

This third edition of the best-selling text, *Object-Oriented Systems Analysis and Design using UML*, has been thoroughly revised and updated. With over 40,000 copies sold world-wide, Bennett, McRobb and Farmer's text remains a key teaching resource for Systems Analysis and Design courses at both undergraduate and postgraduate levels.

The book provides a clear, practical framework for development that uses all the major techniques from UML 2.0. It follows an iterative and incremental approach based on the industry-standard Unified Process, placing systems analysis and design in the context of the whole systems lifecycle. Structured in four parts, the first provides the background to information systems analysis and design and to object-orientation. The second part focuses on the activities of requirements gathering and systems analysis, and the basic notation of the Unified Modelling Language (UML). In part three, the activities of system design and UML notation for object design are covered, and the book concludes with the later stages of the systems development lifecycle, reuse and software development methodologies.

Key features of this exciting new edition include:

- Fully updated notation following the very latest version of the UML standard, reflecting the most up-to-date approaches to information systems development.
- A brand new chapter on System Architecture explaining how information systems are designed to work together in complex modern enterprises.
- New focus on component-based development, Agile software development, eXtreme Programming, Model-Driven Architecture, and other recent advances in the software development process.
- Two realistic case studies are used throughout the book—one for illustrative examples and the other for practical exercises for the reader.

www.mcgraw-hill.co.uk/textbooks/bennett

McGrawHill Education

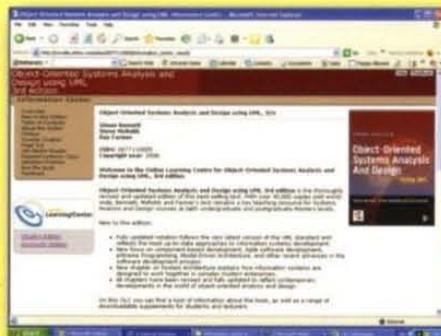


Online Learning Centre

An extensive Online Learning Centre website offers supporting resources for the book at:
www.mcgraw-hill.co.uk/textbooks/bennett.

For students:

- Multiple Choice Self-Test Questions
- More useful learning resources



For lecturers there are PowerPoint presentations and solutions to exercises in the book.

Free access to these resources is available with the textbook.

Simon Bennett is a Systems Architect in the Corporate IT department of Celesio AG. Previously a Principal Lecturer in the Department of Information Systems at De Montfort University, he has also co-authored *Schaum's Outline of UML 2nd edition*, published by McGraw-Hill.

Steve McRobb is a Senior Lecturer in the Faculty of Computing Sciences and Engineering at De Montfort University, Leicester.

Ray Farmer is an Associate Dean in the Faculty of Engineering and Computing at Coventry University.



THIRD EDITION

Object-Oriented Systems Analysis And Design

Using UML

Simon Bennett, Steve McRobb and Ray Farmer



systems and the pace of technological development will continue to accelerate for the foreseeable future. Recent technical developments like the Internet have made possible information systems that would not have been feasible even just a few years ago.

Despite the antiquity of some of its subject material, the academic study of information systems is young, even by the standards of the computer age, and owes its importance today chiefly to the rise of the computer. Those who developed the earliest computer systems gave very little thought to the issues with which this book is primarily concerned. Wartime imperatives drove them to apply the new technology to military problems like code-breaking, naval gunnery calculations and other mathematical tasks. Their focus was on the technical difficulties of implementing hardware based on ideas from the very cutting edge of research in electronics and control logic. It was also necessary to invent efficient techniques for controlling these new machines. In due course these techniques became today's computer programming.

By the time that computer scientists had gained a degree of mastery over their hardware, the world was once more at peace. Businesses began to be aware of the potential that computers held for commercial activities, and attention gradually turned to wider questions. Those with which we are concerned are the following.

- How do we establish the business requirements for a new system (often much subtler and more complex than the role of the earliest machines)?
- What effects will the new system have on the organization?
- How do we ensure that the system we build will meet its requirements?

It is primarily with these questions that the field of information systems is concerned, and they are also the main subjects of this book.

Within the relatively new field of information systems, object-oriented analysis and design are even newer, although they are derived from object-oriented programming, which has been around a lot longer. Object-oriented analysis and design were first conceived of only about a decade ago, and today they are still very much a novelty. Only in the last few years has there been a real surge of interest in object-oriented systems development. This relative youth of object-oriented analysis and design might give cause for concern. Are they no more than a passing fashion? We believe they are no mere fad, but on the contrary, for many information system applications—although not all¹—object-orientation is simply the best way yet found of carrying out the analysis, design and implementation.

So far we have skirted around a question that we must address right at the start. What is an information system? In this chapter we provide the beginnings of an answer (the rest must come through practical experience). We set the scene by describing some information systems, ranging from one that is completely up to date, to others that are considerably older (Section 1.2). We discuss what is meant by 'system' (Section 1.3), by 'information' and finally by 'information system' (Section 1.4). We also put the question in context by raising some business issues, in particular how the managers of an organization decide (or discover) the things that are important for the business to do, and we outline the role of information

¹ We return to this question in Chapter 4.

systems in doing these things successfully (Section 1.5). For business managers, this is the real bottom line. An information system that did not help to ensure the success and prosperity of the organization, its investors and its employees would simply not be worth the cost of its development.

Box 1.1 An online retail system

McGregor plc is a chain of retail stores that sells kitchen appliances, mobile phones and electronic home entertainment equipment. The company has recently created an online shopping centre, accessed via the World Wide Web. After registering their name and address, shoppers can browse through various products, selecting items and placing them in a virtual trolley. At the end of the trip, shoppers can buy what is in their trolley, remove items or quit without making a purchase. Payment is made by submitting credit card details online, or by entering part of the card details and phoning to give the rest. Delivery times are usually within three working days for small items such as mobile phones, but up to three weeks for larger items such as cookers. Goods are dispatched direct to the customer's home. Credit cards are debited only on the day of dispatch and, between purchase and delivery, customers can use the website to check on the progress of their order.

This is how an online shopper interacts with the system, but beneath the surface a great deal more is going on. A whole network of hardware connects the shopper's home PC and modem, through a phone line, a telephone exchange, another phone line and modem, to a computer that acts as a web server. This is connected to other networks at McGregor's head office and shops. Many software applications are also busy processing information captured through the web pages, and feeding various support activities. Some of these are computerized and others are carried out by people.

- Marketing staff keep prices and product details up to date on the electronic product catalogue system. This can also be accessed by touch-screen PCs in the shops.
- Credit card details are stored electronically for relay to the card processing centre when the goods are dispatched.

- Robot forklift trucks in the warehouse fetch items to the loading bay when they are due for dispatch, and warehouse staff load them onto delivery trucks.
- Delivery drivers follow a schedule linked to an electronic map in the vehicle cab. This is updated by radio every few minutes, helping to avoid traffic jams.
- Out-of-stock items are re-ordered from the supplier by electronic data interchange (EDI). On delivery, warehouse, delivery and charging begin quite automatically.
- At each significant point in the sequence, a database entry is automatically updated, and this is displayed on the web page, allowing shoppers to discover what stage their order has reached.

There are many users besides the shopper, each with a different view of the overall system. A network manager monitors the number of hits on the web server and network traffic within McGregor, checking for hardware or software failure and breaches of network security (e.g. if hackers try to break in). Her concern is the efficient and secure flow of information; she is not interested in its content. A financial controller uses a linked accounting system to monitor sales transactions and cash flow. A dispatch clerk regularly checks forklift schedules on her PC, and compares them to delivery truck schedules faxed daily from the courier company. She smoothes the flow of goods through the loading bay, and spends long hours on the phone sorting out delays. A market researcher uses a specialized statistical package on a portable PC to analyse online sales, assessing the success of the web pages' presentation styles. Registration allows individual customers to be tracked as they browse the site; using information about their preferences, the design can be finely tuned to attract high-spending customers.

1.2 Information Systems in Practice

We set the contemporary scene in Box 1.1 by describing a complex modern information system. Many readers will be familiar with systems that resemble this one, and some may have used them to make purchases over the Internet. This particular example is entirely fictitious, although it is loosely based on elements taken from a number of real companies.

The staff described at McGregor all rely on computer systems for the information they need to do their jobs. But only recently have information systems begun to use information technology in such a sophisticated way. Our next example, described in Box 1.2, is almost exactly contemporary with the birth of the electronic digital computer, yet it contains little that we would recognize today as information technology. Checkland and Holwell (1998) describe it aptly as ‘the information system that won the war’.

Box 1.2 Fighter Command’s 1940 information system

This vital communications and control system was used to monitor and control Spitfire and Hurricane fighter squadrons as they flew against Luftwaffe bombing raids during the Battle of Britain. It is quite certain that, without it, the RAF could not have succeeded in the defence of Britain in 1940. The main ‘hardware’ was a large map kept at Command headquarters at Bentley Priory. Coloured counters were used to mark the current position of various aircraft formations, and were moved around the map by hand as reports came in via radar, telephones, teleprinters and radio.

Reports of enemy raids were collected both by radar and from a network of observer posts throughout the country, while information about RAF deployment was sent in via Group and Sector control rooms. A special filter room correlated and checked all reports before passing them through to the main control room for display. Other displays included a blackboard showing at a glance the status of all current Luftwaffe raids, and a row of coloured lights flagged the readiness of RAF fighter squadrons. Carefully co-ordinated duplicates

of these displays were based at other control rooms, including those at Sector level, where the controller spoke directly by radio to the tense young pilots in their aeroplanes. Using this basic, but effective, technology, human operators performed communication, filtering, processing and display tasks that would today be carried out by computers.

Some accounts of the Battle of Britain have claimed that it was won chiefly due to the courage of British pilots or to the RAF’s use of radar. But these miss a most important point. The new radar technology was certainly important, but the British variety was still technically far inferior to its German equivalent, already in use for naval and anti-aircraft gunnery. Equally, there could have been no victory without the bravery of the RAF pilots, but was this really greater than that of their German opponents, who also faced extreme danger each time they flew over the English coast?

Important as these factors were, it was only through Fighter Command’s information system that the British human and technical resources were deployed so effectively.

With a little imagination, it is easy to see how a modern system could use computers to fulfil similar functions to those described in Box 1.2, and indeed this is the situation in most airforces today. Box 1.3 looks even further back in time, introducing briefly an information system that dates back over a century, yet still operates effectively today with similar technology in many parts of the world.

The distinguishing characteristics of the pre-computer information systems described in Boxes 1.2 and 1.3 would change relatively little if they were to be re-implemented today using computers. But the system described in Box 1.1 would not be feasible at all without the use of modern information technology. Thus, while the computer is not a necessary precondition for the existence of many information systems, in some cases today computers have made possible information systems that could not exist without them.

Box 1.3 A railway signalling system

This example is particularly interesting because, according to most classifications of work and social status, an old-fashioned railway signalman would be regarded as essentially a manual worker of the early industrial age. Yet on closer examination he can be seen clearly to be an information worker, differing from many modern computer users only in that the technology available to him was much less sophisticated than would now be the case. According to Webster (1995) ‘... The railway signalman must have a stock of knowledge about tracks and timetables, about roles and routines; he needs to communicate with other signalmen down the line, with station personnel and engine drivers, is required to “know the block” of his own and other cabins, must

keep a precise and comprehensive ledger of all the traffic that moves through his area ...’. In other words, the signalman operates an information system that comprises his ledgers, the levers, signals and lamps and what he knows in his mind.

Nowadays many railway signalling systems are largely automated. However, whether the work is done by networked computers linked electronically to electric motors that switch points and operate signal lights or by a traditional signalman, who pulls levers connected by steel cables to the points and signals and keeps careful records in a hand-written ledger, the tasks that are actually carried out are little different. All that has changed is the technology that implements the system.

1.3 General Systems Theory

We now introduce some basic concepts of general systems theory. This is essentially a way of understanding a system in terms of those components and characteristics that are common to all systems. We use the term *system* here in a very specific sense, although one that has much broader application than just computer systems. In everyday speech, people may refer to the legal system, a tropical storm system, the system of parliamentary democracy, an eco-system, a system for winning at roulette, a computer system in someone’s office, a system for shelving books in a library, a system-built house, and many more. Some of these certainly meet our definition of a system, and others do not. Probably the only thing that they all have in common is that they have some kind of organization. But in general systems theory, a system is rather more than just anything that shows a degree of organization.

1.3.1 Characteristics of a system

So what are systems? For our purposes, a system has the following characteristics.

- A system exists in an environment.

- A system is separated from its environment by some kind of boundary.
- Systems have inputs and outputs. They receive inputs from their environment, and send outputs into their environment.
- Systems transform their inputs in some way to produce their outputs, for example, by combining simple ingredients to create a more complex product. The systems in which we are interested generally have an explicit purpose, and this is closely related to the way that inputs are transformed to produce outputs.
- Systems have interfaces. An interface allows communication between two systems.
- A system may have subsystems. A subsystem is also a system, and may have further subsystems of its own.
- Systems that endure have a control mechanism.
- System control relies on feedback (and sometimes feed-forward). These comprise information about the system's operations, or its environment, that is passed to the control mechanism.
- A system has some properties that are not directly dependent on the properties of its parts. These are called emergent properties, as they only emerge at the level of the system as a whole.

Figure 1.1 presents the relationships between these concepts schematically.

Boundary and environment

Boundary and *environment* are closely related concepts, and understanding them is essential to making sense of any system. In fact, the first step in understanding a system is to choose the system that you wish to understand, and this largely means choosing its boundary.

For example, a cell biologist may think about a single human cell as a system. Her interest, and therefore the system she thinks about in detail, is bounded by the cell membrane. A biochemist may be interested in a slightly larger system, perhaps a chemical reaction that takes place in a particular group of cells. A specialist

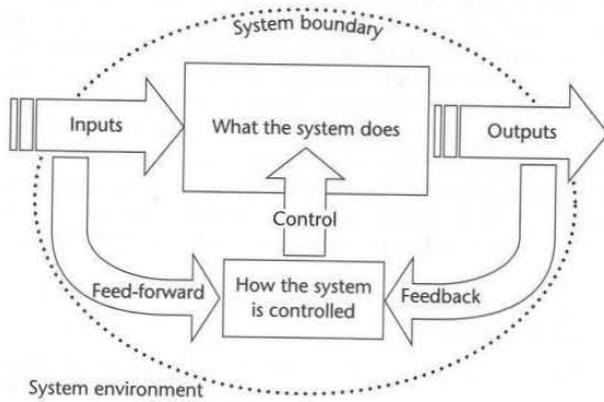


Figure 1.1 Parts of a system, and their relationship to each other.

physician attempting to diagnose an illness may consider a patient's kidney as a system. The boundary of this system may coincide with the organ, or it may be wider still, depending on the particular illness. For a doctor in general practice, a person's whole body may be considered as a system bounded by its skin. These different boundaries are illustrated in Fig. 1.2. We can continue zooming out in this way, perhaps until we reach a cosmologist (whose interest may be bounded by the limits of the physical universe) or a theologian (whose interests are even wider).

But the choice of a system that corresponds to a subject of interest is not simply a matter of scale. Systems can overlap with each other, and this is also an issue for boundary setting. Let us think again about a whole person as a system. Within this image, many other systems can be found. For example, medical specialists each have their own view of what is interesting or important. A neurologist may focus on the nervous system, consisting of the brain, spinal cord and the network of nerves that spreads through the body to just beneath the surface of the skin. While its physical boundary is almost identical to that of the body as a whole, the nervous system contains only various specialized nerve cells. The interests of a haematologist thinking about the circulatory system have a similar boundary, yet this system contains instead blood cells, blood vessels and the heart. A person can also be considered as a variety of non-physical systems. A psychologist may study an individual's cognitive system or emotional system, or may consider a child's

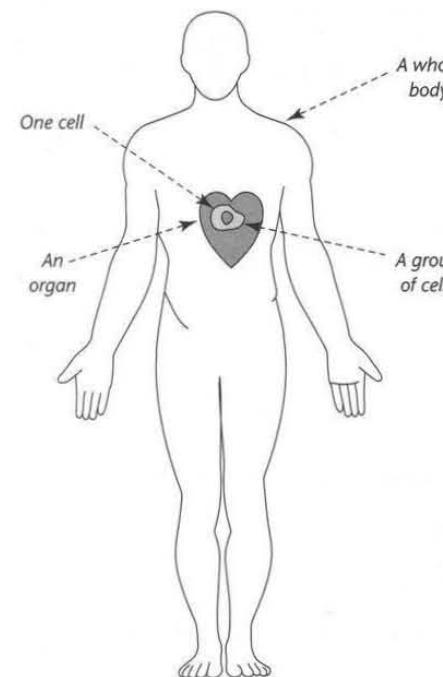


Figure 1.2 Systems at different scales.

intellect as a learning system. If we move along the scale again, a social psychologist may think of a family as many overlapping systems: a child-rearing system, an economic system, a house-maintenance system, etc. These views of the family as a system have purely conceptual boundaries, since family members remain part of their system no matter how distant they are in space.

The boundaries of different systems can overlap or coincide. Indeed, two systems may be closely related, may have identical boundaries, and yet still be distinct. This is potentially a tricky part of the theory. At what point does one system end and another begin? Also, can a thing (e.g. a human organ) simultaneously be part of one system and of another, at one and the same time?

The answer to this lies in the fact that the usefulness of systems is in helping us to understand something about how the world works. They do this by representing selected aspects of the world in an abstract way. In most cases, it is not important whether or not the system corresponds in every detail to the thing it represents. Checkland and Scholes (1990) explain this as follows.

... it is perfectly legitimate for an investigator to say 'I will treat education provision *as if it were* a system', but that is very different from saying that *it is* a system ... Choosing to think about the world as if it were a system can be helpful. But this is a very different stance from arguing that the world *is* a system, a position that pretends to knowledge that no human being can have.²

Figure 1.3 illustrates the same point graphically.

This does not mean to suggest that no system has a basis in reality. On the contrary, many systems are made of real components. For example, it is quite possible to touch physically all the parts of a central heating system. But it is a matter of choice, based on our interest at a given moment, whether we choose to *think* about

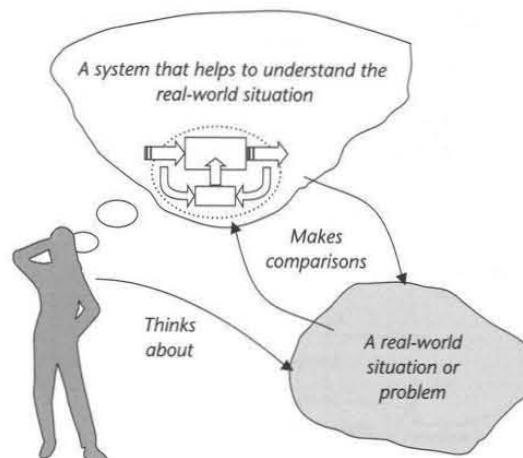


Figure 1.3 The relationship between system and reality (loosely adapted from Checkland and Scholes, 1990).

² Italics in the original.

it as a system. Any system that we think about necessarily exists in our thoughts, rather than in the world, and such a system, however much it corresponds to the world, is still a subjective view of reality, not the reality itself.

Input, output, interface and transformation

Systems have interactions with their environment. For example, human cells take in food and oxygen and transform them into protein, energy and other products. They also output carbon dioxide and other waste. Some cells produce things that are useful to the body as a whole. For example, some white blood cells produce the antibodies that fight infection. The nervous system receives sensory information in the form of light, sound, touch, etc. and transforms these into electrical signals. Some of these signals act as inputs to the body's muscles, generating movement and speech. Each interaction is based on a set of inputs and produces some outputs. Inputs originate outside a system, and are taken in to be used in some way. Outputs are created by the system, and sent into the environment in order to have an effect somewhere else. Outputs act to achieve a purpose of the system. Figure 1.4 shows some inputs and outputs for three systems.

The transformation of inputs into outputs is an important characteristic of purposeful systems, such as businesses. This is how they fulfil their objectives. Depending on your interest in the system, it may be enough to know the relationship between inputs and outputs. A customer of the McGregor online shopping system described in Box 1.1, is probably not interested in what goes on behind the scenes. Her main focus is on her interaction with the system—in other words, the inputs she must provide (information, money) and the outputs she receives (information, a new refrigerator, customer service). This is called a *black box* approach, since it treats the internal workings of the system as being completely hidden. All that can be seen is what goes into the system, and what comes out. This is illustrated in Fig. 1.5.

An output from one system can simultaneously be an input to another. The two systems share part of their boundary, across which inputs and outputs pass between them. The shared boundary is an interface. For example, the web page used by this customer to order her new refrigerator is her interface with the

System	Inputs	Outputs
A student	Information Exercises Guidance	New knowledge New ideas Solutions
A family	Money Social standards and norms (e.g. laws) Purchases Daily news	New citizens (i.e. children) Products of family members' work Social influence Votes in elections
A business	Raw materials and labour Investment Information (e.g. customer orders)	Profit and taxes Finished products Information (e.g. the company report)

Figure 1.4 System inputs and outputs (note that there is not necessarily a one-to-one correspondence between inputs and outputs).



Figure 1.5 A black box view of a system.

McGregor system. The data content and structure of the web page define and limit her interaction with the online shopping system. For example, if there is no field on the web page where she can enter her address, it may not be possible for her new refrigerator to be delivered to her house. We shall see later, in many places throughout this book, that the identification and understanding of interfaces is of particular importance in the development of information systems.

Subsystems

Subsystems are a natural consequence of other systems ideas introduced above. For example, a white blood cell making an antibody can be seen as a system on its own, or as a subsystem that contributes to the larger system. Figure 1.6 illustrates some subsystems that can be found in the description of the Agate case study in Case Study A1 (this case study is used to provide many of the examples throughout the book).

Subsystems are at once part of a larger system, and also coherent systems in their own right³. Communication between subsystems is, by definition, through interfaces. This particular kind of diagram is sometimes called a system map.

Another way of arranging systems and subsystems is as a hierarchy (this is how the Agate subsystems are shown later in Fig. A1.1). Hierarchies are a very important aspect of systems theory, to which we will return frequently (in Chapter 4 we shall also see the importance of hierarchy to understanding object-orientation).

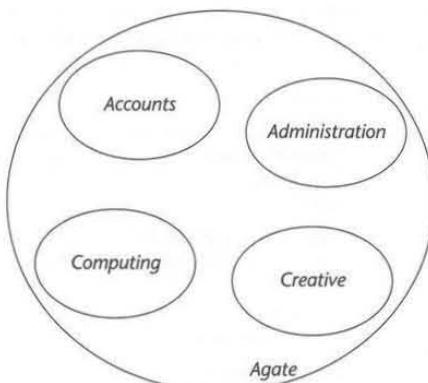


Figure 1.6 Subsystems of Agate.

³ Koestler (1967) coined the word 'holon' to express this duality. A holon is something that, seen from one perspective, looks like a whole thing, but, viewed in another way, looks like a part of something else.

Control in systems

Many systems have a specialist subsystem whose function is to control the operation of the system as a whole. In fact, general systems theory sprang partly from the science of cybernetics, which studies control in natural and artificial systems. A familiar type of cybernetic control is the family of thermostatic devices that control central heating, hot water and air conditioning systems, and also temperature-controlled appliances such as freezers and ovens. Many are based on a simple feedback loop, as shown in Figure 1.7. System control is usually based on a comparison between two or more input values, whose similarity or difference guides a decision about whether any controlling action is required.

Feedback

The term *feedback* indicates that one or more outputs of the system is sampled and literally fed back to the control unit. In Fig. 1.7, a sensor samples the temperature in the main freezer compartment (an output measure) and this is fed back to the control unit. Here a logic mechanism (this could be either mechanical or electronic) decides whether action is required. Possible actions are turning the cooling motor on and turning it off. Both are effected by a signal sent to the motor by the control unit, thus 'closing' the 'feedback loop'. In this way, the control unit monitors the current temperature, compares it with a reference value (in this case probably set by the owner using a dial or switch), and if necessary adjusts the system's operation to bring the difference between the two values within pre-defined limits.

This is *negative feedback*, because it aims to maintain the system's equilibrium by opposing deviations from some norm. Negative feedback is used widely in physical systems, such as electronic equipment, manufacturing systems, etc. By contrast, *positive feedback* works by reinforcing deviations instead of opposing them and therefore tends to increase movements away from equilibrium. Systems

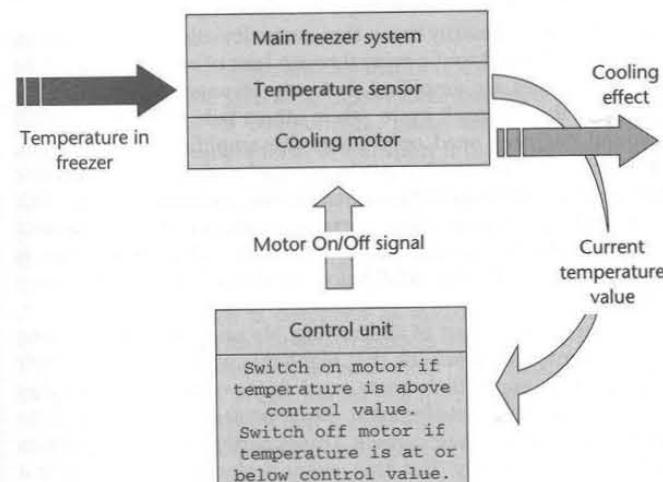


Figure 1.7 Thermostatic control in a freezer—a simple negative feedback loop.

governed by positive feedback are inherently unstable, which is undesirable when the deviation that is reinforced is itself undesirable. For example, the ear-splitting howl sometimes heard from public address systems is the result of a positive feedback loop. It occurs when sound output from a loudspeaker is caught by a microphone and input again to the amplifier. The signal resonates round and round the loop, growing louder at each turn until the amplifier reaches its limit.

Both negative and positive feedback occur in social and business systems, but here they are rarely so clear-cut or simple. For example, the Bank of England regularly reviews the performance of the UK economy to determine whether or not it meets the Government target for monetary inflation. When the economy deviates from the target, action is taken to bring it back on track, usually by altering the bank's base interest rate. This would be a simple example of negative feedback, though of course, in reality the situation is much more complex. Another example of negative feedback is when a business adjusts its manufacturing output to maintain a constant inventory. If sales rise, stock falls, and the rate of production is accordingly increased. If sales fall, stock tends to rise, and the rate of production is reduced. Here the stock level acts as a *buffer* between production and sales subsystems.

Positive feedback is beneficial when a steady state is undesirable. For example, in a soccer game, equilibrium would tend always to result in a goal-less draw. Any competitive edge that one team gains over its opponents is exploited to retain and maximize the advantage. However, this picture is greatly complicated by the fact that both teams continually strive to get the upper hand in a dynamic game, and so it is rare to see the cycle build to an extreme. Instead, the system (i.e. the game) usually tends to continue in a state of dynamic disequilibrium. Negative feedback also plays a role, as teams try to maintain any equilibrium that acts in their favour. A complex mix is also seen in business competition, where firms capitalize on any competitive advantage they can gain over their rivals (positive feedback, aiming to disrupt an unhelpful equilibrium), while also trying to prevent their rivals from gaining an edge (negative feedback, maintaining a beneficial equilibrium).

Positive feedback does not necessarily mean that every deviation from the norm is reinforced. Some may be allowed to die away through lack of encouragement. In audio amplifiers all frequencies are amplified, but only resonant frequencies are disastrously reinforced by the feedback cycle, while others fade naturally without ill-effect. Often a sound engineer need only adjust the amplifier's tone control, reducing slightly the volume in the troublesome frequency band, to cure a feedback problem. A similar type of control can be discerned in businesses that directly link productive output to sales, perhaps because there is a very short-term seasonal market (say, toys at Christmas). In this case, when sales rise, production is increased as far as possible. But when sales fall below a critical level, a product may be dropped altogether.

Effective feedback is an essential part of all learning. No one can develop a new skill without receiving appropriate feedback that highlights which aspects of their performance are satisfactory and which parts need improvement. This applies equally to the activity of software development. Most professionals continue to learn how to do the job throughout their careers. This is partly because techniques and technologies evolve continuously. But also, every project is unique, poses a new set of challenges and demands new approaches from the developers.

Feedback on what worked in the past, and what did not, helps to guide the developer's choices in future.

Feed-forward

Feed-forward information relies on sampling the system's inputs rather than its outputs. For example, the Christmas toy business mentioned earlier may find that the use of feedback from sales (a measure of output) does not allow them to react quickly enough to changing market conditions. This may leave them with unsold stock, bought from manufacturers before they realized that demand had slumped. Ideally, they should adjust their manufacturing to suit the level of demand, and they may be able to use market research to forecast which toys will be popular among children this year. This would allow the firm to avoid buying products for which there is no market. Another way to use feed-forward would be to find out which toys are being manufactured in large numbers, and then advertise these aggressively in order to stimulate demand.

While feed-forward control information can help a system to be more responsive to environmental fluctuations, it is not always easy to implement or manage in a business organization. Difficulties still arise if the rate at which conditions change in the environment is faster than the rate at which the business can adapt. The effects of this are apparent to anyone who visits the kind of specialist book shop where remaindered titles are sold cheaply.

In the Agate case study, the agency must employ and train sufficient staff to cope with the anticipated workload. If there is a serious slump in orders for new work, it may not be possible to reduce the number of staff quickly enough to avoid bankruptcy, because of the need to give staff a period of notice before they can be laid off. The company may also not be able to respond quickly enough to a sudden surge of orders, because of the lead time for recruiting and training new staff. Forecasting the level of demand for a service such as Agate's is an important role of information systems in business.

Emergent properties

One of the truly distinctive characteristics of a system is that it is more than just the sum of its parts. The system possesses some feature or ability of its own that is not present in any of its components. These are called emergent properties. For example, a car is only a form of transport if there is enough of it to drive. It then has the property of being a vehicle, but the wheels, windscreen, motor, etc. do not have this property until they are correctly assembled. In a similar way, the departure of one key member of staff can sometimes cause a business to fail, but that does not mean this person was doing all the work on their own.

This recognition that systems have emergent properties is the main reason why the systems approach is described as *holistic*. This means thinking about each system as a whole, with important aspects that will be overlooked if we think only about its parts in isolation from each other. The holistic world view of the systems approach is manifest in techniques like the rich picture. This aims at capturing in one picture or diagram all that is essential to an initial understanding of a system⁴. An opposing view is taken by reductionists, who begin with the assumption that

⁴ See Checkland and Scholes (1990) for a full explanation of rich pictures with many examples of their use in practical situations.

complex phenomena can be explained by reducing them to their component parts. This approach has a very important place in the physical sciences (e.g. physics and chemistry, although there are dissenters even in those disciplines). We believe it is thoroughly misapplied when used to explain complicated human situations.

1.3.2 Systems of interest to this book

We have used a mixture of natural and artificial systems as examples. General systems theory is applicable to both, but in this book we are principally interested in the artificial kind. Even some of these are more relevant than others. For example, we will probably only be professionally interested in the system of parliamentary democracy if we intend to develop an information system that supports it. Two types of system are of particular importance to us. The first is the *human activity system* (Checkland, 1981). A key feature of this type is that it centres on a purposeful activity, such as we find when we look at a business, a club, a hospital, or any organized activity. There is often wide disagreement among the participants about exactly what that purpose is, and this can be a significant problem for the analysis of information system requirements. Figure 1.8 shows how some of the people who work at Agate see the purpose of one system in which they are engaged.

The second kind of system in which we have an interest is, naturally, the information system. Information systems are constructed to help people in a human activity system to achieve their goals, whatever these might happen to be. Human activity systems are the context of, and provide the meaning for, the activity of information systems development. Anyone who does not understand the meaning and purpose of a human activity system will find that it is impossible to specify, still less to build, an information system that supports it in any useful way. We shall discuss many examples of information systems throughout the main part of the book. Each of these information systems is intended, in some way, to help fulfil the purposes of a human activity system.

The activity of systems development can be usefully regarded as a human activity system. This system transforms various inputs (money, skill, staff time, information from users about how they want the software to work, etc.) and aims to produce an effective software solution to a business problem. Its environment is

System	Purpose of system	As seen from the perspective of...
Agate (a business system)	To become a successful advertising agency on the international stage, thus providing both wealth and prestige for its directors	A director
	To provide varied and interesting work with a good salary, and also to be a useful stepping stone towards the next career move	A copy-writer
	To provide a pleasant and comfortable life until retirement (5 years away), without the need to make too much effort	Another director

Figure 1.8 A human activity system with multiple purposes.

typically the organization in which the developers work, including users of the software and their managers. Its subsystems include the members of the project team, the methodology they follow and also the various analysis and design models that describe the software. Control is exercised by a team leader or project manager, who receives regular feedback on progress and problems. Suitable feed-forward will help to alert the manager to anticipated problems, as well as those that have already happened. One advantage of taking a systems view of any activity is that it encourages those involved to think about the sorts of feedback and control that are needed for everything to run smoothly. This applies just as much to software development as to anything else.

1.4 Information and Information Systems

Some computing professionals see the development of information systems as essentially a matter of designing and building computer technology (including software) that meets a set of clearly understood needs. While this may be the ideal situation, in practice it is often a simplistic view that misses much of importance. It happens that information technology is now the normal technology used to implement an information system, but, as we saw in Section 1.2, information systems were constructed in many other ways prior to the development of digital computers.

Designing and building technology can be the easy part of the job—at least the easy part to understand—while the hard part is often determining the needs that the technology must serve. This involves identifying ways that an information system can support the purposes of a human activity system. An understanding of the information that will be useful to the human actors is an important ingredient in this, as is an understanding of how the information can be used effectively. As a result of these many different concerns, Information Systems has become a multidisciplinary subject that bridges many other fields, in particular computer science and business management, but also psychology, social theory, philosophy and linguistics, among others. In the following sections, we discuss the relationship between information, information systems and the human activity systems they are intended to assist.

1.4.1 Information

Information is conveyed by messages and has meaning. Meaning always depends on the perspective of the person who receives a message. We are always surrounded by a vast mass of potential information, but only some of this ever comes to our attention, and only some of that is actually meaningful in our present context. Checkland and Holwell (1998) describe the process by which raw facts become useful (i.e. become information) through a sequence of stages.

For an example of how this happens, consider four people watching the evening sky. A plume of smoke is rising in the middle distance. For Alex, the smoke is just part of the view, and she does not even consciously notice it. Ben sees it, and it evokes a memory of a camping trip long ago. But he is aware that the only connection between past and present smoke is a coincidence of shape and colour, so he moves on to look at something else. Chetan is thrown into consternation,

when he realizes that the smoke is rising from his house, which is on fire. Dipti runs to phone the Fire Service before doing whatever else she can to help Chetan save his house.

The sight of the smoke is, on the face of it, a single message available to be received by all, yet its meaning is different in each case. Alex does not even notice it, and Checkland and Holwell call this *data* (from the Latin for ‘given’), meaning a fact that has not been selected for any special attention. Ben notices the smoke but does not relate it to any present context. Checkland and Holwell call this *captia* (from the Latin for ‘taken’), meaning a fact that has been selected but has no particular importance or meaning. Chetan gives the smoke a meaning derived from the context in which he sees it (recognizing his house, understanding the implications of the smoke, etc.). This can be called *information* since it has a meaning within the context. The meaning of a fact is always dependent on its relevance to the observer.

There is a final step where information becomes *knowledge*, by being structured into more complex meanings related by a context. We can see this in Dipti’s response to the smoke. She integrates information from many sources: the link between smoke and fire, the effect of fire on houses, the existence and purpose of a Fire Service and the location of nearby phones. All this information comes together in a single framework of knowledge that is relevant to the context. In a word, she *knows* what to do.

1.4.2 Information systems

It is common for information systems textbooks to classify the variety of information systems an analyst may encounter in practice. For example, Laudon and Laudon (2002) describe the main types of information system in business as being: transaction processing; office; knowledge work; management information; decision support; executive support. Such a classification is probably less useful today than it once was, since information systems have become more closely integrated with each other, and the boundaries between categories have become blurred. It is still helpful to present a brief overview of some of the general types of application in organizations, but this really describes the roles that information systems can play, rather than actual types of system.

Operational systems

Operational systems automate the routine, day-to-day record-keeping tasks in an organization. The earliest commercial information systems were operational ones, because routine, repetitive tasks involve little judgement in their execution, and are thus the easiest to automate. Accounting systems are a good example. All organizations need to keep track of money—the amount coming in, the amount going out, the cash available to be spent and the credit that is currently available. Few modern organizations could survive long without a computerized accounting system. Sensible organizations have a ‘disaster recovery plan’ that details how they intend to cope with an emergency that destroys data or renders computer systems inoperable.

The flow of information through an accounting system is based on thousands, or even millions, of similar *transactions*, each of which represents an exchange of a quantity of something, usually a money value (this is why they are often called

transaction processing systems). For example, when you buy a tube of toothpaste in a supermarket, two separate records are made. One records that a tube of toothpaste was sold, and the other records the money you paid in exchange for it. As this repeats day after day for each item, customer, checkout and branch, an overall picture is built up that allows the company’s accountants to compare total income with total costs and to determine whether a profit has been made. Of course, many real accounting systems are more complicated than this, often with subsystems to handle wages, taxation, transport, budget planning and major investments.

Other operational systems record orders received from customers, the number of items in stock, orders placed with suppliers, the number of hours worked by employees, time and cost of mobile telephone calls made by subscribers, and so on.

Management support systems

Information systems intended to support management usually work at a much higher level of complexity than operational systems. This is principally because a manager is likely to be interested in summarized information, such as the total quantity of sales of toothpaste last month, rather than an individual transaction, such as a sale to one customer. However, much of the information used by management to make decisions is derived directly from information stored at the operational level. In practice, therefore, many management support systems are built on top of operational systems. Other systems combine elements of the two, meeting a complex set of needs at different levels of the organization.

Many of the earliest management support systems were developed simply by adding a set of programs (known as a management information system, or MIS) to extract data from existing operational systems, and analyse or combine it to give managers information about the part of the organization for which they were responsible. We can easily see how this happened with an accounting system. Once all routine sales transactions were stored on a computer, it was a short step to the realization that if this data were analysed appropriately, it could tell managers at a glance which products were not selling well, which checkout operators took too long dealing with a customer, which store had the lowest volume of trade, and so on.

This information is useful to managers because they have a responsibility to maximize the performance of an organizational subsystem. An important part of this is identifying and resolving problems as they occur. Thus, one crucial aspect of a management support system is the feedback or feed-forward that it provides, alerting managers to problems and opportunities, and assisting them in the process of tuning the organization’s performance. Let us return for a moment to the diagram in Fig. 1.1, and consider this as a human activity system that represents an organization as a whole. The chief difference, then, between operational and management support systems is that they fit into different parts of this diagram. Operational systems either are located in the central box (labelled ‘what the system does’) or, alternatively, they assist its work by supporting the flow of inputs or outputs. Management support systems either are located in the box in the lower part of the diagram (labelled ‘how the system is controlled’) or, alternatively, they assist its work by supporting the flow of feedback to, or control information from, the control unit.

Office systems

Office systems automate or assist in the work of office workers, such as clerks, secretaries, typists and receptionists. They also support some aspects of the work of managers, for example, communication (word-processing, email, etc.), planning (personal information management, such as the diary facilities included in Lotus Notes or Microsoft Outlook) and decision support (for example, using the built-in functions of any modern spreadsheet package). This might suggest that they are a kind of management support system, except that they are used today by almost every kind of employee, not just by managers. This illustrates the extent to which boundaries between different types of system have become blurred, in part due to the spread of information technology through the modern organization. It also highlights the fact that the introduction of information systems often changes the way that people work—many middle-ranking and senior staff now type their own letters and reports using a word processor, when once they might have expected a secretary or a typist to do this work for them.

Real-time control systems

Real-time systems are explicitly concerned with the direct control of a system's operations, often physical in nature. Examples include lift control systems, aircraft guidance systems, manufacturing systems and the robot forklifts in the McGregor system described in Box 1.1. For this reason, they are perhaps best considered as a control subsystem of a physical processing system. Their role is thus very different from both operational and management support systems. Real-time systems usually have human operators (to date, few are completely independent of human supervision, though this may become common in the future), but they are generally insulated from the surrounding human activity system. In fact, many authors would not agree that real-time systems are information systems at all. We do not regard this as an important issue. The techniques used for the analysis, design and implementation of real-time systems are very similar to those used for other computer systems, so in practical terms any distinction is artificial.

1.4.3 Information technology

We have left information technology until last because decisions about information technology should ideally be made last in the cycle of development. When the human activity system has been understood, the need for an information system has been identified, and the information system requirements have been defined—only then should the emphasis turn to the information technology that will implement it. This is not how things always happen in the real world. Indeed, it is partly for this reason that so many systems in the past have been unsuccessful. But it is how they *should* happen, wherever possible. Section 1.5 of this chapter examines in more detail some of the reasons for this recommendation.

