E.9***.

Figure E.9: General order object schema



The current accounting paradigm, with its origins in paper and ink technology, cannot accommodate this general order/exchange pattern's shape. It typically works around the problem by treating the order element of an order then exchange as another accounting transaction, which generates debit and credit movements for the order date. The accounting paradigm cannot give any firm guidance about what these movements should be, because they only indirectly refer to the transaction. This has resulted in equally 'valid' but different ways of accounting for the overall transaction. For example, bank's treasury operations can choose between a trade (order) and a value (exchange) date accounting approach.

2.8 Generalising the order pattern

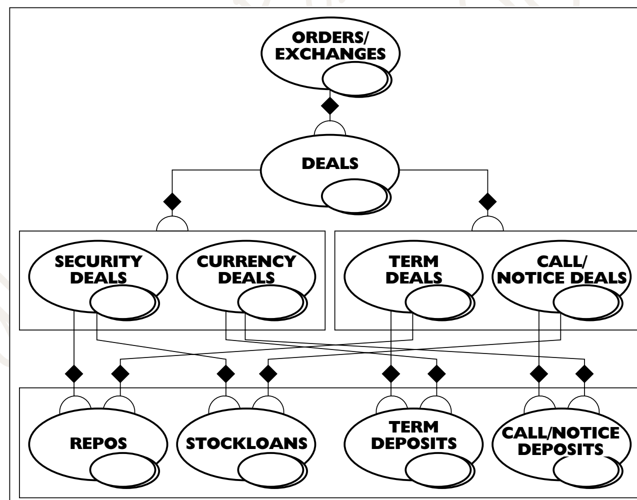
The re-engineering has given us a general order/exchange pattern with the movement event—the object version of accounting movement—at its core. This order/exchange pattern is a basic business pattern. It occurs frequently across a range of businesses. In a re-engineering of an international securities settlement system, we found it in most of the 'transactions', including:

- Security purchases,
- Security sales,
- Dividend entitlements,
- Bonus and rights entitlements,
- Tax entitlements,
- Stock borrowing and lending agreements,
- Term deposits placed and accepted,
- Foreign exchange deals, and
- Call/notice deals.

We found that the generalisation of these classes followed the same pattern as the worked examples in Part Six. As higher level classes were constructed, these became redundant and were purged, compacting the model.

The sub-classes of the general orders/exchanges class fell into a super–sub-class hierarchy similar to the one shown for deals in ***Figure E.10***. In it, we can see how high level patterns combine to construct new sub-classes. For example, when the term deals pattern is combined with the currency deals pattern, it gives a term deposits pattern. When it is combined with the security deals pattern, it gives a repurchase agreements (repos) pattern.

Figure E.10: Deals super–sub-class hierarchy



We also found that at a general level, the order/exchange pattern was a generalisation of the account pattern. So accounts and its various sub-classes are revealed as sub-classes in the orders/exchanges super–sub-class hierarchy. ***Figure E.11*** shows part of the hierarchy and how the higher level accounts classes combine to give lower level classes. It also shows the wide scope of the accounts pattern. This not only covers the more traditional call/notice deposit and stock depot accounts, it also covers investment portfolios and foreign exchange (fx) trading books.

© Copyright chrispartridge.com

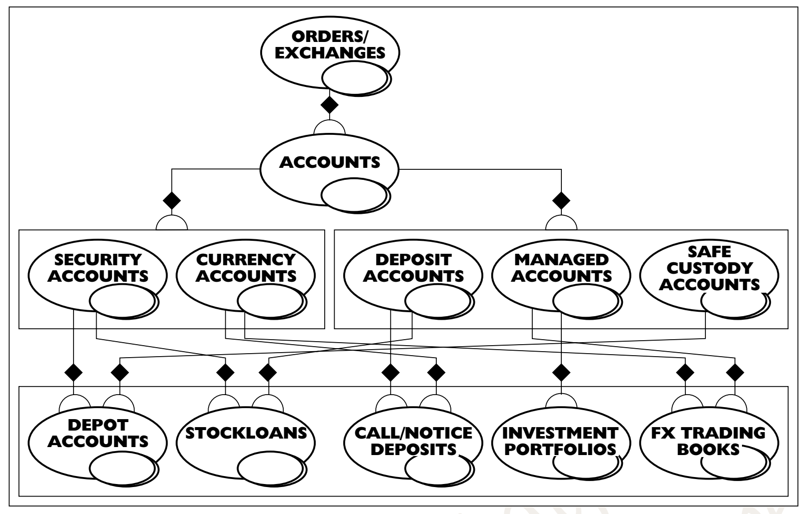


Figure E.11: Accounts super–sub-class hierarchy

There is a certain irony in the fact that the scope of the orders/exchanges pattern includes the accounts pattern. Accountants involved in the specification of some of the first generation of international banking systems stretched the account pattern almost to the breaking point. They tried to fit everything, including the order/exchanges pattern, into it.

For example, in one system they created new ‘currencies’ to accommodate foreign exchange deals (where a sum of money in one currency is exchanged for a sum of money in another). For a US$–Deutsche Mark foreign exchange deal, they would create an accounting movement in a US$–Deutsche Mark ‘currency’. This had the advantage of making it easier to fit a foreign exchange deal into the accounting movement pattern. It was soon found that the disadvantages of distorting the deal to fit the accounting mould more than outweighed the advantages, and the ‘general’ accounting pattern was dropped.

