Domain Architectures

Models and Architectures for UML Applications

Daniel J. Duffy
Datasim Education BV, Amsterdam, Netherlands

John Wiley & Sons, Ltd

Domain Architectures

Domain Architectures

Models and Architectures for UML Applications

Daniel J. Duffy
Datasim Education BV, Amsterdam, Netherlands

John Wiley & Sons, Ltd

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester,
West Sussex PO19 8SQ, England

Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit our Home Page on www.wileyeurope.com or www.wiley.com

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 33 Park Road, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 22 Worcester Road, Etobicoke, Ontario, Canada M9W 1L1

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data

Duffy, Daniel J.
Domain architectures : models and architectures for UML applications /
Daniel J. Duffy.
p. cm.
Includes bibliographical references and index.
ISBN 0-470-84833-2 (alk. paper)
1. Computer software—Development. 2. Business—Data processing. 3. UML
(Computer science) I. Title.
QA76.76.D47D84 2004
005.1—dc22 2004002216

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 0-470-84833-2

Typeset in 10/12.5pt Times by Laserwords Private Limited, Chennai, India
Printed and bound in Great Britain by Biddles Ltd, King's Lynn
This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

Contents

Preface

The last two decennia have witnessed many advances in the area of software development. The advent of object-oriented programming languages and modelling languages such as Unified Modeling Language (UML) has increased our ability as developers to design and realize large and enterprise-wide software systems. However, software engineering, as a discipline seems to be lacking in its support for reference models that can be used in order to help developers create new systems quickly and efficiently. The software development process is still a very context-sensitive and idiosyncratic process. Whereas disciplines such as chemical engineering and mathematics have developed domain models for a range of problems, the IT industry is in general lacking in such models. Software development tends to be a very personal experience and in many cases how a system is to be developed is a product of a single person's insights. This is a potentially dangerous state of affairs because there is no guarantee that the resulting model reflects the problem domain well.

This book introduces a number of so-called models (we call them domain architectures) that act as 'cookie-cutters' or reference models for more specific real-life applications. Working with domain architectures demands a shift in thinking because when designing a new software system we try to categorize it as an *instance system* of one or more domain architectures. Having done that we can reuse and specialize the requirements, viewpoints and generic architecture to the specific systems. This results in massive reuse at the architectural and design levels while the risk of failure is reduced because the reference models in this book are based in real-life applications and experience. They have been used on real projects with real customers.

The reference models can and should be used in much the same way as people reason about the world around them. This is the Ausubel subsumption theory: when developing software systems we relate new knowledge to relevant concepts and propositions we already know.

ACKNOWLEDGEMENTS

Although many of the results in this book are based on my own work it would have been impossible to write this literature without the support and feedback from many organizations and individuals that I've come in to contact with during the last 25 years. First, I would like to thank Datasim's customers who have attended our analysis and design courses since 1992. It is impossible to name them all and we wish to thank them for their feedback. Particular thanks goes to the following individuals (in random order): Paul Langemeijer, Hans Plekker, Henry Rodenburgh, Marten Kramer, Wim van Leeuwen, Robert Demming, Adriaan Meeling, Martijn Boeker, Vladimir Grafov, Jeff Keustermans, Teun Mentzel, Ilona Hooft Graafland and many more. For all others who have had some form of involvement with me throughout the years, many thanks to you as well.

This work has been importantly influenced by several major sources. Firstly, Michael Jackson who is the originator of Problem Frames and who sparked a number of ideas that led to *Domain Architectures.* Secondly, the researchers in the Design Patterns movement (too many to mention) who realize that software development is a repetitive process and that a multitude of patterns can be discovered, documented and used in many different contexts. Finally, to Bjarne Stroustrup, the inventor of C++ for his efforts in making OO more accessible to a wide audience. A word of thanks is due to the 'three amigos' Booch, Jacobson and Rumbaugh for their hugely successful efforts in making UML the *defacto* standard for object-oriented analysis and design.

A special word of thanks is due to the staff at Wiley in Chichester who had infinite patience with me.

Finally, I wish to thank my family, Ilona Hooft Graafland and Brendan Duffy for their patience during the preparation of this book. They probably wondered when the book would finally be finished. Hopefully as I write this sentence...

Daniel J. Duffy
Datasim Education BV, Amsterdam
February 2004
dduffy@datasim.nl

PART I

Background and fundamentals

1 Introducing and motivating domain architectures

'Architecture is born, not made—must consistently grow from within to whatever it becomes. Such forms as it takes must be spontaneous generation of materials, building methods and purpose.'

Frank Lloyd Wright

1.1 WHAT IS THIS BOOK?

This book describes how to analyse large enterprise systems. In particular, we define a process that maps high-level business concerns and business processes to artefacts in the Unified Modelling Language (UML). This is one of the first books that explicitly links the business world with the IT world. We achieve this end by first of all providing the reader with a number of ready-made reference models that he or she can use as a basis for specific applications. These reference models are called domain architectures in this book. Second, and just as important, we adopt, adapt and (hopefully) improve current understanding on how software systems are analysed and designed. In particular, our interest is in creating flexible and maintainable software systems using proven technology. We document the products of our endeavours using the visual notation in UML. This adds to the usability of our process because UML is a *de facto* standard and we shall use it as a universal communication language.