Here we offer only a broad definition of information technology, and this only for the sake of clarity. We understand it to include all the varieties of hardware familiarly known as, based upon or that include within them, a computer or its peripheral devices. For example, obvious things like desktop PCs, pocket electronic organizers, modems, network cabling, file servers, printers and computer-controlled machinery in factories and airliners, and also less obvious things like digital mobile phones, the electronic circuits that calculate fuel consumption in

some cars, the microchips in some cameras that set the aperture and shutter speed—in other words, everything misleadingly described in marketing literature as ‘intelligent’. Due to the unprecedented rate of technical progress, the range of devices that can be described as information technology increases almost daily, and the boundaries between them blur. As digital devices continue to advance in speed and processing power, manufacturers exploit these advances to develop and market new products. For example, consider the mobile phones that combine a digital camera, modem, email software, web browser, diary software, alarm clock, calculator and online gaming. Interface technologies such as voice-activation may soon make it much easier to interact with computers without needing to press keys or click mouse buttons, while wireless networking makes physical connection unnecessary. Mobile commerce using handheld PDA devices, mobile phones and wireless headsets is changing the way that many people access information and communicate. For many, it has already removed the physical restriction that requires a user to be in the same place as a bulky PC when he or she wants to access the Internet. On the whole, it appears likely that over the next few years computers will progressively disappear from view, while their effects will paradoxically be felt in more and more areas of everyday life.

All the examples of information technology that have just been mentioned are really just tools that, like any tool, can be used for many different tasks—and not only those for which they were intended. There is a saying that, if your only tool is a hammer, the whole world looks like a nail. The corollary is also true: if you can see only nails, you will be inclined to use any tool that comes to hand as a hammer, regardless of whether it is actually a wrench, a book or a can of beans. So how a tool will *actually* be used matters much more than how it is *meant* to be used. A modern word-processing package provides a skilled user with the facilities to automate many complex tasks by using macro programs, mailing lists and embedded objects like spreadsheets, sound clips and hyperlinks to the Web. Yet many users have no need of all this and would be just as happy with an electronic typewriter. The question is, then, if an electronic typewriter is all that is required, why install a powerful PC running the latest software?

1.5 Strategies for Success

Business strategy is an essential part of the context for this book, since all information systems analysis and design projects begin with the identification of a business issue that can be aided or a problem that can be solved by the use of an information system. The underlying assumption is that information systems are only worthwhile if they meet the needs of the organization in which they are installed. In this section, we consider some ways that business needs can be identified at a very high level, suggesting possible application areas for information systems and information technology.

1.5.1 Identifying a business strategy

Information systems are useful only when they help a human activity system to fulfil one of its goals. In a business organization, then, one important step in specifying an information system is to be clear about the business goals, and the

strategies that the business will use to achieve them. The development of a business strategy essentially begins with the question: 'Where would we like our organization to be in (say) ten years' time?' Logically, the next question is: 'How do we get from where we are now to where we want to be?' In effect: 'How do we meet our business goals?' Answers to this question will ideally suggest some practical steps that can be taken towards achieving the strategic goals. In Chapter 12, we will return to this point from the perspective of managing information for its strategic value, both in private and in public-sector organizations.

The contents of a strategy (i.e. the actual goals and steps it contains) depend on the characteristics of the organization, its environment, the skills of its workforce and many other factors. In the Agate case study, we are told that the strategy is 'to continue to grow slowly and to develop an international market' (these are the goals). The directors also have a view about how to achieve these objectives: they want to get more business from multinationals, and they hope to succeed in this through the quality of their work and by developing 'campaigns that have a global theme but are localized for different markets around the world' (these are the steps). These elements have been included in the strategy because the directors are confident, for example, that the technical quality of Agate's work and the creativity of their staff are both strengths of the company, and will meet the demands placed on them. They probably also believe that their current client base and contact list is extensive enough for them to win the kind of work they are seeking.

1.5.2 The contribution of information systems

Information systems can contribute to the achievement of business goals in so many different ways that it can seem daunting to decide which are the systems that really matter. Many techniques have been developed to help arrive at useful answers, some adapted from their original purpose as tools for developing a business strategy. One example is the well-known SWOT approach⁵. Usually carried out as a group brainstorming exercise, it involves identifying and categorizing everything about the organization's current circumstances that falls into these categories. The resulting strategy is based on finding ways of exploiting the strengths and opportunities, while counteracting the weaknesses and threats. Another is Porter's Value Chain Model (1985), which we consider here. While not the only technique for information system planning, and not necessarily the best for all situations, it is a useful way of structuring this discussion because of the systemic view of an organization that it presents.

The model is illustrated in Fig. 1.9, and depicts an end-to-end 'chain' or flow of materials through the organization. The flow is transformed at each stage, beginning with raw inputs that become a product through operations applied to them, then are shipped and sold to a customer who finally receives an after-sales service. To apply the model, a group of managers and other staff identify the activities and departments that fit the compartments of the model, and identify the value that each adds to the overall product or service.

Porter used the chain metaphor to reinforce the point that all primary activities are essential, but any weak link negates the value of work done at every other stage. For example, if a business is good at selling its products, but the products

themselves are of poor quality, it is unlikely to be successful. Alternatively, a business that makes excellent products but has very poor arrangements for obtaining its raw materials will also not be successful. In a successful organization, each primary activity (the lower part of the diagram) adds value to the products (i.e. it benefits the company more than it costs). Activities in the upper part of the diagram provide services, but do not directly add value to products. They are not essential, and are therefore only worth doing if they make a contribution to the efficiency or effectiveness of primary activities. Their role must therefore be tuned to support primary activities.

The Value Chain Model can be useful in information systems planning since it focuses attention on activities that are critical to a business, either because they are currently a problem, or because they represent a major source of profit or competitive edge. Information systems development projects can then be targeted at assisting those operations that can directly contribute to the success of the organization as a whole.

Porter's original model was based on his analysis of the typical manufacturing company, and this does not apply well to a service sector business such as Agate. In practice, the basic form of the model is often adapted to fit better with the activities of the organization that is being analysed. Figure 1.10 shows a possible value chain for Agate that takes better account of its service nature. Note that this makes no distinction between primary (value creating) and secondary (support) activities. This means that everything done by Agate's staff contributes to the perceived overall quality of service. For this reason, all activities potentially add to (or diminish) the value on offer to its clients.

1.5.3 Information systems and information technology strategies

The best managed organizations separate their strategic thinking into the three layers illustrated in Fig. 1.11. The key idea is that the development of new information systems should only be considered in the context of a well-thought-out

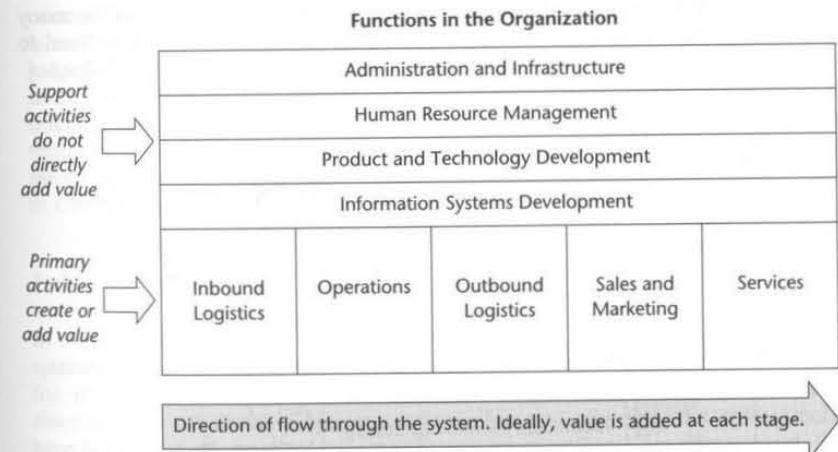


Figure 1.9 The Value Chain Model (adapted from Porter, 1985).

⁵ This stands for Strengths, Weaknesses, Opportunities, Threats.

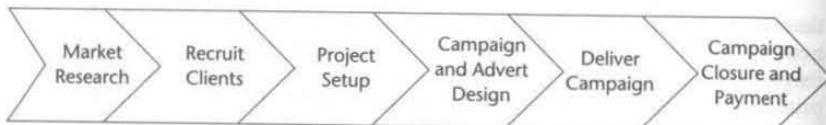


Figure 1.10 Adapted Value Chain Model applied to Agate.

business strategy, while purchases of information technology hardware should only be considered in the context of specific information systems that are planned for development. Thus, the business strategy drives the information system strategy, which in turn drives the information technology strategy. Information flows in the diagram are not only in one direction. In formulating a business strategy, managers need to be advised on those areas of the business where information systems can contribute to fulfilling business goals. Thus the planning cycle is iterative. A similar two-way communication takes place between the information system and information technology strategy planning functions. The role of the information technology strategy is to enable the successful implementation of the systems defined in the information system strategy, while also informing the information system strategists about what is feasible.

The importance of the relationships shown in Fig. 1.11 cannot be overstated. For example, consider the McGregor system described earlier in Box 1.1. One objective of the company's current business strategy is to capture a share of the lucrative online market. This may even be a business imperative. If other online retailers have already taken a significant share of the overall market, they may pose a serious threat to McGregor's continued survival. In order to fulfil this business objective, McGregor's managers must identify, define and then develop an appropriate set of software systems—this includes the online customer order system, the robot warehouse system, the stock control system, the purchase order system, and so on.

It is the main purpose of the IS strategy correctly to identify which of the many possible systems should be chosen for development, and also how these need to

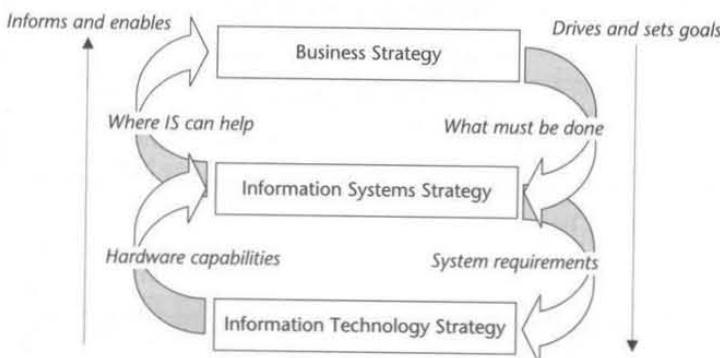


Figure 1.11 The relationship between business, IS and IT strategies.

interface with other systems. Those chosen for development will then become systems development projects. Selection of the wrong systems for development wastes time and resources, and can also distract attention from the business priorities. In this context, 'wrong' may mean systems that are unimportant, but it can also mean systems that are not compatible with other vital systems. In this case, it could cause major business problems for McGregor if the online customer order system cannot correctly transfer details of the items ordered to the warehouse system, resulting in the wrong goods being delivered to the customer. This kind of difficulty could occur simply because a project team is not sufficiently aware of the need for the software it is developing to interface with another system.

Another critical issue might be poor design of the online catalogue, resulting in frustrated customers who buy instead from a competitor's website. While the actual design of the website would not form part of the IS strategy, major business concerns, such as the need for clear navigation and interaction on a customer website, are often spelt out as part of the overall IS strategy.

The IT strategy is responsible for identifying the hardware components and configurations that will allow the software to operate effectively. In McGregor's case, this will include specifying the web servers to ensure that the response time is always fast enough to satisfy customers. Slow response times could also frustrate customers and result in lost sales for the company. The detailed specification of the servers (operating system, number of incoming lines, processors, RAM, etc.) will not be worked out until later, but the strategy will identify this as a business concern, and will also explicitly relate the various hardware components to the software systems they must run.

For many businesses, success depends on finding an appropriate fit between the overall business goals, the information systems that help to fulfil those goals and the IT on which the information systems run. This question of strategy alignment has even greater importance when the business is engaged in e-commerce, whether this is business-to-customer (B2C) or business-to-business (B2B). In either case, for customers, suppliers, partners, collaborators—indeed for any interaction that occurs electronically—the information systems *are* the company, since the website is really all that can be seen. Moreover, an Internet presence can be seen—and judged—by the whole world. Either an inappropriate strategy, at any of the three levels, or a poor implementation can bring swift business failure, as a number of dot.com companies discovered to their cost over the first year of the new millennium. In Chapter 2, we explore in more detail the problems that can occur during information systems development, and even *because of* information systems, while in Chapter 3 we introduce some ways that these problems can be resolved.

1.6 Summary

In this chapter we have introduced the main concepts that underpin information systems. These include some important ideas from general systems theory, such as the ideas of control, communication and emergent properties. We have also discussed the relationship between information and meaning, the central point here being that information only makes any sense within a specific context. For systems analysts and designers this context is usually the human activity system of

an organization. This leads to a necessary set of relationships between the goals of an organization, the steps it must take in order to fulfil them, the information that its staff must have to carry out their activities, the information systems that provide the information, and finally the information technology that supports the information systems. While we recognize that there are many real organizations where this does not happen in the way we describe, we can only wonder how much more could be achieved, and express the hope that readers of this book will start work from firmer foundations.

The idea of information systems is far from new, yet information technology has changed the field almost beyond recognition in the last few years. This process will undoubtedly continue in the future, but however much information systems may change, valuable lessons may still be learned from historical information systems, both those developed since the introduction of the computer and those that pre-date it.

Review Questions

- 1.1** What is the difference between an *information system* and *information technology*?
- 1.2** Identify some things that a computerized *information system* can do, which are difficult or impossible for a non-computerized equivalent.
- 1.3** Why does it not matter whether a system is real, or exists only in someone's mind?
- 1.4** Why are boundary and environment important for understanding a system?
- 1.5** What is the difference between feedback and feed-forward?
- 1.6** Why has a human activity system more than one purpose?
- 1.7** What is the purpose of a management support system?
- 1.8** What is meant by disaster recovery? Why is it important for a business organization?
- 1.9** What are the relationships between business goals, information systems strategy and information technology strategy?
- 1.10** Define information. How does it differ from data?
- 1.11** Describe how knowledge differs from information.
- 1.12** Give an example of some knowledge that you possess. What is its purpose?

Case Study Work, Exercises and Projects

- 1.A** Think of three or four information systems that are not computerized (either historical or contemporary). Identify (or imagine) a computerized equivalent. For each pair, write a brief description of the boundary and the main inputs and outputs. What are the main differences between the computerized and non-computerized versions?
- 1.B** Re-read the description of the online shopping system in Box 1.1, and assume that everything described (computer software, hardware, human activities, etc.) is a single

system. Identify its main subsystems and control mechanisms. What feedback and feed-forward information do you think they would use? Don't be constrained by the description in Box 1.1—use your imagination too. And remember that some control may not be computerized.

1.C Read the first part of the FoodCo case study in Chapter B1, up to and including Section B1.2.1. What do you think are FoodCo's business goals for the next ten years? Make any assumptions that you feel are justified.

1.D Roughly copy out the diagram from Fig. 1.9 (make it large, with plenty of space in each compartment). Try to map FoodCo to the Value Chain Model by identifying people, activities, sections, etc. that fit in each compartment. What flows through your value chain model?

1.E Identify the value that you think is added by each primary activity to FoodCo's products. Which do you think are the weak links?

Further Reading

Checkland and Holwell (1998) is a very accessible account of the relationships between the subjects of systems, information and information systems.

Vidgen et al. (2002) describe approaches to the development of web-based information systems, like some of those described in the McGregor plc example. They also link business strategy with object-oriented analysis and UML (although not the current version).

Chaffey (2004) and Rayport and Jaworski (2003) are both useful sources on e-business and e-commerce (although both are beyond the scope of this book).

Webster (1995) is a scholarly debunking of many of the more exaggerated claims about how the 'information revolution' is changing social relationships.

Turban and Aronson (2001) gives an up-to-date and broad-ranging review of modern software technology for the support of managers at all levels.

Koestler (1967) is a classic text that applies systems concepts to many aspects of life. This book ranges widely over human history and society, and presents some early speculations on the systemic role of evolution in modern social behaviour.

Problems in Information Systems Development

Learning Objectives

In this chapter you will learn

- who the main players are in an information systems project
- the problems in information systems development
- the underlying causes of these problems
- how the concept of a stakeholder helps identify ethical issues in information systems development
- the costs of problems and ethical issues.

2.1 Introduction

It is only by understanding what can go wrong during a system development project that we can hope to avoid failure. Some authors use the metaphor of a journey to describe the process of information systems development (Connor, 1985), highlighting the fact that the main concern in any project is to avoid wrong turns on the way to the destination. At the start of a project, many different routes are available. Some lead to the planned destination, while others lead to a destination that may surprise everyone by turning out to be satisfactory, but is nevertheless unintended. Both are acceptable, although the latter can be nerve-wracking along the way. There are also routes that arrive at the wrong destination, while others are dead ends that lead nowhere. We must recognize and avoid these. This is the explicit purpose of carrying out analysis and design before developing a new system. Analysis is a way of understanding what needs doing before we begin trying to do it, and design is a way of checking that the planned action or solution really meets the needs of the situation, before it is put into practice. An understanding of potential problems is an essential precursor to systems analysis and design.

Projects fail for many different reasons, and in varying degrees. As we shall see, some failures are more visibly catastrophic than others. In this chapter, we look at the question of project failure from the perspective of each of the main players (Section 2.2). We discuss the causes that can lead to failure. These are broadly categorized into those that relate to the quality of what is delivered and those that relate to the productivity of the project (Section 2.3). While not usually a direct cause of project failure, we also consider some ethical issues associated with systems development (Section 2.4). Some of these may act as a delay or an obstacle for the developer, while others only appear as problems when viewed in a wider context. Finally, we mention the costs of the problems, whether or not they result in outright system failure (Section 2.5).

2.2 What Are the Problems?

Many information systems are very successful, and failure is the exception rather than the rule. But the consequences of failure can be severe. One series of reports on information systems projects in the USA estimates that failed projects accounted for \$81bn annually in 1994—almost a third of total information systems development expenditure (Standish Group, 1994, 2001). While this had substantially improved by 2001, the outright failure rate was still one project in four, while 49% of a sample of almost 300,000 projects were described as ‘challenged’—in other words, ‘late and/or over budget’ (Standish Group, 2001). A similar picture prevails in the UK, where a recent survey for Accenture found that approximately one in three IT managers experiences project failure in over 40% of cases (Ranger, 2005). The bill for one project abandoned in 1993 (the London Stock Exchange Taurus system) was estimated at around £480m (Drummond, 1996). Many UK public-sector contracts for information systems have been cancelled in recent years or have failed to deliver their intended benefits, although the true costs are often not known with any certainty. Significant problems have occurred with systems for the UK Passport Agency, the National Probation Service, the Immigration and Nationality Directorate, the Child Support Agency and the National Air Traffic Control Service. Often poor project management and lack of financial control are blamed for the disasters (OGC, 2005). But all potential causes of failure are at least to some extent under the control of the developers. A professional must take the possibility of failure seriously, and work hard to avoid it, even if this sometimes is limited to an awareness of the risks, followed by a damage-reduction exercise.

One difficulty with the question ‘What can go wrong?’ is that the answer depends partly on who gives it. But this is not surprising, as information systems development is a complex activity that always involves people. This is not to say that all people are difficult—this only applies to a very few! But, in any organization, people have varying perspectives that influence their view of a situation, and what, if anything, is to be done about it. Although some differences in attitude depend on psychology or personal history, which are beyond the scope of this book, others relate to an individual’s position within the organization and their link to an information system project. It is useful to discriminate between three categories of people with important relationships to a project. The first is the group

of employees who will become end-users of the information system when it is completed. The second is the group of managers, here called 'clients', who have control (or at least influence) over the initiation, direction or progress of a project. The third is the group of professionals responsible for the development of the information system. For the moment, we ignore differences within each group.

2.2.1 An end-user's perspective

End-users come in many varieties, and can have varying relationships to an information system. The examples that follow concentrate on the experiences and frustrations of those who either consume the products of an information system (i.e. they use its outputs to help achieve a task, such as dispatching an ambulance to an emergency) or are responsible for entering raw data into an information system.

'What system? I haven't seen a new system'

One problem that can be experienced by an end-user is vividly expressed by a term that gained widespread usage in the 1980s. *Vapourware* describes a software product that is much talked about, but never released to its intended users. In other words, instead of arriving, it evaporates. While many businesses are reluctant to talk about information system project failures in public, vapourware may be surprisingly common.

Some surveys have found that an astounding proportion of information systems development projects fail to deliver any product at the end. In the USA, one author has suggested that 'as many as 25% of all projects in large MIS departments are never finished' (Yourdon, 1989), while more recent research found that 23% of projects failed completely (Standish Group, 2001). In the UK, the total cost of a single system for the Wessex Regional Health Authority reached £63m, yet no system was delivered (Collins, 1998b). This last example indirectly hurt patients, as the aim of the system was to help manage hospital resources more effectively throughout the region, and thereby deliver a better, more responsive service. When a project is not completed, the expected benefits to users and other beneficiaries are not achieved.

'It might work, but it's dreadful to use!'

This relates to systems that are unpleasant or difficult to use. Systems may fail to meet the criterion of usability in a number of ways, including: poor interface design, inappropriate or illogical sequence of data entry, incomprehensible error messages, unhelpful 'help', poor response times and unreliability in operation. Figure 2.1 gives some examples (what these mean, and how to avoid them, is explained more in later chapters).

Not long ago one of the authors bought a pair of shoes in a local shoe shop, and saw staff struggling to register the sale correctly with a new cash register system. The difficulty arose indirectly from a promotional offer. Any customer buying this style of shoe was also entitled to a free pair of socks. Since the socks were a normal stock item, correct records of stock had to be maintained. This meant that the socks had to be 'sold' through the till, even though they were actually being given away for free. A simple way to handle this would have been for the assistant to over-ride the price with zero at the time of sale. The assistant tried this, but the

System characteristic	Example
Poor interface design	A web page with yellow text on a white background
Inappropriate data entry	A system where the backspace key sometimes deletes whole words
Incomprehensible error messages	A system message that says 'error #13452'
Unhelpful 'help'	A system message that says 'wrong date format—try again'
Poor response times	Nurses in an Intensive Care Unit complained that their new computerized patient chart system took longer to store and retrieve data, compared with the manual system. This took them away from patients for longer ¹
Unreliability in operation	A national motor vehicle insurance company lost most of its digital records of customers' policies due to a system error. Staff were unable to send renewal notices, but were compelled instead to write to customers asking them to phone in with their policy details ²

Figure 2.1 Aspects of poor system usability with examples.

software specifically prevented the 'sale' of a stock item at a zero price. The assistant called the manager. After some experimentation, it appeared that the only way to deal with this transaction was to reduce the price of the shoes by 1p, and to sell the socks at a cost of 1p, thus giving the correct total for the sale. Now that staff know how to do this, it will not cause so much difficulty in future. But it will always be an unnecessarily awkward way of handling a routine task. There are many examples of this sort of poor design, and they cause a great deal of frustration and lost time for users.

'It's very pretty—but does it do anything useful?'

A system may appear well designed and easy to use, but still not do the 'right' things. This may be a question of the tasks that should be carried out by the system. For example, a library catalogue enquiry system would be of limited use if it could only retrieve shelving information about a book when provided with the title and the author's name, in full and spelt correctly. Readers often do not know the title of the book for which they are searching. Even if the author's name is known, it may be spelt incorrectly. Another way that a system may fail to meet its users' needs is through poor performance (this overlaps with the question of usability, discussed in the previous subsection).

A system may also be of doubtful value to its users because it requires them to work in a way that seems nonsensical. One report describes a warehouse management system, designed partly to increase managers' control over the routine activities of warehouse workers. The workers found that the new system removed much of their discretion in their work, and this prevented them being able to maximize the use of scarce storage space.

¹ Source: Goss et al. (1995).

² One of the authors was a customer at the time.

... because they could see how improvements in these areas would save money for the company they found ways of working around the system.
 ... They were reproved by management for their bad attitude, and yet, it was their commitment to the company as a major employer in their local community which led to their frustration with what they regarded as unnecessarily wasteful rules and procedures.

(Symons, 1990)

It is much more worrying when software errors and failures present a hazard to life. An extreme example was the London Ambulance Service emergency dispatch system, abandoned shortly after delivery in 1992, after incurring a total development cost estimated at £43m.

Far from speeding up the dispatch of London's ambulances and paramedic teams, response times became slower than ever.

After a series of high-profile failures, the system was canned amid claims—never officially substantiated—that it was responsible for several needless deaths.

(Barker, 1998)

Controversy also continues to this day about whether it could have been software errors that caused the crash on the Mull of Kintyre of a Royal Air Force Chinook helicopter in 1994. All 29 people on board were killed, including a number of high-ranking police and military intelligence officers. The official verdict, still accepted by the Government, stated that the helicopter crashed because its pilots were grossly negligent. However, an earlier Board of Enquiry concluded that the crew may have been distracted by a major technical (i.e. software) malfunction. A series of reports in *Computer Weekly* and an item on British TV's Channel 4 News claimed that internal Ministry of Defence reports had raised concerns about the reliability under certain conditions of the engine control software in this type of helicopter. A House of Lords select committee report recommended that the Ministry of Defence view should be set aside (Collins, 2001), but to date the Government has refused to do so and continues to blame the pilots.

2.2.2 A client's perspective

By *client* we mean that person, or group of people, responsible for paying for the development of a new information system. A client usually has influence over whether or not approval is given to a project before it starts. Some clients (but not all) also have the power to stop a project once it is under way. A client may also be a user. If so, we can assume that they share the user's perspective on the kind of things that can be a problem. They may make only indirect use of the system's outputs, which insulates them from the immediate experience of a badly designed interface, for example. While the concerns of a client may overlap with those of an end-user, they also include distinct matters relating to payment, ownership and value-for-money.

'If I'd known the real price, I'd never have agreed'

It is almost routine in many organizations for information systems projects to exceed their budget. We have already mentioned the *Chaos Reports* (Standish Group, 1994; 2001), which show that up to half of all projects run over budget

and/or behind schedule. Any project that overruns its budget can reach a point where the total costs outweigh the benefits that will be provided on completion. This point is not always recognized when it is reached, which may result in the expensive completion of a system that would have been better cancelled. In other cases the project is cancelled, either because it is recognized that costs are escalating out of control, or because it becomes apparent the benefits will not be as great as originally promised. The rationale is summed up in the saying: 'Don't throw good money after bad.' The London Stock Exchange Taurus system mentioned earlier in the chapter was a striking example of this.

The rise of e-commerce since the late 1990s has brought new ways for information systems to cost the organization money, sometimes as a result of what appeared to be no more than routine modifications to the software in use. One example of this caused Barclays Bank serious embarrassment in August 2000 when some customers logged onto the online banking service and found that they could view other customers' accounts (BBC, 2000).

'It's no use delivering it now—we needed it last April!'

A project that is completed late may no longer be of any use. For example, a bricks-and-mortar retailer, threatened by rivals who sell at a lower price on the Internet, may have little use for an e-commerce website if it is not operational until all the customers have defected and the company has been declared bankrupt.

Many other kinds of project are time-critical. This can be due to new legislation that affects the organization's environment. An example of this was the deregulation of the UK electricity supply market in April 1998. This required electricity companies to make extensive modifications to their computer systems so that they would be able to handle customers' new freedom to switch between suppliers. A few years earlier, all local authorities in the UK faced a similar challenge twice in three years, when central government changed the basis for local tax calculations. Each change required hundreds of councils to specify, develop (or purchase) and successfully install new computer systems that allowed them to produce accurate invoices and record income collected from local tax-payers. Failure to implement the new systems in time risked a massive cashflow problem at the beginning of the new tax year.

Commercial pressures can also have an effect. This sometimes translates into whether a business succeeds in being the first to market a new product or service, although the advantage is not always permanent. For some time the continuing success of the Internet bookstore Amazon.com derived, at least in part, from the considerable competitive advantage of being the first of its kind. Some competitors (notably the established US bookseller Barnes and Noble) felt obliged to follow Amazon's lead. For the followers, there is not the same need to take risks with new technology. But attracting customers away from a leader may mean differentiating yourself in some way, perhaps offering new services, or perhaps by being even better at what the leader already does well. At the time of writing (Spring 2005), Amazon seems to have successfully weathered the many dot.com crashes during the year 2000. Some press reports in 2001 suggested that Amazon's commanding lead might be faltering, but in 2003 its annual revenue exceeded \$5 billion. While Amazon still does not show a sustained profit, it recently joined Forbes.com's list of the 25 fastest growing American companies (DiCarlo, 2004) and looks set to continue growing for some time yet.

'OK, so it works—but the installation was such a mess my staff will never trust it'

Once a new system gets a bad press it can be very difficult to overcome the resistance of the people who are expected to use it, particularly if there is an existing alternative available to them. The following scenario is based on a real situation, observed at first hand by one of the authors.

A small company introduced a local area network (LAN) to connect the PCs in its office. Staff were encouraged to store files on the LAN disk-drive, where other staff could also access them (previously, all data was stored on local hard drives, accessible only from one PC). Most saw the mutual benefit of sharing information, and complied. Management claimed that the daily back-up of the LAN drive onto tapes was a further benefit, since there was no longer a need to keep personal back-ups on floppy disks. Then a mechanical fault occurred. This erased all data on the LAN drive, and when the engineer tried to restore it from the tape, it emerged that the tape drive had not operated correctly for several weeks. All tapes recorded over the previous six weeks were useless. Staff were told that all data stored in that time was permanently lost. Re-entering it all took many person-days. The faulty disk and tape drives were replaced, and tapes are now checked after every back-up, but many staff reverted to keeping all important data on their local hard drives. Most keep personal back-ups on floppy disks too. Perhaps nothing will persuade them to trust the LAN again.

'I didn't want it in the first place'

Organizations are complex and political by nature. The politics with which we are concerned here are to do with conflicting ideals and ambitions, and the play of power within the organization. There can be disagreement between management and workers, as in the case of the warehouse management system mentioned earlier in this section. There can also be contention between individual managers, and between groups of managers. One result can be that a manager is sometimes an unwilling client in relation to a project. The following scenario is based on another real-life situation observed by one of the authors.

The head office of a multinational company decided to standardize on a single sales order processing system in all its subsidiaries throughout the world. But the Hong Kong office already had information systems that linked electronically with customers in Singapore, Taiwan and other places in South East Asia. It became apparent that the existing links would not work with the new system. For the Hong Kong management, this meant extra costs to make the new system work, and disruption to established relationships which, in their view, already worked smoothly and did not need to be changed. They therefore had little desire to see the project succeed in their region, but felt they had no other choice. Had they been less scrupulous, they might have tried to find ways of sabotaging its progress, either in the hope that it would be abandoned altogether, or at least that they might be exempted from the global rule.

'Everything's changed now—we need a completely different system'

It is almost inevitable for any information system project that, by the time the system is completed, the requirements are no longer the same as they were thought to be at the beginning. Requirements can change for many reasons.

- Project timescales can be very long (the Taurus project ran for three years) and business needs may change in the meantime.

- Users naturally tend to ask for more, as they learn more about what is available.
- External events can have a dramatic impact—for example, a currency crisis that led to the collapse of a significant market might render a new information system pointless.

This does not apply only to new systems currently under development. Systems that have been in operation for some time may also be affected. This is part of the natural, ongoing process of maintenance, modification, upgrading and eventual replacement that all information systems undergo, and a topic to which we return in Chapters 3 and 19. From a client's perspective, the motivation is usually to make an information system fit better with the business, and therefore provide better support for business activities.

2.2.3 A developer's perspective

The perspective of the developer is quite different both from that of an end-user and from that of a client. This is because the developer adopts the role of 'supplier' to the 'customer' (i.e. client or end-user). For this reason, when problems occur the developer may feel forced into a defensive position of justifying the approach taken during the project. Since at this stage we are discussing only problems, many of the problems identified by a developer tend to centre on blame and its avoidance.

'We built what they said they wanted'

Changes to the requirements for a system, based on sound business reasons, always seem perfectly reasonable from a client's point of view. However, for a developer, given the responsibility for building a system to meet those requirements, they can be a real headache. If we were able to distil the essence of how many developers feel about this, it would read something like the following.

No matter how skilled you are, you can't achieve anything until the users, clients, etc. tell you what they want, and at the start they don't even agree with each other. Eventually, with skill and perseverance, you produce a specification with which everyone is reasonably happy. You work for months to produce a system that meets the specification, and you install it. In no time at all, users complain that it doesn't do what they need it to do. You check the software against the specification, and you find that it does exactly what it was supposed to do. The problem is that the users have changed their minds. They just don't realize that it's not possible to change your mind late in a project. By then, everything you have done depends on everything else, and to change anything you would almost have to start all over again. Or it turns out that they didn't understand the specification when they accepted it. Or there is some ambiguity about what it meant, and you've interpreted it differently from them. Whatever the reason, it's always your fault, even though all you ever tried to do was to give them what they wanted.

In reality, of course, analysts, programmers, etc. often do understand why users and clients change their minds during a project, or after delivery. But this doesn't always make it less frustrating when it happens.

'There wasn't enough time to do it any better'

In every project, there are pressures from outside that limit the ability of the development team to achieve excellence. In the first place, projects almost invariably

have a finite budget, which translates directly into a finite amount of time to do the work. There may also be an externally imposed deadline (for example, a project to develop a student enrolment system that must be ready by the start of the academic year). Another external pressure results from the impatience of users and clients to see tangible results. This, too, is often understandable, since they are not so much concerned with the information system itself, as with the benefits it can bring them—an easier way to do a tedious job, a quicker way to get vital information, and so on. But it can be very counter-productive if it becomes a pressure within the project team to cut short the analysis and get on with building something (anything!) quickly to keep the users happy. The result of haste in these circumstances is usually a poor product that meets few of the needs of its users. Developers know this, but they don't always have the power to resist the pressure when it is applied.

'Don't blame me—I've never done object-oriented analysis before!'

In a successful information system development team, the members must possess a harmonious blend of skills that are appropriate to the needs of the project. These may include the use of techniques (such as object-oriented analysis), knowledge of methodologies (such as the Unified Software Development Process), skill in programming languages (such as C# or Java) or detailed knowledge of hardware performance (such as networking devices). There must be a complementary set of skills within the team for a project to succeed. For example, the analysts must all use the same or related techniques otherwise the results will not be coherent (e.g. the products of object-oriented analysis are entirely incompatible with those of structured analysis). Similarly, all programmers must be able to interpret the designs or else they will not know what the class operations are meant to do. Problems occur when the available staff do not possess adequate expertise in the particular skills required for a project. Many highly skilled and experienced systems analysts may know SSADM very well (a structured methodology used widely in the UK), but have little experience of object-oriented analysis. Many excellent programmers know Visual Basic inside out, but have little experience in Java or Smalltalk. As a consequence, some projects with highly skilled staff are still carried out rather poorly, because the staff are inexperienced with the particular techniques they must use.

'How can I fix it?—I don't know how it's supposed to work'

This complaint is often heard from programmers who have been asked to modify an existing program, and who have then discovered that there is no explanation of what it does or how it works. To modify or repair any artefact, whether a computer system or a bicycle, it is usually necessary to understand how it was intended to work, and thus what the consequences are of changing this or that feature. Anyone who has ever tried to repair anything electronic or mechanical, such as a motor vehicle, washing machine or VCR, will know that much of the time is spent trying to understand what the various parts do, and how they interact with each other. This is true even when a maintenance manual is to hand. The situation is no different for computer software. While software may be more intangible in form than a VCR, it is no less mechanistic in its operation.

'We said it was impossible, but no-one listened'

Just like client managers, systems developers can sometimes be overwhelmed by organizational politics. At times this means that a project is forced on an unwilling

team, who do not believe that it is technically feasible to achieve the project's goals. Alternatively, the team may not believe the project can be completed within the time made available. But if opposing views prevail, the team may find itself committed to trying to achieve what it said could not be done. In these circumstances, it can be very hard for team members to become enthusiastic about the project.

'The system's fine—the users are the problem'

A few information systems professionals, usually those who understand least about business and organizations, are sometimes prone to blame 'the user' for everything. Those who hold this attitude believe that problems that occur in the use of software inevitably result from the fact that most users are far too stupid or ignorant to make proper use of the system. On the other hand, they believe that the design and execution of the software is not open to serious question. Many of these technocrats are undoubtedly very talented, but this view is patently absurd since it assumes that the answer to a problem is known before the situation has even been investigated. In a word, it is a prejudice. We will say no more about it, other than to comment that anyone who hopes to learn the truth about a situation must also be prepared to examine critically their own preconceptions.

2.3 Why Things Go Wrong

Flynn (1998) proposed an analytical framework to categorize project failures, and this is illustrated in Fig. 2.2.

Type of failure	Reason for failure	Comment
Quality problems	The wrong problem is addressed	System conflicts with business strategy
	Wider influences are neglected	Organization culture may be ignored
	Analysis ³ is carried out incorrectly	Team is poorly skilled, or inadequately resourced
	Project undertaken for wrong reason	Technology pull or political push
	Users change their minds	
	External events change the environment	New legislation
Productivity problems	Implementation is not feasible	May not be known until the project has started
	Poor project control	Inexperienced project manager

Figure 2.2 Causes of IS project failure (adapted from Flynn, 1998).

³ To this category, we would add design and implementation. Even when analysis is carried out correctly, this is still no guarantee that the software will be well designed, nor that it will be correctly programmed.

Complete failure is the most extreme manifestation of a problem, but Flynn's framework can also be applied to less catastrophic problems. In Flynn's view, projects generally fail on grounds of either unacceptable quality or poor productivity. In either case, the proposed system may never be delivered, it may be rejected by its users or it may be accepted yet still fail to meet its requirements.

These categories are what are sometimes called 'ideal types'. This means that they are intended to help explain what is found in reality, but that does not imply that any real example, when examined in all its detail, will precisely match any one category. Real projects are complex, and their problems can seldom be reduced to one single cause. Many of the examples in the following sections show some of the characteristics of more than one category of cause.

2.3.1 Quality problems

One of the most widespread definitions of the quality of a product is in terms of its 'fitness for purpose' (see any standard text on software quality assurance, e.g. Galin, 2003). In order to apply this to the quality of a computer system, clearly it is necessary to know (i) for what purpose the system is intended and (ii) how to measure its fitness. As we shall see, both parts of this can be problematic at times.

The wrong problem

Here the emphasis is on the purpose for which a new system is intended. The argument goes like this: if an information system does not contribute anything worthwhile to the aims of the organization, then it is, at best, a waste of resources and a distraction from those things that really matter. At worst, it may do real harm if the objectives of an information system are in direct conflict with the organization's business strategy. The difficulty is in knowing the right viewpoint to take when defining the aims of a project. If the aims of the organization as a whole are unclear, or are not communicated to those responsible for planning information system projects, there is always a risk that a project will fall into this error. It may then be regarded as a complete success by its developers and users, yet appear a failure when seen in a wider frame of reference.

A primary cause of system failures in general is that some projects are started with no clear idea about exactly what are the nature and goals of the client organization. This means that failure, or at least lack of success, is almost inevitable. If an organization itself is not understood, then it is extremely difficult to identify and develop information systems that support it in fulfilling its aims.

Neglect of the context

This emphasizes the fitness of an information system to fulfil its purpose. This can take the form of a system that is too difficult to use, since the designers have taken insufficient account of the environment in which its users work, or the way that they like to work. Some examples given earlier in this chapter can be interpreted in this way, depending on assumptions about the situation. For example, in one case cited earlier (Section 2.2.1) workers and managers held different views about the purpose of the warehouse management system. Managers believed they needed to control the activities of workers more closely. Yet the system designed to do this had also the side effect of obstructing the workers from carrying out their work in an efficient way, to the detriment of the whole company.

Incorrect requirements analysis

We prefer to interpret this category as including not only analysis, but also design and implementation. Here, too, the focus is on a system's fitness to fulfil its purpose, rather than the purpose itself. Even if the aims are clear at the outset, many pitfalls lie along the route, particularly if the development team does not have the right skills, the right resources or enough time to do a good job. However, even when none of these present a difficulty, the project can still fail if the techniques being applied by the team are inappropriate to the project.

The results of this category of failure are generally among the most visible systems problems, when seen from a user's point of view. This may be because they directly affect the external design of the system (e.g. the content or layout of its screens), or the selection of tasks that the system performs (e.g. an essential function may not be included), or the operation of the software (the system as coded may not work in the way that its analysts and designers intended).

Project carried out for the wrong reason

Here, the emphasis is once more on the intended purpose of the system. To give a recent example, over the last few years many organizations seem to have rushed into some sort of e-commerce activity. While this is clearly a great success for some, others have derived little or no benefit. During the year 2000 there seemed to be so many 'dot.com' crashes that the *Guardian* newspaper's website ran a column called 'Dot.com deathwatch', which featured only stories about troubled and failing Internet companies.

It has been pointed out that many organizations did not think carefully enough about some key questions (McBride, 1997).