However, this re-engineering shows that the accountants’ belief in a general pattern underlying the deals is correct. Unfortunately for them, it is not their accounting transaction/movement pattern!

2.9 Fitting the business into the current accounting paradigm

The accounting transaction pattern is a partial view (more correctly, two partial, distorted views) of the money element of the overall transaction pattern. This is typical of a paradigm based on paper and ink technology. We saw something similar when, in ***Chapter 16***, we re-engineered the address pattern. It was also a partial—and distorted—view.

Being restricted to two partial views creates problems. It is difficult, for instance, to give a full rounded picture. Bookkeepers often massage the chart of accounts so that they can fit more into the two views. For example, they create extra accounts, which do not reflect anything directly. They justify the particular rules they use for generating these accounts and their accounting movements by the way they result in final reports that give ‘a true and fair view’ of the business. Whether the accounting movements actually

reflect the business accurately is often not considered. In this environment, it is not surprising that a number of different accounting practices arise—such as the trade and value date accounting methods mentioned earlier. Without the criteria of reflecting the business accurately and directly, it is impossible to arrive at a definitive accounting practice.

We saw something similar happening in ***Chapter 3*** . There we looked at how the entity paradigm was simplified to work within paper and ink technology. We saw how this confused its semantics so that it was no longer able to reflect the real world directly. As a result, people made decisions on whether to use an entity or attribute sign based purely on which made the information processing more effective. Whether the sign directly referred to an entity or attribute in the business was not considered.

3 Accounting's ledger hierarchy

It is not just the underlying accounting movement pattern that is constrained by paper and ink technology. All the patterns in the current accounting paradigm are. Another good example is the ledger balance hierarchy. This is a hierarchy of the balances created by the debits and credits posted to the ledger book. It is traditionally a structure similar to that shown in ***Figure E.12***.

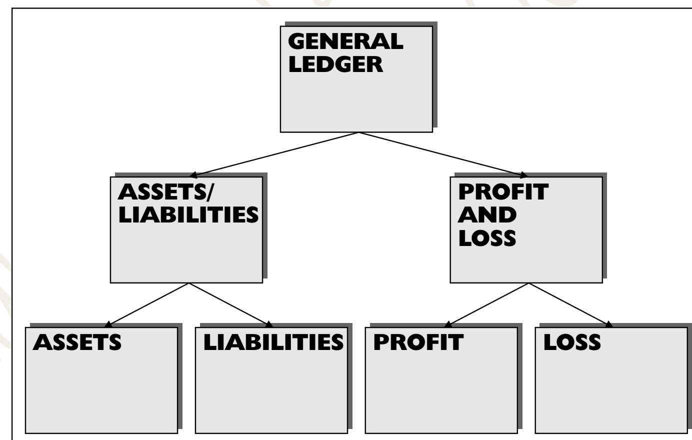


Figure E.12: The traditional structure of the ledger hierarchy

This is a tree structure—much like the secondary substance hierarchy (shown in ***Figures 4.15*** and ***4.16***). As we discussed in ***Chapters 4*** and ***6***, a tree structure means that the hierarchy is constrained (see ***Figure 6.3***). Classification schemes that reflect the world directly, such as the super–sub-class hierarchy, have a less constrained lattice structure. This ledger hierarchy needs to be liberated from its tree structure constraints by re-engineering.

© Copyright chris partridge BORO Centre.com

4 Developing a new object-oriented accounting paradigm

While companies manage their businesses using paper reports, the current paperbound accounting paradigm will have a use. However, when information is routinely supplied electronically, things should change. There will no longer be any technological reason for supplying managers with partial and distorted views of their business.

To see this, consider a business that is building a computer system based on the general orders/exchanges pattern. At a meeting, the requirement to produce standard accounting reports is raised. The system designer suggests that this is done by taking a partial view of the orders/exchanges pattern and using it to generate the traditional accounting entries. He or she explains that these can then be processed in the traditional way to produce the daily journal, balance sheet and other accounting reports. The business modeller then asks what business objects this new information reflects. The answer is that they do not directly reflect anything.

This raises the question—why should managers use this distorted accounting information? Furthermore, why should they be restricted to two views? Shouldn't they be given a multiplicity of views over undistorted information? They should, and this is why the current accounting paradigm needs a thorough re-engineering. When this is done, managers will have undistorted information.

However, this re-engineering will be a substantial task. The shift to the general orders/exchanges pattern described earlier is only a small part of it. We not only have to re-engineer the foundational accounting transaction/movement pattern—as we have started to do here; but we also have to re-engineer the patterns built from it, such as the ledger hierarchy.

In the re-engineered accounting paradigm, complex notions such as assets, liabilities, profit and loss will be transformed. The new paradigm will use the transformed notions to give a more accurate, more relevant vision of the business. However, to re-engineer these requires a thorough knowledge of accounting. We will not find the insights that we need in the entity formats of computer systems. We need to look at the conceptual patterns of people who understand accounting in depth.