A domain architecture is a reference model for a set of applications sharing similar functionality, behaviour and structure. It describes the essential features in some business domain. In this book we introduce five major domain architecture types. These types describe recurring themes in software development. We could loosely define a domain architecture as a pattern that describes structure, functionality and behaviour in the earliest stages of the software lifecycle. We discuss generic architectures for *management information*, *process control*, *access control*, *manufacturing* and *tracking* systems. We devote a chapter to each of these five architecture types. Specific instances of these architectures occur in real-life software development projects and we describe a number of such instances in this book.

Our domain architectures are models in the so-called *problem domain* (roughly speaking, the domain of the sponsor and user of the system) while design and system patterns are models for the *solution domain* (the domain of the object-oriented analyst and designer). Domain architectures fill the gap between the business and the IT worlds. In short, we provide the reader with a set of documented reference models that he or she can specialize to produce analysis artefacts for specific instance systems. We devote six chapters to show how this specialization works; each chapter deals with a well-known application.

1.2 WHY HAVE WE WRITTEN THIS BOOK?

The main reason for writing this book was to describe and document a number of recurring patterns and models that we have discovered in software projects. These models describe a set of applications having similar structure, functionality and behaviour. Each model is documented in handbook form and the reader can use the handbook to 'clone' specific applications. We are primarily interested in large enterprise systems because we have seen that traditional object-oriented technology is not suitable as the driving force for systems of this magnitude. The old maxim of 'looking for the objects and the rest will take care of itself' is not applicable in these situations, in my opinion. It becomes very difficult to manage the object networks that result from this approach. Furthermore, it would seem that the levels of reusability with the object paradigm are quite low; we are interested in reusability at system and architecture level. For example, a system that we have already analysed and designed can be used as a first approximation to some new system that we suspect is similar to it in some way.

Another reason for writing this book is that we wish to integrate the world of business processing modelling, requirements analysis and UML into a coherent whole. In particular, we create a well-defined and hopefully seamless path that maps high-level requirements and business concerns to analysis artefacts such as class diagrams, interaction diagrams and other artefacts in UML. We are not aware of such a process in the literature. This is why we have created the Datasim Development Process (DDP) that *does* provide a step-by-step plan to get you to the UML finish line. The DDP describes the following phases: business processing modelling, architecture, requirements analysis, object-oriented analysis in UML and design. It is a lightweight process and can be used by novice developers. We give an introduction to the DDP in Chapter 3. Each of these topics is discussed in this book with the exception of design.

Finally, we have written this book because we wish to improve the communication lines between customers and developer.

1.3 FOR WHOM IS THIS BOOK INTENDED?

This book is aimed at software architects, (structured and object-oriented) analysts and other software specialists who are involved with the creation of stable architectures for medium and large systems. We describe a step-by-step process that takes the system goals and business processes and maps them to a software architecture consisting of a network of interrelated systems and classes. We describe how to decompose the systems into subsystems and classes. To this end, we use a subset of the UML syntax that is sufficiently rich to allow a detailed design. Thus, this book is also suitable for those developers who analyse and document systems using UML and who wish to integrate them with the 'Gang of Four' (GOF) and system (POSA) patterns. In general, this book focuses on that part of the software lifecycle between business process modelling and object-oriented analysis and it provides a stable architectural framework on which to place customer requirements.

This book is also of interest to analysts who are involved in requirements determination activities and who need to align functional and non-functional requirements with architectural models.

1.4 WHY SHOULD I READ THIS BOOK?

We think that this is one of the first books that attempts to use UML for large enterprise systems. It provides the reader with tools, concepts and advice on how to map the business world to the IT world. We use standards wherever possible, such as UML, standard architectures, business process modelling and patterns. We also improve these standards whenever necessary.

This book should help you produce stable, understandable and high-quality software systems. New key features that we see as important are:

- A defined software process from A to Z
- Integration of proven technology with our software process
- Ready-made reference models that you can use in projects
- Using the UML artefacts in a predictable and usable way
- Reference models that are based on real-world experience
- Software development as a continuous improvement process.