- What does the business aim to achieve by a presence on the Internet?
- How must the business re-organize to exploit the opportunities offered by the Internet?
- How can the business ensure that its presence on the Internet is effective?

The sensible conclusion must be that there is a great deal more to successful trading on the Internet than just writing a few web pages in HTML and placing them on a web server where surfers can find them. Some organizations, in moving onto the World Wide Web, are simply following a trend. They do so not because they understand what it can do for their business, but precisely because they do not understand, and therefore fear the consequences of being left out of something good.

Two possible underlying reasons explain why this can happen. One of these is political push within the organization. For example, a powerful group of managers may wish the business to look modern, even when no clear benefit has been identified. The other is the pull of new technology. An organization is very vulnerable to this if the most senior managers have little understanding of information technology, and therefore no rational basis for evaluating the exaggerated claims that vendors are prone to make about their newest products. In practice these two reasons often combine into a single force that can be irresistible.

Other companies, typically Internet start-up businesses with no history of trading in the physical world, seem simply to have believed that the Internet represented a 'new economy' where established business rules no longer applied. However, while some aspects of business on the Internet are clearly different—in

particular, the speed at which things happen, including business failures—the business fundamentals have not really changed. It is still just as important to plan and design with care, to pay attention to costs and income and to ensure that proper controls are in place.

Much of the naive and sloppy thinking that led to the dot.com crashes may no longer be in evidence, but the central point applies just as much now as it ever did in the past.

2.3.2 Productivity problems

Productivity problems relate to the rate of progress of a project, and the resources (including time and money) that it consumes along the way. If quality is the first concern of users and clients, then productivity is their other vital concern. The questions that are likely to be asked about productivity are as follows.

- Will the product be delivered?
- Will it be delivered in time to be useful?
- Will it be affordable?

Requirements drift

Requirements almost always drift if they are allowed to do so. In the simplest case, this just means that users' requests change over time. In principle, it would be unreasonable to prevent this from happening, but, in extreme cases, change requests can bedevil a project and even prevent its completion. The longer a project proceeds, the more complex both its products and its documentation become. To compound this, each part of the final system probably depends on many others, and the inter-dependencies grow more numerous and complex. It becomes progressively more difficult over time to make changes to a system under development, because any change to one part requires changes to many other parts so that they will continue to work together. A limit is reached when a project is stalled by an escalating workload of making all the changes required as a consequence of *other* changes that were requested by the users. At this point, management have only two choices. They can cancel the project and write off the money spent so far (this happened at the London Stock Exchange in 1993). Alternatively, an effort can be made to bring the project back on track. This is almost always both difficult and expensive, and requires highly skilled management.

External events

This is one cause of failure that is normally beyond the control of both project team and higher management. Depending on the environment in which the organization operates, decisive external events can even be impossible to anticipate. Nevertheless, it is prudent on any project at least to assess the vulnerability of the project to external events, since some are much more at risk than others. For example, a project to build a distributed information system that is to operate on new, state-of-the-art computers communicating over public telephone circuits may be sensitive to external factors such as the reliability of the telephone network and call pricing. By contrast, a project to build an information system that will operate on existing, tried and tested hardware within one building can safely ignore these factors.

Poor project management

The manager of a project is ultimately responsible for its successful completion, and it could therefore be argued that any project failure is also a failure of the project management. To some extent this is true, but there are also some cases where the only identifiable cause of failure overall is a failure of management. This is almost always due to either poor planning at the start, or a lack of care in monitoring progress against the plan. As a result, the manager allows the project to falter, or permits its costs to grow in an uncontrolled way.

Implementation not feasible

Some projects are over-ambitious in their technical aims, and this may not become evident until an attempt is made to implement the system. This is particularly the case when a system is intended to work together with other systems, whether or not these are already in use. The problems of testing and debugging a new system can grow steadily more complex as attention is focused on larger and larger subsystems. Sometimes the task of interfacing several large, complex software systems, written in different programming languages, installed at different sites and running on different makes of computer hardware, can turn out to be impossible to achieve.

An example of the kind of complex installation routinely tackled in many modern projects is a £25m system called Caseman, which has computerized many administrative tasks in the 250 county courts of England and Wales. This system is based on an Oracle database combined with the WordPerfect word-processing package, running on the UNIX operating system on DEC computers. Although each of these components, taken independently, is thoroughly tried and tested, nevertheless implementation was delayed because the company responsible for the project encountered problems in making the different parts work together (Schneider, 1997). A further difficulty occurs when each supplier insists that their product is performing exactly as specified and any problem must lie elsewhere. It can then be very hard to locate the exact origins of the problem.

Technical problems with the implementation do not always become evident until after the system is implemented. One example of this was with the LAS ambulance despatch system. A practical difficulty that staff encountered in its use was with the deliberately high-tech design that used a digital on-screen map. The digital map interface was very unpopular with its users, who found it difficult to pinpoint an ambulance with sufficient accuracy and thus regarded it as dangerous (Barker, 1998). This was a contributory factor to the system's overall failure to dispatch ambulances speedily to emergencies, resulting from the application of a new technology to a critical task that was not sufficiently understood.

An implementation problem led to the crash of the online sportswear retailer Boo.com in May 2000. The software for their website was much delayed in development, but proved a disaster even when delivered. It turned out that very few home PCs were sufficiently advanced to run the sophisticated 3D visualizations without crashing. Even when the software ran without crashing, most images were very slow to download, adding to the users' frustration. As a result, too few customers bought from the website. The company called in the liquidators after reportedly spending £80m over six months, with no realistic prospect of sales increasing to the point where the business would become viable (Hyde, 2000).

Problems of this kind can usually be avoided by sound design practices. Beginning with Chapter 12, we consider later what the software designer can do to avoid costly and damaging failures.

2.4 The Ethical Dimension

Ethics can be loosely defined as a branch of philosophy that is concerned with the rightness or wrongness of human character and conduct, and the establishment of moral rules or principles to guide our behaviour. In practical terms, thinking about ethics normally means that we are trying to establish a way of judging the effects that one person's behaviour has on other people.

Given that all computer-based information systems have a direct effect on someone's life, it is hard indeed to think of one that does not have a significant ethical dimension to its design, construction or use. As with any system of rules, it is necessary to consider the consequences if they are broken. It is not overstating the case to say that a breach of the ethical rules that apply to any aspect of an information system project is necessarily a problem for someone. Sometimes these problems are noticeable at the time and must be handled within the project; they can otherwise lead to a failure of the project overall. For example, a system intended to automate some activities of a business may result in job losses among the staff. At other times there is hidden damage to a project, an organization, the information systems profession as a whole or even in extreme cases to society at large. For example, there is a debate about whether individual users of email systems have a right to privacy from surveillance. Some evidence in the prosecution case that led to the impeachment proceedings against President Clinton was gathered from personal emails. Investigating officers were apparently able to retrieve copies of certain sensitive messages, even though their author believed that all copies had been safely destroyed (*Wall Street Journal*, 1998).

One of the difficulties in assessing the ethical issues in a project is that the person who may have a problem is not necessarily the developer of the system, its user, its client—or indeed anyone at all who is obviously connected with the project. In fact, one of the first problems that must be faced in this area is the identification of all the people who may be affected by the system. These are often called *stakeholders*, since, in their different ways, each has a stake in the outcome of the project.

To illustrate the diversity of stakeholders who may be associated with a project, consider the introduction of a network of bank ATM machines in the branches of a supermarket chain. Figure 2.3 shows a preliminary tracing out of the possible effects of this system, together with the groups of people affected.

You may not feel that every group identified in this analysis is equally affected by the project, and in some cases you may feel that the effect is quite minor. However, until such an analysis has been carried out, it is not possible to speak with any authority about the nature or the extent of impact that a new information system will have.

2.4.1 Ethical issues within a project

The issue of professionalism is at the forefront of any discussion of ethics within information systems development. By their very nature, information systems

Stakeholder affected	Possible consequence of system	Nature of effect on stakeholder
Bank clerks	Automation of bank activities currently carried out manually	Reduced need for staff—redeployment or redundancies
Bank customers	More convenient access to bank services	Improved service
Supermarket customers	More people using supermarket car park	Reduction in service
Bank shareholders	More people attracted to use bank, so greater commercial success	Increased dividends
Supermarket shareholders	More people attracted to use supermarket, so greater commercial success	Increased dividends
Local citizens	More journeys to supermarket to use ATM	Increased pollution

Figure 2.3 Possible stakeholders in a bank ATM network.

projects have deep consequences for the lives and work of many people who either will use the software or will be affected in some way by its use. There are several reasons for this. First, information systems projects are often major investments for the client organization, and money spent on these is necessarily money that now cannot be spent on other worthwhile projects. Second, information systems projects are often concerned with the way that important business activities are carried out, and they can therefore have a direct effect on the overall success or failure of the organization. Third, the introduction or modification of information systems often causes radical changes in the way that employees carry out their work and how they relate to their managers, colleagues and customers. Given these responsibilities, it is important that project team members behave in a professional manner. To some extent this is just a matter of being aware of the rules for behaving in a professional way, and then following them.

Behaving ethically is usually not just a matter of applying a straightforward rule in a mechanical way. We often find ourselves confronting ethical dilemmas, where no one course of action seems entirely right, yet we must find some basis for making a decision. This is compounded when our actions affect different stakeholders in different ways.

Some ethical effects arise from aspects of a project that are not under the direct control of an individual developer. For example, Sachs (1995) describes a 'Trouble Ticketing System' that was intended to improve the work efficiency of testers (telephone repair staff) by tracking the progress of repairs and allocating the next step in a job to any tester that was available at the time. In practice, the system disrupted informal communication between employees, thus damaging their efficiency instead of improving it. According to Sachs, the fault lay in a failure to develop an adequate understanding of the testers' working practices, in particular the way that their informal communications helped in troubleshooting difficult problems. But individual developers are rarely free to choose which aspects of a situation should be analysed.

Legislation adds a further ethical dimension for the members of an information system project team. In the UK the relevant legislation includes the Data Protection Act 1998, the Computer Misuse Act 1990 and the Health and Safety (Display Screen) Regulations 1992. The increasing use of the Internet as a medium for information and exchange also brings cross-border complications. One example was the widely reported 'Internet adoption' case, which involved a British couple who adopted twin baby girls advertised on a website based in the USA. A British court later declared the adoption to be invalid and the twins were returned to the care of their natural father in the USA. In many cases it is far from clear which laws apply when information and services are provided via the Internet to residents of one country, but either the service provider or the information content are hosted within a different country.

2.4.2 Wider ethical issues

Among the wider ethical issues associated with information systems development, one of the longest-running debates is over the effect that IT and IS have had on levels of unemployment throughout the world. Some authors claim that, within the foreseeable future, one effect of the spread of information technology will be an unprecedented, sustained rise in global unemployment that will leave countless millions idle and impoverished (Rifkin, 1995). Others argue that, while IT has certainly destroyed some jobs (tens of thousands of jobs lost in the UK banking sector over the last decade), it has also created many new jobs to replace them, often bringing new opportunities to neglected backwaters in the world economy.

For many years, an ethical debate has raged between the proponents of freedom (some would say anarchy) on the Internet and the big business interests, who already own its infrastructure and may soon dominate most of its content. In its brief history, the Internet has been at the centre of many debates revolving around freedom of access, including the widespread concerns that have been expressed about the ready availability of pornography, political materials and other contentious content. Yet for many others, this chaotic situation has brought unprecedented opportunities to meet (in a virtual sense) and communicate with similarly minded people all over the world.

One issue that is likely to receive more attention in the next few years is the way that some companies use new applications of information technology to gather unprecedented quantities of detailed personal information. For example, at the time that the first edition of this book was being prepared, a South African bank had recently begun to issue its customers with mobile phones that displayed their current account balance each morning. The phones also enabled the bank to monitor all phone numbers dialled, and this added to the bank's profile of data about individual customers. Customers are categorized, and those that the bank feels are least profitable are 'encouraged' to leave the bank, e.g. by higher service charges. The bank was said to be considering plans to add a geographic information system capability, which would also allow customers' daily movements to be tracked, and a link with Internet providers, which would allow the bank to ascertain which websites are visited by their customers (Collins, 1998a).

Questions of access to computers and computerized information have wide implications. There have been arguments over the years about whether the use or the availability of computers has tended to favour men over women, the middle

class over the working class, and those in the affluent north over those from the poor south of the globe. Some even suggest that computers and the Internet have already begun to create a new division in the world's population, dividing people everywhere into the information-rich and the information-poor.

So many things are changing so fast at present that it is not possible to be sure how these questions will seem in a few years' time. What is clear is that future generations will look back on our time as one of great change. Such an era is inevitably one that raises profound ethical questions about the effects on our fellow citizens of the way that we design and apply technology.

2.5 Costs of Failure

Many projects discussed in this chapter, particularly the more famous ones (Taurus, the LAS despatch system and Boo.com), have very large price tags attached. Some companies are widely thought to be reluctant to admit their failures, since this can reflect badly on the business as a whole, perhaps damaging customers' or investors' confidence. For this reason, it is thought that the known, high-profile failures represent only the visible tip of a much larger iceberg. But these are not the only costs associated with project problems. Many costs are associated with projects that have not failed outright, but which, for a variety of reasons, do not fully meet their requirements.

A system that is poorly designed or functionally limited has many consequences for its users. If we take interface design as an example, an ill-considered screen layout (for example, one that compels users to switch back and forth between two screens as they enter data) can have effects that range from mild irritation through to increased error rates, absence due to sickness, and greater staff turnover. Each has an associated cost—but it may be hard to measure accurately, and in some cases may not be recognized at all. If an employee leaves to take another job one reason may be his or her frustration with a poorly designed computer system.

System reliability can be important in determining overall costs. Recall the motor insurance company mentioned above in Fig. 2.1, where customers were asked to resubmit personal details, as a computer system crash had caused most customer data to be lost. This is not a reassuring picture for any insurance company to give its customers, and it is a safe conclusion that many will have switched to another insurer as a result. Some will also have told friends and colleagues, who in turn will be less likely to use this company in the future. It is very unlikely that anyone could accurately determine the full cost to this business in terms of lost customers. In the case of Boo.com, a technically inappropriate implementation resulted in the complete failure of the business.

Some of the more routine effects that can occur are summarized in Fig. 2.4. The list of effects given is by no means exhaustive, but in showing how poor design and operational characteristics of an information system can affect the organization as a whole, it reinforces the importance of getting things right before the problems have a chance to appear. This is true even if we ignore the ethical consequences of some information systems applications under development or already in use today. The full social cost of some ethical issues outlined in Section 2.4 is probably incalculable.

Design aspect	Example	Immediate effects	Other consequences
User interface	Illogical screen layout	Wasted time	Loss of confidence in system
	Difficult to read screens	Increased frustration	Increased sickness
	Unhelpful help messages	Increased error-rate	Increased absenteeism Greater staff turnover
Program execution	System response is slow	As above	Increased operating costs
	Lost data	Extra work re-entering data	Reduced income
Data storage	Inaccurate outputs	Extra work checking outputs	Loss of customer confidence Lost sales

Figure 2.4 Some hidden costs of poor design.

2.6 Summary

In this chapter we have looked at the issue of project failure from many different perspectives, including those of the people who use information systems, the people who purchase them and the people who build them. We have also outlined some of the deeper causes of project and system failure, and considered the costs and wider ethical issues.

The failures in information systems development can teach valuable lessons and, moreover, ignoring a difficulty does not make it go away. Rather, it increases the likelihood of repeating past mistakes. So it is important to understand as much as possible about what can go wrong with an information system development project, the better to be able to avoid it. But after so much concentrated attention on the many dead ends that our unluckier (or more careless) predecessors have encountered, it seems appropriate to end the chapter on a final upbeat note. Two of the failures discussed earlier have since become success stories.

The London Stock Exchange has successfully installed an online share trading system. The new system is much simpler than Taurus—and not all users are entirely happy with its functionality—but it does meet its basic requirements. It was introduced in April 1997, on time and within budget (Philips, 1997).

The London Ambulance Service also has a new ambulance dispatch system, introduced five years after its disastrous predecessor was scrapped in 1992.

Now the service's emergency call room plays host to visitors from around the world who come to see the successful implementation of an emergency response system in one of the biggest emergency call centres in the world. Indeed, the new system won the prestigious BCS Elite Group Information Systems Management Award in May [1997].

(Barker, 1998)

We can also remind ourselves of the many positive contributions that modern information systems make to our lives, enabling so many things that were not possible for earlier generations. It is difficult to imagine an aspect of life in modern

society that is not facilitated in some way by a computerized information system. Our goal, then, should be to ensure that tomorrow's systems build on the successes of the past, while avoiding the problems and failures, as far as it is in our power so to do.

Review Questions

- 2.1 Why do users, clients and developers disagree on the nature and causes of the problems in information systems development?
- 2.2 What are the main underlying causes of problems in information systems development?
- 2.3 Define quality.
- 2.4 What are the main differences between quality problems and productivity problems?
- 2.5 Why do the requirements drift once a project is under way?
- 2.6 What can be the results of ignoring the organizational context of an information system?
- 2.7 Define stakeholder.
- 2.8 What ethical issues might be involved in setting up an online shopping system that has links to an organization's management information systems?

Case Study Work, Exercises and Projects

2.A Do some research in computing trade journals, and find examples of recent projects that failed or ran into difficulties. Draw up a table with four separate columns. Give these titles like: 'Nature of problem', 'Who sees it as a problem', 'Probable cause' and 'Flynn category'. Enter the projects into your table, using your own intuition to complete column 3. Then complete the 4th column using Flynn's categories (summarized in Fig. 2.2). How do your causes compare with Flynn's categories?

2.B The British Computer Society (BCS) publishes a Code of Conduct for its members, who include thousands of computing and information systems professionals in the UK. In the USA, similar codes are published by the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronic Engineers (IEEE). In many other countries around the world, including India, Zimbabwe and Singapore, there is a national professional society with an equivalent code of professional ethics. Write down some ethical problems associated with the development of an information system to support staff who dispatch ambulances to medical emergencies, and use this to identify a list of issues you would expect to be covered in a professional code of conduct for information systems developers. Then obtain a copy of the BCS Code (BCS, 2001—or, if you are a reader in another country, your nearest local equivalent). Compare it to your list of issues. What are the main differences?

- 2.C Write down all the stakeholders who you think are associated with an emergency ambulance dispatch system. How are they affected?

2.D Review the ethical issues you identified in Exercise 2.B, and identify one or more issues that appear as a problem from the perspective of one stakeholder, but do not appear as a problem from the perspective of another stakeholder.

2.E Find out what legislation applies to information systems development activity in your country, and what implications it has for developers. Would the South African bank customer profiling system described in Section 2.4.2 be fully legal under these laws? Which particular aspects of the system make it ethically questionable, and from whose perspective?

Further Reading

Trade magazines such as *Computing* and *Computer Weekly* regularly publish articles that report on problems in current projects as they occur.

Sauer (1993) discusses reasons for project failures in terms of the project's organizational environment, and includes several practical case studies that illustrate the problems in practice.

Galin (2003) is one of many books that deal with software quality assurance (SQA), a discipline that endeavours to ensure that software systems fully meet their defined requirements. The latest edition of Pressman's standard software engineering text (2004) also has a good chapter on SQA.

De Montfort University's Centre for Computing and Social Responsibility runs a website that focuses on ethical issues in information systems development. This has many links to other sites, and also carries the full text of a number of academic papers in this field. It can be found at www.ccsr.cse.dmu.ac.uk.

Avoiding the Problems

Learning Objectives

In this chapter you will learn

- the stages in the waterfall lifecycle
- about prototyping and incremental lifecycles
- the importance of project management
- how users may be involved in a project
- the role of CASE tools in systems development.

3.1 Introduction

Problems that commonly affect the development of computerized information systems have been described in Chapter 2. These problems can significantly reduce the success of a software development project, and it is clearly important to adopt strategies and procedures that will minimize their occurrence. In this chapter we consider the steps that can be taken to make a computerized information system development project as successful as possible.

The problems identified in Chapter 2 fall into two major categories: those that are concerned with the management of the project and those that relate to poor quality in the delivered system. These are the two major areas on which we should focus if we wish to produce information systems within budget, on time and providing the required functionality. We discuss project lifecycles in Section 3.2 and consider some of the issues regarding the management of information systems development in Section 3.3.

There is much debate about what approach to information systems development is most likely to produce a quality system, a system that is fit for purpose. One major source of difficulty is the inherent complexity of software development. In an object-oriented approach complexity is managed in such a way as to help address this problem.

From a more general perspective, building computerized information systems can be viewed as a form of problem solving. Figure 3.1 shows a general problem-solving model adapted from Hicks (1991). The phases *Data gathering* and *Problem redefinition* are concerned with understanding what the problem is about; the *Finding ideas* phase attempts to identify ideas that help us to understand more about the nature of the problem and possible solutions. *Finding solutions* is concerned with providing a solution to the problem and *Implementation* puts the solution into practice. This approach to problem solving divides a task into sub-tasks, each with a particular focus and a particular objective.

The information systems development process may be subdivided simply into three main tasks, identifying what is required, planning how to deliver what is required and delivering what is required. There are many other ways of subdividing an information systems project but they all include an analysis activity that identifies what the system should do, a design activity that determines how best to do it and some construction that builds the system according to the design. The phases that contain these activities are given various names but the core activities remain the same.

An important consequence of subdividing the development process is that techniques and skills specific to the different phases can be identified. Teams of developers with these specialized skills can be allocated to the particular phases or activities, maximizing the chance that the activities are completed as well as possible. The importance of user involvement is considered in Section 3.4. Subdividing the development process results in smaller tasks that can be managed more easily to achieve the appropriate quality standards and to stay within the allocated resource budget. The importance of methodological approaches to systems development is discussed in Section 3.5 and the emergence of lightweight methodologies in Section 3.6.

A further source of help to overcome the problems the software developer faces is computer support. Software development is an activity that, like many others, may benefit from computerized information systems. Computerized support for software development did not become realistic in terms of cost and technology until the mid-1980s. It is now accepted that computerized support is an important

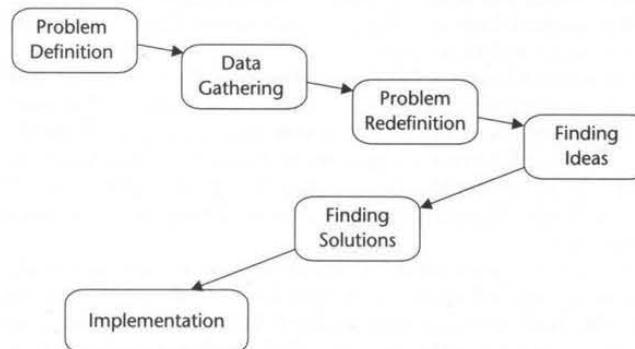


Figure 3.1 General problem-solving model (adapted from Hicks, 1991).

component of a development project (Allen, 1991). This is covered in more detail in Section 3.7.

We have already alluded to the benefits to be gained from managing the software development process effectively and have explicitly identified poor project management as a source of many of the problems. Building a software system is very different from building almost any other human artefact. Software is intangible, it cannot be weighed, its strength cannot be measured, its durability cannot be assessed, and its resistance to physical stress cannot be gauged. Of course we try to find (and with some success) measures of a software system that enable us to make judgements about its size, its complexity, the resource required to build it, and so on. But these measures are much less well understood than their counterparts in the design and construction of tangible artefacts such as buildings.

3.2 Project Lifecycles

Subdividing the process of software development produces what is known as a lifecycle model. Just as an animal goes through a series of developmental stages from its conception to its demise so, it is argued, does a computerized information system. Various project lifecycles can be applied to computerized information systems development. We will discuss some of the most commonly used lifecycle models.

Arguably two activities precede an information systems development project. They are *strategic information systems planning* and *business modelling*. The successful completion of these activities should ensure that the information system that is developed is appropriate to the organization. It can be argued that these are part of the information systems development lifecycle. However, their focus is not on computerization, per se, but rather the identification of organizational requirements. Whether they are viewed as part of the lifecycle or not, their importance is almost universally accepted for commercially oriented computer systems development. If a system is not commercially oriented (e.g. an embedded control system) strategic information systems planning and business modelling are not precursors to systems development. There is a distinction to be made between systems development, where a system may incorporate human, software and hardware elements, and software development that focuses primarily on software construction, although it involves the human users and the hardware upon which it executes. It is perhaps a matter of perspective. Thus a software development project is, by definition, focused solely on producing a software system that will satisfy the user requirements, whereas strictly speaking a systems development project has a wider scope and may not even include software as part of the solution.

Strategic information systems planning. As we saw in Chapter 1, information systems work within the context of an organization and must satisfy its current requirements as well as provide a basis from which future needs can be addressed. In order to do this, strategic plans are developed for the organization as a whole and within their context a strategic view of information systems needs can be formed. For example, in the Agate case study a strategic decision may be made to target multinational companies for international advertising campaigns.

This has consequences for campaign management and its supporting information systems.

Business modelling. In order to determine how an information system can support a particular business activity it is important to understand how the activity is performed and how it contributes to the objectives of the organization. Campaign management is an important business function for Agate and it should be modelled in order to determine how it is carried out, thus providing some of the parameters for subsequent information systems development.

3.2.1 Traditional lifecycle

A diagrammatic view of the traditional lifecycle (TLC) for information systems development is shown in Fig. 3.2. This model is also known as the waterfall lifecycle model (the difficulty of returning to an earlier phase once it is completed has been compared to the difficulty of swimming up a waterfall). It was developed in the late 1960s in an attempt to introduce a more systematic engineering approach (Royce, 1970) to software development. The individual phases of a TLC are described in the following sections.

System engineering. An information system involves human, software and hardware elements. The first stage of an information systems project is to identify the major requirements for the whole system and then to identify those parts of the system that are best implemented in software, those parts that are best implemented in hardware and those components that should be allocated to human

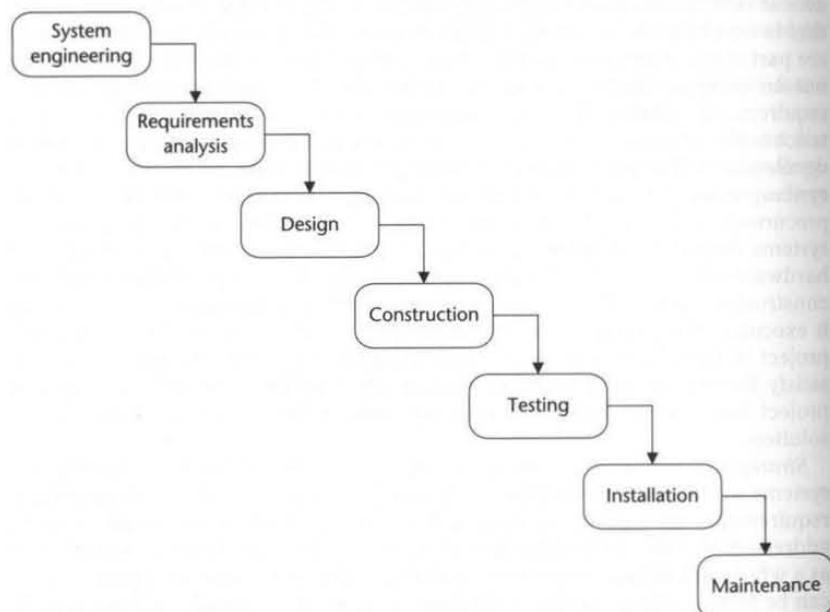


Figure 3.2 Traditional waterfall lifecycle model.

participants. This phase produces a high-level architectural specification that defines how these major parts of the system will interact with each other.

Requirements analysis. All the system's requirements need to be defined clearly and requirements gathering becomes more intensive. If the project is concerned largely with the development of elements that will be implemented in software, then these are the main focus of requirements analysis. The objective is then to identify what the users would require from the software elements of the system. Requirements gathering for large or complex systems benefits from fact-finding techniques that help to identify the requirements. Further techniques are also important to document these requirements appropriately and to aid their analysis¹. User requirements can also be used to develop the criteria for user acceptance tests later in the lifecycle.

Design. Once it is known what is required from the system, the design process determines how best to construct a system that delivers these requirements. Design is first concerned with the specification of a software architecture that defines the major software components and their relationships. Design involves reaching a balance between requirements that conflict with each other (e.g. maintainability and performance) within implementation environment constraints. Human activities may also be designed at this stage.

Construction. The completed design is now translated into program code. Depending on how the design phase has been completed, part of the construction may be automated. Construction may utilize different programming languages and database management systems for different parts of the system.

Testing. The system is tested to ensure that it satisfies the user requirements accurately and completely. Typically, several levels of testing are performed. Individual components are tested independently, then they are tested together as a subsystem and then the subsystems are tested together as a whole system. Often users perform some form of acceptance testing before the system is finally accepted as complete.

Installation. Once the system has been tested satisfactorily it is delivered to the customer and installed for use. The introduction of the system has to be managed carefully so as not to cause unnecessary disruption and to minimize the attendant risk of change. For example, one approach is to run both the old and new systems in parallel to ensure that the new system operates effectively before discontinuing the operation of the old system. This is an expensive strategy and it may be impractical to adopt such a fail-safe approach. Nonetheless, a contingency plan appropriate to the level of risk should be in place. These aspects of project management are discussed further in Chapter 22 available from the book website.

Maintenance. It is likely that the system will be subject to change during its operating life. The delivered system may operate erroneously and corrections may have to be made to the software (corrective maintenance). Certain aspects of the system's behaviour may not have been fully implemented (because of cost or time constraints, for instance), but are then completed during the maintenance phase (perfective maintenance). The operating environment may also change in various ways causing requirements changes that have to be accommodated (adaptive maintenance).

There are many variations of the waterfall model (e.g. Pressman, 2004; Sommerville, 1992), differing chiefly in the number and names of phases, and the

¹ We describe the techniques provided by UML for these purposes in Chapters 5–8.

Phase	Output deliverables
System engineering	High-level architectural specification
Requirements analysis	Requirements specification Functional specification Acceptance test specification
Design	Software architecture specification System test specification Design specification Subsystem test specification Unit test specification
Construction	Program code
Testing	Unit test report Subsystem test report System test report Acceptance test report Completed system
Installation	Installed system
Maintenance	Change requests Change request report

Figure 3.3 Lifecycle deliverables (adapted from Sommerville, 1992).

activities allocated to them. A major justification for lifecycle models like this is that their phases have explicitly defined products or *deliverables*. Sommerville (1992) suggests a series of deliverables produced by different phases of development, shown in Fig. 3.3.

These products can be used to monitor productivity and the quality of the activity performed. Several phases have more than one deliverable. If we need to show a finer level of detail to assist in the monitoring and control of the project, phases can be split so that each subphase has only one deliverable. Alternatively, a phase may be viewed as comprising a series of activities, each of which has a single deliverable and can be managed individually. Different types of project and different styles of organization may suit different styles of project lifecycle. When an organization embarks upon a systems development project it should specify the stages in the systems development process, their associated deliverables and the type of lifecycle to be used in a way that is applicable to its organizational context and the nature of the systems being developed.

The traditional lifecycle has been used for many years but is the subject of several criticisms.

- Real projects rarely follow such a simple sequential lifecycle. Project phases overlap and activities may have to be repeated.
- Iterations are almost inevitable, because inadequacies in the requirements analysis may become evident during design, construction or testing.
- A great deal of time may elapse between the initial systems engineering and the final installation. Requirements will almost inevitably have changed in the meantime and users find little use in a system that satisfies yesterday's requirements but may hamper current operations.

■ The TLC tends to be unresponsive to changes in client requirements or technology during the project. Once architectural decisions have been made (e.g. during systems engineering), they are difficult to change. A technological innovation that may make it feasible to automate different parts of the whole system may become available after the project has been running for some time. It may not be possible to incorporate the new technology without redoing much of the analysis and design work already completed.

The diagram in Fig. 3.4 shows possible paths for feedback loops within the TLC but the resulting iterations can be very costly. This would not be considered an iterative lifecycle. For example, if a major deficiency in the requirements analysis is discovered during construction, iterating back to requirements analysis may result in significant redesign making some of the construction so far completed inappropriate. If problems such as this do occur, project constraints may result in the missed requirements being addressed using ad hoc coding solutions instead of applying a suitable redesign. Ad hoc coding solutions frequently result in unmaintainable systems. However, despite its drawbacks, the TLC does provide a very structured approach to systems development that has the following advantages.

- The tasks in a particular stage may be assigned to specialized teams. For example, some teams may specialize in analysis, others in design and yet others in testing.

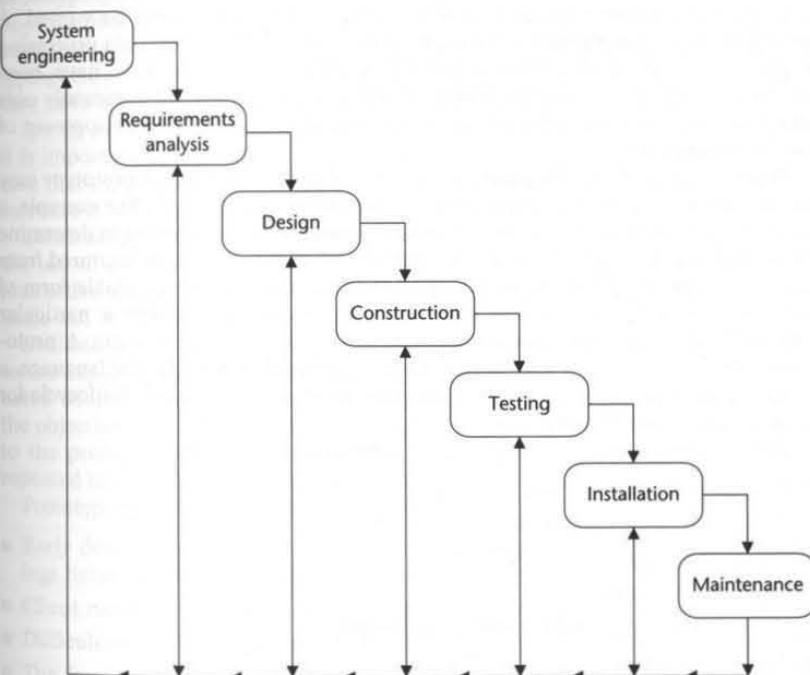


Figure 3.4 Waterfall lifecycle with feedback loops.

- The progress of the project can be evaluated at the end of each phase and an assessment made as to whether the project should proceed.
- The controlled approach can be effective for managing the risks on large projects with potentially high levels of risk.

3.2.2 Prototyping

Many approaches to systems development incorporate some iteration. For instance, analysis may involve a series of tasks that are repeated iteratively until the analysis models are deemed complete. The waterfall model discussed so far aims to deliver a final working system as the finished product. However, such approaches have a significant difficulty, in that the user only actually experiences how the system operates once it is delivered. Sometimes users find it difficult to imagine how their requirements will be translated into a working system and it is certainly the case that different implementations can be produced from the same set of requirements. The prototyping approach overcomes many of the potential misunderstandings and ambiguities that may exist in the requirements.

In software development a prototype is a system or a partially complete system that is built quickly to explore some aspect of the system requirements and that is not intended as the final working system. A prototype system is differentiated from the final production system by some initial incompleteness and perhaps by a less resilient construction. If the prototype is to be discarded once it has fulfilled its objectives, the effort required to build a resilient prototype would be wasted. A prototype will typically lack full functionality. It may have limited data processing capacity, it may exhibit poor performance characteristics or may have been developed with limited quality assurance. Prototype development commonly uses rapid development tools, though such tools are also used for the development of production systems.

Prototypes may be constructed with various objectives in mind. A prototype may be used to investigate user requirements as described in Chapter 5. For example, a prototype may be focused on the human-computer interface in order to determine what data should be presented to the user and what data should be captured from the user. A prototype might also be used to investigate the most suitable form of interface. A prototype may be constructed to determine whether a particular implementation platform can support certain processing requirements. A prototype might be concerned with determining the efficacy of a particular language, a database management system or a communications infrastructure². A lifecycle for prototyping is shown in Fig. 3.5.

The main stages required to prepare a prototype are as follows.

- Perform an initial analysis.
- Define prototype objectives.
- Specify prototype.
- Construct prototype.
- Evaluate prototype and recommend changes.

These are described in more detail below.

² For example, an Object Request Broker (these are described in Chapter 18).

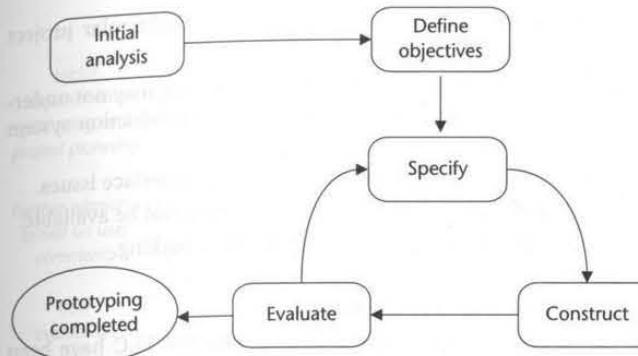


Figure 3.5 A prototyping lifecycle.

Perform an initial analysis. All software development activity utilizes valuable resources. Embarking upon a prototyping exercise without some initial analysis is likely to result in an ill-focused and unstructured activity producing poorly designed software.

Define prototype objectives. Prototyping should have clearly stated objectives. A prototyping exercise may involve many iterations, each iteration resulting in some improvement to the prototype. This may make it difficult for the participants in a prototyping exercise to determine if there is sufficient value to continue the prototyping. However, with clearly defined objectives it should be possible to decide if they have been achieved.

Specify prototype. Although the prototype is not intended for extended operation it is important that it embodies the requisite behaviour. It is almost certainly the case that the prototype will be subject to modification and this will be easier if the software is built according to sound design principles.

Construct prototype. Since it is important that prototype development is rapid, the use of a rapid development environment is appropriate. For example, if an interactive system is being prototyped, environments such as Delphi or Visual Basic can be most effective.

Evaluate prototype and recommend changes. The purpose of the prototype is to test or explore some aspect of the proposed system. The prototype should be evaluated with respect to the objectives identified at the beginning of the exercise. If the objectives have not been met, then the evaluation should specify modifications to the prototype so that it may achieve its objectives. The last three stages are repeated until the objectives of the prototyping exercise are achieved.

Prototyping has the following advantages:

- Early demonstrations of system functionality help identify any misunderstandings between developer and client.
- Client requirements that have been missed are identified.
- Difficulties in the interface can be identified.
- The feasibility and usefulness of the system can be tested, even though, by its very nature, the prototype is incomplete.

Prototyping also has several problems and their impact on a particular project should be estimated before engaging in prototyping:

- The client may perceive the prototype as part of the final system, may not understand the effort that will be required to produce a working production system and may expect delivery soon.
- The prototype may divert attention from functional to solely interface issues.
- Prototyping requires significant user involvement, which may not be available.
- Managing the prototyping lifecycle requires careful decision making.

3.2.3 Iterative and incremental development

Several suggestions on how to overcome the problems with the TLC have been made. A common thread in many current approaches to software development is to adopt an iterative lifecycle. In such an approach the development project comprises many iterations, each of which can be viewed as a mini-project in its own right. An iteration may result in an iteration release comprising some elements of functionality. An iteration may provide new elements of functionality in addition to that from previous iteration releases or it may improve some aspects of operation of previously delivered functionality.

Gilb (1988) suggests that successful large systems start out as successful small systems that grow incrementally. An incremental approach performs some initial analysis to scope the problem and identify major requirements. Those requirements that will deliver most benefit to the client are selected to be the focus of a first increment of development and delivery. The installation of each increment provides feedback to the development team and informs the development of subsequent increments.

Boehm's (1988) spiral model can be viewed as supporting incremental delivery. However, Gilb (1988) argues that the spiral model does not fully support his view of incremental development, as there are aspects of systems development that it does not emphasize or include, namely:

- the production of high-value to low-cost increments
- the delivery of usable increments of 1% to 5% of total project budget
- a limit to the duration of each cycle (e.g. one month)
- a measure of productivity in terms of delivered functionality or quality improvements
- an open-ended architecture that is a basis for further evolutionary development.

Figure 3.6 shows how Boehm's spiral model can be adapted to suit incremental delivery. Note that prototyping may be used either during the risk analysis or during the software development part of the development cycle.

Iterative development of course does in fact result in incremental delivery, though some approaches produce the increments only for internal release to the development team and not for external release to the client. Many current approaches are categorized as iterative and incremental to reflect the iterative nature of the development process and the incremental nature of the system delivery.