Undoubtedly, when businesses start using the new accounting paradigm there will be resistance. Though some people will welcome the new paradigm, others will oppose it. There was a similar reaction when computers were first introduced. Like computers, the superiority of the new paradigm will ensure that, in the long run, it establishes itself.

5 Industrialising information

The transformation of business paradigms (of which the accounting paradigm is an example) is going to be a vital part of an overall industrialisation of information. This industrialisation will need new skills applied to new standards of accuracy. I expect that a new profession of information engineers will need to be created to do this.

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

This is a direct parallel with the rise of a ‘physical’ engineering profession in the 18th century Industrial Revolution. The two revolutions are similar in many ways. In ***Chapter*** , we compared the physical accuracy that drove the Industrial Revolution’s development of interchangeable parts with the conceptual accuracy that is driving the current development of general, re-usable business objects.

5.1 The rise of an engineering profession

Not only was the ‘physical’ engineer instrumental in making the Industrial Revolution, it can equally be said that the Industrial Revolution created the modern engineering profession. As the revolution emerged, it demanded new technical skills. Ones that were not taught to the pre-revolution craftsmen in their craftshops. When it became apparent that these skills could be codified, it was also realised that the best way to learn them was a formal technical training. Originally this was provided in military academies, but eventually established universities followed suit. This formal training set the ‘engineers’ apart, and from this, the engineering profession naturally developed.

We can see a similar pattern emerging in business modelling. Currently business modelling is a craft. Modellers are not given much, if any, formal training. They are certainly not given any training in information paradigms and how they work. Most of them are recruited from the ranks of programmers and system analysts. Some are recruited from the operational parts of the business. For all practical purposes, business modelling can be considered a craft carried out by craftsmen.

Business objects require information ‘engineers’ with a more professional technical training. For a start (as this book has shown), they need to be able to see and model business objects with a high degree of accuracy. For this, they need a good understanding of what they are. This is sufficient for the simple re-engineering of entity formats in existing systems. But, to take advantage of business objects’ flexibility to handle far more powerful patterns, information engineers will need to re-engineer the conceptual patterns of experts in the business. This will involve either training the experts in business objects or, more likely, the information engineers developing a deep understanding of the experts’ conceptual patterns. In other words, the engineers will have to become business experts.

Business analysts already have to develop a good understanding of the business to do their job. This is typically learnt in a similar informal way to the pre-revolution engineer–craftsmen. Information engineers will need a much deeper understanding. Formal technical training will be the simplest way for them to develop the required in-depth knowledge of the business.

5.2 Where will the information engineers come from?

If information engineering follows the same path as physical engineering, then we can expect information engineering professions to emerge. An interesting question is—where will they come from? One obvious source is the current computer system developers. This will inevitably lead to a segregation of information engineering from the rest

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

of system building. I realised quite early on that this is a natural divide. Business modelling and computer system design are very different kinds of activities.

When we were developing our re-engineering approach, the team distinguished between It (big I for information, small t for technology) and iT (small i, big T). Business modelling was It; computer system design was iT (interestingly, some years ago the Post Office renamed its IT department iT). If you think about it, the scope of most IT department's work does not include all the kinds of information technology used by the business. Information technology covers more than computers; it includes paper and ink technology as well as human brains. What most IT departments deal in is computing technology. So it is by no means a foregone conclusion that IT departments will supply the professional information engineers of the future.

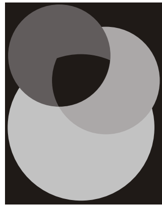
IT people are not the only candidates for information engineers. Areas of the business that have traditionally belonged to powerful professions will need the information engineer's skills. For example, matters relating to the accounting paradigm will have to be decided by people with information engineering skills. The accounting profession is unlikely to want ex-computer people to take over this task. To make sure that they do not 'miss the boat', accountants will have to develop the information engineering skills needed to manage the re-engineering of the accounting paradigm. This is more of an opportunity than a threat. The re-engineered paradigm will give accountants much more powerful tools than they currently have, with which to help managers run their businesses.

Wherever the professional information engineers are drawn from, whether it is the ranks of accountants, lawyers, systems analysts or business people, the roles that people play within businesses will change. New responsibilities will arise from the industrialisation of information. Old responsibilities will change or become irrelevant. Undoubtedly, when these responsibilities are shared out, information engineers will not only have some of the new responsibilities, but also take over some currently held by other professionals.

6 21st century information industries

Re-engineering entity oriented legacy systems into simpler and better systems is interesting and useful work. But it is using business objects to re-engineer the business that really offers really exciting opportunities for information engineers. It will put them at the centre of a business and social revolution, where they will be shaping the future. They will help change the way businesses work, shifting them from the paper-bound information processing institutions of the 20th century into the industrialised information industries of the 21st century.

© Copyright Chris Partridge chris.partridge@BORO Centre.com



Bibliography

This bibliography points you towards books that cover, in more detail, the main topics raised in the various parts of the book. These books generally have their own extensive bibliography that can be used to find additional material if needed.

Preface and Prologue

Only one other book I know of squarely faces up to the semantic problems of business modelling; that is, Bill Kent's, *Data and Reality; Basic Assumptions in Data Processing Reconsidered* (published by North-Holland in 1978, ISBN 0-444-85187-9). In it, he argues quite forcefully for a reference based paradigm of information and identifies a number of problems that this kind of paradigm needs to resolve; these are the kind of problems that this book addresses. He deserves a lot of credit for realising the importance of these issues in computing so early on.