1.5 WHAT IS A DOMAIN ARCHITECTURE, REALLY?

A domain architecture is a reference model for a range of applications that share similar structure, functionality and behaviour. It is not an application as such but

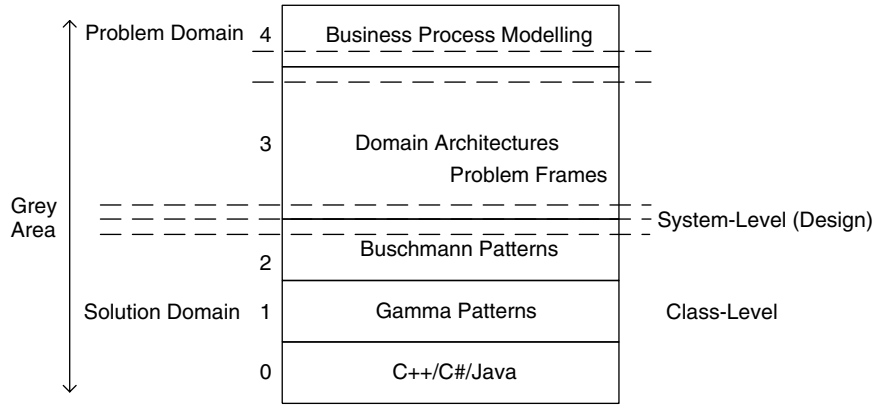

Figure 1.1 Taxonomy of domain architectures.

is in fact a meta model that describes how more specific instance systems ('real applications for real customers') are created. A domain architecture subsumes much of the current techniques in software development and is positioned between a number of other methods as shown in Figure 1.1. This diagram should help the reader position domain architecture in the galaxy of methods. For a discussion of the methods in Figure 1.1 and how they have influenced our work, we refer to Appendix 3 at the end of this book. We thus see that our work and results are positioned between the problem domain and the solution domain. Once you have determined in which domain architecture type (or types) your application falls, you can then use the ready-made templates to map the business artefacts to UML artefacts. You have a foot in both camps, as it were. This can't be a bad thing.

We discuss five basic forms and one 'composite' form in this book:

- *MIS* (*Management Information Systems*): Produce high-level and consolidated decision-support data and reports based on transaction data from various independent sources.
- *PCS* (*Process Control Systems*): Monitor and control values of certain variables that must satisfy certain constraints.
- *RAT* (*Resource Allocation and Tracking*) *systems*: Monitor a request or some other entity in a system. The request is registered, resources are assigned to it, and its status in time and space is monitored.
- *MAN* (*Manufacturing*) *systems*: Create finished products and services from raw materials.
- *ACS* (*Access Control Systems*): Allow access to passive objects from active subjects. They are similar to security systems.
- *LCM* (*Lifecycle Model*): A 'composite' model that describes the full lifecycle of an entity; an aggregate of MAN, RAT and MIS models.

We realize that some of the above names may be confusing to some readers, or that readers may infer some wrong conclusions based on those names. For example, the author once spoke to a software engineer who developed reporting functionality in the telecom industry. For example, the system to be developed should create invoices at different levels. The author suggested analysing the system as a MIS category. The response from the engineer was 'Oh no, my system is technical!'.

In order to fit domain architectures in a hierarchy that improves understandability and discovery we create a semantic network model as shown in Figure 1.2. This is an application of well-known techniques in cognitive psychology (Eysenck and Keane 2000). There are three main categories:

- *Superordinate level* (level 1): This is a high level of abstraction in a conceptual hierarchy and corresponds to a very general type. In our case we have categories for object creation, aligning objects in some structure, and modelling object behaviour. The basic assumption is that these three categories model the lifecycle of *any* object in any phase of the software lifecycle.
- *Subordinate level* (level 3): This is the lowest level in the conceptual hierarchy and contains specific objects and systems. This is, for example, where all the specific applications that we discuss in Part III of this book are to be found.
- *Basic level* (level 2): This is an intermediate level of abstraction in the conceptual hierarchy and fits between the superordinate and subordinate levels. This is the level where the current domain architecture types are placed.

The reader can use the hierarchy in Figure 1.2 as a navigational aid. For example, he or she can try to place a system to be developed as a subordinate level system under a more general basic level category. For example, a system that produces invoices

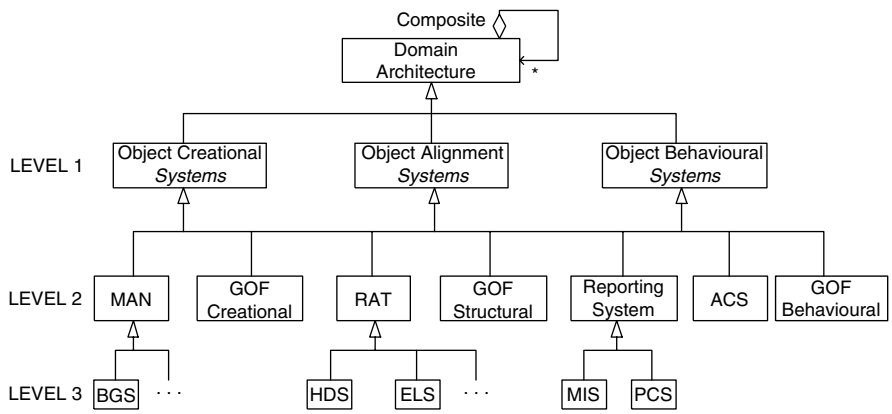

Figure 1.2 Hierarchy of patterns and reference models.

on mobile telephone usage is seen as an instance of an Object Reporting System. This can be refined by modelling the system as an instance of a MIS category.