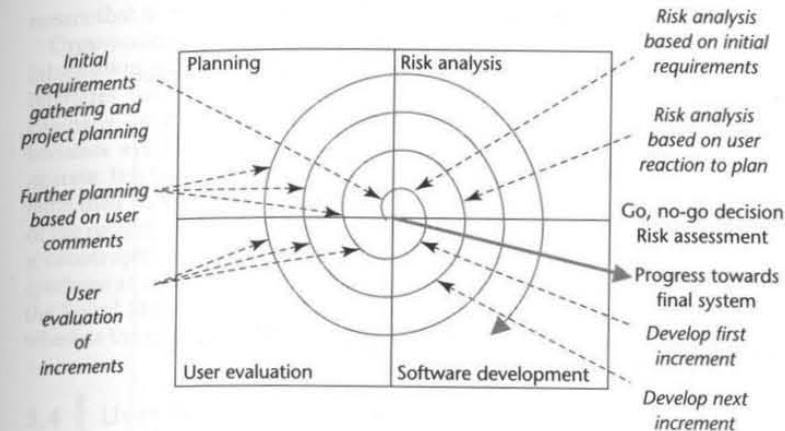


Figure 3.6 Spiral model for incremental delivery (adapted from Boehm, 1988).

3.2.4 The Unified Software Development Process

The Unified Software Development Process (USDP) (Jacobson et al., 1999) reflects the current emphasis on iterative and incremental lifecycles. It builds upon previous approaches by Jacobson et al. (1992), Booch (1994) and Rumbaugh et al. (1991). The USDP incorporates the UML and comprises much good advice on software development. The USDP will be discussed in more detail later (Chapter 21). However, an overview of its lifecycle is presented here.

A development cycle for the USDP is illustrated in Fig. 3.7 and comprises four phases.

- *Inception* is concerned with determining the scope and purpose of the project.
- *Elaboration* focuses on requirements capture and determining the structure of the system.
- *Construction*'s main aim is to build the software system.
- *Transition* deals with product installation and rollout.

A particular development cycle may be made up of many iterations. In Fig. 3.7 there are two iterations in the inception phase and four in the construction phase. This is purely illustrative and the number of iterations in each phase is determined on a project by project basis. At the end of each iteration an increment is delivered and its composition may range from elements of a requirements model to working program code for a portion of the system. In the USDP an increment is not necessarily additive; it may be a reworked version of a previous increment.

The diagram also illustrates that a phase may involve a series of different activities or workflows. This is different from the waterfall lifecycle, in which each phase largely comprises a single activity. The inception phase may include elements of all of the workflows, though it is likely that design, implementation (i.e. constructing the software) and test would be focused on any necessary exploratory prototyping. However, most commonly, inception would involve primarily the requirements and analysis workflows.

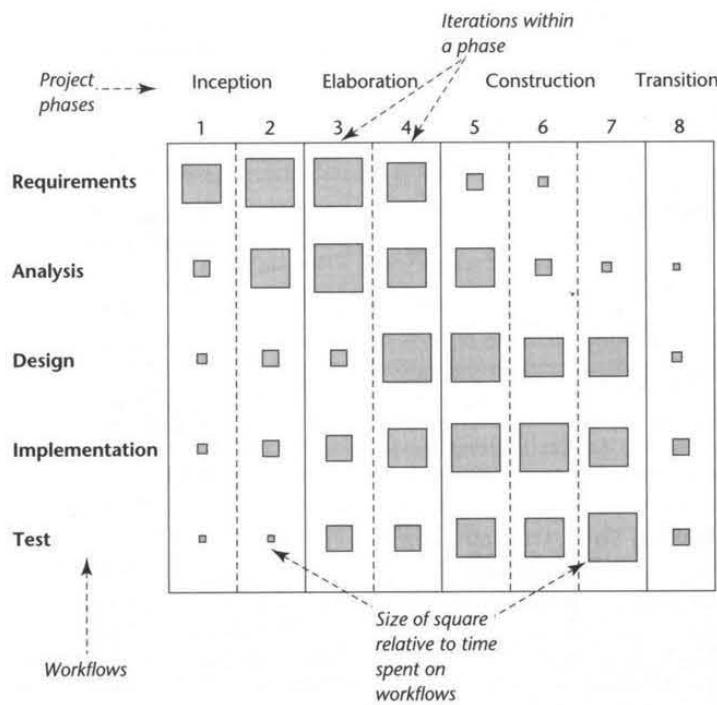


Figure 3.7 The Unified Software Development Process (adapted from Jacobson et al., 1999).

3.3 Managing Information Systems Development

An information system development project has many attendant risks. Late delivery may result in lost income and missed opportunities for the client, and punitive legal action against the software developers. The continuing ability of an enterprise to compete effectively may be dependent upon a new information system. Failure to deliver a new system may even cause a company to fail. There are also risks associated with adopting a new information system. For example, suppose an organization chooses to automate the payment of all invoices. An automatic invoice payment system may be designed to balance the commercial and moral duties to pay suppliers promptly with a requirement to maximize the interest that is earned on the organization's assets. Should the system fail to achieve this balance in practice, the company may suffer serious consequences. On the one hand, late payment of an invoice may lead to damaging legal action. Of course, one way to avoid this particular risk is not to adopt the new information system, but the alternatives will also have attendant risks and costs. For instance, a manual system for processing invoices may have higher costs and greater risk of error. On the other hand, if invoices are paid too quickly, the company's interest earnings will be reduced. Use of any new information system must be controlled and monitored to

ensure that it continues to operate within the parameters defined by the company.

Organizations are increasingly dependent on the operation of computerized information systems to manage the business, to the point where a new system that interferes with the successful operation of existing systems may prevent the company from being able to operate at all. As a project proceeds, more detail becomes available as to what the proposed system should do and how it will operate. It is important at each stage to identify risks to the organization and determine their significance so that judgements can be made on the continuing viability of the project. If a new system is thought likely to place the organization at risk of a catastrophic failure, then either the system should be redesigned to avoid this (perhaps at significant extra cost) or the project should be cancelled. Note that in the Spiral Model (Fig. 3.6) each iteration includes a risk analysis that determines whether the next cycle should proceed.

3.4 User Involvement

A key factor in maximizing the chance of success is ensuring that there is continued and effective user involvement throughout the project. The traditional waterfall lifecycle is less amenable to user involvement throughout the whole project and hence is less likely to stay in line with the changing user requirements. A prototyping approach is normally dependent upon continuing user involvement and by its very nature encourages user involvement. However, care has to be taken to ensure that the users have sufficient time to perform their roles effectively. In prototyping, the evaluation of the prototype requires significant time from the users.

Users can be involved in projects at various levels and can play many different roles. Some approaches to software development (e.g. DSDM³) directly involve users in the development team, so that they then have a significant opportunity to influence the way the project proceeds, perhaps by identifying difficulties and suggesting more acceptable alternatives. It is important that users who have such a significant influence on the direction of the project should understand the organizational requirements and the needs of fellow users. Direct involvement of users is more likely to be successful if they are considered as full members of the project team and if they are genuinely empowered to represent the organization and make decisions within clearly defined operating parameters. There is always a danger that users who become members of a project team cease over time to represent the user perspective effectively, as they begin to view themselves more as a team member and less as a user representative. One way of overcoming this tendency is to rotate membership of the development team among a group of users. However, this can result in a loss of continuity. A more satisfactory approach is to scope each activity so that a user team member can see it through to completion in a reasonably short time, say within three months.

At the other end of the spectrum are users whose only involvement in the project is as participants in fact gathering. Such a user may provide information about current working practice but they have little or no influence on the design of the

³ DSDM is introduced in Chapter 21.

new system. In this situation users are likely to be concerned about the project and may fear the effect it will have on their jobs. As a result, they may be less cooperative and will be less willing to take ownership of the new system when it is installed.

Even when users are not invited to join the project team, effective participation can still be encouraged through a consultative approach. Procedures are set up so that users are able to review certain aspects of the systems development, provide feedback, give their views of the system and then be able to see the response of the project team to their feedback. In some circumstances a large part of the task of requirements capture may be delegated to users. They are likely to feel a strong affinity to the delivered system, provided that it satisfies the requirements they specified.

Whatever form of involvement users have with the project, it is important that their role is carefully explained and that training is given as required. Some large organizations have gone to the trouble of training users so that they understand the terminology and models used by the systems developers. Furthermore, users must be given the time to participate. It is no good expecting users to review requirements documents effectively if the only time they have available is during their lunch break.

Users who participate in a systems development project can be selected in various ways. They can be designated by management as being the most appropriate representatives or they can be selected by their peers. In either circumstance they must be genuine representatives of the user perspective.

3.5 Methodological Approaches

One of the major influences on the quality of the systems developed is the software development approach adopted. If the approach used is not appropriate for a particular type of application then it may limit the quality of the system being produced. In part at least, the object-oriented approach provides a mechanism for mapping from real-world problems to abstractions from which software can be developed effectively. Furthermore, object-orientation provides conceptual structures that help to deal with modelling complex information systems. As information systems requirements are becoming increasingly complex the use of an object-oriented approach is more necessary. It is a sensible strategy to transform the development of a large, complex system into the development of a set of less complicated subsystems. Object-orientation offers conceptual structures that support this subdivision. Object-orientation also aims to provide a mechanism to support the reuse of program code, design and analysis models.

A methodology consists of an approach to software development (e.g. object-orientation), a set of techniques and notations (e.g. the Unified Modelling Language—UML) that support the approach, a lifecycle model (e.g. spiral incremental) to structure the development process and a unifying set of procedures and philosophy. In this text we do not espouse a particular methodology but apply object-oriented techniques in a co-ordinated fashion using UML. Methodologies are discussed in more detail in Chapter 21.

3.6 Agile Approaches

Problems with some of the methodologies in the 1980s and 1990s that incorporated the Traditional Lifecycle included unresponsiveness to change and a highly bureaucratic approach to analysis and design that was also heavy on documentation. In order to overcome some of these problems, iterative lightweight approaches emerged that were typically used for small to medium business information systems where there was appreciable requirements change during the life of the project. XP (Extreme Programming) is an early example of such an approach (Beck, 2004). XP does not emphasize early analysis and design documentation and is an iterative and incremental approach. It is an example of a group of lightweight methodologies that are now termed *Agile*. One of the key features of Agile approaches is the acceptance that user requirements will change during development and that this must be accommodated by the development process. DSDM (Dynamic System Development Method) is viewed by some as another example of an Agile approach (Fowler, 2003). Both XP and DSDM are further described in Chapter 21. The IBM-Rational Unified Process (and the Unified Software Development Process) can also be used in an Agile manner (Pollice, 2001).

In many ways Agile approaches are the antithesis of methodologies based on the Traditional Lifecycle, which assume a belief that requirements are, relatively speaking, fixed. The degree to which requirements are subject to change during software development varies from project to project and it is reasonable that the approach that is being used should be capable of coping with this degree of change. Nonetheless, the importance of effective requirements capture, documentation, analysis and design should not be underestimated irrespective of the size of project. Boehm (2002) draws a distinction between plan-based and Agile methods and suggests that both have their place. (As one might expect, the emphasis on planning early in the project varies from one Agile methodology to another.) The need for systems that dependably deliver agreed requirements suggests that planning and effective analysis and design with appropriate documentation have an important role in software development.

A group of software developers and methodology authors met in February 2001 and produced a manifesto for Agile software development. This manifesto is shown in its entirety in Fig. 3.8. The preference for customer collaboration over contract

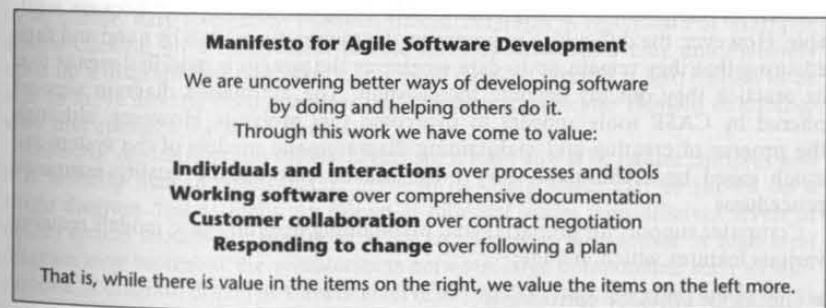


Figure 3.8 The Manifesto for Agile Software Development

negotiation in the manifesto suggests the need to adopt new forms of customer relationship that acknowledge requirements change during software development. This in turn emphasizes the key importance of strong customer or user involvement.

3.7 CASE

Computer Aided Software Engineering (CASE) tools have been widely available since the mid-1980s and now provide support for many of the tasks the software developer must undertake. The broadest definition of CASE includes the use of software tools for any technical, administrative or managerial aspects of software development.

Typically CASE (or modelling) tool vendors offer a series of products that are integrated to offer extensive lifecycle coverage. Such CASE tools are known as I-CASE (the I stands for Integrated). CASE tools tend to be designed for use with a particular approach to software development, particular notations and implementation environments. In the 1980s and early 1990s CASE tools were being developed to support an apparently ever-increasing range of different development approaches. The growth in popularity of object-oriented approaches exacerbated the situation with the emergence of many new methodologies with different notations. Although it is acknowledged that no single approach to software development is appropriate for all projects, it has become accepted that a common notation set can be used for requirements and design modelling. The move to a common notation has resulted in the development of the Unified Modelling Language, which, it must be emphasized, is not a methodology. Modern CASE tool-sets provide an increasingly wide range of facilities and cover most lifecycle activities. These are discussed in turn below.

3.7.1 Model and technique support

Most approaches to information systems development utilize models, frequently diagrammatic, to represent the relationships between different elements of the requirements and to document the design structure before the system is constructed. Analysis and design diagrams are models of the system and of its requirements. Diagrammatic analysis and design techniques were used in information systems development for many years before computerized support became available. However, the difficulties in preparing diagrammatic models by hand and then ensuring that they remain up to date whenever the system is modified means that in practice they quickly become inconsistent. The automated diagram support offered by CASE tools appears to overcome this problem. However, although the process of creating and maintaining diagrammatic models of the system are much eased by CASE tools, they must still be enforced by quality assurance procedures.

Computer support for preparing and maintaining diagrammatic models requires various features which include:

- checks for syntactic correctness
- repository support

- checks for consistency and completeness
- navigation to linked diagrams or models
- layering
- traceability
- report generation
- system simulation
- performance analysis.

These features are described in turn below.

Syntactic correctness. The CASE tool checks that the correct symbols are being used on the diagrams and that they are being linked in permissible ways. This ensures that the correct vocabulary is being used but does not ensure that it is meaningful or relevant to client requirements.

Repository. Originally repositories were only designed to hold the definitions of the data attributes of a system. They were then termed data dictionaries because in the same way that a language dictionary provides clear definitions of the words in a language, the data dictionary contains definitions of all the elements in an information system. However, increasingly it became clear that all elements in a system development project are open to ambiguous interpretation unless clearly defined in a repository of information about the system. A repository may contain diagrams, descriptions of diagrams and specifications of all the elements in the system. Some CASE vendors use the term 'encyclopaedia' instead of repository.

Consistency and completeness. Most CASE tools support various diagrammatic models that capture different aspects of the system. As all relate to the same system, it is important that any one element that appears on several diagrams or models (perhaps viewed from different perspectives) should be consistent with itself. Most approaches to analysis and design stipulate that certain diagrams must be completed and that the elements referred to in those diagrams must all be documented in the repository. To manually check the consistency and completeness of a system of any significant size is a task that is very onerous, time-consuming and error-prone. A good CASE tool may check the consistency and completeness of a large model in seconds and provide the developer with a comprehensive report on any inconsistencies found or omissions identified.

Navigation to linked diagrams. A complex system is likely to require many diagrammatic models to describe its requirements and its design. For a CASE tool to be usable, easy navigation between linked diagrams is essential. For example, double-clicking on a component at one level of abstraction may automatically open up a diagram that describes it at a more detailed level. It is also helpful to be able to move directly from one view that contains a particular element to another view that contains the same element.

Layering. An information system of any significant size is by nature complex and it is unlikely that all relationships between its components can be shown on a single diagram. Just as maps are drawn at different scales with different levels of detail, system models are produced at various levels of abstraction. A high-level diagram may represent the relationships between large components such as subsystems. A diagram drawn at a lower level of abstraction may describe the elements within a particular component in detail. In order to cope with complexity, we divide the system into manageable chunks and link them in layers. A good CASE

tool provides a capability to layer the models of the system at different levels of abstraction. The consistency and completeness checking discussed earlier should also check that the representations of one element at different levels of abstraction are consistent.

Traceability. Most of the elements created during the development of an information system are derived from other elements, and the connections between them should be maintained. It must be possible to trace through from the repository entries that describe a particular requirement to the program code that provides the functionality that satisfies the requirement. If a requirement changes, the maintenance activity is easier if all the code that implements that requirement can be readily identified. It should be possible to trace all requirements from the analysis documentation, through the design documentation to the implemented code. This feature is known as requirements traceability.

Report generation. Complex systems involve modelling many elements. Comprehensive reporting capabilities improve the usability of a CASE tool by ensuring that the developer can easily obtain information about the models for a system in suitable formats. In fact, a CASE tool would be of little use if the information it held about a project were not readily available, no matter how effective it was in other respects.

System simulation. When a CASE tool has been populated with models of an application it should be possible to simulate some aspects of system behaviour. For example, how does the system respond to a particular event? Some CASE tools provide capabilities that enable a software developer to examine the consequences of a design decision without the need to actually build the software.

Performance analysis. The performance of a system is an important ingredient in its success. For example, a system that supports staff who deal directly with customer enquiries should be able to respond quickly to a query about the availability of a certain product. If customers are kept waiting for too long, this will probably result in lost sales. The analysis of performance is particularly difficult for an application that runs on multiple processors and uses a complex communications infrastructure. Some CASE tools provide the capability to perform a 'What if' analysis to examine the implications of alternative implementation architectures.

3.7.2 Software construction

CASE tools can offer a range of features to support software construction and maintenance. These include code generation and maintenance tools.

Code generators. The capability to generate code directly from a design model is a major benefit to the developer for several reasons. First, a working software system is likely to be produced more quickly. Second, one source of error is largely removed when the code is produced automatically and consistent with the design. Third, when requirements change, a consequent change to the design documentation can be followed by automatic code generation. If the application logic is defined completely and precisely in the design model, full code generation is possible. If a design model contains detailed operation specifications⁴, then it is likely that a code framework can be generated to which further code can be added. In order to reduce the level of detail required for the design model, code

generators may make certain assumptions concerning the implementation. Code generators are available for many different languages and development environments and are likely to include the capability to generate database schemata for the major proprietary database management systems.

Maintenance tools. Software maintenance is a major issue. All systems are subject to change as the enterprise changes, perhaps in response to legislative change. For example, the introduction of the euro in Europe has necessitated significant change to many software systems. Various tools are available to help with systems maintenance. For some programming languages, reverse engineering tools may also be available that can generate design documentation directly from program code (although if the program code is poorly structured the resulting design documentation may be of little use). Tools are also available that can analyse program code and identify those parts that are most likely to be subject to change.

3.7.3 Benefits and difficulties of using CASE

CASE tools can bring many benefits to the development activity. They help to standardize the notation and diagramming standards used within a project, and this aids communication among the team members. They can perform automatic checks on many aspects of the quality of the models produced by analysts and designers. The report generation capabilities of a CASE tool reduce the time and effort that needs to be spent by analysts and designers in retrieving data about the system upon which they are working. Where a CASE tool can carry out automatic code generation, this further reduces the time and effort that is required to produce a final system. Finally, the electronic storage of models is essential to the reuse of models, or components of them, on other projects that address similar analysis or design problems.

Like any other technology, CASE tools also have their disadvantages. These include limitations in the flexibility of the documentation that they can provide. However, some CASE tools include the capability to specify and tailor documentation templates to suit particular reporting requirements. The development approach may also be limited by the need to work in a particular way in order to fit in with the capabilities of the CASE tool. The ability of a CASE tool to check all models for their consistency, completeness and syntactic correctness can in itself give rise to a danger. Developers may make the erroneous assumption that, because their models are correct in those specific senses, they are therefore also necessarily relevant to user requirements. There are also certain costs attached to the installation of a CASE tool. Aside from the cost of the software and manuals, there is also likely to be a significant cost in additional training for developers who will be expected to use the CASE tools.

On balance CASE tools can provide useful and effective support for the software development activity, but it requires appropriate management for this to be achieved without any damaging side-effects.

3.8 Summary

We have considered how to avoid the problems that typically arise during information systems development. Several strategies have been discussed. Lifecycle

⁴ These define how the system will function.

models are used to provide structure and management leverage for the development process itself. User involvement is crucial to ensure relevance and fitness for purpose of the delivered system. Furthermore, many of the difficulties that may occur during installation are reduced if ownership of the proposed system has been fostered by effective participation during development. The modelling requirements of the activities involved in the software development are well supported by object-orientation. An evolving range of methodological approaches to information systems development is available and a key factor for success is matching the most appropriate approach to project and organizational requirements. Finally, we discussed the importance of CASE tool support for the software developer.

Review Questions

- 3.1** What are the advantages of the traditional waterfall lifecycle?
- 3.2** What are the disadvantages of the traditional waterfall lifecycle?
- 3.3** How are some of the disadvantages listed in your answer to Question 3.2 overcome?
- 3.4** What is prototyping?
- 3.5** How does prototyping differ from incremental development?
- 3.6** What are the different ways of involving users in the systems development activity? What are potential problems with each of these?
- 3.7** How do 'syntactic correctness', 'consistency' and 'completeness' differ from each other?
- 3.8** What does requirements traceability mean?
- 3.9** Why is it not enough for a diagram to be syntactically correct, consistent and complete?
- 3.10** What is the purpose of a repository?

Case Study Work, Exercises and Projects

- 3.A** Read the Case Study Chapter B1. What lifecycle model would you recommend for the development of the production control system for FoodCo? Justify your decision.
- 3.B** For a CASE tool with which you are familiar, explore and critically assess the consistency and completeness checking facilities available.
- 3.C** For a CASE tool with which you are familiar, explore and critically assess its system generation capabilities.
- 3.D** In your library find references for three lifecycle models not discussed in this chapter. Briefly review each of these lifecycle models.
- 3.E** Research at least one Agile Methodology and explore its capability for large-scale information systems development.

Further Reading

Hicks (1991) provides a comprehensive introduction to problem-solving skills that are valuable to the software developer. Sommerville (2004) and Pressman (2004) provide good discussions of lifecycle issues. Gilb (1988) contains much good advice concerning software development and is well worth reading. Texel and Williams (1997) suggest a detailed object-oriented lifecycle model that includes 17 stages. A lifecycle with such a large number of delineated stages is probably more suited to large projects. If used for small projects the management overhead is likely to be excessive.

Jacobson, Booch and Rumbaugh (1999) provide a description of the USDP and further information on the IBM-Rational variant of the USDP can be found at <http://www-306.ibm.com/software/rational/index.html>. Beck (2004) offers the seminal description of XP and Fowler (2003) provides a very readable review of Agile approaches at <http://www.martinfowler.com/articles/newMethodology.html>. Larman (2003) discusses Agile and iterative and development approaches. DeMarco and Boehm (2002) conduct an interesting dialogue about Agile approaches. More information regarding the Agile manifesto can be found at <http://agilemanifesto.org/>.

Many CASE tools are in widespread use and increasingly are providing support for the UML standard, though the styles of implementation do vary.

CHAPTER

4

What Is Object-Orientation?

Learning Objectives

In this chapter you will learn

- the fundamental concepts of object-orientation
- the justifications for an object-oriented approach
- how object-orientation is used in practice.

4.1 Introduction

It is a major premise of this book that the object-oriented approach to systems development helps to avoid many of the problems and pitfalls described in earlier chapters. In this chapter, we lay the foundations for understanding object-orientation by presenting an explanation of the fundamental concepts. This is important for all readers who are new to object-orientation, but particularly so for those who are already familiar with other approaches to systems analysis and design, for example structured analysis. There are important differences between the object-oriented approach and earlier approaches. It is necessary to have a sound grasp of the basic concepts in order to be able to apply the techniques of object-orientation in an effective way.

At this stage the most important concepts to grasp are object, class, instance, generalization and specialization, message passing and polymorphism. We explain each of these concepts in turn, showing both what the parts of an object-oriented system are and how they use message passing to isolate different parts of a system effectively from each other, thus controlling the complexity of the interfaces between subsystems (Section 4.2). The discussion is necessarily conceptual in places, but practical examples and analogies are used to illustrate the theoretical points wherever this is appropriate. We then place object-orientation in its historical context, and look at some of the reasons why it has been able to make a contribution to successful information systems development (Section 4.3). Finally,

we conclude by summarizing the state of evolution that object-oriented languages have reached today (Section 4.4).

4.2 Basic Concepts

The most important concept addressed in this section is the *object* itself, and it is to this that we first pay attention. The other concepts explained in this section are strongly dependent on each other, and all contribute to an adequate understanding of the way that objects interact, and thus to their significance for information systems. The following explanations concentrate on giving an understanding of the territory that later chapters will explore in greater depth.

4.2.1 Objects

In an early book on object-oriented analysis and design, Coad and Yourdon (1990) define *object* as follows:

Object. An *abstraction* of something in a problem domain, reflecting the capabilities of the system to keep information about it, interact with it, or both.

This may not immediately appear to help very much, as parts of the definition are themselves a little obscure and raise further questions. In particular, what exactly do we mean by ‘abstraction’, and which ‘system’ are we talking about?

A useful definition of abstraction in this context might be: ‘A form of representation that includes only what is important or interesting from a particular viewpoint’. To give a familiar example, a map is an abstract representation. No map shows every detail of the territory it covers (impossible, in any case, unless it were as large as the territory, and made from similar materials!). The intended purpose of the map guides the choice of which details to show, and which to suppress. Road maps concentrate on showing roads and places, and often omit landscape features unless they help with navigation. Geological maps show rocks and other subsurface strata, but usually ignore towns and roads. Different projections and scales are also used to emphasize parts of the territory or features that have greater significance. Each map is an abstraction, partly because of the relevant features it reveals (or emphasizes), and also because of the irrelevant features it hides (or de-emphasizes). Objects are abstractions in much the same way. An object represents only those features of a thing that are deemed relevant to the current purpose, and hides those features that are not relevant.

This brings us to our second question. To which system are Coad and Yourdon referring? The context of the quotation makes it reasonably clear that their answer would be something like this: ‘The proposed object-oriented software system, whose development is under consideration’. This seems appropriate—after all, this book is about the development of object-oriented software—but we should note that other systems are also involved. The most important of these is the human activity system that we must understand before beginning the construction of any software.

We should also note that we are using one tool to serve two purposes. As we shall see particularly in Chapter 7, objects are used to model an understanding of the application domain (essentially part of a human activity system), yet we shall

also see in later chapters that objects are also understood as parts of the resulting software system. These are distinct purposes and there will be some occasions when we may need to take care to be clear about which meaning is intended.

Rumbaugh et al. (1991) take a slightly different slant on defining an object. These authors also explicitly recognize the dual purpose noted in the last paragraph.

We define an *object* as a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. Objects serve two purposes: They promote understanding of the real world and provide a practical basis for computer implementation.

This definition contains a repetition (although this does no harm)—by ‘concept’ we actually mean an abstraction, something that is logical rather than physical. A concept may be intangible, even imaginary (in the sense that not every participant agrees on its meaning, or sometimes even its existence). In the Agate case study, one concept is the ‘campaign’. While they are clearly important, campaigns are intangible and difficult to define with precision. They really exist only as a relationship between a client (say Yellow Partridge, a jewellery company), the Account Manager, some other staff, some advertisements and various tasks and components that go into creating advertisements.

Concepts include many kinds of relationship between people, organizations and things. In information systems development, it is often necessary to recognize relationships such as contracts, sales or agreements. While intangible, some of these relationships can be quite long-lasting, and can have a complex influence on how people and other things in the application domain are able to act.

Let us take a simple transaction as an example. Imagine buying a tube of toothpaste in your local supermarket. On one level, this is just a sale, an exchange of money for goods. On a deeper level you may be entering into a complicated relationship with the shop and the manufacturer. This probably depends on other factors, e.g. the warranty may vary depending on the country that you are in at the time of the purchase, and perhaps the sale will earn points for you on a loyalty card. Perhaps the packaging includes a money-off coupon for your next purchase, or a contest entry form that must be accompanied by a valid proof of purchase. Now suppose you find something wrong with the toothpaste—you may be able to claim a refund or replacement. Maybe you can even sue the shop for damages. The point here is that we cannot understand the business without understanding these possible consequences of the sale in some appropriate way. In this case, the real-world ‘sale’ will almost certainly be modelled as an object in the system.

At a fairly abstract level, when choosing the objects we wish to model—in fact, at the level that corresponds to a mapmaker—we need to ask: ‘What sort of map is this, what details should it show, and what should it suppress?’ In the real world, probably the only common characteristic of all objects is that they exist. However, all objects in a model or in an information system have certain similarities to all other objects, summarized by Booch in the statement that an object ‘has state, behaviour and identity’ (Booch, 1994). Here, ‘state’ represents the particular condition that an object is in at a given moment, ‘behaviour’ stands for the things that the object can do (and that are relevant to the model) and ‘identity’ means simply that every object is unique.

For example, Fig. 4.1 lists some characteristics of a person, a shirt, a sale and a bottle of ketchup. Supposing that we wished to model these as objects, we can

Object	Identity	Behaviour	States
A person	‘Hussain Pervez’	Speak, walk, read	Studying, resting, qualified
A shirt	‘My favourite button-down white denim shirt’	Shrink, stain, rip	Pressed, dirty, worn
A sale	‘Sale no 0015, 15/02/05’	Earn loyalty points	Invoiced, cancelled
A bottle of ketchup	‘This bottle of ketchup’	Spill in transit	Unsold, opened, empty

Figure 4.1 Characteristics of some objects.

identify some possible identities, behaviours and states (but note that these are for illustration only, and do not indicate any particular system perspective).

In some texts (e.g. Wirsits-Brock et al., 1990), objects are deliberately characterized as if each were a person, with a role in the system that is based on its answers to three questions.

- Who am I?
- What can I do?
- What do I know?

This approach defines an object in terms of its responsibilities and its knowledge, themes that we will encounter again in later chapters.

4.2.2 Class and object

Let us begin by being clear that we are here dealing with objects as abstractions within an information system—either a model or the resulting software—and not with the real-world objects that these represent.

Class is a concept that describes a set of objects that are specified in the same way. According to the UML specification, all objects of a given class share a common specification for their features, their semantics and the constraints upon them (OMG, 2004c). This does not quite mean that all objects of a class are identical in every way, but it does mean that their specification is identical. Objects that are sufficiently similar to each other are said to belong to the same class, and, in turn, the class is an abstract descriptor for the specified logical similarities between those objects.

This corresponds to the idea of a class in object-oriented programming (which is where the concept originated). For example, in a Java program a class acts as a kind of template from which individual objects are constructed when they are needed. (This is not the whole story, as software classes can also do other things that need not concern us here.)

Instance is another word for a single object, but it also carries connotations of the class to which that object belongs: every object is an instance of some class. So, just like an object, an instance represents a single person, thing or concept in the application domain.

Figure 4.2 shows some classes that might be identified from the Agate case study (Chapter 7 describes a practical approach to identifying classes).

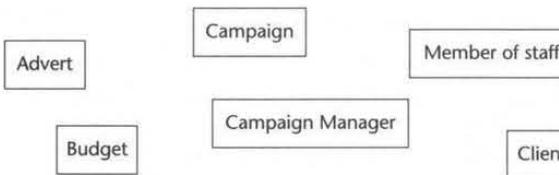


Figure 4.2 Some possible object classes in the Agate case study.

A class and its instances are related in the following manner. For staff at Agate, the idea of ‘a campaign’ is an abstraction that could represent any one of several specific campaigns. In an object-oriented software system, the class *Campaign* represents the relevant features that all campaigns have in common. There is one instance of the class to represent each real-world campaign. Some examples of campaigns are: a series of adverts for various Yellow Partridge jewellery products broadcast on various satellite and cable TV channels during Spring 2005; a national series of TV, cinema, magazine and Internet adverts for the launch of the Soong Motor Co’s Helion sports utility vehicle in August 2005.

Each instance of a class is unique, just as every living person is unique, however closely they resemble someone else. This is true even when two instances have identical characteristics. For example, there could be two members of staff at Agate with the same name—say ‘Ashok Patel’, as this is a fairly common name. It is even possible (though unlikely) that both joined the company on the same date and work in the same section. Yet each remains a separate individual, and would therefore be represented by a separate instance of the class *StaffMember*.

4.2.3 Class membership

The idea that instances belong to a class logically implies that there must be a test that determines to which class an instance belongs. Since membership is based on similarity, such a test will also be capable of determining whether any two instances belong to the same class. There are two distinct types of logical similarity that must be tested. First, all objects in a class share a common set of descriptive characteristics.

For example, the staff at Agate record a client’s company name, address, telephone number, fax number, email address and so on. Each item is included in the list because it is useful in some way to the users of the system¹, and the full list gives a complete description of a client. The value of each item (e.g. the actual company name) will vary from one client to another. But while the values may differ, this information *structure* is the same for every client.

To take another example, users of the Agate system also need to know about the member of staff assigned as staff contact to a client. A staff member might be described by a name, staff number and start date. Again, the full list gives a complete description of a staff member. Here, too, the value of each item (e.g. the staff

name) will normally vary from one person to another, but again the structure is the same for all staff members.

Now compare the two descriptions, summarized in Fig. 4.3.

Both staff members and clients have a name, but otherwise there is little in common. The information structure used to describe a client would not be capable of describing a member of staff, and vice versa. Even the sorts of name that would be valid in each case are likely to be different (apart from the relatively unusual cases where a company is known by the name of a person)². It is reasonable to consider all clients as members of one class, and all staff as members of one class, but it would not be justified to consider them as members of the *same* class. When two objects cannot be described by a single set of features, they cannot belong to the same class.

A second logical similarity must also be tested. All objects in a class share a common set of valid behaviours. For example, clients may initiate a campaign, may be assigned a staff contact, may pay for a campaign, terminate a current campaign, and so on. Perhaps no one client will actually do all this in reality, but that does not matter. The point is that any client could do any of these things, and the information system must reflect this.

If we consider staff members, we find a different set of valid behaviours. Staff can be assigned to work on a campaign, be assigned to a campaign as staff contact, change grade and maybe other things we do not yet know about. It may be more likely for staff members than for clients that they will actually go through the same sequence of behaviours, but this, too, does not matter. The point again is that a member of staff could do these things.

All clients, then, have a similar set of possible behaviours, and so do all staff members. But clients can do some things that a member of staff cannot do, and vice versa. On this count, too, we would consider clients as a class, and staff as a class, but we cannot make a case for considering clients and members of staff as instances of the same class. To summarize the examples, *Client* is a valid class, and *StaffMember* is a valid class. We can also note that, while the discussion here has been rather formal, an informal version of this dual test also works. In fact, we have already met this in Section 4.2.1: all members of a class give the same answers to the questions ‘What do I know?’ and ‘What can I do?’

Class	Characteristics	Class	Characteristics
Staff member	Name Staff number Start date	Client	Name Address Telephone number Fax number Email address

Figure 4.3 Information structures for two classes.

2 The values that an attribute can take are said to be contained within the *namespace* of the class. This means that each attribute value (for example, a company name) must be unique for its class. An attribute of another class might take the same value (for example, a staff member’s name) but, since this is in a separate namespace, there is no conflict between the two.

1 What appears in the list depends on the needs of the application. Another team may be developing a system to monitor whether client companies comply with environmental laws. Their list may share some characteristics with ours, but may also add others of no interest to our model.

Box 4.1 Class and object

Different books use the same terminology in different ways, and this can be confusing for the novice. For example, some authors use 'object' and 'class' interchangeably to mean a group of similar objects. But 'object' can also mean a single individual object, with 'class' reserved for the definition of a group of similar objects.

A further confusion arises if it is not clear whether 'object' refers to application domain things and concepts, or to their representation in an object-oriented model. The map is not the territory, even when the same words are used in both.

A third source of confusion can occur when the project focus shifts from analysis to design. During this transition, what was a model of the application domain is transformed into a model of software components. However, as we have not yet reached that point in the life cycle, we defer further discussion until later.

The UML Specification (OMG, 2004c) defines **class** as '... a set of objects that share the same specifications of features, constraints and semantics'. Moreover, 'The purpose of a class is to specify a classification of objects and to specify the features that characterize the structure and behaviour of those objects'.

Object is no longer explicitly defined in the UML Specification, but 'instance' (strictly, an instance specification) is described as something that 'specifies existence of an entity in a modelled system and completely or partially describes that entity' (OMG, 2004c).

We follow these definitions as closely as we can, and we use 'object' consistently to mean a single instance of a class in a model. 'Class' is used consistently to mean the descriptor of (specification of) a set of similar object instances in a model. When we need to refer

explicitly to people, things or concepts in the application domain, we will say so explicitly, as in this sentence.

A further distinction should be made between a *class* and its *type*. The UML Specification defines a **type** as 'A stereotyped class that specifies a domain of objects together with the operations applicable to the objects, without defining the physical implementation of those objects. A type may not contain any methods ... however it may have attributes and operations' (OMG, 2004c).

Basically, this means that a class represents a particular software implementation while a type is more abstract, and represents only a specification of some kind, but never an actual implementation in software. A type may be implemented by more than one class. This could be, for example, because a model must be implemented in such a way that the software can execute on different operating systems or hardware platforms. This means that, if we were to be completely strict about things, it would make more sense to use 'type' rather than 'class' when what we want to describe is an element in a domain model or a specification. Indeed, Cook and Daniels (1994) note that they prefer to use the term 'type' rather than 'class' to denote a collection of similar real-world objects.

However, we prefer to take a pragmatic approach and use 'class' for the most part as synonymous with 'type'. This is in keeping with the majority of authors on the subject, many of whom do not even discuss or define 'type'. For example, Maciaszek (2005) uses the term 'class' interchangeably to denote domain, model and software objects. In any case, where the context does not make it sufficiently clear which meaning we intend, further explanation will be given.

4.2.4 Generalization

In this section, we look at the concept of *generalization* and its application within object-orientation. In the UML Specification, the definition of generalization is written in such a way that it can apply to any *classifier*—a general concept that includes other modelling concepts such as interfaces, datatypes and components, as well as classes (which is itself an example of generalization). For the moment, we will concentrate on classes.

Generalization occurs where there is a taxonomic relationship between two classes. This means that the specification of one class is more general and applies also to the other class, while the specification of the latter is more specific and includes some details that do not apply to the former. Another way to put this is that any instance of the more specific class is also indirectly an instance of the more general class. The specific class inherits all the features of the more general class, while also adding some features that are uniquely its own (OMG, 2004c).

This is quite a lot to grasp in one take if you are new to the concept, so in the sections that follow we will dismantle the definition into its main components, and examine each part on its own. We will highlight the general principles by looking first at an example of species classification (note that this is intended only to illustrate the concept of generalization and is not a technical presentation of modern biological taxonomy). Later, in Chapter 8, we present an extended practical example from the Agate case study.

Taxonomic relationship

'Taxonomy' literally means a scheme of hierarchic classification—either an applied set of classifications or the principles by which that set is constructed. The word was originally used for the hierarchic system of classification of plant and animal species; hence the example shown in Fig. 4.4.

The taxonomic relationship between the two elements in this hierarchy labelled 'cat' and 'mammal' can be simply rephrased as 'the cat belongs to the class of mammals', or even more simply 'a cat is a kind of mammal'. Many other relationships are also identified in the diagram. For example, a domestic cat is a cat, as is a tiger, and both are also animals and living things. We can summarize this by saying that in each case the common relationship is that one element 'is a kind of' the other element.

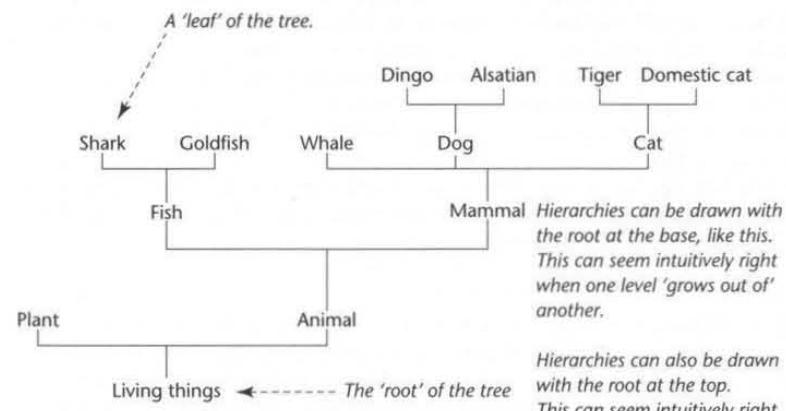


Figure 4.4 A simple taxonomy of species.