A book I found useful was Joseph D. Novak and D. Bob Gowin's, *Learning how to learn* (published by Cambridge University Press in 1984, ISBN 0-521-31926-9). Novak has sections in this on a technique he calls concept mapping, which he believes can be used to map people's understanding. I found this useful when trying to understand what business modelling should be doing. A more technical book that looks at the structures of concepts is John F. Sowa's, *Conceptual Structures* (published by the Addison-Wesley in 1984, ISBN 0-201-14472-7).

© Copyright Chris Partridge
chrispartridge@BOROCentre.com

Part One—Re-Engineering the Computing EntityParadigm's Semantics

Thomas S. Kuhn introduced the idea of re-engineering in the sense it is used in this book, though he called it paradigm shifting. His book, *The Structure of Scientific Revolutions* (published by University of Chicago Press in 1970, Second Edition, ISBN 0-226-45803-0) is a useful introduction to what a paradigm is. In particular, it has many illuminating examples from the history of science. A flavour of what re-engineering is in a business context can be found in Michael Hammer and James Champy's, *Reengineering the Corporation—A Manifesto for Business Revolution* (published by Nicholas Brealey Publishing in 1993, ISBN 1-86373-505-4).

Two other useful books are Howard Margolis's, *Paradigms and Barriers* (published by University of Chicago Press in 1993, ISBN 0-226-50523-5) and *Patterns, Thinking and Cognition* (also published by University of Chicago Press in 1987, ISBN 0-226-50528-6). These give a useful description of both paradigms and patterns and how they link together.

A good description of the evolution of semantics is in Ian Hacking's, *Why Does Language Matter to Philosophy?* (published by Cambridge University Press in 1975, ISBN 0-521-09998-6).

Parts Two and Three—Shifting From the Entity Paradigm to the Logical Paradigm

One of the best ways of researching the shift toward objects is to read the more accessible of the original texts. These include:

- Aristotle's *The Categories*
- John Locke's *An Essay Concerning Human Understanding*
- David Hume's *A Treatise of Human Nature*
- Gottlob Frege's *Foundations of Arithmetic*
- Charles Peirce's *Reasoning and the Logic of Things*

Parts Four and Five—Object Semantics and Syntax

You could look at Willard Van Orman Quine's, *Word and Object* (published by the M.I.T. Press in 1964, ISBN 0-262-67001-1) and *Roots of Reference* (published by Open Court Publishing Co. in 1974, ISBN 0-87548-123-X); but these can be quite heavy going. An easier book to read is Mark Heller's, *The ontology of physical objects: Four-dimensional hunks of matter* (published by Cambridge University Press in 1990, ISBN 0-521-38544-X). It gives a good explanation of the notion of four-dimensional physical objects. David Lewis's, *Parts of Classes* (published by Basil Blackwell in 1991, ISBN 0-631-17656-X) has a discussion of the relationship between mereology and classes, including a discussion of the similarity between the sub-class and whole-part patterns generalised.

© Copyright Chris Partridge chris.partridge@BORCentre.com

INDEX

A–C

A

accuracy (and inaccuracy) - - 31, 57–58, 90, 99, 122, 163, 193, 303, 312
benefits of - - - 17–18
checking - - - 392
fruitful - - - 384
functionally richer systems - - - 17
idea of signs and sameness - - - 81–82, 132
identity - - - 158–159
information paradigm evolution - - - 80, 384
interchangeable parts - - - 18–20, 384
modelling the business - - - xvii, 182
physical vs. referential (conceptual) 18–20, 384–385, 411
reflecting the business xxiii–xxiv, xxvi, xxix, 12, 48, 52, 393
semantic - - - 119, 131, 268, 349–354
trend towards greater - - - xxix, 18
appearance and reality - - - 92
application level (of model) - 260–262, 265, 297, 332, 339, 349, 379, 384, 393–394
classes - - - 261
Aristotelian categories - - - 76, 263
Aristotle - - - xxvi, 16, 34, 40, 50, 57, 65
causes explaining an event - - - 176–177, 242, 403
pattern for motion - - - 70
syllogism - - - 200, 204
The Categories - - - 57, 66
attribute
attribute types - - - 42
change - - - 67, 174
close link to substance - - - 94
essential vs. accidental - - - 67–68, 82, 95, 129, 157
individual attributes - - - 41
natural way of seeing - - - xxv–xxvi
primary level semantics - - - 64–65
range of values - - - 121, 157
re-engineer - - - 278, 285
order - - - 320
relational - - - 82, 243
as states - - - 168
correlational 57–58, 102–103, 105–107
implicit - - - 278, 280, 286, 345
many to many - - - 361
re-engineering 57, 96, 100–107, 131, 156, 168, 348
secondary level semantics - - - 66
signed as an entity - - - 48