The domain architecture types are fully documented in Part II. The documentation style is similar to how the patterns community document their design and system patterns (see GOF 1995, POSA 1996). The structure is roughly as follows ('DA' stands for Domain Architecture):

- *Motivation section*
  - — Background to DA and its history
  - — Motivational examples (one small example, one larger example)
  - — The general applicability of the DA
- *Functional modelling, architecture and structure*
  - — The goals, processes and activities for the DA
  - — Context diagram, system discovery and system decomposition
- *Behavioural modelling*
  - — Stakeholders and their viewpoints
  - — Requirements and use cases
- *Object-Oriented Analysis* (*OOA*)
  - — Class architecture UML classes in the DA
  - — Use cases (and possibly sequence diagrams)
- *Extensions to the DA*
  - — Specializations of the DA
  - — Using the DA with other systems (as client, server, collaborator).

Each of the artefacts in the above list is documented using UML whenever possible.

1.6 THE DATASIM DEVELOPMENT PROCESS (DDP)

This book would not be complete if we did not pay some attention to the actual process of mapping high-level concepts and requirements to lower-level artefacts that we use in UML. We describe a step-by-step *constructive* process that actually shows you *how* to do this. This topic is discussed in Chapter 3. In particular, we develop processes for the following important phases:

- Architecture discovery and decomposition
- Requirements analysis
- Object-oriented analysis.

Furthermore, we discuss the integration problems when we wish to align the artefacts of the different phases. We note that it is possible to use the DDP as described here without having to refer to domain architectures at all! This makes the book useful

for those readers who do not have the time to study the domain architectures in detail but who will still want to use a solid software process.

A full treatment of project management issues for DDP is outside the scope of this book.

1.7 THE STRUCTURE OF THIS BOOK

This book consists of four main parts and 18 chapters. In Part I (Chapters 1–4) we motivate domain architectures by describing what they are and how to use and document them and by giving examples. In Part II we discuss and document the six basic forms of domain architecture. We discuss these categories in Chapters 5–10. Each chapter in this part is documented using a standard template structure. Part III analyses six instance systems of the domain architecture types from Part II and these instances are described in Chapters 11–16. The cases are well known in the software literature or have been distilled from real-life software projects in the past. Finally, Part IV contains two chapters that summarize the similarities and differences between the different domain architecture types and how to use them in your software projects.

The chapters in Parts II and III have been written in handbook form. We have written several chapters and appendices to help the reader understand the rationale behind the structure of the book.

An important feature in this book is that we resurrect information models that have been used for many years to help systems analysts design software systems and we have dressed them in a more object-oriented suit. In this way we hope to save these useful models for future applications.

How do we use this book? We attempt to answer this question by posing a number of standard questions that we hope will encompass those that readers might ask, and then directing the reader to the most appropriate chapters:

- *Question*: Where can I find a summary of domain architectures and their instance systems?
  Answer: Chapter 2.
- *Question*: Where are domain architectures and UML artefacts documented?
  Answer: Chapter 4.
- *Question*: Where are domain architectures and their instances documented in detail?
  Answer: Parts II and III of this book. Furthermore, Chapter 17 summarizes the domain architectures and the client/server relationships between them and their instance systems.
- *Question*: How do I start?

Answer: Chapter 18 discusses the different ways of tackling software projects. We develop a number of practical techniques to help you get up to speed.

- *Question*: Does this book help me to develop interviewing skills?
  Answer: Yes. Please read Appendix 1.

1.8 WHAT THIS BOOK DOES *NOT* COVER

First, this book is not a UML tutorial and we assume that the reader has experience of UML syntax. Second, this book is not concerned with design issues or design patterns, although the artefacts can be mapped to the GOF and POSA patterns. This topic is beyond the scope of this book.

Finally, this book does not deal with component technology, although it is possible to first model domain architectures using this technology and then create systems in which the component and object technologies dovetail. We thus see the object paradigm evolving into something to adapt to the realities of the modern software development environment.

2 Domain architecture catalogue

'Any problem in geometry can easily be reduced to such terms that a knowledge of the lengths of certain straight lines is sufficient for its construction. Just as arithmetic consists of only four or five operations, namely, addition, subtraction, multiplication, division and the extraction of roots ... so in geometry, to find required lines it is merely necessary to add or subtract other lines.'

René Descartes, *The Geometry*

2.1 INTRODUCTION AND OBJECTIVES

This chapter summarizes the major domain architectures that we discuss in this book as well as several instance systems in each category. It has been included mainly for reference purposes and it may be skipped on a first reading. The added value of this chapter is that the reader can use it as a kind of *Yellow Pages* to help find applications that are similar to his or her current applications. This topic will be further developed in Chapter 18 when we develop some guidelines to help us discover the structure and functionality of an application by comparing it with known applications. This is called *analogical reasoning*.

In short, this chapter is a quick reference to the domains and instance systems in this book. It is *not* meant to be read from start to finish but gathers all the domain architectures and their instances in one place for perusal and reference.