A more general class

'Mammal' is a word we use to classify any animal with certain characteristics. For example, all mammals have warm blood, suckle their babies on milk, have hair on their bodies and so on. 'Cat' is a word that classifies a subgroup of mammals, usually with thick fur and retractable claws. Many also have an ability to purr. Mammal is a more general class than cat, which in turn is more general than the domestic cat or the tiger. Any description of a mammal must apply to many different animals: domestic cats, tigers, dogs, whales, etc. A description of a 'general' mammal really means just a description of the common characteristics shared by all mammals. This may be rather brief, perhaps consisting of only one characteristic, e.g. that all mammals suckle their young. In the tree diagram, the more general an element is, the nearer it is to the root of the tree.

A more specific class

'Cat' is more specific than 'mammal' and 'domestic cat' still more specific than cat. The more specific word in any pair of related classes conveys more information. Knowing that an animal is a cat, we can guess something about its diet, general body shape, size (within certain limits), number of legs (barring accidents, etc.) and so on. If we know only that it is a mammal, we can guess little, if anything at all, of its physical description or diet. The more specialized elements of a hierarchy are those that are further from the root of the tree and closer to the leaves. The most specialized elements of all are those that actually form the leaves of the tree. In Fig. 4.4, the leaves are individual species³.

Any instance of the specific class is also indirectly an instance of the more general class

Whatever is true for a mammal is also true for a domestic cat. If the defining characteristic of a mammal is that it suckles its young, then a domestic cat also suckles its young, and so does a tiger, a dog or a whale. This is an important feature of any hierarchic taxonomy. As an illustration, we can think about what would happen if a zoologist discovers that a defining mammalian characteristic does not apply to an animal previously thought to be a mammal. For example, suppose a research project found conclusive evidence that common field mice lay eggs instead of bearing live young. Imagine the consternation and argument! Zoologists would have to decide whether to reclassify the field mouse (perhaps as a previously unknown species of monotreme) or to redefine what distinguishes a mammal from other animals. Perhaps a brand new classification would be invented specifically to accommodate egg-laying mice (as occurred following the discovery of the duck-billed platypus, which in most respects resembles a mammal, except that it lays eggs instead of bearing live young).

Specialization adds additional information

A full description of a domestic cat would contain a great deal more information than needed for a general member of the mammal class. For example, we might define a domestic cat by saying that it suckles its young, has a certain skeletal structure, a particular arrangement of internal organs, carnivorous teeth and habit,

thick fur, the ability to purr, and so on. Apart from suckling its young, none of these characteristics applies to *all* other mammals. A full zoological description of any species contains at least one characteristic (or a unique combination of characteristics) that differentiates it from all other species. Otherwise it would not make sense to consider it a species in the first place.

Practical uses of generalization

The main use of *generalization* in object-orientation is to describe relationships of similarity between classes. Object classes can be arranged into hierarchies much the same as the species example. This has two main benefits.

The first results from the use of object classes to represent different aspects of a real-world situation that we wish to understand. Using generalization, we can build logical structures that make explicit the degree of similarity or difference between classes. This is an important aspect of the *semantics* of a model—in other words, it helps to convey its meaning. For example, to know what hourly-paid and monthly-salaried employees in a business have in common with each other may be just as important as to know how they differ. The former may help to understand that some types of information must be recorded in identical ways for both types of employee. Figure 4.5 illustrates this with an example that might be suitable for a payroll system.

In this model, each employee is represented by their date of appointment, date of birth, department, employee number, line manager and name, and some details that depend on whether they are paid weekly or monthly (the only significant differences shown). A hierarchically structured model allows the close similarity to be shown clearly.

A second benefit comes from the relative ease with which a hierarchy can be extended to fit a changing picture. If this company were to decide that a new,

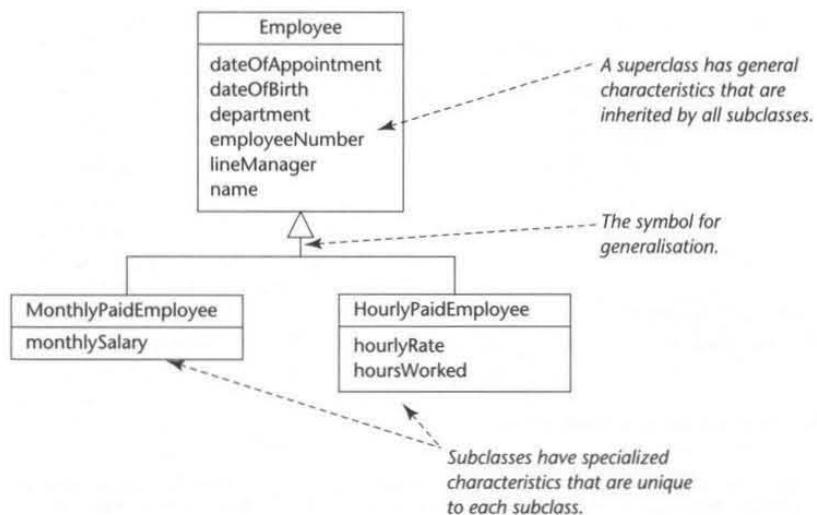


Figure 4.5 Hierarchy of employee types.

³ If 'species' is itself considered as a class, then bear in mind that this is another sort of abstraction of its individual members.

weekly-paid type of employee is required, it is a simple matter to add a new subclass to the hierarchy to cater for it, as shown in Fig. 4.6.

Other features of generalization

Some features of generalization that are not mentioned explicitly in the definition given earlier are sufficiently important to deserve some discussion at this point. These are: the mechanism of *inheritance*, the *transitive* operation of inheritance and the *disjoint* nature of generalization hierarchies.

Inheritance. This is a mechanism for implementing generalization and specialization in an object-oriented programming language. When two classes are related by the mechanism of inheritance⁴, the more general class is called a *superclass* in relation to the other, and the more specialized is called its *subclass*. As a first approximation, the rules of object-oriented inheritance generally work as shown below.

1. A subclass inherits all the characteristics of its superclass⁵.
2. A subclass definition always includes at least one detail not derived from its superclass.

Inheritance is very closely associated with generalization. Generalization describes the logical relationship between elements that share some characteristics, while inheritance describes an object-oriented mechanism that allows the sharing to occur.

Transitive operation. This means that the relationship between two elements at adjacent levels of a hierarchy ‘carries over’ to all more specialized levels. Thus, in Fig. 4.4, the definition of an animal applies in turn to all mammals, and thus by a series of logical steps to a domestic cat. So we can rewrite the rules of inheritance given above, as follows.

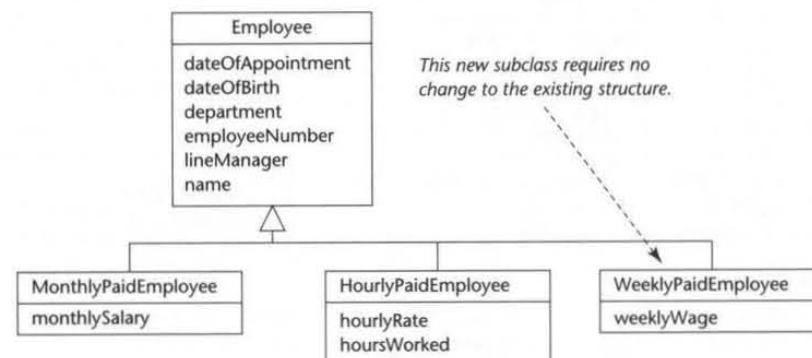


Figure 4.6 Hierarchies are easy to extend.

⁴ The name comes from a superficial similarity to biological inheritance, which also takes the form of a hierarchy. But object-oriented inheritance is very different from inheritance in living things, and care should be taken not to confuse the two.

⁵ We shall see in Chapter 8 that inherited characteristics are sometimes overridden in a subclass.

1. A subclass always inherits all the characteristics of *all* its superclasses.
2. The definition of a subclass always includes at least one detail not derived from *any* of its superclasses.

Disjoint nature. In a hierarchic system, the branches of the tree diverge as they get further away from the root and closer to the leaves. They are not permitted to converge. This means, for example, that a cat cannot be both a mammal and a reptile. In other words, each element in a hierarchy can only be a member of one classification at any given level of the hierarchy (although, of course, it can be a member of other classifications at other levels of the hierarchy, due to the transitive nature of the relationship).

The disjoint aspect of generalization means that we sometimes need to be careful about the characteristics chosen to express a generalization. For example, we could not use ‘Has four feet’ as the only defining characteristic of a mammal, even supposing that it were true of all mammals—because many lizards also have four feet, and this would make it possible to classify a lizard as a mammal. A class must be defined in terms of a unique set of characteristics that differentiate it from all other classes at its level of the hierarchy.

It is worth stressing here that generalization structures are abstractions that we choose to apply, since they express our understanding of some aspects of an application domain. This means that we can also choose to apply more than one generalization structure to the same domain, if it expresses a relevant aspect of the situation. Thus, for example, a human might be simultaneously classified as a creature (*Homo sapiens*), as a citizen (a voter in a city electoral division) and as an employee (an Account Manager in Agate’s Creative Department). If each of these were represented as a hierarchy in an object-oriented model, the position of a human might be an example of multiple inheritance, which means that a subclass is at once a member of more than one hierarchy and inherits characteristics from every superclass in each hierarchy.

We should also note that real-world structures are not compelled to follow the logical rules applied in object-oriented modelling. Sometimes they are not disjoint or transitive, and therefore not strictly hierarchic. This does not detract from the usefulness of hierarchic structures in object-oriented development.

4.2.5 Message passing

In an object-oriented system, objects communicate with each other by sending messages. Objects often represent things in a real-world system that collaborate to carry out a collective task. This is also how people communicate with each other: we send messages too. In the case of human communication, the means by which we transmit our messages are diverse, and sometimes it is not obvious who is the sender or receiver for a particular message, but all human communication can be seen as made up of messages. For example, everything we say to our friends and family, the emails we read when we log onto the network, advertising posters on the bus, games shows and cartoons on TV, even the clothes we wear, our tone of voice and our posture—these are all messages of one sort or another. If this is so, then just what is special about saying that objects only communicate via messages?

What makes it special is the fact that software was not constructed in this way until quite recently. Earlier approaches to systems development tended to separate

data in a system from the processes that act on the data. This was done for sound analytical reasons and is still appropriate for some applications, but it can give rise to difficulties. Chief among these is the need for the person who designs a process to understand the organization of the data that it uses; for such a system, processes are said to be dependent on the structure of the data.

In itself, this is not necessarily a problem—in fact, it is to a certain extent inescapable. But if taken too far, and for insufficient reason, dependency of process upon data structure can cause many problems. For example, if the data structure were changed for any reason (a not uncommon occurrence in business), those processes that use the data may also need to be changed. This is an example of coupling between subsystems, and we must minimize coupling if we are to construct systems that are reliable, that can be upgraded or modified, and that can be easily repaired if they break down (coupling is discussed in Chapters 13 and 14). Object-oriented systems avoid these problems by locating each process, as far as is practicable, with the data it uses. This is another way of describing an object: it is really little more than a bundle of data together with some processes that act on the data. These processes are called *operations* and each has a specific *signature*. An operation signature is a definition of its interface. In order to *invoke* an operation, its signature must be given (signatures are sometimes also called *message protocols*).

In practice, it is not usually possible for all processes to be located with all the data that they must access, and data and processes are distributed among many different objects. Message passing is a way of insulating each object from the need to ‘know’ any of the internal details of other objects. Essentially, an object knows only its own data, and its own operations. But in order for collaboration to be possible, the ‘knowledge’ of some objects includes knowing how to request services from other objects, which may include the retrieval of data. In this case, an object knows which other object to ask, and how to formulate the question. But it is not necessary for an object to know anything about the way that another object will deliver the service. Such knowledge would introduce an undesirable and unnecessary degree of coupling into the relationship between the two objects.

When an object receives a message it can tell instantly whether the message is relevant to it. If the message includes a valid signature to one of its operations, the object can respond. If it does not, the object cannot respond. So we can think of operations as located within objects, only able to be invoked by a message that gives a valid operation signature. The object’s data lies even deeper inside, and can only be accessed by an operation of that object. The organization of an object’s data can be changed without affecting any collaborating objects which use that data, as long as the signatures to that object’s operations are not changed. Such changes would not be visible from the outside. For this reason, the complete set of signatures for an object is sometimes known as its *interface*. This hiding of information behind an interface is called *encapsulation* and is illustrated in Fig. 4.7.

We can illustrate the advantages of designing software in this way by an example. Consider a simple system to print pay cheques for employees in a business. Suppose that there is a class *Employee*, with an instance that represents each person on the payroll. Each *Employee* object is responsible for knowing about the salary earned by the real employee it represents. Suppose also that a *PaySlip* object is responsible for printing each employee’s payslip each month. In order to print the payslip, each *PaySlip* object must know how much the corresponding employee has earned. One object-oriented approach to this is for each *PaySlip*

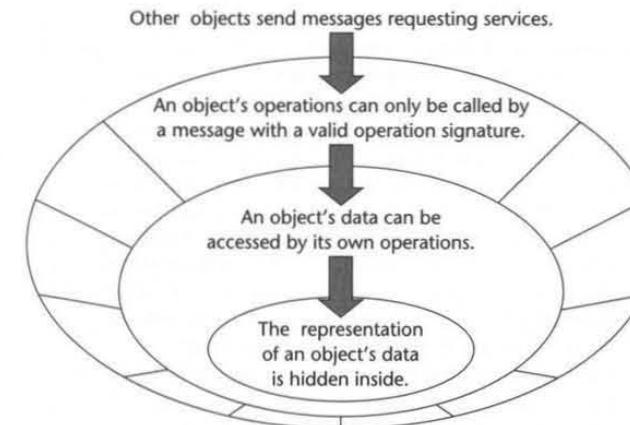


Figure 4.7 Encapsulation: the layers of protection that surround an object.

object to send a message to the associated *Employee* object, asking how much salary should be paid. The *PaySlip* object need not know how the *Employee* object works out the salary, nor what data it stores. It only needs to know that it can ask an *Employee* object for a salary figure, and an appropriate response will be given. Message passing allows objects to encapsulate (i.e. hide) their internal details from other parts of the system, thus minimizing the knock-on effects of any changes to the design or implementation.

4.2.6 Polymorphism

When one person sends a message to another, we often ignore the details of how the other person might respond. For example, a mother might use the same phrasing to tell her child to ‘go to bed now’, whether the child is a five-year-old boy or a thirteen-year-old girl. But the precise tasks to be carried out by each child in going to bed may be very different. The five-year-old may set off towards bed by himself, but perhaps then requires help with washing his face, brushing his teeth and putting on his pyjamas; he may also expect to be read a bedtime story. The thirteen-year-old may not require any further help, once convinced that it really is bedtime.

This is rather like *polymorphism*, which is an important element in the way that object-oriented approaches encourage the decoupling of subsystems. Polymorphism literally means ‘an ability to appear as many forms’ and it refers to the possibility of identical messages being sent to objects of different classes, each of which responds to the message in a different, yet still appropriate, way. This means the originating object does not need to know which class is going to receive the message on any particular occasion. The key to this is that each receiving object is responsible for knowing how to respond to messages.

Figure 4.8 uses a communication diagram to illustrate polymorphism in a business scenario (communication diagrams appear again in Chapter 7 and their notation is described fully in Chapter 9). The diagram assumes that there are different ways of calculating an employee’s pay. Full-time employees are paid a

salary that depends only on their grade; part-time staff are paid a salary that depends in a similar way on grade, but must also take into account the number of hours worked; temporary staff differ in that no deductions are made for the company pension scheme, but the salary calculation is otherwise the same as for a full-time employee. An object-oriented system to calculate pay for these employees might include a separate class for each type of employee, each able to perform the appropriate pay calculation. However, following the principle of polymorphism, the message signature for all `calculatePay` operations could be the same. Suppose one output from this system is a print-out showing the total pay for the current month: to assemble the total, a message is sent to each employee object, asking it to calculate its pay. Since the message signature is the same in each case, the requesting object (here called `MonthlyPayPrint`) need not know the class of each receiving object, still less how each calculation is carried out.

Polymorphism is a powerful concept for the information systems developer. It permits a clear separation between different subsystems that handle superficially similar tasks in a different manner. This means that a system can easily be modified or extended to include extra features, since only the interfaces between classes need to be known. The way that each part of the system is implemented (its internal structure and behaviour) is hidden from all parts of the system that do not absolutely need this knowledge.

Program designers and programmers have struggled for many years to achieve such a high level of modularity in software. Object-orientation gives greater promise of practical success than did any of its predecessors.

4.2.7 Object state

In the real world, people and objects do not always behave in exactly the same way in response to similar stimuli. For example, if you have just eaten a good lunch, you

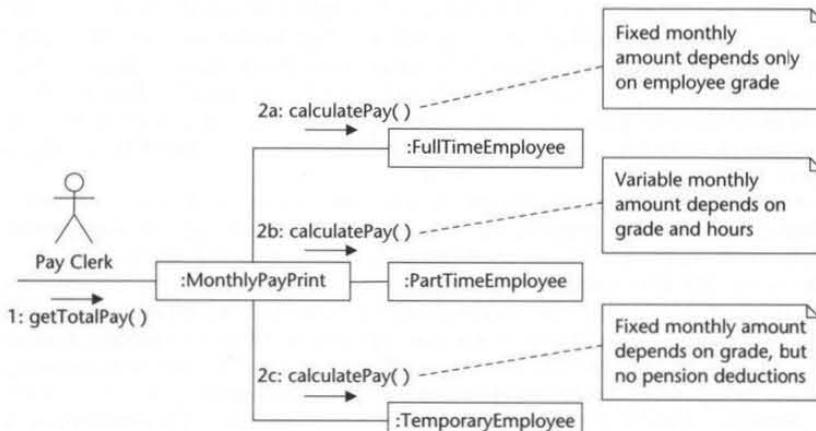


Figure 4.8 Polymorphism allows a message to achieve the same result even when the mechanism for achieving it differs between different objects.

will be much more likely to refuse the offer of a large, sticky gateau. However, if you happen to be very hungry at the time, this offer may be much more tempting. A simple way to represent this difference is to say that at any given time you can be in either of two *states*: well-fed or hungry. Each state is characterized by data that describes your condition—in this case your level of hunger. Each state is also characterized by a difference in behaviour. In other words, your response to certain messages differs according to the current value of your data—when well-fed you refuse food, while when hungry you accept it. Your behaviour in response to a stimulus can also change your state. After eating a large gateau, your state changes from hungry to well-fed (perhaps for some people more than one would be required).

Objects can also occupy different states, and this affects the way that they respond to messages. Each state is represented by the current values of data within the object, which can in turn be changed by the object's behaviour in response to messages. Thus a state is a steady or stable period in the life of the object, during which it carries out an activity, waits for some event to occur, or meets some specified condition (Booch et al., 1999).

The concept of object state is fundamental to an understanding of the way that the behaviour of an object-oriented software system is controlled, so that the system responds in an appropriate way when an external event occurs. The significance of control is perhaps most apparent for safety-critical real-time systems. For example, consider the engine and flight controls in a modern airliner. Normally, for the greater part of the flight, an onboard computer flies the aircraft entirely automatically. During take-off and landing the pilot and flight crew take control, but even then it is still the software that directly operates the engine throttles, elevators and so on. It could be disastrous if all the engines of an airliner were shut down during the final approach to land, and the software is designed to prevent this occurring. However, it may sometimes be necessary for the pilot to override this constraint during an emergency. The software must also be designed to permit this to occur and to correctly discriminate between the different situations. In order to ensure safety, the control software must be designed so that only the appropriate control behaviours (shut down engine, full throttle, climb, descend, turn) can occur in each possible state of the aircraft (parked, climbing, flying level on auto-pilot, landing). This must also take account of all external events that could trigger a change in behaviour or state (pilot operates throttle, turbulence causes course deviation, sudden loss of cabin pressure, etc.).

The identification of object states can also be critical to the correct operation of business information systems. Here, the consequences of error are not so often life threatening, but they can threaten the survival of the organization and thus have an impact on the lives and livelihoods of customers, workers, investors and others associated with the enterprise. We return to this topic again in Chapter 7 and it is explained in detail in Chapter 11.

4.3 The Origins of Object-Orientation

Object-orientation is only one stage, and almost certainly not the last, in a long development path. We review below some strands in the history of computing that have led to object-oriented analysis and design.

Increasing abstraction

From the very earliest days of digital computers, there has been a steady increase in the level of abstraction at which programmers have been able to operate. The increase in abstraction applies both to the activity of programming itself and to the tasks that computer programs are expected to perform. In a relatively short time, the purposes to which computers are applied have become enormously more complex and demanding, thus greatly increasing the complexity of the systems themselves. Some of the waymarks in this progress are summarized in Fig. 4.9.

Event-driven programming

Early work on computer simulation led directly to the object-oriented paradigm of independent, collaborating objects that communicate via messages. A typical simulation task is to model the loading of vehicles onto a large ferry ship, in order to determine the quickest safe way to do this. This simulation would be run many times under differing assumptions, for example, the sequence of loading, the arrangement of vehicles on the storage decks, the speed at which vehicles are driven onto the ship, the time separation between one vehicle and the next, and so on. The real-world situation that is being modelled is very complex and consists of a number of independent agents, each responding to events in ways that are easy to model at the level of individual agents, but very difficult to predict in the large, with many agents interacting with each other all at once. This kind of task is very

Period	Programming languages	Interface technology	Typical applications
1940s	Machine code	Hardwired input	Military and scientific
1950s	Assembly language First third-generation languages (3GLs), e.g. Fortran, Algol	Punched tape and cards	Simple commercial software, such as early project planning programs
1960s	More mature 3GLs, e.g. COBOL, BASIC, CPL (early precursor of C)	Punched cards, teleprinters	Commercial systems such as accounts and payroll spread; scientific applications grow more complex
1970s	Non-procedural languages: Prolog, ADA, Smalltalk	VDU terminals	Mainframe networks begin to support early management information systems
1980s	Object-oriented languages, e.g. Eiffel, Objective-C, C++	PCs Early GUIs	Spread of office automation and spreadsheets; email introduced in some businesses
1990s	Java, Python, C#	GUIs Web applets	Enterprise resource planning, data warehouses, multimedia applications, e-business applications

Figure 4.9 The increasing abstraction of programming (greatly simplified).

difficult to program effectively in a procedural language (Fortran, for example), since program designs for procedural languages are based on the assumption that the program structure controls the flow of execution. Thus for a procedural program to tackle the simulation task described above, it must have separate routines that test for, and respond to, a vast number of alternative conditions.

One solution to this problem is to structure the program in a similar way to the problem situation itself: as a set of independent software agents, each of which represents an agent in the real-world system that is to be simulated. This insight evolved into early simulation languages such as Simula and remains one of the key ideas in object-oriented software development: that the structure of the software should mirror the structure of the problem it is intended to solve. In this way, the tension between the model of the application domain and the model of the software (mentioned in Section 4.2.1) is resolved, turning a potential weakness into a strength.

The spread of GUIs

The rapid spread of graphical user interfaces (GUIs) in the 1980s and 1990s posed particular difficulties for contemporary development methods. GUIs brought some of the problems encountered earlier in simulation programming into the world of mainstream business applications. The reason for this is that users of a GUI are presented on their computer screen with a highly visual interface that offers many alternative actions all at once, each one mouse-click away. Many other options can be reached in two or three more clicks via pull-down menus, list boxes and other dialogue techniques. Interface developers naturally responded by exploiting the opportunities offered by this new technology. As a result, it is now almost impossible for a system designer to anticipate every possible route that a user might take through a system's interface. This means that the majority of desktop applications are now very difficult to design or control in a procedural way. The object-oriented paradigm offers a natural way to design software, each component of which offers clear services that can be used by other parts of the system quite independently of the sequence of tasks or the flow of control.

Modular software

In an object-oriented system, classes have two kinds of definition. From an external perspective, a class is defined in terms of its interface, which means that other objects (and their programmers) need only know the services that are offered by objects of that class and the signature used to request each service. From an internal perspective, a class is defined in terms of what it knows and what it can do—but only objects of that class need to know anything about this internal definition. It follows that an object-oriented system can be constructed so that the implementation of each part is largely independent of the implementation of other parts, which is what modularity means. This contributes to solving some of the most intractable problems in information systems development.

- It is easier to maintain a system built in a modular way, as changes to a subsystem are much less likely to 'ripple' through the rest of the system.
- For the same reason, it is easier to upgrade a modular system. As long as replacement modules adhere to the interface specifications of their predecessors, other parts of the system are not affected.

- It is easier to build a system that is reliable in use. This is because subsystems can be tested more thoroughly in isolation, leaving fewer problems to be addressed later when the whole system is assembled.
- A modular system can be implemented in small, manageable increments. Provided each module is designed to provide a useful and coherent package of functionality, they can be introduced one at a time.

Life cycle problems

Most structured methodologies of the 1980s and early 1990s were based on the waterfall life cycle model originally designed for large engineering projects, such as constructing new jet aircraft and building road bridges. The difficulties with this life cycle model have already been discussed in Chapter 3. Object-orientation addresses them by encouraging a cyclic development approach. In this, there is little difficulty in revisiting earlier stages and revising earlier products. This iterative process can repeat, if necessary, until everyone is satisfied with the quality of the software—subject, of course, to time and budget constraints.

This aspect is strongly linked to the highly modular character of an object-oriented system, described in the previous section, and also to the ‘seamless’ development of models throughout an object-oriented life cycle. The latter point is discussed in the next section.

Model transitions

In structured approaches to information systems development, the models of process that are developed during the analysis phase (e.g. data flow diagrams) have only an indirect relationship to the process models developed during the design phase (e.g. structure charts or, in SSADM, update process models). This has meant that designs for new systems, however good in their own right, have been hard to trace back to the original requirements. Yet, what makes a design successful is that it meets the requirements in a way that is functional, efficient, economic, and so on (see Chapter 13). This is particularly frustrating when one considers that a major strength of the structured methods was their concentration of effort on achieving a good understanding of users’ requirements.

Object-oriented analysis and design avoid these transition problems by using a core set of models throughout analysis and design, adding more detail at each stage and avoiding the awkward discontinuities that arise when one model must be discarded to be replaced by another with a different, incompatible structure. In UML, the fundamental analysis models are the use case and the class diagram (described in Chapters 6 and 7, respectively) and these continue as the backbone of the design, with other design models derived directly or indirectly from them.

Reusable software

Information systems are very expensive, yet in the past their developers have tended to reinvent new solutions to old problems, many times over. This wastes time and money, and has led to the demand for reusable software components, which can eliminate the need to keep reinventing the wheel. Object-oriented development methods offer great potential, not yet fully realized, for developing software components that are reusable in other systems for which they were not originally designed. This is as a result of the highly modular nature of object-ori-

mented software, and also of the way that object-oriented models are organized. Inheritance is of particular importance in this context and we say more about this in Chapter 8.

4.4 Object-Oriented Languages Today

A number of object-oriented programming languages are available today, with some significant differences between their capabilities and the extent to which they adhere to the object-oriented paradigm. Figure 4.10 lists some of the main characteristics of the most widely used of these languages. This is not a programming textbook, so we offer only a very brief description of each feature, as follows.

Strong typing refers to the degree of discipline that a language enforces on the programmer when declaring variables. In a strongly typed language (most modern languages are strongly typed), every data value and object that is used must belong to an appropriate type for its context. Static typed languages enforce this with type-checking at compile time. Dynamic typed languages check types at run-time, but some languages offer a hybrid approach that allows the flexibility of loading classes at run-time. Garbage collection is important for memory management in systems that create and delete many objects during their execution. If objects are not removed from memory when they are deleted, the system may run out of memory in which to execute. When this is provided automatically, it removes the responsibility for this task from the programmer. Multiple inheritance refers to the capacity of an object to acquire features from more than one hierarchy. This is important as it minimizes the amount of code duplication and hence reduces inconsistencies that can cause maintenance problems. In static typed languages, multiple inheritance can allow a new class to stand in for any of its superclasses, and this reduces the amount of special-case programming required elsewhere in the system. Languages in which all constructs are implemented as classes or objects are said to be ‘pure’ object-oriented languages. Some languages permit data values that are not objects, but this introduces extra complexities for the programmer. Other languages allow unencapsulated types, but this gives the sloppy programmer opportunities to bypass the safe encapsulation of classes. Both of these circumstances can cause a system to be difficult to maintain and extend.

Feature	Smalltalk	C++	Eiffel	Java	C#
Strong typing	✓	optional	✓	✓	✓
Static/dynamic typing (S/D)	D	S	S	S + D	S + D
Garbage collection	✓	X	✓	✓	✓
Multiple inheritance	X	✓	✓	X	X
Pure objects	✓	X	✓	X	X
Dynamic loading	✓	X	X	✓	✓
Standardized class libraries	✓	X	✓	✓	X
Correctness constructs	X	X	✓	X	X

Figure 4.10 Characteristics of some widely used object-oriented languages.

Dynamic loading refers to the ability of a language to load new classes at run-time. This can be used to allow software to reconfigure itself, for example to cope with hardware or environment changes. It can also be used in a client–server environment to allow the client software to evolve by loading it with different classes from the server. Dynamic loading can help to propagate improvements and bug fixes, by concentrating maintenance efforts on the server side. Standardized class libraries give the programmer access to classes that are known to run successfully on a variety of hardware platforms, and under a variety of operating systems. When these are not available, it can be difficult to modify an application so that it will run on another platform, or in conjunction with applications that have used a different library. Finally, correctness constructs include pre-conditions and post-conditions on methods, forming a contract between any client–supplier pair. Contracts are important to the development of robust software and are discussed in Chapter 10.

Limitations of object-orientation

Some applications are not ideally suited to object-oriented development and in this section we make a few comments about these. There are two main examples. The first kind includes systems that are strongly database-oriented. By this we mean both that they have a record-based structure of data that is appropriate to a relational database management system (RDBMS) and also that their main processing requirements centre on the storage and retrieval of the data (e.g. a management information system used mainly for querying data in the database). Such applications cannot easily be adapted to an object-oriented implementation without losing the many benefits of using a RDBMS for data storage. Commercial RDBMS are a very mature form of technology that organize their data according to sound mathematical principles. This ensures a good balance of efficiency of retrieval, resilience to change and flexibility in use. However, RDBMS are limited in their capabilities for storing and retrieving certain kinds of complex data structure, such as those that represent multimedia data. Graham et al. (1998) mention the spatial (map-based) data that forms the basis of a geographic information system (GIS) as a particular example of data structures to which RDBMS are not well suited, but which are ideally suited to object-oriented development. We return to the relative advantages and disadvantages of RDBMS as compared to object-oriented database systems in Chapter 18.

Applications that are strongly algorithmic in their operation are less suited to an object-oriented development approach. For some scientific applications that involve large and complex calculations (for example, satellite orbit calculations) it may be neither feasible nor desirable to split the calculation down into smaller parts. Such a system, if developed in an object-oriented manner, might contain very few objects, but each would be extremely complex. This would not be sound object-oriented design, and so either a procedural or a functional approach (these are alternative styles of programming) is recommended instead.

4.5 Summary

In this chapter we have introduced the most important concepts in object-orientation, in particular, object and class, generalization and specialization,

message passing, object state and polymorphism. Understanding these gives an essential foundation for later chapters that deal with the practical application of object-oriented analysis and design techniques. There is a great deal of synergy in the way that the different fundamental concepts contribute to the success of object-orientation. For example, message passing and polymorphism both play a significant role in achieving sound modularity in a system. But there is no clean break with the past; instead, the characteristics of object-orientation are best seen as the result of a gradual process of evolution that can be traced back to the earliest days of electronic digital computers. This evolutionary process is by no means finished yet, but rushes onwards into the future. As applications and computing environments grow ever more complex, there is a continuing need for reliable, maintainable, modifiable information systems.

Review Questions

- 4.1** Define object, class and instance.
- 4.2** What do you think is meant by 'semantics'?
- 4.3** How does the object-oriented concept of message passing help to encapsulate the implementation of an object, including its data?
- 4.4** What is polymorphism?
- 4.5** What is the difference between generalization and specialization?
- 4.6** What rules describe the relationship between a subclass and its superclass?
- 4.7** What does it mean to say that an object-oriented system is highly modular?
- 4.8** Why is it particularly hard for a designer to anticipate a user's sequence of tasks when using a GUI application?
- 4.9** What does 'object state' mean?
- 4.10** What is an operation signature?

Case Study Work, Exercises and Projects

- 4.A** Section 4.2.1 mentions the human activity system and the proposed software system as particularly important systems to consider, but these are not the only systems that an analyst will encounter or work with. Make a list of any other systems you can think of that might be involved in the process of software development. What interfaces exist between them?
- 4.B** Re-read the description of generalization given in Section 4.2.4. How does object-oriented inheritance differ from inheritance between a parent and a child: (i) in biology and (ii) in law?
- 4.C** Arrange the following into a hierarchy that depends on their relative generalization or specialization: person, thing, green, shape, primary school teacher, cub, polar bear, square, law, child, colour, animal. Add more classifications as necessary so that it is clear what is generalized or specialized at each level.

4.D Read the first section of the case study material for FoodCo (Section B1.1), and identify classes that represent FoodCo's whole business environment.

4.E List all FoodCo's products that are identified in the case study material in Case Study B1 and arrange these into a hierarchy. Imagine some more products that make your hierarchy more interesting, and add these to your diagram.

Further Reading

Most standard texts on object-oriented analysis and design contain a section that introduces the fundamental concepts of object-orientation. Although predating the development of UML, Jacobson et al. (1992) and Rumbaugh et al. (1991) remain good general introductions.

Most recent books use UML notation, for example, Maciaszek (2005). Naturally, the most authoritative are those from the Rational stable, in particular Jacobson et al. (1999). (Rational Software became part of IBM early in 2003.)

Readers who are interested in learning more about object-oriented programming will find almost too many books on this subject to be able to count them all. In the Java field, Bishop (2001) is among the more readable, although note that this book has not yet been updated to UML 2.

Agate Ltd Case Study—Introduction

Agate Ltd

A1.1 Introduction to Agate

Agate is an advertising agency in Birmingham, UK. Agate was formed as a partnership in 1982 by three advertising executives, Amarjeet Grewal, Gordon Anderson and Tim Eng (the name is a combination of their initials). Amarjeet and Gordon had previously worked for one of the UK's largest and most successful advertising companies in London, but felt frustrated at the lack of control they had over the direction of the campaigns they worked on. As a result, they moved to the West Midlands region of the UK and set up their own business in 1981. Shortly afterwards they were joined by Tim Eng, with whom they had worked on a project in Hong Kong, and Agate was formed.

In 1987, the three partners formed a UK limited company and between them own all the shares in it. Gordon Anderson is Managing Director, Amarjeet Grewal is Finance Director and Tim Eng is Creative Director. They now employ about fifty staff at their office in the centre of Birmingham (see Fig. A1.1) and a further hundred or so at seven offices around the world. Each of these other offices is set up locally as a company with the shares owned jointly by Agate and the local directors.

Initially the company concentrated on work for the UK motor industry, much of which is located in the West Midlands region. However, as the company has expanded and internationalized, the type of work it takes on has changed and it now has clients across a wide range of manufacturing and service industries.

The company strategy is to continue to grow slowly and to develop an international market. The directors would like to obtain business from more large multinational companies. They feel that they can offer a high standard of service in designing advertising campaigns that have a global theme but are localized for different markets around the world.

The company's Information Systems strategy has a focus on developing systems that can support this international business. Currently, as well as considering the



Figure A1.1 Staff at Agate Ltd UK office.

development of a new business information system, the directors are deciding whether to invest in hardware and software to support digital video editing. They have until now subcontracted video editing work, but with relatively cheap broadband access they have the capability for fast file-transfer of digital video between offices.

A1.2 Existing Computer Systems

Agate already uses computers extensively. Like most companies in the world of design and creativity, Agate uses Apple Macintosh computers for its graphic designers and other design-oriented staff. The secretaries and personal assistants also use Apple Macs. However, the company also uses PCs to run accounts software in Microsoft Windows.

Last year, Agate had a basic business system for the UK office developed in Delphi for Windows. However, after the system was developed, the directors of Agate decided that it should have a system developed in Java, the object-oriented language originated by Sun Microsystems Inc. One of the reasons for the choice of Java was that it is portable across different hardware platforms and the company wants software that could run both on the PCs and on the Macs. Another reason is that Agate foresees the possibility of using Java to deliver information to clients via the World Wide Web. Unfortunately, the person who developed the Delphi software for the company (and was going to rewrite it in Java) was headhunted by an American software house, because of her skills in Java, and has moved to the USA. Fortunately, this developer, Mandy Botnick, was methodical in her work and has left Agate with some object-oriented system documentation for the system she designed and developed.

This existing system is limited in its scope: it only covers core business information requirements within Agate. It was intended that it would be extended to cover most of Agate's activities and to deal with the international way in which the business operates.

A1.3 Business Activities in the Current System

Agate deals with other companies that it calls clients. A record is kept of each client company, and each client company has one person who is the main contact person within that company. His or her name and contact details are kept in the client record. Similarly, Agate nominates a member of staff—a director, an account manager or a member of the creative team—to be the contact for each client.

Clients have advertising campaigns, and a record is kept of every campaign. One member of Agate's staff, again either a director or an account manager, manages each campaign. Other staff may work on a campaign and Agate operates a project-based management structure, which means that staff may be working on more than one project at a time. For each project they work on, they are answerable to the manager of that project, who may or may not be their own line manager.

When a campaign starts, the manager responsible estimates the likely cost of the campaign, and agrees it with the client. A finish date may be set for a campaign at any time, and may be changed. When the campaign is completed, an actual completion date and the actual cost are recorded. When the client pays, the payment date is recorded. Each campaign includes one or more adverts. Adverts can be one of several types:

- newspaper advert—including written copy, graphics and photographs
- magazine advert—including written copy, graphics and photographs
- Internet advert—including written copy, graphics, photographs and animations
- TV advert—using video, library film, actors, voice-overs, music etc.
- radio advert—using audio, actors, voice-overs, music etc.
- poster advert—using graphics, photographs, actors
- leaflet—including written copy, graphics and photographs.

Purchasing assistants are responsible for buying space in newspapers and magazines, space on advertising hoardings, and TV or radio air-time. The actual cost of a campaign is calculated from a range of information. This includes:

- cost of staff time for graphics, copy-writing etc.
- cost of studio time and actors
- cost of copyright material—photographs, music, library film
- cost of space in newspapers, air-time and advertising hoardings
- Agate's margin on services and products bought in.

This information is held in a paper-based filing system, but the total estimated cost and the final actual cost of a campaign are held on the new computer system.

The new system also holds the salary grades and pay rates for the staff, so that the cost of staff time on projects can be calculated from the timesheets that they fill out. This functionality has been partially implemented and is not used in the existing system.

A1.4 Summary of Requirements

This section summarizes the requirements for the new system.

- 1. To record details of Agate's clients and the advertising campaigns for those clients.**
 - 1.1** To record names, address and contact details for each client.
 - 1.2** To record the details of each campaign for each client. This will include the title of the campaign, planned start and finish dates, estimated costs, budgets, actual costs and dates, and the current state of completion.
 - 1.3** To provide information that can be used in the separate accounts system for invoicing clients for campaigns.
 - 1.4** To record payments for campaigns that are also recorded in the separate accounts system.
 - 1.5** To record which staff are working on which campaigns, including the campaign manager for each campaign.
 - 1.6** To record which staff are assigned as staff contacts to clients.
 - 1.7** To check on the status of campaigns and whether they are within budget.
- 2. To provide creative staff with a means for recording details of adverts and the products of creative process that leads to the development of concepts for campaigns and adverts.**
 - 2.1** To allow creative staff to record notes of ideas for campaigns and adverts.
 - 2.2** To provide other staff with access to these concept notes.
 - 2.3** To record details of adverts, including the progress on their production.
 - 2.4** To schedule the dates when adverts will be run.
- 3. To record details of all staff in the company.**
 - 3.1** To maintain staff records for creative and administrative staff.
 - 3.2** To maintain details of staff grades and the pay for those grades.
 - 3.3** To record which staff are on which grade.
 - 3.4** To calculate the annual bonus for all staff.
- 4. Non-functional requirements.**
 - 4.1** To enable data about clients, campaigns, adverts and staff to be shared between offices.
 - 4.2** To allow the system to be modified to work in different languages.

FoodCo Ltd Case Study—Introduction

FoodCo Ltd



B1.1 Introduction to FoodCo

FoodCo produces a range of perishable foods for supermarkets and is based in the flat agricultural lands of the East Anglia region of the UK. John Evans, the present Chairman, started the company when he left the Royal Air Force. He borrowed money to buy 200 acres (81 hectares) of arable farmland, but his ambition was to be more than a farmer. As soon as Home Farm was running he opened a factory in a converted barn.