B

book–keeping - - - 399–400
Boole, George - - - 88
BORO Centre - - - xiv, xvii
BORO Methodology - - - xvii
BORO Program - - - xiv
business object meta–model - - - 395
business object modelling–training - - - 385
business object model–technology independent 394
business objects - - - xxiv, xxvi
accuracy - - - 20
key to - - - 24
missing the wider potential for - - - 398
require information engineers - - - 412
vs. concepts - - - 91
vs. system objects - - - 29
wider application - - - 398
business paradigm - - - xxix
accounting - - - 399–412
actively revising vs. describing - - - xix
benefits of re-engineering - - - xxix
changing radically - - - xvii
entity – re-engineering - - - 254
transformation of - - - 411
business patterns
natural stage to generalise - - - 382
salvaging investment in - - - xvi, 376
business re-engineering - - - xix

C

Cantor, Georg - - - 88, 98, 105, 114
cardinality pattern - - - 244, 246
change
See also four key types of things, changes happening to things

© Copyright Chris Partridge
chris.partridge@BOROCentre.com

logical bodies' problem with ---------- 127–128
needing a firm semantics ---------------- 138
not persisting over time ------------------ 27
substance paradigm's explanation ------- 67–69
transformed into an object ----------- 130, 169
unchanging substance ------------------- 69
chunking ------------------------------ 387–389
class
Cantor's definition ---------------------- 98
of four-dimensional objects --------------- 151
referring to a collection of extensions ------- 99
sign for ------------------------- 189–190
weak pattern for ---------------------- 114
class of classes
country full names --------------------- 286
country's weekends class --------------- 369
distinct and overlapping pattern objects 224–225
examples ---------------------------115–117
sign for ---------------------------- 196
unnatural idea ------------------------ 133
classification
class–member ------------------------ 120
single vs. multiple ---------------------- 78
static vs. dynamic ---------------------- 79
class–member
deducing distant-class–member sign ------ 204
hierarchy ---------------------------- 196
sign for ------------------------- 191–197
sign for distant- and nearest-class–member - 203
virtual distant-class–member signs -------- 205
Coad, Peter --------------------------xxiv
compacting
benefit of introducing early --------------- 382
classes – pattern for --------------- 248, 347
combined chunks ---------------------- 388
de-duplication ------------------------ 352
definition ---------------------------- xvii
encouraging environment ---------------- 122
examples -------------------------- 20–22
fewer, simpler components -------------- 378
increases in scope --------------------- xxx
metrics for --------------------------- 265
potential for conceptual economy --------- 122
purging redundant objects -------- 297, 326, 407
re-using the secondary hierarchy ----------73
simpler and functionally richer ------------20
with multiple classification ------------ 120–122
with patterns of extensions ----------223, 230
with the general naming model -----------355
with tuples -----------------------------107
with virtual tuples ------------------201, 205
complexity
building business systems ----------------14
conceptual patterns ----------------370, 379
cost of automating ----------------------302
fruitful patterns -------------------------383
not inherent ----------------------------379
re-engineering -- xviii, xxx, 20, 302–303, 379, 383
traditional development methodologies ------xv
computer technology xxvi–xxvii, 34, 45, 186–187, 412
conceptual economy – increased potential for --122
conceptual patterns -------------- 301–303, 308
consciousness – time-based -------- 179–180, 215
constructive nature of modelling -------------196
Cook, Steve ---------------------------- xxiv
Copernicus, Nicolaus ---------------------144
correlational attribute, *See attributes, relational, correlational*
current tuple ----------------------180, 316
sign for -------------------------------215

D

Daniels, John -------------------------- xxiv
data–process distinction ----- 27–29, 34, 36, 90, 216
derived object – sign for ------------ 227, 235, 249
Descartes, René -----------------------94–95
distinct and overlapping pattern --223–224, 230–231
distinct pattern -------------------------221, 228
use caution signing --------------------231
distorted
accounting pattern -----------------400, 409
address pattern -------------------345, 352
by a relational attribute ------------------58
by a tree structure ---------------------199

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

by paper and ink technology 356
by the entity paradigm xxvi, 90, 268, 286, 290, 300, 357, 399
deal pattern 409
manager's view of the business 410
process's view of the business 257
documentation – ephemeral 391
Dummett, Michael 90
dynaclass 179
dynamic classification 72, 79, 120
See also classification, static vs. dynamic
logical changes 129–130
mimicked by attributes 121
re-engineer into an event 169, 176
dynamic object 180, 186
implementing 180
sign for 214–215

E

economies of scope 386
Edwards, John xi
efficient cause, See *Aristotle, causes explaining an event*
egocentricity – general trend away from 144
Einstein, Albert 17, 143–144, 148, 173
Empedocles 80–81
encapsulation 173–175
entity
attributes belong to 41
business xxix
corresponds to substance 50
individual entities belong to entity types 41
natural type level 54
re-engineering order rules 258, 271, 320
relational 59, 361
entity life history diagram 242
entity paradigm
based upon paper and ink technology xxvii, 53
entity–based computer systems 254–255
framework 43
fundamental particles 40–42
general entity–attribute pattern 43–44
ignoring semantic problems 61
links to the file–record paradigm 45–49
problem with 25
re-engineering 254, 256
re-use 43
simplifying relationships 56–58
simplifying semantics 54–56
types restricted to a single level 51, 54
way of seeing xxvi
entity–attribute–relation model 60
event
Aristotelian causes 177
as a physical object 169
as three-dimensional objects 169
complex/encapsulated 169, 173–177, 181
distinct from body 170
encapsulation 174
object paradigm's shift to 169
explicit
business model xxix, 200, 260, 303, 309, 339, 355, 384
class–member tuple 194
data 257
implicit entity model xvi
implicit pattern 114, 117, 349–354, 361
mapping sense explicitly 90
pattern for change 67, 72, 131, 181
relationship link 58, 60, 103
re-usable patterns xxix
sense patterns 264
sign for patterns 186
states 157
strong reference principle 91, 351
time ordering patterns 178
extension
benefits of 94
collection vs. fusion of 99–100, 173–174, 227
Descartes 94
disconnected 173
four-dimensional 143–144, 157, 163, 216, 272, 276, 279, 312, 346, 373
instantaneous 140