We assemble all the domain architectures, their instances and exercises in one place. This is where you can begin before you consult the chapters in Parts II and III because your specific application will hopefully be analogous to one or more categories or instances. You can use this chapter as you would a real catalogue, namely by browsing until you come across something that interests you.

One of the assumptions in this book is that each new application is similar to an instance of some domain architecture (or category as we shall sometimes call it). In order to help the reader determine which category is 'best' we have introduced domain architectures and their instances. A domain architecture encapsulates the

assumption that all applications in a given domain have a central description that then stands for all of them. An application is a member of the category if there is a good correspondence between its attributes and that of the more general architecture. For example, we suggest that the following applications are good prototypes for their respective domain architecture types:

- Manpower Control (MPC) is a prototype for MIS
- Home Heating System (HHS) is a prototype for PCS
- Order Realization System (ORS) is a prototype for RAT
- A compiler is a prototype for MAN
- The Reference Monitor model is a prototype for ACS
- The Product Lifecycle Model (LCM) is a prototype for lifecycle and composite models.

We note that the domain architectures themselves may be used as prototypes for new systems. The disadvantage is that it may not be possible to fit your application to a prototype. Then we must resort to the so-called *exemplar-based view*. Rather than working from an abstraction of the central tendency of all the instances of a category, people simply make use of particular instances or exemplars of the category (Eysenck and Keane 2000). For example, some exemplars in the RAT category are:

- Help Desk System (HDS)
- Order Realization System (ORS)
- Call handling systems.

People relate to instance systems more quickly than to abstract reference models. However, you have a choice! Basically, we choose between one representative application and several exemplars as the target when using analogical reasoning to help us discover the architecture and behaviour of the system under discussion (SUD). A prototype approach assumes that there is a single 'best' system that is representative of all other systems in the same category, while the exemplar-based view contradicts this assumption. Instead, we need several instance systems to help us discover system structure and behaviour. We discuss prototypes and exemplars in more detail in Chapter 18.

In order to focus on the problem at hand we attempt to define the major defining features of a system or domain architecture type. We reduce the scope by focusing on the initial business and analysis phases of the software lifecycle. To this end, we think that the following set will provide a good starting point:

- C1: What are the main goals of a system?
- C2: What are the main core processes and key systems?
- C3: What are external stakeholder systems and their viewpoints?
- C4: What are the most important use cases?

For point C3 we are using the term 'viewpoint' as defined in Sommerville and Sawyer 1997, for example. This is a term that we use in the very early stages of the software lifecycle to denote perspectives taken by different system stakeholder groups. We give a fuller definition in Chapter 3.

Note that these questions are mainly of relevance during the early analysis phases. Unfortunately, these are the problems that tend to get glossed over in large systems in the rush to UML nirvana.

Answers to Questions C1 to C4 should be forthcoming as soon as possible and before commencing with object-oriented analysis. The risks are great if you gloss over or 'fudge' these issues.

2.2 MANAGEMENT INFORMATION SYSTEMS (MIS) (CHAPTER 5)

Management Information Systems produce decision-support information that can be used as input to other systems such as data mining, statistical analysis and executive information systems. The motivation and vocabulary for MIS date from the 1960s and 1970s (see Section 5.2 for a description) and we have subsumed the corresponding models under an object-oriented framework. The core process is to produce decision-support information based on low-level or transaction input data from various sources. The output is presented in various ways. The main activities in the core process are:

- Register, validate and create basic transaction objects
- Consolidate and aggregate transaction objects
- Present, dispatch and report on consolidated data.

The MIS category subsumes many industrial, technical and administrative applications. The word 'management' should not be interpreted as just being of relevance to business domains. It has a broader scope.

We now give a brief discussion of the MIS instances in this book. These are useful for reference purposes.

1. Simple Digital Watch (SDW) (Section 5.3.1)
SDW accepts pulses (one pulse every second). The pulses are buffered until the number of pulses reaches 60. Then the current time (in hours and minutes) is (re)calculated and the new time is displayed on an output panel. SDW can be configured on a 12-hour or 24-hour time regime.

SDW contains a panel consisting of two buttons for setting the time. We see the current version of SDW as an instance of MIS for a number of reasons. First, low-level data (seconds) is registered and merged to high-level data (time, that is hours and minutes). Second, we need different kinds of merging and consolidation

algorithms to create this high-level data. Finally, this data is displayed on a LED and is in fact decision-support information (for example, it's time to get up!).

2. Instrumentation and control systems (Section 5.3.2)
This technical problem occurs in many industrial applications. Nonetheless, it can still be modelled as a MIS instance. All instrumentation and control systems convert physical quantities and display the converted information on a recording device or recorder. The recorder stores the results of the measurements. The difference between a recorder and a display is that the former produces a permanent (persistent) record while the latter shows the results in volatile form. In general, we use a database system to store results permanently while displays can be implemented by some kind of light-emitting diode (LED) display or a graphics screen.