The first product was a pickle made to a traditional family recipe. It sold well, and success financed expansion. Two years later, John acquired derelict land next to the farm and the company moved into a larger, purpose-built factory. The product range soon extended to pre-packed vegetables and salads, and later a wide range of sauces, pickles and sandwich toppings, in fact almost anything made of vegetables that can be sold in jars. FoodCo's traditional customers are major UK supermarket chains. Some lines (e.g. washed salads) sell to all customers, while others (most of the cooked products) are produced for a specific supermarket chain. Most are packaged under the supermarket's 'own brand' label.

The pickle started a company tradition that, as far as possible, ingredients are grown on the company's own farm. This now covers 1,500 acres (607 hectares) and includes a market garden growing tomatoes, peppers, courgettes, chillies and other exotic vegetables under glass, and an extensive herb garden. Ingredients that do not grow well in the UK climate are supplied from carefully selected farms abroad, in Mediterranean Europe, East Africa, the USA and the Far East.

The company's annual turnover and employee numbers are summarized in Fig. B1.1.

There are now three factories on the site. Beechfield is the oldest, and this is where raw vegetables are prepared. This work is relatively unskilled. The newer Coppice and Watermead factories concentrate on the more complex cooking processes involved in making sauces, pickles and the like. These need more skilled

FoodCo Limited	
Number of employees (actual)	324
Number of employees (full-time equivalent)	223
Annual turnover (current projection for 2004/2005)	£7.5m

Figure B1.1 FoodCo's current staff complement and annual turnover.

and experienced staff. A bottling plant is also located in Watermead, and there are two warehouses in separate buildings. One is refrigerated and stores fresh vegetable and salad packs, while the other stores dry and bottled products. Figure B1.2 shows a recent organization structure chart.

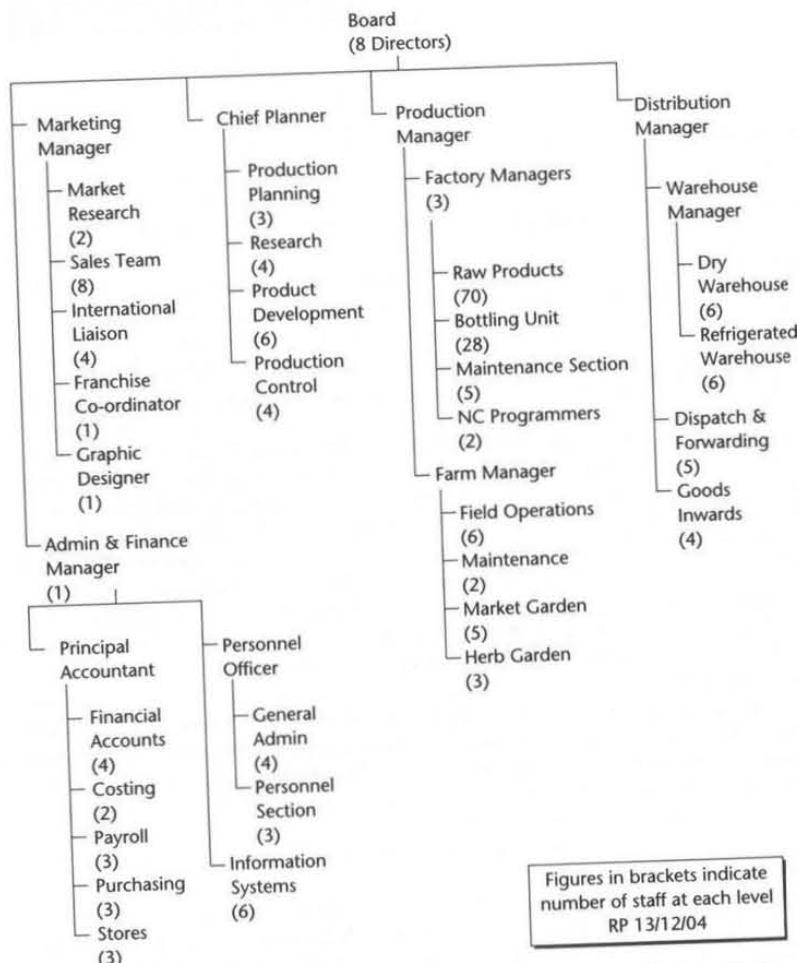


Figure B1.2 FoodCo's current organization structure, showing the distribution of staff.

Box B1.1 Scene: the Board Room, May 1996. John Evans speaks to the Directors about FoodCo's problems and a possible solution

'Fellow directors, we all know the company faces great difficulties. This year's profits will be the lowest ever. If we don't do something drastic, we'll be bankrupt in another year. But if we are to turn the situation round, we must know why things got so bad.'

'I believe the reason goes right back to our beginnings. Those of you who were with me in 1967, when I started this company, will remember we had a degree of control that seems incredible now. Everything ran the way we wanted: farm, production, sales, distribution. We made consistently high quality goods, and by 1972 the new supermarkets were clamouring to buy. That was all a long time ago, but I think our early success is a direct cause of our present predicament. Let me explain.'

'Remember 1978? When we borrowed heavily to finance expansion to meet that demand? Those loan repayments squeezed our profits hard. And then in 1984? When the TrustMart chain emerged as our dominant customer, and began driving down prices? We simply hadn't the financial muscle to fight them. We were still paying off a huge loan! That's why for ten years or more, TrustMart has dictated prices that are so low they have crippled us. And we've been unable to do a thing, because we've simply been scared they'll buy elsewhere. So last year TrustMart bought 65% of our total production—altogether over £4m in sales—and we'll be lucky to clear £200,000 profit on it!'

'That's also why TrustMart calls all the shots on new products. We don't have the cash to develop products for other customers. Now, I know we've grown in spite of that. It's not all been bad, but let's not kid ourselves it's good. We haven't really run the game since 1990. We all know it! We've been towed along behind

TrustMart—and the supermarket sector—like a child dragged along by its father. We've only survived this long because TrustMart had no other source of supply. Now that's changing. We have serious new rivals for the supermarket supply business, and TrustMart has driven our prices still lower, to the point where we may make no profit at all this year.'

'We can beat off this attack, but only if we develop new products and sell in a wider market. There is no argument about that, but there is a problem. Our *real* customers are not the supermarkets, but *their* shoppers. And they don't know we exist, because our name is hidden behind all the TrustMart own-brand labels on all our packs and jars.'

'The answer is to reach the consumers directly. Our market can only expand if they know our name, if they ask for our products. So here's what we do. We're going to launch our own brand name, and promote it so well that everyone knows about us. Customers will begin to insist on our brand, and TrustMart will have to pay *our* price, for a change.'

'It won't be cheap. We'll need serious market research. We'll need more staff in the Product Development team, and we'll need time. We'll need a new corporate image. We'll need TV advertising. But it will be worth it. There's a vast market out there, and I'm not just thinking of the UK.'

'So can we finance it? Certainly! It means heavy borrowing again, but we can repay the loans out of profits on increased sales. Sure, it's a risk, but we'll sink if we don't take it. There are many details to work out, but this plan can succeed. It *will* succeed! When we started, we were the best in the business. I believe we can be the best again.'

'Thank you. Are there any questions?'

The company is still privately owned and managed. In 1993 John made way for his elder son Harold to become Managing Director. At the time, it was generally agreed by the Board that the company must improve profitability but there was no consensus on how to achieve this. In May 1996, John persuaded the Board that FoodCo must bypass its traditional supermarket customers and market directly to

consumers. Partly as a result of John's speech to the Board (Box B1.1 gives an edited version), the 'World Tradition' range was launched in 1997. This now sells successfully at the quality end of the UK market. Helped by the growing reputation of British food and cooking, the range has also begun to penetrate continental European and North American markets.

B1.2 FoodCo Today

B1.2.1 Current thinking

John Evans still believes that the company's major difficulty was over-reliance on one customer, and that this will be solved over time as the 'World Tradition' market expands. His son, Harold, feels that management procedures are now the main problem, particularly management information. He sees the systems as hopelessly inadequate and thinks the company has simply outgrown them. For him, this is an extremely serious issue since it will inevitably worsen as the company grows. But father and son are each as stubborn as the other, so they have never settled their differences on this vital point.

Late in 1995 a new Finance Director, Clare Smythe, was appointed. Less than one year later she achieved a compromise that averted open war in the family. First, she championed the 'World Tradition' brand that successfully met John's concerns. This is a range of international condiments, prepared to traditional recipes from many cultures and using only the finest ingredients. Growing numbers of people in the affluent world want to be able to prepare authentic dishes from world cuisine, ranging from aloo brinjal (Indian potato and eggplant curry) to Yucatan-style cod (Mexican fish cooked in orange, lime and coriander). The new range allowed the company to reposition itself in a new international market, where growth has been highly profitable. It also helps FoodCo to free itself from dependence on TrustMart, still by far their largest customer.

Second, Clare recently helped Harold to persuade the Board that the introduction of a new product range compelled the company to manage its information more effectively and efficiently. The Board agreed to undertake a major review and updating of all information systems, and a national firm of consultants was commissioned to recommend a strategy.

B1.2.2 Information systems

The current systems are a mixed set of applications, some dating back to the late 1960s, that run on diverse hardware platforms. An ageing mini-computer runs an inflexible suite of accounting programmes, a sales order processing system and a stock control system. The stock control system also generates product barcodes for the jar and bottle labels. The mini-computer is accessible from VT100 character-only terminals dotted throughout the factories and offices. Payroll is run off-site on a local computer bureau's server.

Some managers and other staff have networked PCs with standard office software. A handful of proprietary packages includes the Computer Aided Design programme used to design production line layouts. The PC network is not linked to the mini-computer or to the outside world for Internet, email or Electronic Data Interchange.

Some production is automated, including some washing and chopping operations in the Beechfield factory. The automated machines are of the numerical-control type and are now obsolescent. Although they still do a reasonable job, by modern standards they are awkward to re-program, and maintaining them requires particular specialist skills.

The consultants' report

After some months of investigation the consultants submitted their report early in 2005. This identified serious failings in a number of areas and recommended a phased approach to change. The top priority was to develop new product costing and production planning systems that would interface to a new in-house payroll package. The improved product costing and production information would give tighter control of production costs. Price negotiations with customers could be conducted on a more realistic basis, and better management information would help managers respond to the volatile international market for 'World Tradition' products.

The second main recommendation in the report was for a substantial investment in upgraded hardware. Many more PCs were to be installed, networked to each other and to the mini-computer. These needed to be in place prior to phase 2 of the information systems plan, which called for a rapid spread of automation through the production and distribution departments. Finally, all new software development was to fit in with a medium-term plan to make the most of new technology opportunities. For this reason, a move to an object-oriented development method was seen as a critical aspect. This would help later with building an integrated set of systems, ultimately to include factory automation, management information, electronic links with suppliers and customers and an exploration of online Internet marketing. The Board accepted all the recommendations, and a detailed investigation into the requirements for the first systems was begun by FoodCo's in-house IS team, who had recently been trained in object-oriented development methods. Two staff were seconded from the consultants to act as mentors to the first two or three projects.

Product costing: current operations

This section describes the way that product costing activities are currently carried out at FoodCo. It concentrates particularly on the Beechfield factory, as this was an area identified by the consultants' report as a priority for action. Further information is given within chapters, where necessary, as part of case study exercises or review questions. Some information that is not strictly necessary to the completion of the exercises in this book has been included, in order to give a broader view of the overall operations.

Line operations. The nature of production control varies between the various factory and farm departments, depending on the operations undertaken and the nature of the product. At Beechfield, the main products are packs of washed salads and prepared raw vegetables, and some uncooked products such as coleslaw and Waldorf salad. There are three production lines. Each can be adapted to produce different products as the need arises, but only one at a time. Operatives' pay and the overall production costs for these lines are based on the entire batch produced during a single run, which often, although not always, equates to a single eight-hour shift. The line is switched on at the beginning of the run and temporarily halted for coffee breaks and lunch, or when a problem occurs. When a line is switched to a different product, this is treated as a separate run. If operatives are

required to wait while the line is changed over to another product, or while a problem with the line is sorted out, they are paid a standing rate to compensate them for lost earnings.

Payroll and costing. For workers on the older lines at Beechfield, earnings are calculated using an algorithm that has as its input the following variables: the piecework rate for each item, the quantity of that item produced, the number of productive hours spent on the line by each employee and the employee's grade. For each run, the line supervisor completes a *daily production record sheet* (see Fig. B1.3). These are sent to the costing section for analysis before being passed on to the payroll section.

The supervisors also complete a *weekly timesheet* (see, Fig. B1.4) for each employee. These are passed direct to the payroll section. Each Tuesday, the entire week's production record sheets and timesheets are batched up and sent to the computer bureau. Data from the production sheets and timesheets is input to a piecework system at the bureau to produce a weekly earnings figure for the payroll calculation. After the payroll has been run, all paperwork is returned to FoodCo's costing section for analysis. In practice, however, only a limited amount of analysis is carried out.

Some parts of the overall product costing function are outside the scope of this initial project and will be included either in a later increment or possibly in phase 2 of the plan. These are the Coppice and Watermead factories, where the problems with product costing are not as significant as at Beechfield, and Home Farm, where the operations are very different in nature.

Problems in product costing

The mini-computer accounting system includes a product costing module. This meets only some of the information needs of the Finance Director and very few of the requests of any other managers, least of all the factory and farm managers who

Daily Production Record Sheet No:		
Line:	Supervisor:	Date:
Product:	Job No:	Circle Day: M T W T F S S
Run Start:	Run Finish:	
Problem Report		When & where problem occurred:
Brief Description:		
Fault Log No:		Downtime (H/m):
		(Only if applicable)
Signed:		Production Control use only
(Supervisor)		Total Qty Produced:
		Checked by:
FR5/1.3		

Figure B1.3 Daily production record sheet for Beechfield factory

Figure B1.4 The timesheet used in Beechfield factory.

have direct control of most operations. Since the existing product costing system cannot answer most of the queries that are put to it, the costing clerks attempt to provide additional reports by using a spreadsheet application. But the sheer volume of data available for input each week is impossible to process accurately for all products. As a result of ongoing staff shortages in the office, it has only usually been possible to produce actual costs for one production line each week. Making the best of a bad job, each line is costed accurately every fifth week and estimates produced for the other four weeks in between. As a result, the 'actual' costs quoted in management reports are often really no more than estimates derived from samples of the data available.

Both Harold Evans and Clare Smythe are convinced that their inability to get accurate costs is a major contributory factor in the company's decline in profitability. In effect, it means that senior management cannot say with confidence which operations are profitable and which are not. Until they have better information, they cannot even tell where their real problems are.

B1.3 | The Proposal

The first system proposed for development is one to automate product costing. In Box B1.2, some staff from FoodCo's IS team are heard during a meeting early in the project. Louise Lee is the project manager, Ken Ong and Rosanne Martel are both analyst/developers on the team and Chris Pelagrin is a consultant in object-oriented development (he works for the consultancy firm that produced the IS report).

Box B1.2 Scene: FoodCo's IS Team Meeting Room

Louise Lee: I'll begin by welcoming Chris Pelagrin. Chris is a consultant on object-oriented development, and he will be working closely with us on this project.

Chris Pelagrin: Thank you, Louise. Yes, that's right, I've been seconded to you for the duration. Provided, that is, we complete in six months (laughs).

LL: Don't worry, we can hit that deadline. OK, let's get started. Today I mainly want to set the scope for the project. Rosanne, you had a meeting with the Beechfield factory manager, Hari Patel. What did you find out?

Rosanne Martel: Yes, I met Hari on Thursday. He's the principal client for this system, and he'll be a user too. He confirmed the reasons why we picked this as our first object-oriented project. It's strategically important, but not so critical that the whole company will fold if we mess up. It's tightly scoped, and only really affects operations in Beechfield and the costing office. But it does need access to a lot of data currently on the mini-computer and it's a feeder system for payroll and production planning. If we develop a system that integrates with these, we'll have a sound basis for re-engineering the entire IS provision.

LL: Good. This confirms the consultants' report too. Did you get any idea of the main functionality of the system? We'll need this to estimate timescales with any confidence.

RM: Ken, you've done some work on this. How far did you get?

Ken Ong: Well, it's too early to be precise, but I've sketched out some use cases and a rough class diagram. Users include Hari, his line

supervisors, the sales team, production control and the costing office. The main system inputs are staff timesheets and production record sheets, and data we can import from payroll records and the existing costing system. The main system outputs will be production run costs. One obvious problem is that we don't hold any payroll data electronically, so we'll need access to the bureau's files at some point. I would say that as a whole it is not highly complex. My first class diagram has about a dozen classes. There are a few interactions that involve most classes—for example, producing the final cost for a production line run—but most are simpler.

LL: So this is a fairly small system with relatively few users, but lots of interfaces to other systems. Can you show us some of this on the whiteboard?

KO: Yes, of course. Just give me a few minutes (goes to whiteboard and starts to draw).

LL: (while Ken draws) What do you think so far, Chris? Perhaps you could say a little about how you see your role.

CP: My task is to help you apply to this project the object-oriented techniques that you have all learned on the training courses. You all know there is a big difference between knowing the techniques and understanding how they fit together. I'm here to help when you're unsure about anything. Rosanne's summary suggests this project is an ideal start and I'm confident we will make it a complete success.

LL: That's great, Chris, coming from the expert. OK Ken, now let's see your diagrams.

Modelling Concepts

Learning Objectives

In this chapter you will learn

- what is meant by a model
- the distinction between a model and a diagram
- the UML concept of a model
- how to draw *activity diagrams* to model processes
- the approach to system development that we have adopted in this book.

5.1 | Introduction

Systems analysts and designers produce models of systems. A business analyst will start by producing a model of how an organization works; a systems analyst will produce a more abstract model of the objects in that business and how they interact with one another; a designer will produce a model of how a new computerized system will work within the organization. In Section 5.2 we explain what is meant by a model and the relationship between models and diagrams. In UML, the term 'model' has a specific meaning, and we explain the UML concept of a model and how it relates to other UML concepts, such as the idea of a package.

The best way to understand what we mean by a model is to look at an example. In the Unified Process (the method of developing systems that is promoted by the developers of UML) activity diagrams are used to model the development process itself. We introduce the basic notation of activity diagrams in Section 5.3. Activity diagrams are one of the techniques that can be used to model the dynamic view of a system, and their use in systems analysis and design is explained in more detail in Chapter 10, where they are used as one way of specifying operations. Finally, in Section 5.4 we give an overview of the approach to system development that we have adopted in this book. We should stress that we are not presenting this development approach as a method in its own right, but only introducing it as a way of providing some structure to the process that we follow in the book.

5.2 Models and Diagrams

In any development project that aims at producing useful artefacts, the main focus of both analysis and design activities is on models (although the ultimate objective is a working system). This is equally true for projects to build highways, space shuttles, television sets or software systems. Aircraft designers build wooden or metal scale models of new aircraft to test their characteristics in a wind tunnel. A skilled furniture designer may use a mental model, visualizing a new piece of furniture without drawing a single line.

In IS development, models are usually both abstract and visible. On the one hand, many of the products are themselves abstract in nature; most software is not tangible for the user. On the other hand, software is usually constructed by teams of people who need to see each other's models. However, even in the case of a single developer working alone, it is still advisable to construct visible models. Software development is a complex activity, and it is extremely difficult to carry all the necessary details in one person's memory.

5.2.1 What is a model?

Like any map, models represent something else. They are useful in several different ways, precisely because they differ from the things that they represent.

- A model is quicker and easier to build.
- A model can be used in simulations, to learn more about the thing it represents.
- A model can evolve as we learn more about a task or problem.
- We can choose which details to represent in a model, and which to ignore.
- A model is an abstraction.
- A model can represent real or imaginary things from any domain.

Many different kinds of thing can be modelled. Civil engineers model bridges, city planners model traffic flow, economists model the effects of government policy and composers model their music. This book is a model of the activity of object-oriented analysis and design.

A useful model has just the right amount of detail and structure, and represents only what is important for the task at hand. This point was not well understood by at least one character in *The Restaurant at the End of the Universe* by Douglas Adams (1980). A group of space colonists are trying to re-invent things they need after crash-landing on a strange planet, and are unable to proceed with a project to design the wheel, because they cannot come to an agreement on what colour it should be.

Real projects do get bogged down in this kind of unnecessary detail if insufficient care is taken to exclude irrelevant considerations (though this example is a little extreme). What IS developers must usually model is a complex situation, frequently within a human activity system. We may need to model what different stakeholders think about the situation, so our models need to be rich in meaning. We must represent functional and non-functional requirements (see Section 6.2.2). The whole requirements model must be accurate, complete and unambiguous. Without this, the work of designers and programmers later in the project would be much

more difficult. At the same time, it must not include premature decisions about how the new system is going to fulfil its users' requests, otherwise designers and programmers may later find their freedom of action too restricted. Most systems development models today are in the form of diagrams, with supporting textual descriptions and logical or mathematical specifications of processes and data¹.

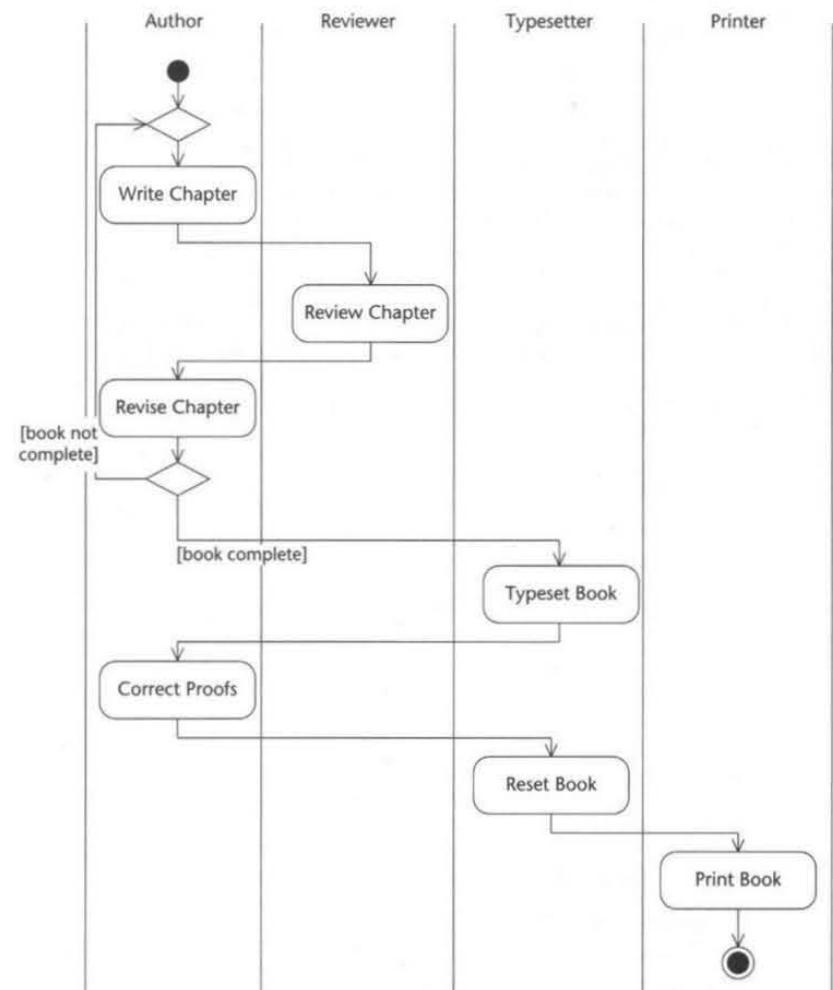


Figure 5.1 Activity diagram for producing a book.

¹ Some approaches rely primarily on formal logic techniques and rigorous mathematical specification. These are most often applied to real-time and safety-critical systems, such as those that control aircraft in flight or manage nuclear power plants, and are not covered in this book.

5.2.2 What is a diagram?

Analysts and designers use diagrams to build models of systems in the same way as architects use drawings and diagrams to model buildings. Diagrammatic models are used extensively by systems analysts and designers in order to:

- communicate ideas
- generate new ideas and possibilities
- test ideas and make predictions
- understand structures and relationships.

The models may be of existing business systems or they may be of new computerized systems. If a system is very simple, it may be possible to model it with a single diagram and supporting textual descriptions. Most systems are more complex and may require many diagrams fully to capture that complexity.

Figure 5.1 shows an example of a diagram (a UML activity diagram) used to show part of the process of producing a book. This diagram alone is not a complete model. A model of book production would include other activity diagrams to show other parts of the overall system such as negotiating contracts and marketing the book. This diagram does not even show all the detail of the activities carried out by authors and the other participants in the process. Many of the activities, shown as rectangles with rounded corners in Fig. 5.1, could be expanded into more detail. For example, the activity Write Chapter could be broken down into other activities such as those shown in Fig. 5.2.



Figure 5.2 Activity diagram for the activity Write Chapter.

We might break some of the activities shown in Fig. 5.2 down into more detail, though it will be difficult to show the detail at a lower level, as activities like Write a Paragraph, Add a Figure, Revise a Paragraph and Move a Figure do not lend themselves to being represented in the flowchart notation of the activity diagram. There is also a limit to what we want to show in such a diagram. There are many activities such as Make Coffee, Change CD and Stare out of Window that are part of the process of writing, but like the colour of the wheel in the example from *The Restaurant at the End of the Universe*, they represent unnecessary detail.

The diagrams of Figs 5.1 and 5.2 are typical of the kind of diagrams used in systems analysis and design. Abstract shapes are used to represent things or actions from the real world. The choice of what shapes to use is determined by a set of rules that are laid down for the particular type of diagram. In UML, these rules are laid down in the *OMG Unified Modeling Language Specification 2.0* (OMG, 2004c). It is important that we follow the rules about diagrams, otherwise the diagrams may not make sense, or other people may not understand them. Standards are important as they promote communication in the same way as a common language. They enable communication between members of the development team if they all document the information in the same standard formats. They promote communication over time, as other people come to work on the system, even several years after it has been implemented, in order to carry out maintenance. They also promote communication of good practice, as experience of what should be recorded and how best to do that recording builds up over time and is reflected in the techniques that are used.

Modelling techniques are refined and evolve over time. The diagrams and how they map to things in the real world or in a new system change as users gain experience of how well they work. However, for the designers of modelling techniques, some general rules are that the techniques should aid (and enforce):

- simplicity of representation—only showing what needs to be shown
- internal consistency—within a set of diagrams
- completeness—showing all that needs to be shown
- hierarchical representation—breaking the system down and showing more detail at lower levels.

Figure 5.3 shows some symbols from a label in an item of clothing. These icons belong to a standard that allows a manufacturer of clothing in Argentina to convey to a purchaser in Sweden that the item should be washed at no more than 40 °C, should not be bleached and can be tumble dried on a low setting.

While not following the UML standards will not cause your T-shirts to shrink, it will cause you problems in communicating with other analysts and designers—at

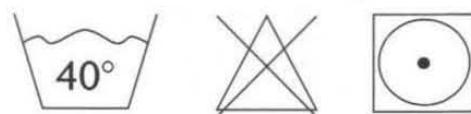


Figure 5.3 Example of a diagram standard.

least if they are using UML as well. We have chosen to use UML in this book, as it has become the industry standard for modelling information systems.

UML consists mainly of a graphical language to represent the concepts that we require in the development of an object-oriented information system. UML diagrams are made up of four elements:

- icons
- two-dimensional symbols
- paths
- strings.

These terms were used in the 1.X specifications, and are no longer used in UML 2.0. However, they are useful to explain the graphical representation of UML diagrams.

UML diagrams are *graphs*—composed of various kinds of shapes, known as *nodes*, joined together by lines, known as *paths*. The activity diagrams in Figs 5.1 and 5.2 illustrate this. Both are made up of two-dimensional symbols that represent activities, linked by arrows that represent the control flows from one activity to another and the flow of control through the process that is being modelled. The start and finish of each activity graph is marked by special symbols—icons: the dot for the initial node and the dot in a circle for the final node. The activities are labelled with strings, and strings are also used at the decision nodes (the diamond shapes) to show the conditions that are being tested.

The UML Specification (OMG, 2004c) provides the formal grammar of UML—the syntax—and the meaning of the elements and of the rules about how elements can be combined—the semantics. It also explains the different diagrams in more detail and provides examples of their construction and use (although with fewer examples than previous versions). Box 5.1 explains with the aid of an example how the UML specification defines the syntax and semantics of UML. It may be difficult to follow at this stage in your understanding of UML, so feel free to skip it and come back to it when you know more about UML.

Box 5.1 The UML metamodel

The metaclass diagrams use the notation of UML class diagrams (see Chapter 7) to define the *metamodel*—the model of how the elements of UML fit together. Rules in a special formal language called Object Constraint Language (OCL) formally define constraints on how these elements can be used. The textual part of the UML Specification explains the meaning of the elements and how they work in English text.

As an example, Fig. 5.4 shows a simplified extract from the diagrams in the UML Semantics section that define the rules about activity diagrams.

The diagram shows us that an ActivityFinalNode is a subtype of FinalNode, which is a subtype of ControlNode, which is a subtype of ActivityNode, and that ActivityNode has two associations with ActivityEdge. We can tell from this that each ActivityEdge goes from one and only one ActivityNode (its source) to another (its target). However, each ActivityNode can have many (depicted by the asterisks) incoming and outgoing edges.

Because FinalNode is a subtype of ActivityNode via ControlNode, it appears that it can also have many incoming and

Box 5.1 The UML metamodel (continued)

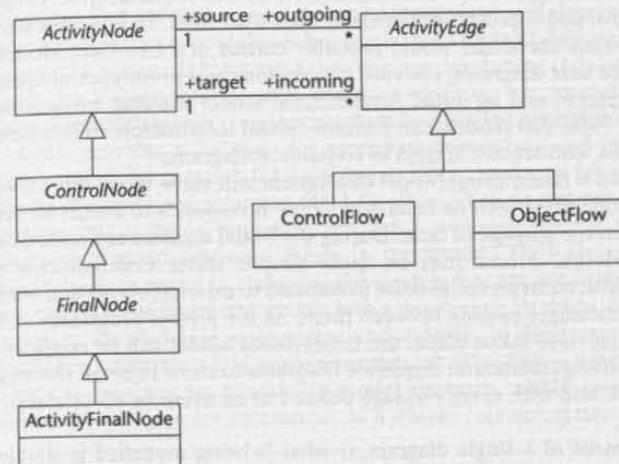


Figure 5.4 Extract from UML metamodel for activity diagrams.

outgoing edges. However, it does not make any sense for a final node in an activity diagram to have any outgoing edges. If it has outgoing edges, then there must be a node after it, so it is not a final node. OCL can be used to add a constraint to the metaclass FinalNode that stops it from having outgoing transitions. This is written as follows.

```

context FinalNode inv:
  self.outgoing->size = 0
  
```

This means that this OCL statement is an *invariant* (something that cannot be changed) of FinalNode, and says that the size of the set of ActivityEdges linked to any FinalNode by the outgoing association must be zero. That is, it cannot have any outgoing transitions.

This is explained in text in the UML Specification as follows.

FinalNode

A final node is an abstract control node at which a flow in an activity stops.

...

Constraints

[1] A final node has no outgoing edges.
(OMG, 2004b, Section 12.3.21)

You do not need to worry about the implications of the first part of this definition, but if you want to understand it, take a look at Figs 5.1 and 5.2 again. All the activities in Fig. 5.2 take place within the Write a Chapter activity of Fig. 5.1. When the activities of Fig. 5.2 have been completed, then the enclosing activity, Write a Chapter, has been completed. At the end of Fig. 5.1 there is a final state icon and, since this activity chart is not enclosed within some other activity, it means that the entire activity diagram has completed.

5.2.3 The difference between a model and a diagram

We have seen an example of a diagram in the previous section. A single diagram can illustrate or document some aspect of a system. However, a model provides a complete view of a system at a particular stage and from a particular perspective.

For example, a Requirements Model of a system will give a complete view of the requirements for that system. It may use one or more type of diagram and will most likely contain sets of diagrams to cover all aspects of the requirements. These diagrams may be grouped together in models in their own right. In a project that uses UML, a Requirements Model would probably consist of a Use Case Model, which comprises use case diagrams, use case descriptions and prototypes of some use cases (see Chapter 6) and an initial Architectural Model showing subsystems (see Section 5.2.4). Note that models can contain textual information, in this case use case descriptions, and are not limited to containing diagrams.

On the other hand a Behavioural Model of a system will show those aspects of a system that are concerned with its behaviour—how it responds to events in the outside world and to the passage of time. During the initial analysis activities of a project, the Behavioural Model may be quite simple, using Communication Diagrams to show which classes collaborate to respond to external events and with informally defined messages passing between them. As the project progresses and more design activities have taken place, the Behavioural Model will be considerably more detailed, using Interaction Sequence Diagrams to show in detail the way that objects interact, and with every message defined as an event or an operation of a class.

A model may consist of a single diagram, if what is being modelled is simple enough to be modelled in that way, but most models consist of many diagrams—related to one another in some way—and supporting data and textual documentation. Most models consist of many diagrams because it is necessary to simplify complex systems to a level that people can understand and take in. For example, the class libraries for Java are made up of hundreds of classes, but books that present information about these classes rarely show more than about twenty on any one diagram, and each diagram groups together classes that are conceptually related.

5.2.4 Models in UML

The UML 2.0 Superstructure Specification (OMG, 2004b) defines a model as follows:

A model captures a view of a physical system. It is an abstraction of the physical system, with a certain purpose. This purpose determines what is to be included in the model and what is irrelevant. Thus the model completely describes those aspects of the physical system that are relevant to the purpose of the model, at the relevant level of detail.

In UML there are a number of concepts that are used to describe systems and the ways in which they can be broken down and modelled. A *system* is the overall thing that is being modelled, such as the Agate system for dealing with clients and their advertising campaigns. A *subsystem* is a part of a system, consisting of related elements, for example the Campaigns subsystem of the Agate system. A *model* is an abstraction of a system or subsystem from a particular perspective or *view*. An example would be the use case view of the Campaigns subsystem, which would be represented by a model containing use case diagrams, among other things. A model is complete and consistent at the level of abstraction that has been chosen. Different views of a system can be presented in different models, and a *diagram* is a graphical representation of a set of elements in the model of the system.

Different models present different views of the system. Booch et al. (1999) suggest five views to be used with UML: the use case view, the design view, the process view, the implementation view and the deployment view. The choice of diagrams that are used to model each of these views will depend on the nature and complexity of the system that is being modelled. Indeed, you may not need all these views of a system. If the system that you are developing runs on a single machine, then the implementation and deployment views are unnecessary, as they are concerned with which components must be installed on which different machines.

UML provides a notation for modelling subsystems and models that uses an extension of the notation for *packages* in UML. Packages are a way of organizing model elements and grouping them together. They do not represent things in the system that is being modelled, but are a convenience for packaging together elements that do represent things in the system. They are used particularly in CASE tools as a way of managing the models that are produced. For example, the use cases can be grouped together into a Use Cases Package. Figure 5.5 shows the notation for packages, subsystems and models. In diagrams we can show how packages, subsystems and models contain other packages, subsystems and models. This can be done by containing model elements within larger ones. Figure 5.6 shows the notation for an example of a system containing two subsystems.

5.2.5 Developing models

The models that we produce during the development of a system change as the project progresses. They change along three main dimensions:

- abstraction
- formality
- level of detail.

During a particular phase of a project we may extend and elaborate a model as we increase our understanding of the system that is to be built. At the end of each

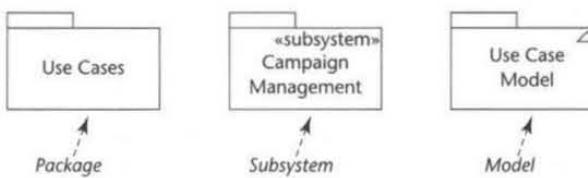


Figure 5.5 UML notation for packages, subsystems and models.

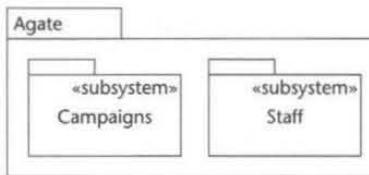


Figure 5.6 UML notation for a system containing subsystems, shown by containment.

phase we hope to have a model that is complete and consistent, within the limitations of that phase of the project. That model represents a view of our understanding of the system at that point in the project.

In a system development project that uses an iterative lifecycle, different models that represent the same view may be developed at different levels of detail as the project progresses. For example, the first use case model of a system may show only the obvious use cases that are apparent from the first iteration of requirements capture. After a second iteration, the use case model may be elaborated with more detail and additional use cases that emerge from discussion of the requirements. Some prototypes may be added to try out ideas about how users will interact with the system. After a third iteration, the model will be extended to include more structured descriptions of how the users will interact with the use cases and with relationships among use cases. (Use cases are explained in Chapter 6.) Figure 5.7 illustrates this process of adding detail to a model through successive iterations. The number of iterations is not set at three. Any phase in a project will consist of a number of iterations, and that number will depend on the complexity of the system being developed.

It is also possible to produce a model that contains a lot of detail, but to hide or suppress some of that detail in order to get a simplified overview of some aspect of the system. For example, class diagrams (explained in Chapter 7) can be shown with the compartments that contain attributes and operations suppressed. This is often useful for showing the structural relationships between classes, using just the name of each class, without the distracting detail of all the attributes and operations. This is the case in the diagrams that show the classes in the Java class

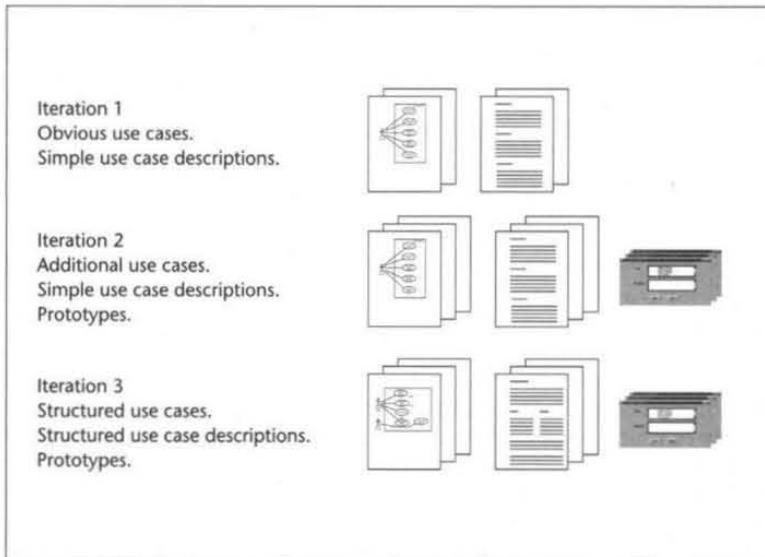


Figure 5.7 Development of the use case model through successive iterations.

libraries (referred to in Section 5.2.3), where the intention is to show structural relationships between classes rather than the detail.

As we progress through analysis and design of a system, elements in the model will become less abstract and more concrete. For example, we may start off with classes that represent the kinds of objects that we find in the business, Campaigns, Clients etc., that are defined in terms of the responsibilities that they have. By the time that we get to the end of design and are ready to implement the classes, we will have a set of more concrete classes with attributes and operations, and the classes from the domain will have been supplemented by additional classes such as collection classes, caches, brokers and proxies that are required to implement mechanisms for storing the domain classes (see Chapter 18).

In the same way, the degree of formality with which operations, attributes and constraints are defined will increase as the project progresses. Initially, classes will have responsibilities that are loosely defined and named in English (or whatever language the project is being developed in). By the time we reach the end of design and are ready to implement the classes, they will have operations defined using Object Constraint Language (see Chapter 10), with pre-conditions and post-conditions for each operation.

This iterative approach, in which models are successively elaborated as the project progresses, has advantages over the waterfall model, but it also has shortcomings. First, it is sometimes difficult to know when to stop elaborating a model and, second, it raises the question of whether to go back and update earlier models with additional information that emerges in later stages of the project. Issues like these are addressed either as part of a methodology (Chapter 21) or as part of a project management approach (see supporting website). For now, we shall look at a first example of a UML diagram and see how it is developed.

5.3 Drawing Activity Diagrams

We have used activity diagrams earlier in this chapter to illustrate what is meant by a diagram. In this section we explain the basic notation of activity diagrams in UML and give examples of how they are used.

5.3.1 Purpose of activity diagrams

Activity diagrams can be used to model different aspects of a system. At a high level, they can be used to model business activities in an existing or potential system. For this purpose they may be used early in the system development lifecycle. They can be used to model a system function represented by a use case, possibly using object flows to show which objects are involved in a use case. This would be done during the phase of the lifecycle when requirements are being elaborated. They can also be used at a low level to model the detail of how a particular operation is carried out, and are likely to be used for this purpose in later analysis or system design activities. Activity diagrams are also used within the Unified Software Development Process (USDP) (Jacobson et al., 1999) to model the way in which the activities of the USDP are organized and relate to one another in the software development lifecycle. We use them for a similar purpose in later chapters to show how the activities of the simplified process that we have

adopted for this book fit together. (This process is described in Section 5.4.)