© Copyright Chris Partridge chris.partridge@BORO centre.com

of bodies - 95
of classes - 98, 101, 152
of dynamic classifications - 130
of events - 170, 173–175
of states - 158, 165
of stuff - 150
of tuples - 104, 107, 153
of wholes and parts - 129
patterns for the connections between - 154, 220
same extension - 97, 100, 140–141, 147–148, 152, 160, 238
shift to timeless objects - 151–153
shifting from substance to - 91–96, 128, 149
three-dimensional - 143

F

file–record paradigm - 45–47
final cause, See *Aristotle, causes explaining an event*
flexibility - 28, 77, 83, 122, 165, 186, 300, 412
Fluidity - xiv
formal cause, *See Aristotle, causes explaining an event*
Forty Two Objects Ltd - xiv
forward engineering - xvi–xvii
four key types of thing - 31, 40, 61, 133
changes happening to things - 34, 127–131, 138
general types of thing - 32–33, 130
particular things - 31–32, 40, 130, 153
relationships between things - 33, 56, 131
framework level (of model) - 260, 262–265, 328, 355
example model - 270
meta–model - 260
Frege, Gottlob - 88, 98, 129, 171, 264
fruitful patterns - 143
beyond a project - 383
chunking - 389
from complex entity formats - 383
general lexicon - 263
ignoring - 10
more accurate - 384
prioritise construction of - 376–377, 383
re-use - 74, 303
functional decomposition - 380–381
fundamental particle(s) - 12–13
entity paradigm - 11, 40–42, 46, 54
information particle - 11–14
logical paradigm - 88, 108
object paradigm - 156, 169
physical particle - 11
re-engineering - 4, 10–14, 22, 101, 109
substance paradigm - 50, 74, 88
vs. complex business objects - 13
fusion - 147, 369
collection vs. fusion of - 99–100
creating components - 156, 163
encapsulated events - 173–175, 181
of extensions - 99
of stuff - 151
sign for pattern - 227, 235

G

general lexicon - 74, 262–264, 355
Aristotelian categories - 263
border - 264
example - 270
general types of thing, *See four key types of thing, general types of thing*
generalisation
compacting - 264
friendly environment - 122, 376, 380
introduce during business modelling - 381
less costly components - 378
metrics for - 265
potential for - 18–20, 72, 125, 383
produces compacting - 294, 378
re-use - 29
generalising
re-used patterns - 294
substance and attribute patterns - 72

H

happens–at (whole–part) tuple 172
happens–to (whole–part) tuple 171
Heraclitus of Ephesus 68
here event class 215
Huichol Indians 81–82, 132, 154, 384
Hume, David 92–94, 127

I

identity
- continuity 139, 143, 146, 148, 150, 159, 346
- more accurate way of seeing 154
- of a class over time 151
- of a tuple over time 153
- of disconnected physical bodies 148
- of logical bodies persisting through change 127–128
- of physical bodies 139, 145
- problem with whole–part pattern 129

increases in scope xxx, 125
- opportunities for generalisation 385–386

individual object – sign for 187, 189
Industrial Revolution 18, 411
industrialisation of information 20, 398, 411–413
information engineering profession 411
information paradigm 11, 31–34, 88, 411
- evolution 384
- major elements of 24, 35
- particles 11, 14
- separate evolution of semantics 35
- vs. business paradigm level xxviii–xxix

inheritance 197, 203, 209
- cardinality patterns 248
- distinct and overlapping patterns 222, 229
- logical class 113–114
- partitioning patterns 226
- secondary hierarchy 72
- single vs. multiple 77, 122

interchangeable parts 18–19, 384, 411
intersection pattern 226, 234

J

Jacobson, Ivar xxiv–xxv

K

Kent, William xii
Kepler, Johannes 21
Kuhn,Thomas K. 416

L

Laboratory for Applied Ontology xiv
lattice structure, *See structure – lattice and tree*
Leibnitz, Gottfried 133
Lewis, David 149
Linnaeus (Carl von Linné) 76
Locke, John 92–93
logical dependency – sign for 227, 235
logical paradigm
- changes 129–130
- class–member pattern 114–122, 193
- encouraging compacting 122
- fundamental particles 88
- halfway house 36
- new way of seeing 131–133, 384
- origins 88
- problem with changes 127–128
- sense element of 112
- shift to extension 91–95
- simplifying the substance framework 107–108
- strong reference principle 97
- super–sub-class pattern 112–114, 122, 197