3. Noise control engineering (Section 5.10.1)
This is another technical example of MIS. In this case we imagine a petrochemical plant consisting of various noise-producing equipment. The equipment is grouped into various areas, clusters and assemblies. The system calculates noise levels (in decibels) in the petrochemical plant and the main goal is to produce high-level decision information for health inspectors and local authorities. Typing reporting functions are:

- What are the noise levels at various distances from the plant?
- What are the noise levels caused by various assemblies?
- Compare actual noise levels with levels allowed by the law.

4. Reporting activities in the 'Rent-a-machine' system (Section 10.3.1)
This system is an instance of a lifecycle model (LCM) and its core process is the tracking of a customer request from A to Z. The lifecycle system has the following subsystems:

- Reservation: create the basic customer order (MAN instance)
- Contracting: create a binding contract between the customer and garden centre (RAT instance)
- Reporting: marketing and sales information on rented equipment (MIS instance).

This last system is an instance of a MIS because we are interested in monitoring the status of each rented machine. Some typical questions to be answered are:

- Report on the usage levels for a given group of machines
- How many machines need repair?
- What is the garden centre's profit in the last six months?

5. Manpower Control (MPC) system (Chapter 11)
An engineering company works on projects for internal and external customers. A project represents the sequence of activities that are executed by the different departments. The project is deemed to be complete when each activity has been completed. An employee works on several activities in a project and is allocated a certain number of hours and other resources for each activity. Each department has its own area of expertise.

Departments are grouped into divisions. Customers are the sponsors of external projects. The resources (in this case hours) are allocated to departments and employees on a project basis.

A system needs to be built that registers, validates and monitors project resource usage (in this case man-hours). In particular, the following requirements must be supported in the system:

- MPC processes transaction data (resource usage) once per period (e.g. per month)
- Resource utilization must be monitored
- Status reporting capabilities must be available to stakeholders.

We model this problem as a MIS instance because we wish to monitor project status. We could have modelled this as an instance of RAT (a kind of time-tracking) but the fit may be less clear. For example, RAT does not say much about high-level reporting and consolidation algorithms, while MIS does.

6. Portfolio management
A financial instrument (or instrument for short) is an entity that can be traded in the marketplace. Examples of instruments are cash, equities, equity options, index options, bonds and futures (see Jarrow and Turnbull 1996). We can create MIS systems for a given instrument type, for example:

- Calculate the value of the instrument
- Get the instrument history (historic prices of a selected instrument).

Thus, we can monitor instrument behaviour using a MIS, albeit at the level of a single instrument. You could also model it using a RAT, in which case you have a competing solution.

A portfolio is a set of instruments. We now wish to monitor the performance of the portfolio so that we can generate an optimal return on the portfolio. In particular, we wish to calculate a strategy of buys and sells and we achieve this by using simulation techniques, for example using the Monte Carlo method (see Wilmott 1998).

The main reporting functions in a portfolio system are:

- Get portfolio history (display the historic values in a graph)
- Calculate performance (sum performance of instruments in portfolio)
- Calculate the Value At Risk (VAR) of the portfolio.

Some other examples of MIS applications are discussed in Section 5.10.

2.3 PROCESS CONTROL SYSTEMS (PCS) (CHAPTER 6)

Process Control Systems model differences between the scheduled and actual values of certain attributes and variables in a system. The main objective is to keep these two sets of values within close proximity to each other. The system monitors the values and corrective or control action is taken if the values drift too much away from each other. Process control systems are well understood and we discuss the basic model and its variants in Sections 6.4 (reference model and main components) and 6.4.2 (control engineering). We subsume these models under a domain architecture that we call PCS. The core process in PCS is the activation of actuators that ensure that the system returns to equilibrium. The main concurrent activities are:

- Monitor disturbances and other changes in the system's environment
- Activate actuators to bring the system to a steady state
- Monitor and control the life of the system (for example, via an operator panel).

As we shall see in Chapter 6, we map each activity to a subsystem that contains the necessary structure, functionality and behaviour to approximate the corresponding activity.

Process Control Systems occur in many industrial, real-time and business domains. In fact, any application where part of the problem is to monitor and control disparities between actual and ideal values of some variable will almost certainly be a candidate for one or more PCS instance systems.

1. Water level control (Section 6.3.1)
The water level in a tank must be monitored and controlled. If the water level is too high we open a valve to let the water escape, while if the level is too low we close the valve and start a pump motor that consequently delivers water to the tank in order to increase the level.

2. Bioreactor (Section 6.3.2)
This problem is similar to the previous problem. Instead of monitoring water level the bioreactor system monitors and controls the temperature of the water (or other liquid) in the tank. An example of a bioreactor system is a sewage plant.

Real applications monitor several variables such as temperature, pressure, pH level and percentage of oxygen in the liquid. We then speak of a *multi-parameter* problem.

3. Barrier options (Section 6.3.3)
In this case we are interested in situations where stock price fluctuates between critical 'barrier' values. Upper and lower barriers may be defined and stock value is measured against these scheduled values. For example, a so-called knock-out option becomes worthless if its underlying stock value reaches the barrier value.