In summary, activity diagrams are used for the following purposes:

- to model a process or task (in business modelling for instance)
- to describe a system function that is represented by a use case
- in operation specifications, to describe the logic of an operation
- in USDP to model the activities that make up the lifecycle.

Fashions change in systems analysis and design—new approaches such as structured analysis and design and object-oriented analysis and design replace old approaches and introduce new diagrams and notation. One diagram type that is always dismissed by the inventors of new approaches but always creeps back in again is the flowchart². Activity diagrams are essentially flowcharts in an object-oriented context.

Activity diagrams are really most useful to model business activities in the early stages of a project. For modelling operations, interaction sequence diagrams are closer to the spirit of object-orientation. However, there may be occasions when the analyst wants to model the activities that must be carried out, but has not yet identified the objects or classes that are involved or assigned responsibilities to them. In such circumstances, activity diagrams may be an appropriate tool to use.

5.3.2 Notation of activity diagrams

Activity diagrams at their simplest consist of a set of actions linked together by flows from one activity to the next, formally called *ActivityEdges*. Each action is shown as a rectangle with rounded corners. The name of the action is written inside this two-dimensional symbol. It should be meaningful and summarize the action. Figure 5.8 shows an example of two actions joined by a flow.

Actions exist to carry out some task. In the example of Fig. 5.9, the first action is to add a new client into the Agate system described in Chapter A1. The flow to the second action implies that as soon as the first action is complete, the next action is started. Sometimes there is more than one possible flow from an action to the next.

In this example from the Agate system, the flow of work is summarized by this brief statement from an interview with one of the directors of Agate.



Figure 5.8 Example of two activities joined by a control flow.

² Flowcharts are useful because they model the way that people perform tasks as a sequence of actions with decision points where they take one of a set of alternative paths, in which some actions are repeated either a number of times or until some condition is true and some actions take place in parallel. In UML 2.0 activity diagrams have the semantics of Petri Nets.

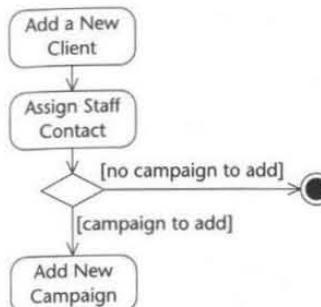


Figure 5.9 Activities with a decision node.

When we add a new client, we always assign a member of staff as a contact for the client straightaway. If it's an important client, then that person is likely to be one of our directors or a senior member of staff. The normal reason for adding a new client is because we have agreed a campaign with them, so we then add details of the new campaign. But that's not always the case—sometimes we add a client before the details of the campaign have been firmed up, so in that case, once we have added the client the task is complete. If we are adding the campaign, then we would record its details, and if we know which members of staff will be working on the campaign, we would assign each of them to work on the campaign.

This transcript from an interview describes some choices that can be made, and these choices will affect the actions that are undertaken. We can show these in an activity diagram with an explicit *decision node*, represented by a diamond-shaped icon, as in Fig. 5.9.

In UML 1.X, it was not necessary to use an explicit decision node like this. The diagram could just show the alternative flows out of the action *Assign Staff Contact*, as in Fig. 5.10.

However, this is no longer possible in UML 2.0. In UML 1.X, if there was more than one flow out of an action, it was treated as an OR, i.e. only one flow would be taken. In UML 2.0, it is treated as an AND, i.e. all of the flows must be taken.

The alternative flows are each labelled with a *guard condition*. The guard condition is shown inside square brackets and must evaluate to either true or false.

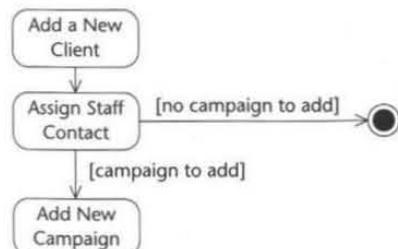


Figure 5.10 UML 1.X choice represented without an explicit decision point.

Alternative guard conditions from a single decision node do not have to be mutually exclusive, but if they are not, you should specify the order of evaluation in some way, otherwise the results will be unpredictable. We would recommend that they should be mutually exclusive.

Figures 5.9 and 5.10 illustrate another element of the notation of activity diagrams: when an activity has completed that ends the sequence of activities within a particular diagram, there must be an *activity edge* or *control flow* to a *final node*, shown as a black circle within a white circle with a black border. Each activity diagram should also begin with another special icon, a black circle, which represents the start of the activity. Figure 5.11 shows the addition of the *initial node* into the diagram of Fig. 5.9. It also shows an additional action—to assign a member of staff to work on a campaign—and additional guarded flows.

Note that there is a loop or iteration created at the bottom of this diagram, where the activity *Assign Staff to Campaign* is repeated until there are no more staff to assign to this particular campaign.

Activity diagrams make it possible to represent the three structural components of all procedural programming languages: sequences, selections and iterations. This ability to model processes in this way is particularly useful for modelling business procedures, but can also be helpful in modelling the operations of classes.

In an object-oriented system, however, the focus is on objects carrying out the processing necessary for the overall system to achieve its objectives. There are two ways in which objects can be shown in activity diagrams:

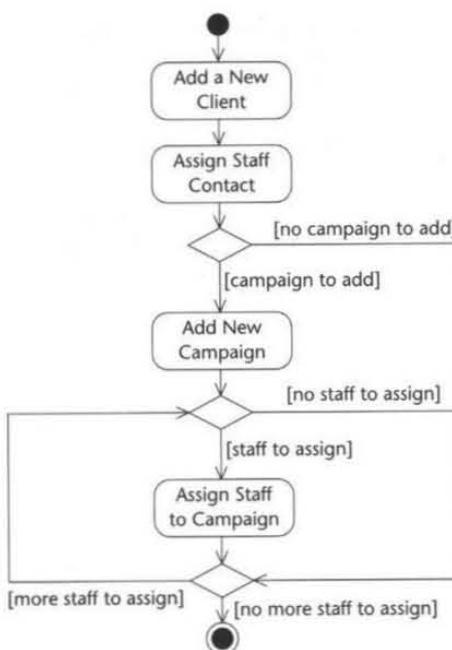


Figure 5.11 Activity diagram with initial node.

- the operation name and class name can be used as the name of an action
- an object can be shown as providing the input to or output of an action.

Figure 5.12 shows an example of the first of these uses of objects in activity diagrams. In this example, the total cost of a campaign is calculated from the cost of all the individual adverts in the campaign added to the campaign overheads. The names of the classes involved are shown followed by double colons in brackets beneath the names of the actions. If the name of the action is not the same as the name of an operation of the class, then the operation name can be shown after the colons.

The second way that objects are shown in activity diagrams is by using *object flows*. An object flow is an arrow between an object and an action that results in a change to the state of that object. The state of the object can be shown in square brackets within the symbol for the object. Figure 5.13 shows an example of this for the activity *Record Completion* of a *Campaign*, which changes the state of a *Campaign* object from *Active* to *Completed*. (Objects and classes are covered in much more detail in Chapters 7 and 8, and the idea of ‘state’ is covered in more detail in Chapter 11, where we explain state machine diagrams.)

A final element of the notation of activity diagrams that it is useful to understand at this stage is the idea of *activity partitions*, which were called *swimlanes* in UML 1.X and are generally known by this name. Activity partitions are particularly useful when modelling how things happen in an existing system and can be used to show where actions take place or who carries out the actions.

In the Agate system, when an advertising campaign is completed, the campaign manager for that advertising campaign records that it is completed. This triggers off the sending of a record of completion form to the company accountant. An invoice is then sent to the client and, when the client pays the invoice, the payment is recorded. (Some of these actions are documented as use cases in Figs A1.2 and A1.3.)

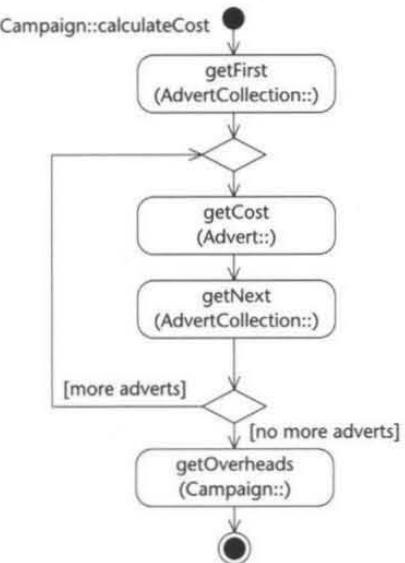


Figure 5.12 Activity diagram with operations of classes as actions.

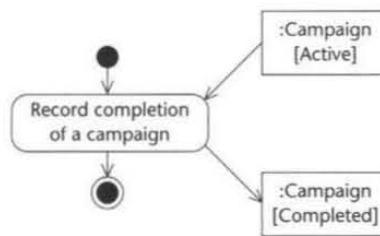


Figure 5.13 Activity diagram with object flows.

In order to model the way that the system works at the moment, we might draw an activity diagram like the one in Fig. 5.14 in order to show these actions taking place. The brief for this project is to concern ourselves with the campaign management side of the business, as there is an existing accounts system in the company. However, the act of drawing this diagram raises the question of what happens to the payment from the client.

- Does the payment go to the accountant, and is there some way in which the campaign manager is notified?
- Does the payment go to the campaign manager, and does he or she record the payment and then pass it on to the accountant?

Clarifying points like these is part of the process of requirements capture, which is covered in detail in Chapter 6.

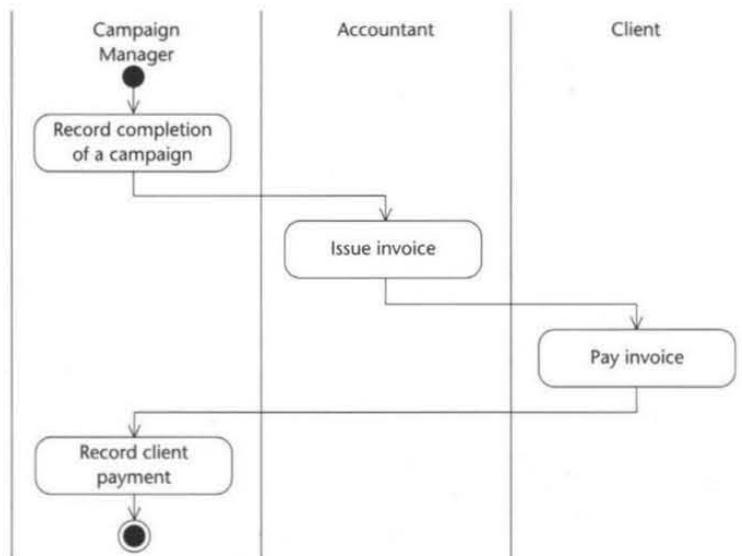


Figure 5.14 Activity diagram with activity partitions.

One of the reasons for introducing activity diagrams at this point is that they are used in the Unified Software Development Process to document the activities of the software development lifecycle. In USDP, the diagrams are *stereotyped*—the standard UML symbols are replaced with special icons to represent actions and the inputs and outputs of those actions. In the next section, we describe the simplified process model that we have adopted in this book. We use activity diagrams to summarize this process in the case study chapters later in the book.

5.4 A Development Process

A development process should specify what has to be done, when it has to be done, how it should be done and by whom in order to achieve the required goal. Project management techniques (see the supporting website) are used to manage and control the process for individual projects. One of the software development processes currently in wide use is the Rational Unified Process, a proprietary process now owned by IBM but based on the Unified Software Development Process (USDP) (Jacobson et al., 1999). The USDP was originally developed by the team that created UML. It is claimed that the USDP embodies much of the currently accepted best practices in information systems development. These include:

- iterative and incremental development
- component-based development
- requirements driven development
- configurability
- architecture centred
- visual modelling techniques.

The USDP is explained in more detail in Chapter 21 on System Development Methodologies. USDP is often referred to as the *Unified Process*.

The USDP does not follow the Traditional Lifecycle shown in Fig. 3.2 but adopts an iterative approach within four main *phases*. These phases reflect the different emphasis on tasks that are necessary as systems development proceeds (Fig. 5.15). These differences are captured in a series of *workflows* that run through the development process. Each workflow defines a series of activities that are to be carried out as part of the workflow and specifies the roles of the people who will carry out those activities. The important fact to bear in mind is that in the waterfall lifecycle activities and phases are one and the same, while in iterative lifecycles like the USDP the activities are independent of the phases and it is the mix of activities that changes as the project proceeds. Figure 5.16 illustrates how a simplified waterfall lifecycle would look using the same style of diagram as Fig. 5.15.

5.4.1 Underlying principles

In order to place the techniques and models described in this book in context we have assumed an underlying system development process. We are not attempting to invent yet another methodology. The main activities that we describe here appear in one form or another in most system development methodologies. The

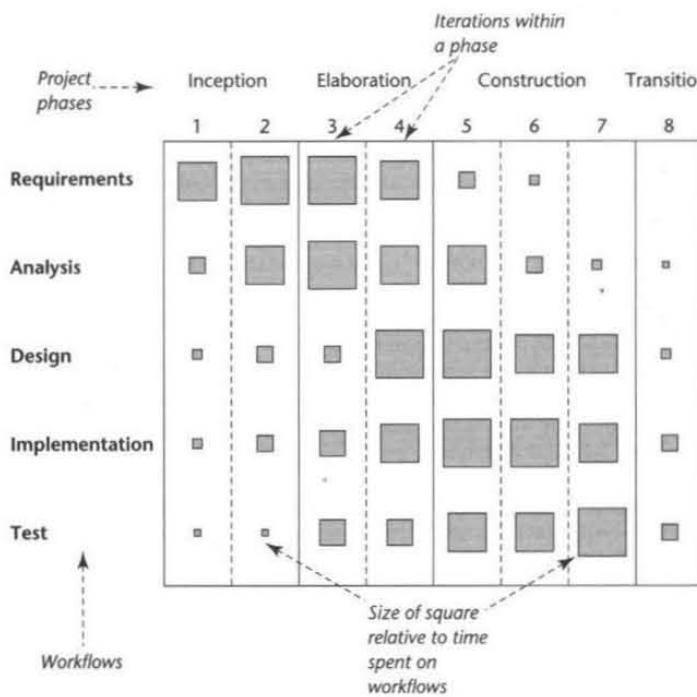


Figure 5.15 Phases and workflows in the Unified Software Development Process.

system development process that we adopt is largely consistent with USDP, although it incorporates ideas from other sources. This approach incorporates the following characteristics. It is:

- iterative
- incremental
- requirements driven
- component-based
- architectural.

These principles are embodied in many commonly used methodologies and are viewed as elements of best practice.

5.4.2 Main activities

The systems development process embodies the following main activities:

- requirements capture and modelling
- requirements analysis
- system design

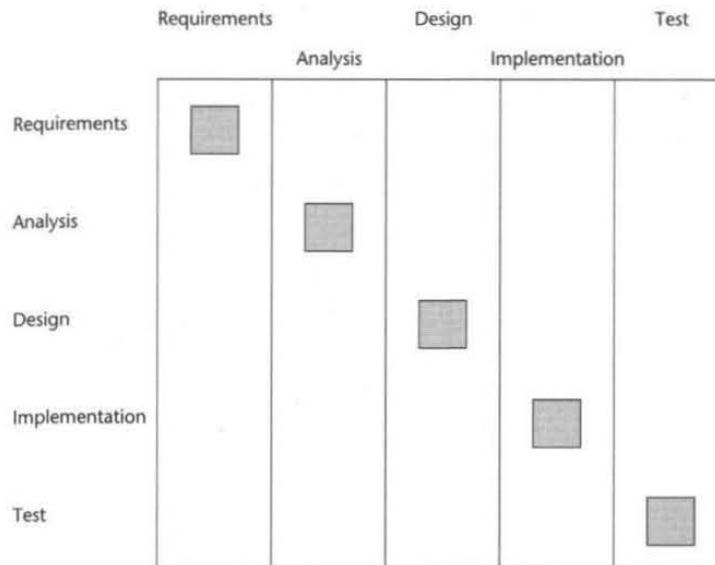


Figure 5.16 Phases and activities in a simplified waterfall process.

- class design
- interface design
- data management design
- construction
- testing
- implementation.

These activities are interrelated and dependent upon each other. In a waterfall development process they would be performed in a sequence (as in Fig. 5.16). This is not the case in an iterative development process, although some activities clearly precede others. For example, at least some requirements capture and modelling must take place before any requirements analysis can be undertaken. Various UML techniques and notations are used, as well as other techniques, and these are summarized in the table in Fig. 5.17.

Only the key deliverables are listed in the table and are likely to be produced in a series of iterations and delivered incrementally. A brief summary of each activity follows. The models that are produced and the activities necessary to produce them are explained in more detail in subsequent chapters.

5.4.2.1 Requirements capture and modelling

Various fact-finding techniques are used to identify requirements. These are discussed in Chapter 6. Requirements are documented in use cases. A use case captures an element of functionality and the requirements model may include

Activity	Techniques	Key Deliverables
Requirements capture and modelling	Requirements elicitation Use case modelling Architectural modelling Prototyping	Use case model Requirements list Initial architecture Prototypes
Requirements analysis	Communication diagrams Class and object modelling Analysis modelling	Analysis models
System design	Deployment modelling Component modelling Package modelling Architectural modelling Design patterns	Overview design and implementation architecture
Class design	Class and object modelling Interaction modelling State modelling Design patterns	Design models
Interface design	Class and object modelling Interaction modelling State modelling Package modelling Prototyping Design patterns	Design models with interface specification
Data management design	Class and object modelling Interaction modelling State modelling Package modelling Design patterns	Design models with database specification
Construction	Programming Component reuse Database DDL Programming idioms Manual writing	Constructed system Documentation
Testing	Programming Test planning and design Testing	Test plans Test cases Tested system
Implementation		Installed system

Figure 5.17 Table of system development process activities.

many use cases. For example, in the Agate case study the requirement that the accountant should be able to record the details of a new member of staff on the system is an example of a use case. It would be described initially as follows:

Use Case: Add a new staff member

When a new member of staff joins Agate, his or her details are recorded. He or she is assigned a staff number, and the start date is recorded. Start date defaults to today's date. The starting grade is recorded.

The use cases can also be modelled graphically. The use case model is refined to identify common procedures and dependencies between use cases. The objective of this refinement is to produce a succinct but complete description of requirements. Not all requirements will be captured in use cases. Some requirements that apply to the whole system will be captured in a list of requirements. Requirements that are concerned with how well the system performs rather than what it does (non-functional requirements) are also captured separately. It is also common to capture rules that reflect how the business works (business rules) in a separate document and cross-reference them from use cases.

Prototypes of some key user interfaces may be produced in order to help to understand the requirements that the users have for the system.

An initial system architecture (see Fig. 5.18 for part of the Agate system) may be developed to help guide subsequent steps during the development process. This initial architecture will be refined and adjusted as the development proceeds.

5.4.2.2 Requirements analysis

Essentially each use case describes one major user requirement. Each use case is analysed separately to identify the objects that are required to support it. The use case is also analysed to determine how these objects interact and what responsibilities each of the objects has in order to support the use case. Communication diagrams (Fig. 5.19) are used to model the object interaction. The models for each use case are then integrated to produce an analysis class diagram, as described in Chapters 7 and 8. Figure 5.20 shows an example of an analysis class. The initial system architecture may be refined as a result of these activities.

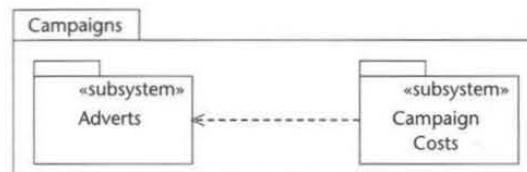


Figure 5.18 Part of the initial system architecture for the Agate system.

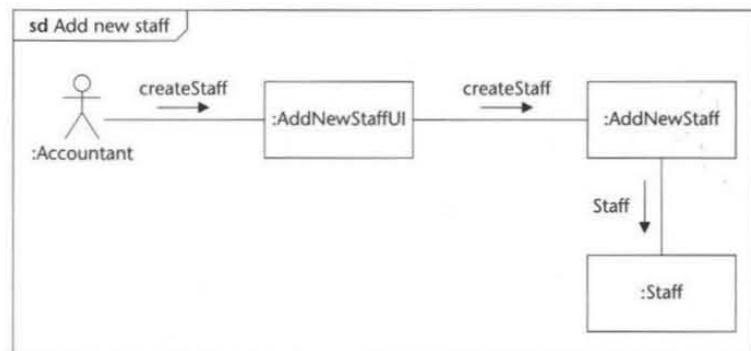


Figure 5.19 Part of a communication diagram for the use case Add New Staff.

Client
companyName
companyAddress
companyTelephone
«responsibilities»
addClient
updateClientDetails
getClientDetails
addNewCampaign
getClientCampaigns

Figure 5.20 Partly completed sample analysis class.

5.4.2.3 System design

At this stage various decisions concerning the design process are made, including the further specification of a suitable systems architecture. For example, a possible architecture for the system in the Agate case study is shown in Fig. 5.21. This architecture has four layers. The two bottom layers provide common functionality and database access for the campaign costing and advert planning subsystems. Part of the architectural specification may include the identification of particular technologies to be used. In this case it may be decided to use a client–server architecture with the subsystem interfaces operating through a web browser to give maximum operational flexibility.

System design is also concerned with identifying and documenting suitable development standards (e.g. interface design standards, coding standards) for the remainder of the project. System Design is explained in Chapter 13.

5.4.2.4 Class design

Each of the use case analysis models is now elaborated separately to include relevant design detail. Interaction sequence diagrams may be drawn to show detailed object communication (Chapter 9) and state diagrams may be prepared for objects

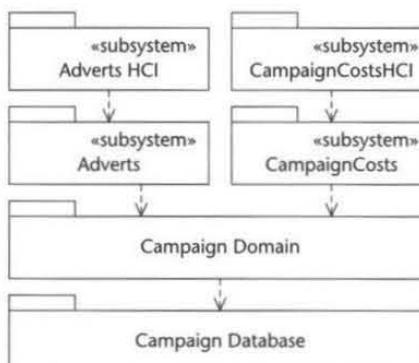


Figure 5.21 Possible architecture for part of the Agate system.

with complex state behaviour (Chapter 11). The separate models are then integrated to produce a detailed design class diagram. Design classes have attributes and operations specified (Fig. 5.22) to replace the less specific responsibilities that may have been identified by the analysis activity (Fig. 5.20). The detailed design of the classes normally necessitates the addition of further classes to support, for example, the user interface and access to the data storage system (typically a database management system). Class design is explained in Chapter 14.

5.4.2.5 User interface design

The nature of the functionality offered via each use case has been defined in requirements analysis. User interface design produces a detailed specification as to how the required functionality can be realized. User interface design gives a system its look and feel and determines the style of interaction the user will have. It includes the positioning and colour of buttons and fields, the mode of navigation used between different parts of the system and the nature of online help. Interface design is explained in Chapter 17 and is very dependent on class design.

5.4.2.6 Data management design

Data management design focuses on the specification of the mechanisms suitable for implementation of the database management system being used (see Chapter 18). Techniques such as normalization and entity–relationship modelling may be particularly useful if a relational database management system is being used. Data management design and class design are interdependent.

5.4.2.7 Construction

Construction is concerned with building the application using appropriate development technologies. Different parts of the system may be built using different languages. Java may be used to construct the user interface, while a database management system such as Oracle would manage data storage and handle commonly used processing routines.

5.4.2.8 Testing

Before the system can be delivered to the client it must be thoroughly tested. Testing scripts should be derived from the use case descriptions that were previously agreed with the client. Testing should be performed as elements of the system are developed.

Client
– companyName: String
– companyAddress: Address
– companyTelephone: Phone
+ addClient(name: String, address: Address, phone: Phone)
+ addNewCampaign(campaign: Campaign)
+ getClientCampaigns(): Campaign[]

Figure 5.22 Partly completed sample design class.

5.4.2.9 Implementation

The final implementation of the system will include its installation on the various computers that will be used. It will also include managing the transition from the old systems to the new systems for the client. This will involve careful risk management and staff training.

5.5 Summary

As in many kinds of development projects, we use models to represent things and ideas that we want to document and to test out without having to actually build a system. Of course, our ultimate aim is to build a system and the models help us to achieve that. Models allow us to create different views of a system from different perspectives and, in an information systems development project, most models are graphical representations of things in the real world and the software artefacts that will be used in the information system.

These graphical representations are diagrams, which can be used to model objects and processes. In UML a number of diagrams are defined and the rules for how they are to be drawn are documented. UML diagrams are made up of four graphical elements: icons, two-dimensional symbols, paths and strings. Diagrams are also supported with textual material, some of which may be informal, for example in natural language, while some may be formal, for example written in Object Constraint Language.

As a project progresses a variety of models are produced in order to represent different aspects of the system that is being built. A model is a complete and consistent view of a system from a particular perspective, typically produced using diagrams. An example of a diagram notation that is used in UML is the activity diagram. Activity diagrams model activities that are carried out in a system and include sequences of actions, alternative paths and repeated actions. As well as being used in system development projects, activity diagrams are also used in the Unified Software Development Process to document the sequence of activities in a workflow.

The Unified Software Development Process provides a specification of a process that can be used to develop software systems. It is made up of phases, within which models of the system are elaborated through successive iterations in which additional detail is added to the models until the system can be constructed in software and implemented. For the purpose of this book, we have broken the software development process into a number of activities that must be undertaken in order to develop a system. These activities are described in more detail in subsequent chapters.

Review Questions

- 5.1** What is the difference between a diagram and a model?
- 5.2** What are the four elements of a UML diagram?
- 5.3** Why do we use models in developing computerized information systems and other artefacts?

5.4 Why do we need standards for the graphical elements of diagrams?

5.5 What is the UML notation for each of the following: package, subsystem and model?

5.6 In what way can we show in UML that something is contained within something else, for example a subsystem within another subsystem?

5.7 What is the notation used for an activity in a UML activity diagram?

5.8 What links activities in an activity diagram?

5.9 In what way can a decision be represented in a UML activity diagram?

5.10 What are the two special nodes shown in an activity diagram?

5.11 What is meant by a guard condition?

5.12 What is an object flow?

5.13 What is the notation for an object flow?

5.14 What is the difference between the USDP and the waterfall lifecycle in the relationship between activities and phases?

Case Study Work, Exercises and Projects

5.A Some people suggest that information systems are models or simulations of the real world. What are the advantages and disadvantages of thinking of information systems in this way?

5.B Think of other kinds of development project in which models are used. For each kind of project list the different kinds of models that you think are used.

5.C Choose a task that you carry out and that you understand, for example preparing an assignment at college or university, or a task at work. Draw an activity diagram to summarize the actions that make up this task. Use activity partitions if the task involves actions that are carried out by other people.

5.D Choose some of the actions in your activity diagram and break them down into more detail in separate diagrams.

5.E Read about the Rational Unified Process (RUP) (see references in the Further Reading section). Identify some of the differences between RUP and USDP.

Further Reading

Booch et al. (1999) discuss the purpose of modelling and the differences between models and diagrams. They also describe the notation of activity diagrams. Jacobson et al. (1999) describe the Unified Software Development Process and explain the notation of the stereotyped activity diagrams that they use to model the workflows in the USDP.

An alternative to the USDP is the Rational Unified Process, see Kruchten (2004), Kroll and Kruchten (2003) or the IBM Corporation website (<http://www-130.ibm.com/developerworks/rational/products/rup/>).

Requirements Capture

Learning Objectives

In this chapter you will learn

- the distinction between the current and required systems
- when and how to apply the five major fact-finding techniques
- the need to document requirements
- how to draw *use case diagrams* to document requirements
- how to write *use case descriptions*.

6.1 Introduction

Part of the job of the systems analyst is to find out from users what they require in a new information system. Indeed, identifying what a new system should be able to do is one of the first steps in its development, whether you are developing some simple programs for your own use or embarking on the development of a large-scale system for a commercial client. The user requirements can be classified in different ways (Section 6.2), and analysts use a range of techniques to identify and document the requirements (Sections 6.3 and 6.5). Each of the main fact-finding techniques has advantages and disadvantages and is appropriate for different situations. Stakeholders are the people who have an interest in the new system and whose needs must be considered (Section 6.4). UML provides a diagramming technique that can be used to document the stakeholders' requirements. This is the *use case diagram*, a relatively simple diagram that is supported by written information in the form of *use case descriptions* (Section 6.6).

6.2 User Requirements

The aim of developing a new information system must be to produce something that meets the needs of the people who will be using it. In order to do this, we must

have a clear understanding both of the overall objectives of the business and of what it is that the individual users of the system are trying to achieve in their jobs. Unless you are in the rare position of developing a system for a new organization, you will need to understand how the business is operating at present and how people are working now. Many aspects of the current system will need to be carried forward into the new system, so it is important that information about what people are doing is gathered and documented. These are the requirements that are derived from the 'current system'. The motivation for the development of a new information system is usually problems with and inadequacies of the current system, so it is also essential to capture what it is that the users require of the new system that they cannot do with their existing system. These are the 'new requirements'. The requirements should identify benefits of buying or building the new system that can be included in the 'Business Case' or 'Cost-Benefit Analysis', which is used to justify the expense of the new system in terms of the benefits that it will bring.

6.2.1 Current system

The existing system may be a manual one, based on paper documents, forms and files; it may already be computerized; or it may be a combination of both manual and computerized elements. Whichever it is, it is reasonably certain that large parts of the existing system meet the needs of the people who use it, that it has to some extent evolved over time to meet business needs and that users are familiar and comfortable with it. It is almost equally certain that there are sections of the system that no longer meet the needs of the business, and that there are aspects of the business that are not dealt with in the existing system.

It is important that the analyst, gathering information as one of the first steps in developing a new system, gains a clear understanding of how the existing system works: parts of the existing system will be carried forward into the new one. It is also important because the existing system will have shortcomings and defects, which must be avoided or overcome in the new system. It is not always easy or possible to replace existing systems. So-called *legacy systems* may have been developed some time ago and may contain millions of lines of program code, which have been added to and amended over a period of time. One approach to dealing with such systems is to create new front-ends, typically using modern graphical user interfaces and object-oriented languages, and *wrap* the legacy systems up in new software. If this is the case, then it is also necessary to understand the interfaces to the legacy systems that the new *wrappers* will have to communicate with.

It is not always possible to leave legacy systems as they are and simply wrap them in new code. Sometimes the suppliers of legacy systems withdraw support for older versions of applications; sometimes the technology in which they were written becomes obsolete and unsupported. In these cases, the system may need to be replaced for IT reasons rather than because of business requirements for new functionality. It was not possible to ignore the problems that faced companies at the turn of the century, when it was realized that many systems were in danger of catastrophic collapse as a result of the decision to use two decimal digits to store the year. However, the process of changing the program code in such systems is a matter of understanding the internal working of existing systems rather than

- performance criteria such as desired response times for updating data in the system or retrieving data from the system
- anticipated volumes of data, either in terms of throughput or of what must be stored
- security considerations.

Usability requirements

Usability requirements are those that will enable us to ensure that there is a good match between the system that is developed and both the users of that system and the tasks that they will undertake when using it. The International Standards Organization (ISO) has defined the usability of a product as 'the degree to which specific users can achieve specific goals within a particular environment; effectively, efficiently, comfortably and in an acceptable manner'. Usability can be specified in terms of measurable objectives, and these are covered in more detail in Chapter 16 on Human–Computer Interaction. In order to build usability into the system from the outset, we need to gather the following types of information:

- characteristics of the users who will use the system
- the tasks that the users undertake, including the goals that they are trying to achieve
- situational factors that describe the situations that could arise during system use
- acceptance criteria by which the user will judge the delivered system.

Booth (1989) describes the issues surrounding system usability in more detail.

6.3 Fact-Finding Techniques

There are five main fact-finding techniques that are used by analysts to investigate requirements. Here we describe each of them in the order that they might be applied in a system development project, and for each one we explain the kind of information that you would expect to gain from its use, its advantages and disadvantages, and the situations in which it is appropriate to use it.

6.3.1 Background reading

If an analyst is employed within the organization that is the subject of the fact-gathering exercise, then it is likely that he or she will already have a good understanding of the organization and its business objectives. If, however, he or she is going in as an outside consultant, then one of the first tasks is to try to gain an understanding of the organization. Background reading or research is part of that process. The kind of documents that are suitable sources of information include the following:

- company reports
- organization charts
- policy manuals
- job descriptions

- reports
- documentation of existing systems.

Although reading company reports may provide the analyst with information about the organization's mission, and so possibly some indication of future requirements, this technique mainly provides information about the current system.

Advantages and disadvantages

- + Background reading helps the analyst to get an understanding of the organization before meeting the people who work there.
- + It also allows the analyst to prepare for other types of fact finding, for example, by being aware of the business objectives of the organization.
- + Documentation on the existing system may provide formally defined information requirements for the current system.
- Written documents often do not match up to reality; they may be out of date or they may reflect the official policy on matters that are dealt with differently in practice.

Appropriate situations

Background reading is appropriate for projects where the analyst is not familiar with the organization being investigated. It is useful in the initial stages of investigation.

6.3.2 Interviewing

Interviewing is probably the most widely used fact-finding technique; it is also the one that requires the most skill and sensitivity. Because of this, we have included a set of guidelines on interviewing that includes some suggestions about etiquette in Box 6.1.

A systems analysis interview is a structured meeting between the analyst and an interviewee who is usually a member of staff of the organization being investigated. The interview may be one of a series of interviews that range across different areas of the interviewee's work or that probe in progressively greater depth about the tasks undertaken by the interviewee. The degree of structure may vary: some interviews are planned with a fixed set of questions that the interviewer works through, while others are designed to cover certain topics but will be open-ended enough to allow the interviewer to pursue interesting facts as they emerge. The ability to respond flexibly to the interviewee's responses is one of the reasons why interviews are so widely used.

Interviews can be used to gather information from management about their objectives for the organization and for the new information system, from staff about their existing jobs and their information needs, and from customers and members of the public as possible users of systems. While conducting an interview, the analyst can also use the opportunity to gather documents that the interviewee uses in his or her work.

It is usually assumed that questionnaires are used as a substitute for interviews when potential interviewees are geographically dispersed in branches and offices around the world. The widespread use of desktop video conferencing may change

Box 6.1 Guidelines on Interviewing

Conducting an interview requires good planning, good interpersonal skills and an alert and responsive frame of mind. These guidelines cover the points you should bear in mind when planning and conducting an interview.

Before the interview

You should always make appointments for interviews in advance. You should give the interviewee information about the likely duration of the interview and the subject of the interview.

Being interviewed takes people away from their normal work. Make sure that they feel that it is well spent.

It is conventional to obtain permission from an interviewee's line manager before interviewing them. Often the analyst interviews the manager first and uses the opportunity to get this permission.

In large projects, an interview schedule should be drawn up showing who is to be interviewed, how often and for how long. Initially this will be in terms of the job roles of interviewees rather than named individuals. It may be the manager who decides which individual you interview in a particular role.

Have a clear set of objectives for the interview. Plan your questions and write them down. Some people write the questions with space between them for the replies.

Make sure your questions are relevant to the interviewee and his or her job.

At the start of the interview

Introduce yourself and the purpose of the interview.

Arrive on time for interviews and stick to the planned timetable—do not overrun.

Ask the interviewee if he or she minds you taking notes or tape-recording the interview. Even if you tape-record an interview, you are advised to take notes. Machines can fail! Your notes also allow you to refer back to what has been said during the course of the interview and follow up points of interest.

Remember that people can be suspicious of outside consultants who come in with clipboards and stopwatches. The cost-benefit analyses of many information systems justify the investment in terms of savings in jobs!

During the interview

Take responsibility for the agenda. You should control the direction of the interview. This should be done in a sensitive way. If the interviewee is getting away from the subject, bring them back to the point. If what they are telling you is important, then say that you will come back to it later and make a note to remind yourself to do so.

Use different kinds of question to get different types of information. Questions can be open-ended—'Can you explain how you complete a timesheet?'—or closed—'How many staff use this system?'. Do not, however, ask very open-ended questions such as 'Could you tell me what you do?'

Listen to what the interviewee says and encourage him or her to expand on key points.

Keep the focus positive if possible. Make sure you have understood answers by summarizing them back to the interviewee. Avoid allowing the interview to degenerate into a session in which the interviewee complains about everyone and everything.

You may be aware of possible problems in the existing system, but you should avoid pre-judging issues by asking questions that focus too much on problems. Gather facts.

Be sensitive about how you use information from other interviews that you or your colleagues have already conducted, particularly if comments were negative or critical.

Use the opportunity to collect examples of documents that people use in their work, ask if they mind you having samples of blank forms and photocopies of completed paperwork.

After the interview

Thank the interviewee for their time. Make an appointment for a further interview if it is necessary. Offer to provide them with a copy of your notes of the interview for them to check that you have accurately recorded what they told you.

Transcribe your tape or write up your notes as soon as possible after the interview while the content is still fresh in your mind.

If you said that you would provide a copy of your notes for checking then send it to the interviewee as soon as possible. Update your notes to reflect their comments.

this and make it possible to interview staff wherever they are. Even then, questionnaires can reach more people.

Interviewing different potential users of a system separately can mean that the analyst is given different information by different people. Resolving these differences later can be difficult and time-consuming. One alternative is to use group interviews in order to get the users to reach a consensus on issues. Dynamic Systems Development Method (DSDM) is a method of carrying out systems development in which group discussions are used (Stapleton, 1997). These discussions are run as workshops for knowledgeable users with a facilitator who aims to get the users to pool their knowledge and to reach a consensus on the priorities of the development project.

Advantages and disadvantages

- + Personal contact allows the analyst to be responsive and adapt to what the user says. Because of this, interviews produce high quality information.
- + The analyst can probe in greater depth about the person's work than can be achieved with other methods.
- + If the interviewee has nothing to say, the interview can be terminated.
- Interviews are time-consuming and can be the most costly form of fact gathering.
- Interview results require the analyst to work on them after the interview: the transcription of tape recordings or writing up of notes.
- Interviews can be subject to bias if the interviewer has a closed mind about the problem.
- If different interviewees provide conflicting information, it can be difficult to resolve later.

Appropriate situations

Interviews are appropriate in most projects. They can provide information in depth about the existing system and about people's requirements from a new system.

6.3.3 Observation

Watching people carrying out their work in a natural setting can provide the analyst with a better understanding of the job than interviews, in which the interviewee will often concentrate on the normal aspects of the job and forget the exceptional situations and interruptions that occur and which the system will need to cope with. Observation also allows the analyst to see what information people use to carry out their jobs. This can tell you about the documents they refer to, whether they have to get up from their desks to get information, how well the existing system handles their needs. One of the authors has observed staff using a telesales system where there was no link between the enquiry screens for checking the availability of stock and the data entry screens for entering an order. These telesales staff kept a pad of scrap paper on the desk and wrote down the product codes for all the items they had looked up on the enquiry screens so that they could enter them into the order-processing screens. This kind of information does not always emerge from interviews.

People are not good at estimating quantitative data, such as how long they take to deal with certain tasks, and observation with a stopwatch can give the analyst

plentiful quantitative data, not just about typical times to perform a task but also about the statistical distribution of those times.

In some cases where information or items are moving through a system and being dealt with by many people along the way, observation can allow the analyst to follow the entire process through from start to finish. This type of observation might be used in an organization where orders are taken over the telephone, passed to a warehouse for picking, packed and dispatched to the customer. The analyst may want to follow a series of transactions through the system to obtain an overview of the processes involved.

Observation can be an open-ended process in which the analyst simply sets out to observe what happens and to note it down, or it can be a closed process in which the analyst wishes to observe specific aspects of the job and draws up an observation schedule or form on which to record data. This can include the time it takes to carry out a task, the types of task the person is performing or factors such as the number of errors they make in using the existing system as a baseline for usability design.

Advantages and disadvantages

- + Observation of people at work provides first-hand experience of the way that the current system operates.
- + Data are collected in real time and can have a high level of validity if care is taken in how the technique is used.
- + Observation can be used to verify information from other sources or to look for exceptions to the standard procedure.
- + Baseline data about the performance of the existing system and of users can be collected.
- Most people do not like being observed and are likely to behave differently from the way in which they would normally behave. This can distort findings and affect the validity.
- Observation requires a trained and skilled observer for it to be most effective.
- There may be logistical problems for the analyst, for example, if the staff to be observed work shifts or travel long distances in order to do their job.
- There may also be ethical problems if the person being observed deals with sensitive private or personal data or directly with members of the public, for example in a doctor's surgery.

Appropriate situations

Observation is essential for gathering quantitative data about people's jobs. It can verify or disprove assertions made by interviewees, and is often useful in situations where different interviewees have provided conflicting information about the way the system works. Observation may be the best way to follow items through some kind of process from start to finish.