M

managing large re-engineering projects 385–392
material cause, *See Aristotle, causes explaining an event*
meaning – Frege's analysis 89–91
membership information – modelling lack of 193–196

© Copyright Chris Partridge
www.partridge@BORO.com

mereology ---- 129, 159
  See also whole-part patterns
migration of business patterns ---- 393–395
model levels (framework, etc.) ---- 260
(modelling)2 model ---- 216–217
multiple classification, *See classification, single vs. multiple*

N

naming patterns – a general model for ---- 355
new way of seeing ---- 7, 286, 377
  bodies ---- 153
  business objects ---- xviii–xix
  changes ---- 181
  logical paradigm ---- 122, 131–133
  object paradigm ---- 138, 143, 163
Novak, Joseph ---- 415
now event class ---- 215

O

object paradigm
  characteristics of ---- 148
  enabling compact models ---- xxx
  generalisation friendly environment ---- 122, 126
  origins of ---- 35, 143
  preliminary definition of ---- xviii
object syntax ---- 182, 186, 191, 236, 239, 242, 248, 373
occupied class place
  as an object ---- 246
  constraints on tuple places ---- 208
  sign for ---- 207
Odell, James ---- xi
ontic commitment ---- 12
ontology ---- 12
O-O programming language ---- xx, xxviii, 10
  group attribute ---- 61
  halfway house ---- xxviii, 175
  single inheritance ---- 77
  static classification ---- 80
operational level (of model) ---- 260–262, 265, 273
operational level (vs. understanding level)
  See also understanding vs. operational
operational re-use vs. generalisation ---- 29
oral culture ---- 81–82, 154
overall stuff objects ---- 151
overlapping pattern ---- 221, 228
  confirming ---- 231

P

Pacioli, Fra Luca ---- 399
paper and ink technology 35, 131, 201, 279, 400, 412
  accounting paradigm ---- 399–407, 409
  address format ---- 356
  computers ---- xxvii
  entity paradigm ---- xxvi, 34, 40, 53, 57, 409
  object paradigm ---- 35, 186
  paper forms ---- xxvii
  paper's rows and columns ---- xxvi–xxvii, 45, 53, 107
paradigm
  as a holistic framework ---- 7
  business and information paradigm level ---- xxviii–xxix
paradigm shifts
  leading to radically different questions ---- 6
particular things, *See four key types of thing, particular things*
partitioning patterns ---- 225–226, 232–234
Pasteur, Louis ---- 9
pattern
  aspects of ---- 356
Peirce, Charles Sanders ---- 88
physical bodies
  as four-dimensional objects ---- 137
  disconnected ---- 150
place (Aristotelian category) ---- 75, 272
  benefits ---- 143
  replaced by Descartes' extension ---- 94
  two things in the same ---- 140, 142, 148
Plato ---- 80
popular (general) consciousness ---- 80, 153, 181
Porphyry ---- 76

© Copyright Chris Partridge
chris.partridge@BORO Centre.com

power class - 307
primary attribute
  problems in the logical paradigm - 96
  transformed into a logical class - 98, 102
primary substance
  as underlying matter - 64–65
  re-engineer - 95–100
Ptech - xi

Q

Quine, Willard van Orman - xi, 99–100, 143–144, 149, 416

R

Ray, John - 76
redundant patterns - 210, 310, 315–316, 334, 384
  classifying - 296–297, 326–328, 331, 336
  example - 347
  purging - 297
  recording - 250
  sign for - 250
re-engineer
  attributes
    relational - 278, 282, 287, 289
  benefits - xvii, xxix, 17–22
  complex patterns - 20
  conceptual patterns - xvi
  data and process - 257
  existing systems - xv
  framework - 270
  paradigms - 4
  rules for ordering elements - 258, 271
  systematic approach - 256, 259, 270
  the same object twice - 352–353
  things in the business - 36
  underlying business - xv
reference - 188, 279, 373
  changing - 141
  diagram - 197, 202, 276–277
  need to determine - 90
  of classes - 152
  one name referring to two objects - 354
  problem with relational attribute - 102
  unchanging - 149
  weak - 100
relational databases - 107
relationships
  See also four key types of thing, relationships between things
  between more than two objects - 105
  many-to-many - 58, 106
  problem with attributes - 103
re-usable–business objects - xvii, xxi, 17, 411
re-use
  accuracy and interchangeable parts - 18
  Aristotelian categories - 74
  complex patterns - 302
  country/region patterns - 294–297, 320, 322, 339
  explicitness and accuracy - xxix
  general lexicon - 263
  high levels of - xvi
  metrics for - 265
  operational vs. understanding level - 29
  patterns 125–126, 209, 302–303, 344, 354, 376–377, 384
  potential for - xviii, xxi, 20, 29, 64, 72, 83, 312, 315
  whole–part patterns in space-time - 149
  within the secondary hierarchy - 72–73
  working down entity framework - 43
REV-ENG method - 253–256, 276, 297
  stages - 257–259
reverse engineer - 256
revisionary vs. descriptive approach - xix
Russell, Bertrand - 71

S

sameness, *See identity*
secondary attribute
  hierarchy - 51–52, 56
  independent - 52, 56, 74
  inheritance - 72