Whereas a plain option is unconstrained, a barrier option is constrained by the predefined barrier values of the stock. Control action is executed when these barriers are reached, thus confirming that we are indeed looking at an instance of the PCS category.

4. Control engineering (Section 6.4.2)
This is a specialized discipline and it is concerned (among other things) with the definition of models that ensure that a system behaves in a certain way. We distinguish between open, closed, feedback and feedforward systems.

You can skip this section on a first reading. It may not be to everybody's taste.

5. Complexity of object-oriented applications
Systems built using objects and classes tend to become more complex and difficult to maintain as time goes on. In particular, classes may have associations with several other classes. The more relationships a class has with other classes, the less understandable and maintainable this class becomes. In order to redress this problem, we can define a number of so-called *software metrics*, define target values for them and describe the problem of defining the resulting system as an instance of the PCS category. For example, we could define an upper threshold value for the number of attributes in a class; a warning message is sent to the software risk manager if this value is exceeded. Of course, risk and quality managers are interested in risks and potential calamities. Modelling their world using PCS systems may not be a bad idea after all because these systems inform the managers when things start to go wrong.

6. Home Heating System (HHS) (Chapter 12)
This system is a prototype for the PCS category. It is a standard benchmark case in the software literature. Our approach to the HHS is unique, in our opinion. Some of the issues that we address in a comprehensive manner are:

- Integration of HHS with process-control terminology (from Chapter 6)
- Benchmark previous analyses of HHS (Booch, Hatley and Pirbhai)
- Thorough description of behaviour with use cases
- Integration of the PAC model with use cases.

Furthermore, we have used HHS as a reference model for new systems. We can employ a form of analogical reasoning to 'morph' HHS into the current system under discussion. This is easier than approaching the analysis of HHS using traditional object-oriented technology and its related methods such as using nouns for classes, CRC cards and so on. Our approach is better because we have decomposed HHS into loosely coupled systems and each system encapsulates a difficult and volatile design decision. Furthermore, we have integrated this approach with the object paradigm.

2.4 RESOURCE ALLOCATION AND TRACKING (RAT) SYSTEMS (CHAPTER 7)

The main added value of the RAT category is that it provides us with a model for registering and tracking entities in a system. It must be possible to query the status of the entity at all times. The primary input to RAT systems is some kind of request. The core process produces status information and the main activities are:

- Register and verify the request
- Assign resources to execute the request
- Monitor the status of the request and present this to stakeholders.

RAT systems occur in many industrial and business applications and we consider the RAT category to be one of the most important categories in our repertoire. We now summarize the specific RAT instances that are discussed in this book.

1. Help Desk System (HDS) (Section 7.3.1)
This is a good prototypical instance of a RAT category and it contains enough information to allow us to generalize it to other applications. We discuss the viewpoints and requirements of a number of stakeholder groups. Furthermore, we create a context diagram for HDS that is able to support stakeholder requirements and that can be used as a prototype for other applications in the same category.

2. Discrete manufacturing (Section 7.3.2)
This real-life problem discusses the process of trimming and forming computer chips once they have been manufactured. To this end, pallets of chips are loaded into a machine, the chips are trimmed and formed and finally unloaded. There is a clear tracking metaphor in this problem.

3. Tracking systems in financial risk management (Section 7.11)
This is a large system in general but there is a strong tracking element and this is modelled as a number of 'layered' RAT systems. One layer tracks real-time market data, the next layer tracks individual portfolios, while the highest-level layer tracks all portfolios in an organization.

4. Elevator Control System (Chapter 13)
We devote a chapter to this problem. We discuss how the RAT category is a good fit to this problem. We analyse the problem as three loosely coupled RAT instances, one for elevator reservation (by would-be passengers), the second for elevator utilization (by passengers) and finally a RAT system that is responsible for the actual scheduling and dispatching of physical elevators.

A thorough discussion of goals, processes, stakeholders and requirements is given in this chapter and we document these artefacts using the standard templates as discussed in Chapter 4.

5. Order Realization System (ORS) (Chapter 14)
This is a RAT instance that is embedded in a Lifecycle Model (LCM). We create the context diagram for ORS in order to reduce scope and risk. Furthermore, we show how to construct a PAC model for ORS and we integrate this model with the requirements and use cases. We also discover a number of critical classes in ORS and we document them using UML. Finally, we discuss how ORS should be designed and we place particular emphasis on database design and how the software components actually communicate.

6. Rent-a-machine (Section 10.3)
This is an application from the retail industry. We wish to track the whereabouts of a machine that is rented from a garden centre.

2.5 MANUFACTURING (MAN) SYSTEMS (CHAPTER 8)

This category defines applications where there is a clear idea of *creating* products and services. In general, a MAN instance creates a product from raw materials. This is the core process and its activities are:

- Process and check raw materials
- Convert raw materials to 'half-products'
- Package and dispatch half-products.