© Copyright Chris Partridge chris.partridge@BORO Centre.com

making dependent - - - 56
re-engineering - - - 101
secondary substance
hierarchy - - - 51, 56, 72–74, 113, 232, 410
re-engineer - - - 100
semantic re-engineering route - - - 35
sense and reference - - - 89–91, 171, 264
signs as objects – modelling the model - - - 216–217
single classification, *See classification, single vs. multiple*
Sowa, John F. - - - xii
space-time - - - 146, 148, 172–173, 181
amalgamation of space and time - 149, 154, 167, 373
connected in - - - 150
Einstein's notion of - - - 143
map - - - 145, 170–172
temporal slice
example - - - 372
time objects - - - 362–365
state tuples – tuples with state object places - - - 168
states
as physical body objects - - - 156
components as fusions of - - - 163
consequences of timelessness - - - 163–165
contiguous - - - 167
distinct - - - 161
hierarchy - - - 159–162, 236–237
identity - - - 158
life history - - - 239, 241
objects that are states of themselves - - - 164
overlapping - - - 161, 238
re-engineer from dynamic classifications - - - 169
substance paradigm's view - - - 157
time ordered connections - - - 165
time ordered patterns - - - 239
static classification, *See classification, static vs. dynamic*
stored information – current way of seeing - - - 61
strong reference principle - - - 91, 133, 231
and extention - - - 97, 99–100, 122
applying - - - 150, 262, 351–352
breaches of - - - 103, 130, 148, 352
mapping objects directly - - - 118
strong sense of object - - - 127
structure – lattice and tree - - - 30, 335–336
accounting ledger - - - 410
Aristotlian categories - - - 75–76
functional decomposition - - - 381
life history - - - 242
logical paradigm - - - 114, 120, 122
object paradigm - - - 162
O-O programming language - - - 77
super–sub-class hierarchy - - - 199
sub-part - - - 220
sign for - - - 211
substance paradigm
David Hume's scepticism - - - 93
John Locke's qualms - - - 92
relationships - - - 56–58
simplified into entity paradigm - - - 50
superior semantics - - - 52
substance semantics
primary level particles - - - 64
secondary level particles - - - 66
super–sub-class - - - 112–113, 159, 306
confused with class–member pattern - - - 132, 205
deducing descendant–sub-class signs - - - 200
inheritance - - - 199
new way of seeing - - - 384
sign for a tuple - - - 197–198
sign for a tuples class - - - 206
sign for child– and descendant–sub-class - 199–200
virtual descendant–sub-class signs - - - 201
super–sub-class hierarchy - - 112, 162, 200–203, 326, 347, 370, 405, 407–410
class membership inheritance - - - 203
natural structure - - - 199
non-circular structure - - - 202

T

tables paradigm - - - 46
temporal–whole–part - - - 236–237, 242, 315, 369
sign for - - - 237

testing conceptual correctness - - - 122
thing and stuff patterns - - - 149–151
things in the business - - - 50
  See also four key types of thing
  business model reflecting - - - 12–13
  forcing the re-engineering on - - - 24
  identifying similar - - - 30
  ignoring - - - 27–29
  problems identifying - - - 25–26
  re-engineering object semantics for - - - 35
thought experiment - - - 14–17, 64, 115, 130, 139–142, 145–148, 158
  Descartes' - - - 95
  Einstein's - - - 17
  Hume's - - - 128
  Zeno's - - - 71
Time - - - 243
time dimension– translated into a spatial dimension 145
time objects - - - 172–173
  See also space-time, time objects
time ordering - - - 239–240, 242–243
timelessness 148–149, 152, 154, 163–165, 179–181, 214
time-line - - - 147, 164, 170, 181
time-slice - - - 147, 158, 164, 170–172, 179, 181, 215
Tree of Porphyry, *See Porphyry*
tree structure, *See structure – lattice and tree*
tuple
  of four-dimensional objects - - - 153
  sign for - - - 206–209
tuple place - - - 207
tuples class - - - 207–208
  cardinality patterns for - - - 243–248
  super–sub-class hierarchy - - - 209

U

understanding
  as meaning - - - 88
  vs. operational level - - - xxvi, 29–30, 40, 138

V

validation system - - - 122, 309, 392, 395
Venn diagram - - - 59, 160–161
Venn, John - - - 88

W

webby pattern - - - 203, 308, 355, 377
well–defined scope - - - 377
Whitney, Eli - - - 18–20
Whitworth, Joseph - - - 19–20
whole–part
  child– and descendant–part - - - 212
  deducing descendant–part signs - - - 213
  hierarchy - - - 212
  signs for - - - 210
whole–part pattern - - - 149, 171–173, 224, 310, 350
  and causes - - - 178
  and identity - - - 129
  and overlapping pattern - - - 221
  and states - - - 158–159
  and super–sub-class pattern - - - 197, 210–211
  conceptually more accurate - - - 131, 351–352
  extending spatial to spatio-temporal patterns 147, 149
  for stuff - - - 151
  happens–at pattern - - - 172
  implicit - - - 350

Y

Yourdin, Ed - - - xxiv

Z

Zeno of Elea - - - 71, 130, 175

@Copyright Chris Partridge
chris.partridge@BORO Centre.com