There is a clear idea of procuring raw materials, designing a product based on these materials and packaging the product for different kinds of customers. We are not interested in tracking the manufacturing process as such (this is done by a RAT system), nor in historical information concerning the product (this is done by a MIS system). We could say that a MAN is a MIS or a RAT without memory; in other words, we create a product but we have no historical data on it and we do not know how, when or by whom it was created. Of course, complementary RAT and MIS systems will be needed in real applications if we do wish to model these requirements.

The MAN category is needed by other applications because we must first create objects before we can do something with them.

1. Reference models in manufacturing domains (Section 8.2)
Models for manufacturing processes are well known in the literature. We use these models to describe and document the MAN category. We note that there are many

flows in MAN systems, for example material, cost and information flows. We must model these flows.

2. Compiler construction (Section 8.3.1)
This is probably the prototypical MAN instance. Compiler models are well documented in the literature.

3. Graphics and CAD applications (Section 8.3.2)
These are applications that create entities that are then displayed on a screen. The raw input data is usually an ASCII or binary file that describes graphics objects.

4. Human memory models (Section 8.3.3)
These are models that describe how long-term memory works and how we remember events based on sensory perception. We see this problem as a MAN instance because we are interested in how long-term memory is created and stored.

5. Rent-a-machine (Section 10.3)
This is a lifecycle model and it has an 'embedded' MAN component. In this case we create a basic request object. This object will then be assigned to resources in an upstream RAT system.

6. Tracking plastic manufacturing processes (Section 16.3)
This is a lifecycle model and it has an 'embedded' MAN component. In this case we create a basic request object. This object will then be assigned to certain resources in an upstream RAT system.

We note that there are many similarities between this problem and Rent-a-machine; in the latter case we are tracking rented machines while in the former case we are tracking a customer request for a supply of processed plastic film.

2.6 ACCESS CONTROL SYSTEMS (ACS) (CHAPTER 9)

This class of applications includes security systems and systems where controlled access to valuable resources must be defined. These systems are well understood because there are many reference models for them. There are two main processes in ACS systems:

- Authorization: securely identifying principals
- Authentication: controlling which principals can execute which operations on which resources.

The main activities in the Authentication process are:

- Accept a request from a subject to gain access to an object
- Check whether access is allowed
- If successful, execute the request on behalf of the subject.

ACS systems are 'helper' systems for other applications because they realize requirements such as Security (a sub-characteristic of Functionality) and to a lesser extent Reliability.

1. The Reference Monitor model (Section 9.3.1)
This can be seen as the original model for this class of problems. We can learn a lot about ACS systems by looking at the model and its corresponding architecture. We have modified this model to suit an object-oriented context. In particular, we have mapped the architecture in the Reference Monitor model to a context diagram in ACS.

2. Security issues in Web applications (Section 9.10.1)
Here we give a short description of some modern versions of the Reference Monitor model from Section 9.3.1, including role-based access mechanisms that we conveniently document by an UML class diagram.

3. The proxy design pattern as a special ACS system (Section 9.10.2)
We subsume the well-known proxy pattern under the ACS banner. In particular, the different kinds of proxy as described in POSA 1996 are discussed in relation to the ISO 9126 quality characteristics.

4. Drink Vending Machine (Chapter 15)
A classic! This problem is discussed in many books on software development. We model the problem as an instance of ACS and we show how our solution compares well to the somewhat *ad hoc* approaches taken to analyse this problem. Just looking for the objects is no longer good enough!

2.7 LIFECYCLE AND COMPOSITE MODELS (CHAPTER 10)

The systems in this category have three components, namely a MAN instance, a RAT instance and a MIS instance. Lifecycle Models (LCM) are very important because most real-life applications are in fact composed of multiple lifecycle models.

Many reference models exist for this class of applications. These models have been standardized and institutionalized in mature disciplines such as retail, manufacturing, marketing and oil (where the author got the model).

1. Product lifecycle in general (Section 10.2)
This is a general discussion of the lifecycle model for any kind of product. The reader should consult technical marketing literature to understand just how organizations view this problem.

2. Rent-a-machine (Section 10.3)
This problem discusses the lifetime of a request from a customer to rent a machine at a garden centre. We sketch the core processes in this system as well as the context diagram and main activities in each subsystem. Special emphasis is paid to how customer-defined features (which are always a bit fuzzy) are mapped to more concrete requirements.

3. Order Processing System (OPS) (Chapter 14)
This is a large chapter that discusses a lifecycle model that tracks a request or order from the moment that it is created to when it is completed and archived. We concentrate on the structure of the subsystems and the different kinds of stakeholders that have their own specific viewpoints on the system.

4. Plastics extrusion (Chapter 16)
This chapter describe how we have applied the LCM to an industrial application, namely the production of plastic film. We pay attention to defining robust context diagrams and black-box interfaces between the systems and components in this problem. Some design topics are introduced to show the reader how the artefacts from the DDP map to design patterns. Special emphasis is paid to how user-defined features (which are always a bit fuzzy) are mapped to more concrete requirements.