6.3.4 Document sampling

Document sampling can be used in two different ways. First, the analyst will collect copies of blank and completed documents during the course of interviews and observation sessions. These will be used to determine the information that is used

by people in their work and the inputs to and outputs from processes which they carry out, either manually or using an existing computer system. Ideally, where there is an existing system, screen shots should also be collected in order to understand the inputs and outputs of the existing system. Figure 6.1 shows a sample document collected from Agate, our case study company.

Second, the analyst may carry out a statistical analysis of documents in order to find out about patterns of data. For example, many documents such as order forms contain a header section and a number of lines of detail. (The sample document in Fig. 6.1 shows this kind of structure.) The analyst may want to know the distribu-

Agate					
Campaign Summary					
Date	23rd February 2005				
Client	Yellow Partridge Park Road Workshops Jewellery Quarter Birmingham B2 3DT U.K.				
Campaign	Spring Collection 2005				
Billing Currency	GBP £				
Item	Curr	Amount	Rate	Billing amount	
Advert preparation: photography, artwork, layout etc.	GBP £	15,000.00	1	15,000.00	
Placement French Vogue	EUR €	6 500,00	1.47	4,421.77	
Placement Portuguese Vogue	EUR €	5 500,00	1.47	3,741.50	
Placement US Vogue	USD \$	17,000.00	1.77	9,604.52	
Total				32,767.79	
This is not a VAT Invoice. A detailed VAT Invoice will be provided separately.					

Figure 6.1 Sample document from the AGATE case study.

tion of the number of lines in an order. This will help later in estimating volumes of data to be held in the system and in deciding how many lines should be displayed on screen at one time. While this kind of statistical sampling can give a picture of data volumes, the analyst should be alert to seasonal patterns of activity, which may mean that there are peaks and troughs in the amount of data being processed.

Advantages and disadvantages

- + Can be used to gather quantitative data, such as the average number of lines on an invoice and the range of values.
- + Can be used to find out about error rates in paper documents.
- If the system is going to change dramatically, existing documents may not reflect how it will be in future.

Appropriate situations

The first type of document sampling is almost always appropriate. Paper-based documents give a good idea of what is happening in the current system. They also provide supporting evidence for the information gathered from interviews or observation.

The statistical approach is appropriate in situations where large volumes of data are being processed, and particularly where error rates are high and a reduction in errors is one of the criteria for usability.

6.3.5 Questionnaires

Questionnaires are a research instrument that can be applied to fact finding in system development projects. They consist of a series of written questions. The questionnaire designer usually limits the range of replies that respondents can make by giving them a choice of options. (Figure 6.2 shows some of the types of question.) YES/NO questions only give the respondent two options. (Sometimes a DON'T KNOW option is needed as well.) If there are more options, the multiple choice type of question is often used when the answer is factual, whereas scaled questions are used if the answer involves an element of subjectivity. Some questions do not have a fixed number of responses, and must be left open-ended for the respondent to enter what they like. Where the respondent has a limited number of choices, these are usually coded with a number, which speeds up data entry if the responses are to be analysed by computer software. If you plan to use questionnaires for requirements gathering, they need very careful design. Box 6.2 lists some of the issues that need to be addressed if you are thinking of using questionnaires.

Advantages and disadvantages

- + An economical way of gathering data from a large number of people.
- + If the questionnaire is well designed, then the results can be analysed easily, possibly by computer.
- Good questionnaires are difficult to construct.
- There is no automatic mechanism for follow-up or probing more deeply, although it is possible to follow up with an interview by telephone or in person if necessary.
- Postal questionnaires suffer from low response rates.

YES/NO Questions			
Do you print reports from the existing system? (Please circle the appropriate answer.)	YES	NO	10
Multiple Choice Questions			
How many new clients do you obtain in a year? (Please tick one box only.)	a) 1–10	<input type="checkbox"/>	11
	b) 11–20	<input type="checkbox"/>	
	c) 21–30	<input type="checkbox"/>	
	d) 31+	<input type="checkbox"/>	
Scaled Questions			
How satisfied are you with the response time of the stock update? (Please circle one option.)	1. Very satisfied	2. Satisfied	3. Dissatisfied
	4. Very dissatisfied		
12			
Open-ended Questions			
What additional reports would you require from the system?			
<hr/> <hr/> <hr/>			

Figure 6.2 Types of question used in questionnaires.

Appropriate situations

Questionnaires are most useful when the views or knowledge of a large number of people need to be obtained or when the people are geographically dispersed, for example in a company with many branches or offices around the country or around the world. Questionnaires are also appropriate for information systems that will be used by the general public, and where the analyst needs to get a picture of the types of user and usage that the system will need to handle.

6.3.6 Remembering the techniques

For those who like mnemonics, these techniques are sometimes referred to as SQIRO—Sampling, Questionnaires, Interviewing, Reading (or Research) and Observation. This order has been chosen to make it possible to pronounce the mnemonic. However, this is not the order in which they are most likely to be used. This will depend on the situation and the organization in which the techniques are being used.

6.3.7 Other techniques

Some kinds of system require special fact-finding techniques. *Expert systems* are computer systems that are designed to embody the expertise of a human expert in solving problems. Examples include systems for medical diagnosis, stock market trading and geological analysis for mineral prospecting. The process of capturing

the knowledge of the expert is called *knowledge acquisition* and, as it differs from establishing the requirements for a conventional information system, a number of specific techniques are applied. Some of these are used in conjunction with computer-based tools.

Box 6.2 Guidelines on Questionnaires

Using questionnaires requires good planning. If you send out 100 questionnaires and they do not work, it is difficult to get respondents to fill in a second version. These guidelines cover the points you should bear in mind when using questionnaires.

Coding

How will you code the results? If you plan to use an optical mark reader, then the response to every question must be capable of being coded as a mark in a box. If you expect the results to be keyed into a database for analysis, then you need to decide on the codes for each possible response. If the questions are open-ended, how will you collate and analyse different kinds of responses?

Analysis

Whatever analysis you plan should be decided in advance. If you expect to carry out a statistical analysis of the responses, you should consult a statistician **before** you finalize the questions. Statistical techniques are difficult to apply to responses to poorly designed questions.

You can use a special statistical software package, a database or even a spreadsheet to analyse the data.

Piloting

You should try out your questionnaire on a small pilot group or sample of your respondents. This enables you to find out if there are questions they do not understand, they misinterpret or they cannot answer.

If you plan to analyse the data using statistical software, a database or a spreadsheet, you can create a set of trial data to test your analysis technique.

Sample size and structure

If you plan to use serious statistical techniques,

then those techniques may place lower limits on your sample size. If you want to be sure of getting a representative sample, by age, gender, department, geographical location, job grade or experience of existing systems, then that will help to determine how many people to include. Otherwise it may be down to you to choose a sensible percentage of all the possible respondents.

Delivery

How will you get the questionnaires to your respondents, and how will they get their replies back to you?

You can post them, or use internal mail in a large organization, fax them, email them or create a web-based form on the company intranet and notify your target group by email. If you use the intranet, you may want to give each respondent a special code, so that only they can complete their own questionnaire.

Your respondents can then post, fax or email their responses back to you, or complete them on the intranet.

Respondent information

What information about the respondents do you want to gather at the same time as you collect their views and requirements? If you want to analyse responses by age, job type or location, then you need to include questions that ask for that information.

You can make questionnaires anonymous or you can ask respondents for their name. If the questionnaire is not anonymous, you need to think about confidentiality. People will be more honest in their replies if they can respond anonymously or in confidence.

If you ask for respondents' names and you store that information, then in the UK you should consider the provisions of the Data Protection Act (1998). (See also Chapter 13.)

Box 6.2 Guidelines on Questionnaires (continued)

There are similar requirements in other countries.

Covering letter or email

In a covering letter you should explain the purpose and state that the questionnaire has management support. Give an estimate of the time required to fill in the questionnaire and a deadline for its return. Thank the respondents for taking part.

Structure

Structure the questionnaire carefully. Give it a title, and start with explanatory material and notes on how to complete it. Follow this with questions about the respondent (if required). Group questions together by subject. Avoid lots of instructions like 'If you answered YES to Q. 7a, now go to Q. 13'. Keep it reasonably short.

Return rate

Not everyone will necessarily respond. You need to plan for this and either use a larger sample than you need or follow up with reminders. If you use a form on the intranet, you should be able to identify who has not responded and email them reminders. Equally, you can email a thank you to those who do respond.

Feedback

This needs to be handled carefully—telling everyone that 90% of the company cannot use the existing system may not go down well—but people do like to know what use was made of the response they made. They may have spent half an hour filling in your questionnaire, and they will expect to be informed of the outcome. A summary of the report can be sent out to branches, distributed to departments, sent to named respondents or placed on the company intranet.

6.4 User Involvement

The success of a systems development project depends not just on the skills of the team of analysts, designers and programmers who work on it, or on the project management skills of the project manager, but on the effective involvement of users in the project at various stages of the lifecycle. The term *stakeholders* was introduced in Chapter 2 to describe all those people who have an interest in the successful development of the system. Stakeholders include all people who stand to gain (or lose) from the implementation of the new system: users, managers and budget-holders. Analysts deal with people at all levels of the organization. In large projects it is likely that a steering committee with delegated powers will be set up to manage the project from the users' side. This will include the following categories of people:

- senior management—with overall responsibility for running the organization
- financial managers with budgetary control over the project
- managers of the user department(s)
- representatives of the IT department delivering the project
- representatives of users.

Users will be involved in different roles during the course of the project as:

- subjects of interviews to establish requirements

- representatives on project committees
- those involved in evaluating prototypes
- those involved in testing
- subjects of training courses
- end-users of the new system.

Case Study Example

The section that follows applies what has been covered in this chapter so far to the case study.

Objective	Technique	Subject(s)	Time commitment
To get background on the company and the advertising industry	Background reading	Company reports, trade journals	0.5 day
To establish business objectives. Interview Agree likely scope of new system. Check out involvement of non-UK offices	Interview	Two directors	2 x 1 hour each
To gain understanding of roles of each department. Check out line management and team structure in the Creative department. Agree likely interviewees among staff	Interview	Department heads (only 1 account manager)	2 x 1 hour each
To find out how the core business operates	Interview	1 account manager 1 graphic designer 1 copy writer 1 editor	1.5 hours each
To follow up development of business understanding	Observation	2 creative staff	0.5 day each
To determine role of support/admin staff and relationship to core business	Interview	2 admin staff (based on experience with the company)	1.5 hours each
To establish what records and resources are kept	Interview/document sampling	Filing clerk Resource librarian	2 x 1 hour each
To determine what use is made of current computer system. To determine functionality of current system	Interview	Computer manager	2 x 1 hour
To establish additional requirements for new system	Interview	2 account managers 3 staff from Creative department	3 x 1 hour each
To establish accounting requirements for new system	Interview	Accountant Credit controller 1 purchasing assistant 1 accounts clerk	1.5 hours each

One of the first tasks in fact finding is to draw up a plan that outlines what information is being sought, which techniques will be used, who is involved and how long the fact finding will take. A draft plan for fact finding at Agate is shown above. The jobs of the subjects are those shown in the list in Figure A1.1 in the Agate case study.

6.5 Documenting Requirements

Information systems professionals need to record facts about the organization they are studying and its requirements. As soon as the analysts start gathering facts, they will need some means of documenting them. In the past the emphasis was on paper forms, but now it is rare for a large-scale project to depend on paper-based documentation. As we have explained in Chapter 5, systems analysts and designers model the new system in a mixture of diagrams and text. The important thing to bear in mind is that within a project some set of standards should be adhered to. These may be the agreed standards of the organization carrying out the analysis and design project or they may be a requirement of the organization that is having the work done. For example, government and military projects usually require that developers conform to a specific set of standards. We are using UML to produce models of the system from different perspectives. Computer Aided Software Engineering (CASE) tools are normally used to draw the diagrammatic models and to maintain in a repository the associated data about the various things that are shown in the diagrams.

However, there will also be other kinds of documents, not all of which fit into the UML framework. In large-scale projects a librarian or configuration manager may be required to keep track of these documents and ensure that they are stored safely and in a way that enables them to be retrieved when required. Such documents include:

- records of interviews and observations
- details of problems
- copies of existing documents and where they are used
- details of requirements
- details of users
- minutes of meetings.

Even in smaller projects that cannot justify a librarian, a filing system with an agreed set of conventions on how material is to be filed, and for recording who has taken items from the filing system, is good practice.

In many projects, these documents will be stored digitally, using a document management system or a version control system. Handwritten documents and sample documents can be scanned with an image scanner and held in digital form. In this case, many people can access the same document simultaneously. The system enforces control over whether a document can be updated, and ensures that no more than one person at a time is able to 'check out' a document in order to amend it.

Not all of the documents listed above represent requirements, and it is necessary to maintain some kind of list or database of requirements. There are software tools

available to hold requirements in a database and some can be linked to CASE tools and testing tools. This makes it possible to trace from an initial requirement through the analysis and design models to where it has been implemented and to the test cases that test whether the requirement has been met.

Use cases, which are explained in the next section, can be used to model requirements, but because they focus on the functionality of the system are not good for documenting non-functional requirements. Jacobson et al. (1999) suggest that the use case model should be used to document functional requirements and a separate list of 'supplementary requirements' (those not provided by a use case) should be kept. They say that, together, the use case model and the list of supplementary requirements constitute a traditional requirements specification. Rosenberg and Scott (1999) argue that use cases are not the same as requirements: use cases describe units of system behaviour, whereas requirements are rules that govern the behaviour of the system; one requirement may be met by more than one use case, and one use case may meet more than one requirement; some non-functional requirements are difficult to attribute to any particular use case.

Some people try to document requirements in use cases by writing long use case descriptions using templates that enable them to include non-functional requirements as well as functional requirements.

We favour the view that use cases can be used to model functional requirements, but a separate list of requirements should be kept, containing all requirements—functional and non-functional—for the system. Where there is a relationship between a particular use case and a particular requirement, this should be recorded. Moreover, some requirements describe very high-level units of behaviour and may need to be broken down into low-level requirements that describe more precisely what is to be done. Any database of requirements should make it possible to hold this kind of hierarchical structure of requirements.

There are software tools available to assist in the documentation of requirements. Essentially, they provide a database of requirements of different types with standard attributes of those requirements, and can usually be customized to allow the users to record other attributes of the requirements that are needed for a particular project or organization. In some cases, these tools can be integrated with CASE tools or modelling tools in order to provide for 'traceability of requirements'. This means that model elements that deliver a requirement can be linked to the

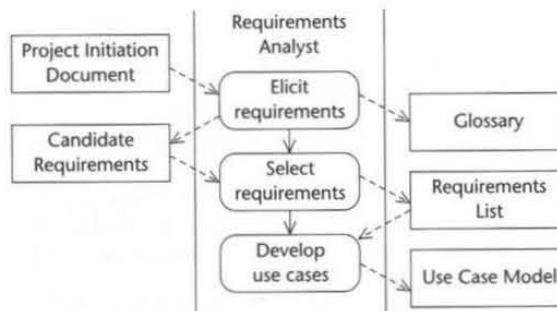


Figure 6.3 Activity diagram to show the activities involved in capturing requirements.

requirement in question. If a requirement subsequently changes, it is possible to trace the impact of that requirement change on all the model elements that it affects, such as use cases and classes.

Sometimes the process of requirement gathering throws up more requirements than can be met in a particular project. They may be outside the scope of the project, over-ambitious, too expensive to implement or just not really necessary at this stage. The process of building a requirements model for a system involves going through all the candidate requirements to produce a list of those that will be part of the current project. Figure 6.3 shows this as an activity diagram. (An approach such as the *MoSCoW rules* explained in Chapter 21 can be used to help prioritize requirements.)

6.6 Use Cases

Use cases are descriptions of the functionality of the system from the users' perspective. Use case diagrams are used to show the functionality that the system will provide and to show which users will communicate with the system in some way to use that functionality. Figure 6.4 shows an example of a use case diagram. This is a relatively simple diagramming technique, and its notation is explained below in Section 6.6.2.

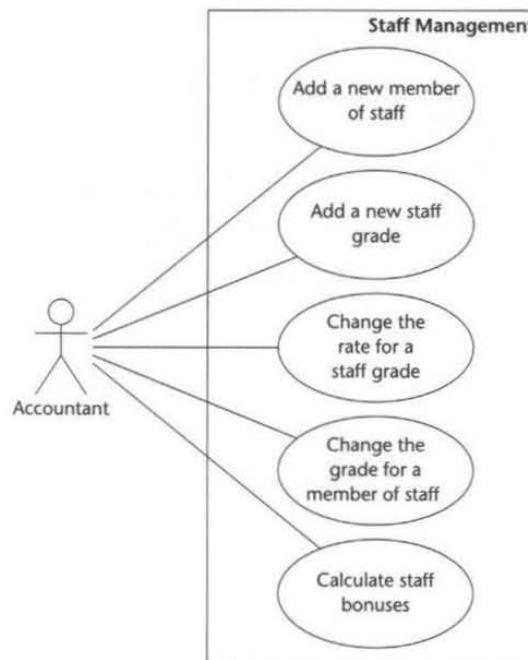


Figure 6.4 Example use case diagram.

Use case diagrams were developed by Jacobson et al. (1992), and the subtitle of the book in which they are presented is *A Use Case Driven Approach*. Jacobson and his co-authors offer a complete approach to the development of object-oriented software systems, but use case diagrams are the starting point for much of what follows in their approach.

6.6.1 Purpose

The use case model is part of what Jacobson et al. (1992) call the requirements model; they also include a problem domain object model and user interface descriptions in this requirements model. Use cases specify the functionality that the system will offer from the users' perspective. They are used to document the scope of the system and the developer's understanding of what it is that the users require.

Use cases are supported by *behaviour specifications*. These specify the behaviour of each use case either using UML diagrams, such as *communication diagrams* or *sequence diagrams* (see Chapter 9), or in text form as *use case descriptions*.

Textual *use case descriptions* provide a description of the interaction between the users of the system, termed *actors*, and the high-level functions within the system, the use cases. These descriptions can be in summary form or in a more detailed form in which the interaction between actor and use case is described in a step-by-step way. Whichever approach is used, it should be remembered that the use case describes the interaction as the user sees it, and is not a definition of the internal processes within the system or some kind of program specification.

6.6.2 Notation

Use case diagrams show three aspects of the system: actors, use cases and the system or subsystem boundary. Figure 6.5 shows the elements of the notation.

Actors represent the roles that people, other systems or devices take on when communicating with the particular use cases in the system. Figure 6.5 shows the actor Staff Contact in a diagram for the Agate case study. In Agate, there is no job title Staff Contact: a director, an account manager or a member of the creative team can take on the role of being staff contact for a particular client company, so one actor can represent several people or job titles. Equally, a

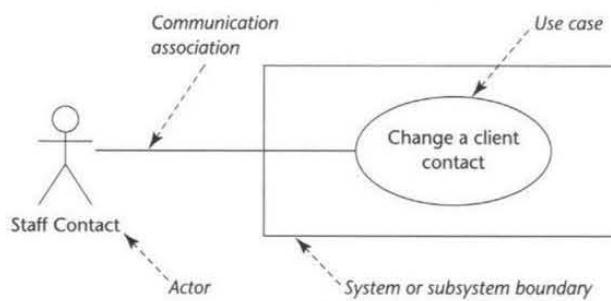


Figure 6.5 The notation of the use case diagram.



Figure 6.6 Use case showing Campaign Manager actor.

particular person or job title may be represented by more than one actor on use case diagrams. This is shown in Figs 6.5 and 6.6 together. A director or an account manager may be the Campaign Manager for a particular client campaign, as well as being the Staff Contact for one or more clients.

The use case description associated with each use case can be brief:

Assign staff to work on a campaign

The campaign manager selects a particular campaign. A list of staff not already working on that campaign is displayed, and he or she selects those to be assigned to this campaign.

Alternatively, it can provide a step-by-step breakdown of the interaction between the user and the system for the particular use case. An example of this extended approach is provided below.

Assign staff to work on a campaign

- | Actor Action | System Response |
|------------------------------------------------------------------|---------------------------------------------------------------------------------|
| 1. The actor enters the client name. | 2. Lists all campaigns for that client. |
| 3. Selects the relevant campaign. | 4. Displays a list of all staff members not already allocated to this campaign. |
| 5. Highlights the staff members to be assigned to this campaign. | 6. Presents a message confirming that staff have been allocated. |

Alternative Courses

Steps 1–3. The actor knows the campaign name and enters it directly.

Constantine (1997) makes the distinction between *essential* and *real* use cases. Essential use cases describe the 'essence' of the use case in terms that are free of any technological or implementation details, whereas real use cases describe the concrete detail of the use case in terms of its design. During the analysis stage, use cases are almost always essential, as the design has not yet been decided upon. In a real use case, Step 2 in the use case description for Assign staff to work on a campaign could be described as 'Lists all campaigns for the client in a list box, sorted into alphabetical order by campaign title'.

Each use case description represents the usual way in which the actor will go through the particular transaction or function from end to end. Possible major alternative routes that could be taken are listed as *alternative courses*. The term *scenario* is used to describe use cases in which an alternative course is worked through in detail, including possible responses to errors. The use case represents the generic case, while the scenarios represent specific paths through the use case.

As well as the description of the use case itself, the documentation should include the purpose or intent of the use case, that is to say details of the task that the user is trying to achieve through the means of this use case, for example:

The campaign manager wishes to record which staff are working on a particular campaign. This information is used to validate timesheets and to calculate staff year-end bonuses.

One way of documenting use cases is to use a template (a blank form or word-processing document to be filled in). This might include the following sections:

- name of use case
- pre-conditions (things that must be true before the use case can take place)
- post-conditions (things that must be true after the use case has taken place)
- purpose (what the use case is intended to achieve)
- description (in summary or in the format above).

Cockburn (2000) provides examples of templates and guidance on how to write good use cases.

Two further kinds of relationship can be shown on the use case diagram itself. These are the *Extend* and *Include* relationships. They are shown on the diagram using a piece of UML notation that you will come across in other diagrams: stereotypes.

Stereotypes—a stereotype is a special use of a model element that is constrained to behave in a particular way. Stereotypes can be shown by using a keyword, such as ‘extend’ or ‘include’ in matched guillemets, like «extend». (Guillemets are used as quotation marks in French and some other languages. They are not the same as guillemots, which are seabirds found in the North Atlantic and North Pacific oceans!) Stereotypes can also be represented using special icons. The actor symbol in use case diagrams is a stereotyped icon—an actor is a stereotyped class and could also be shown as a class rectangle (see Chapter 7) with the stereotype «actor» above the name of the actor. So by stereotyping classes as «actor» we are indicating that they are a special kind of class that interacts with the system’s use cases. Note, however, that actors are external to the system, unlike use cases and classes.

The Extend and Include relationships are easy to confuse. «extend» is used when you wish to show that a use case provides additional functionality that may be required in another use case. In Fig. 6.7, the use case *Print campaign summary* extends *Check campaign budget*. This means that at a particular point in *Check campaign budget* the user can optionally invoke the behaviour of *Print campaign summary*, which does something over and above what is done in *Check campaign budget* (print out the information in this case). There may be more than one way of extending a particular use case, and these possibilities may represent significant variations on the way the user uses the system. Rather than trying to capture all these variations in one use case, you would document the core functionality in one and then extend it in others. Extension points can be shown in the diagram, as in *Check campaign budget* in Fig. 6.7. They are shown in a separate compartment in the use case ellipse, headed *Extension points*. The names of the extension points are listed in this compartment. If an extension point exists, it must have a name. A condition can be shown in a UML *comment* attached to the

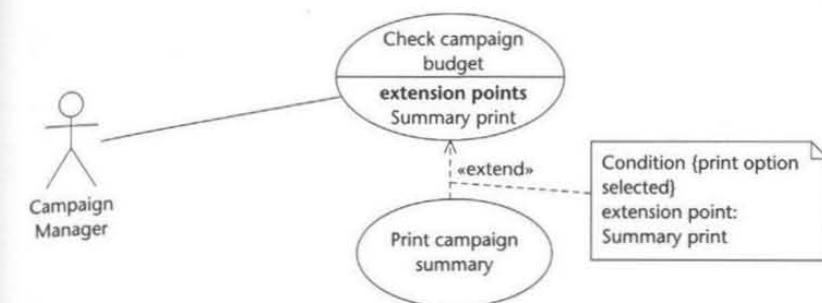


Figure 6.7 Use case diagram showing «extend».

relationship. Comments are used to add to a diagram information that is not part of other graphical elements in the diagram. The condition must be true for the extension to take place in a particular instance of the use case.

«include» applies when there is a sequence of behaviour that is used frequently in a number of use cases and you want to avoid copying the same description of it into each use case in which it is used. Figure 6.8 shows that the use case *Assign staff to work on a campaign* has an «include» relationship with *Find campaign*. This means that when an actor uses *Assign staff to work on a campaign* the behaviour of *Find campaign* will also be included in order to select the relevant campaign. Note the direction of the arrows in the examples of *Include* and *Extend* relationships. The arrow always points at the use case that is being included or extended.

It is important not to overuse *Include* and *Extend* relationships. Overuse can result in the decomposition of the functionality of use cases into many small use cases that deliver no real value to the users of the system.

As well as describing the use cases, it is worth describing who the actors are in terms of job titles or the way in which they interact with the system. Although at the moment we are concentrating on requirements, later we shall need to know who the actual users are for each high-level function that is represented by a use case. This may help in specifying the security for different functions or in assessing the usability of the functions.

Bear in mind that actors need not be human users of the system. They can also be other systems that communicate with the one that is the subject of the systems development project, for example other computers or automated machinery or equipment.



Figure 6.8 Use case diagram showing «include».

Figure 6.9 shows a use case diagram for the Campaign Management subsystem with both Extend and Include relationships. Note that you do not have to show all the detail of the extension points on a diagram: the Extension points compartment in the use case can be suppressed. Of course, if you are using a CASE tool to draw and manage the diagrams, you may be able to toggle the display of this compartment on and off and, even if the information is not shown on a particular diagram, it will still be held in the CASE tool's repository.

In Chapter 4, the concepts of generalization, specialization and inheritance were introduced. They are explained in more detail in Chapter 8. However, generalization and specialization can be applied to actors and use cases. For example, suppose that we have two actors, Staff Contact and Campaign Manager, and a Campaign Manager can do everything that a Staff Contact can do, and more. Rather than showing communication associations between Campaign Manager and all the use cases that Staff Contact can use, we can show Campaign Manager as a specialization of Staff Contact, as in Fig. 6.10. Similarly, there may be similar use cases where the common functionality is best represented by generalizing out that functionality into a 'super-use case' and showing it separately. For example, we may find that there are two use cases at Agate—Assign individual staff to work on a campaign and Assign team of staff to work on a campaign—which are similar in the functionality they offer. We might abstract out the commonality into a use case Assign staff to work on a campaign, but this will be an abstract use case. It helps us to define the functionality of the other two use cases, but no instance of this use case will ever exist in its own right. This is also shown in Fig. 6.10.

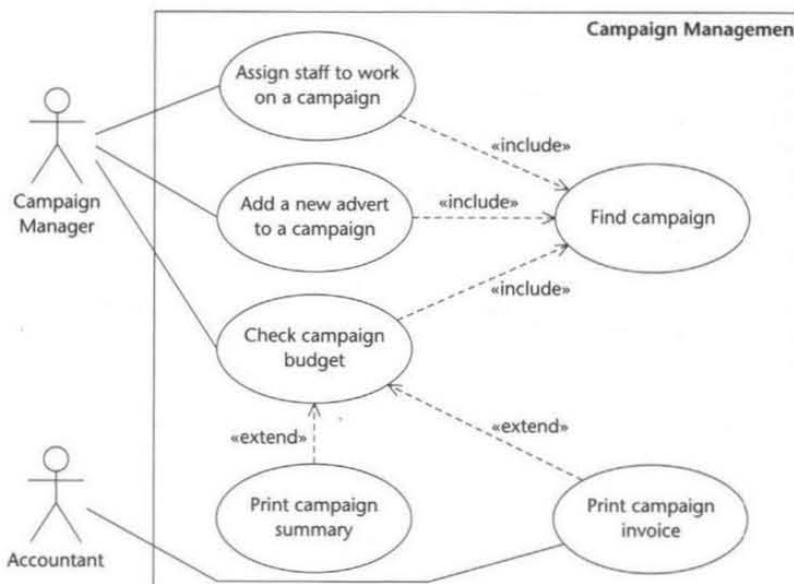


Figure 6.9 Use case diagram showing both «extend» and «include».

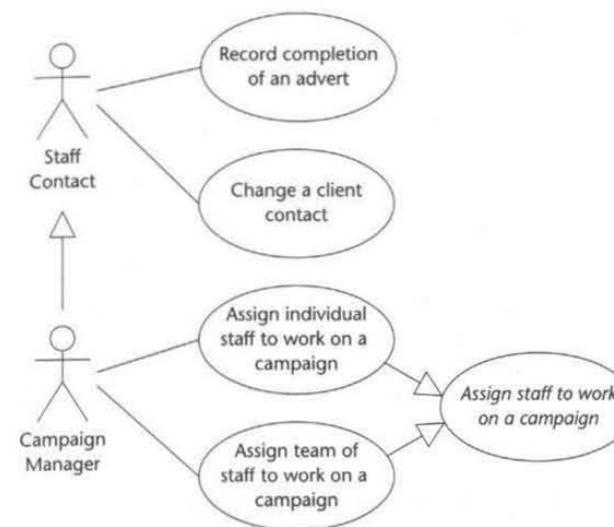


Figure 6.10 Generalization of actors and use cases.

6.6.3 Supporting use cases with prototyping

As the requirements for a system emerge in the form of use cases, it is sometimes helpful to build simple prototypes of how some of the use cases will work. A prototype is a working model of part of the system—usually a program with limited functionality that is built to test out some aspect of how the system will work. (Prototypes were discussed in Section 3.2.2 and are explained in more detail in Chapter 17 on the design of the user interface.)

Prototypes can be used to help elicit requirements. Showing users how the system might provide some of the use cases often produces a stronger reaction than showing them a series of abstract diagrams. Their reaction may contain useful information about requirements.

For example, there are a number of use cases in the Campaign Management subsystem for Agate that require the user to select a campaign in order to carry out some business function. The use case diagram in Fig. 6.9 reflects this in the «include» relationships with the use case Find campaign. The use case Find campaign will clearly be used a great deal, and it is worth making sure that we have the requirements right. A prototype could be produced that provides a list of all the campaigns in the system. A possible version of this is shown in Fig. 6.11.

Showing this prototype interface design to the users may well produce the response that this way of finding a campaign will not work. There may be hundreds of campaigns in the system, and scrolling through them would be tedious. Different clients may have campaigns with similar names, and it would be easy to make a mistake and choose the wrong campaign if the user does not know which client it belongs to. For these reasons, the users might suggest that the first step is to find the right client and then display only the campaigns that belong to that client. This leads to a different user interface as shown in Fig. 6.12.

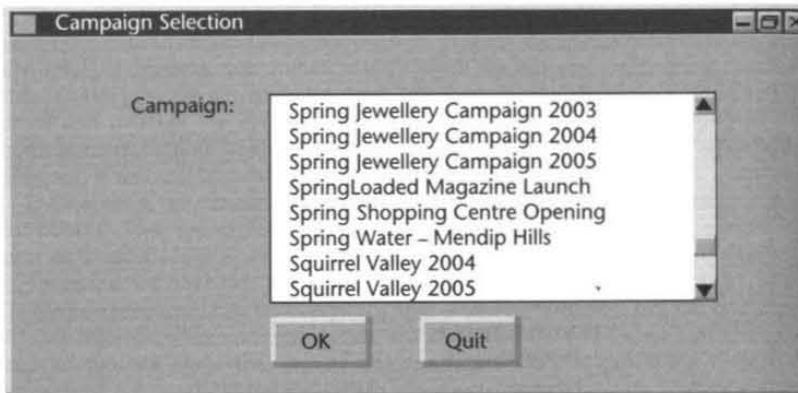


Figure 6.11 Prototype interface for the Find campaign use case.

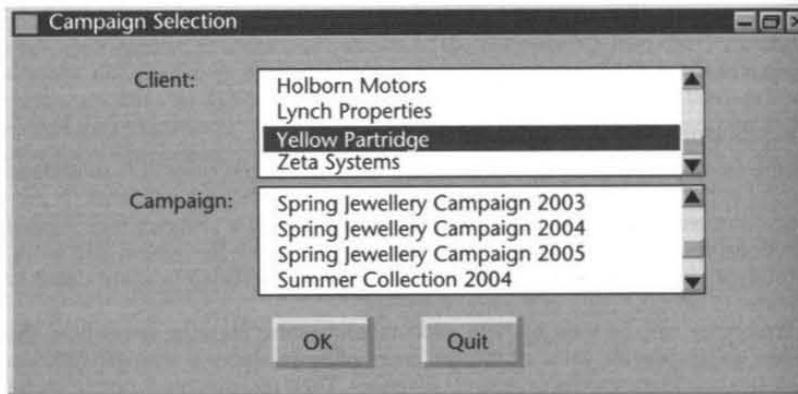


Figure 6.12 Revised prototype interface for the Find campaign use case.

The information from this prototyping exercise forms part of the requirements for the system. This particular requirement is about usability, but it can also contribute to meeting other, non-functional requirements concerned with speed and the error rate: it might be quicker to select first the client and then the campaign from a short-list than search through hundreds of campaigns; and it might reduce the number of errors made by users in selecting the right campaign to carry out some function on.

Prototypes can be produced with visual programming tools, with scripting languages like TCL/TK, with a package like Microsoft PowerPoint or even as web pages using HTML.

Prototypes do not have to be developed as programs. Screen and window designs can be sketched out on paper and shown to the users, either formally or informally. A series of possible screen layouts showing the steps that the user

would take to interact with a particular use case can be strung together in a storyboard, as in Fig. 6.13.

6.6.4 CASE tool support

Drawing any diagram and maintaining the associated documentation is made easier by a CASE tool, as described in Section 3.6.

As well as allowing the analyst to produce diagrams showing all the use cases in appropriate subsystems, a CASE tool should also provide facilities to maintain the repository associated with the diagram elements and to produce reports. Automatically generated reports can be merged into documents that are produced for the client organization. The behaviour specification of each use case forms part of the requirements model or requirements specification, which it is necessary to get the client to agree to.

6.6.5 Business modelling with use case diagrams

We have used use case diagrams here to model the requirements for a system. They can also be used earlier in the life of a project to model an organization and how it operates. Business modelling is sometimes used when a new business is being set up, when an existing business is being 're-engineered', or in a complex project to ensure that the business operation is correctly understood before starting to elicit the requirements.

In the examples that we have shown above, the actors have all been employees of the company interacting with what will eventually be at least in part a computerized system. In business modelling, the actors are the people and organizations outside the company, interacting with functions within the company. For example, Fig. 6.14 shows the Client as an actor and use cases that represent the functions of the business rather than functions of the computer system.

A full business model of Agate would show all the functions of the company, and the actors would be the other people and organizations with which Agate interacts, for example the media companies (TV stations and magazine and newspaper publishers), from which Agate buys advertising time and space, and the subcontractors that Agate uses to do design work and printing. There are other approaches to business modelling, the most prominent of which use process charts, which are similar to activity diagrams. Although this kind of business process modelling has been around for a few decades, it has recently acquired a new significance with the development of web services and service-oriented architectures and the idea that it is possible to model business processes and then have them automated directly

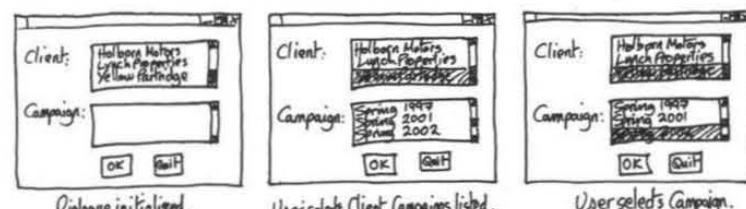


Figure 6.13 Prototype storyboard.

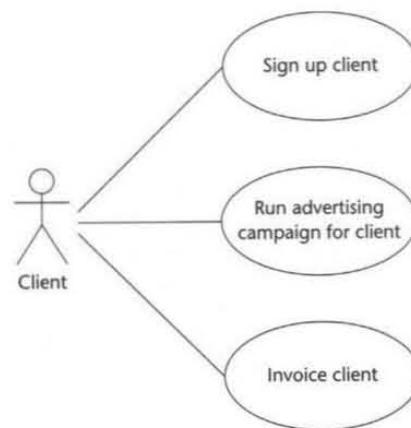


Figure 6.14 Example of business modelling with use cases.

through the use of workflow tools and services. Notations such as Business Process Modelling Notation (BPMN) and XML-based languages such as Business Process Execution Language for Web Services (BPEL4WS) have been developed as a result of this interest in business process automation.

6.7 Requirements Capture and Modelling

The first stage of most projects is one of capturing and modelling the requirements for the system. As we progress through the book, we shall include activity diagrams to illustrate the main activities in and products of each phase. These diagrams link back to the table in Fig. 5.18, which summarizes the approach that we are taking in this book. Figure 6.15 shows the first such diagram.

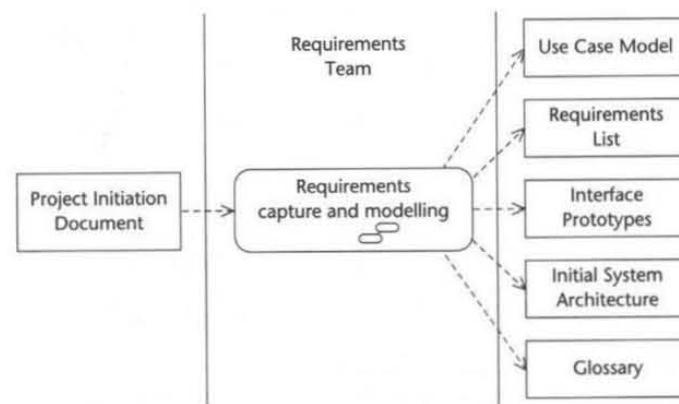


Figure 6.15 Activity diagram for Requirements capture and modelling.

In this case we have not broken the activity Requirements capture and modelling down into more detail, though it could potentially be broken down into separate activities for the capture of the requirements (interviewing, observation, etc.) and for the modelling of the requirements (use case modelling, prototyping, etc.).

We have used object flows to show the documents and models that are the inputs to and outputs from activities, and activity partitions to show the role that is responsible for the activities. In this case, one or more people in the role of Requirements Team will carry out this activity. In a small project, this may be one person, who carries out many other analysis and design activities; in a large project or organization, this may be a team of requirements analysis specialists taking more specialist roles.

The Case Study Chapter A2, which follows this one, provides more extended examples of the outputs of the Requirements capture and modelling activity.

6.8 Summary

Analysts investigating an organization's requirements for a new information system may use five main fact-finding techniques—background reading, interviews, observation, document sampling and questionnaires. They use these to gain an understanding of the current system and its operation, of the enhancements the users require to the current system and of the new requirements that users have for the new system.

Using agreed standards to document requirements allows the analysts to communicate these requirements to other professionals and to the users. Use case diagrams are one diagramming technique that is used to summarize the users' functional requirements in a high-level overview of the way that the new system will be used.

Case Study Example

You have already seen several examples from the case study in this chapter. The use cases are determined by the analyst from the documentation that is gathered from the fact-finding process. What follows is a short excerpt from an interview transcript, which has been annotated to show the points that the analyst would pick up on and use to draw the use case diagrams and produce the associated documentation. The interview is between Dave Harris, a systems analyst, and Peter Bywater, an Account manager at Agate. It is from one of the interviews with the objective 'To establish additional requirements for new system' in the fact-finding plan in the earlier case study section in this chapter.

Dave Harris: You were telling me about concept notes. What do you mean by this?

Peter Bywater: At present, when we come up with an idea for a campaign we use a word-processor to create what we call a concept note. We keep all the note files in one directory for a particular campaign, but it's often difficult to go back and find a particular one.

DH: So is this something you'd want in the new system?

PB: Yes. We need some means to enter a concept note and to find it again.
(This sounds like two possible use cases. Who are the actors?)

DH: So who would you want to be able to do this?

PB: I guess that the staff working on a campaign should be able to create a new note in the system.

DH: Only them?

(Any other actors?)

PB: Yes, only the staff actually working on a campaign.

DH: What about finding them again? Is this just to view them or could people modify them?

PB: Well, we don't change them now. We just add to them. It's important to see how a concept has developed. So we would only want to view them. But we need some easy way of browsing through them until we find the right one.

(Who are the actors for this?)

DH: Can anyone read the concept notes?

PB: Yes, any of the staff might need to have a look.

DH: Would you need any other

(Thinking ahead to Chapter 7!)

PB: Yes. It would be good to be able to give each one a title. Could we use the titles then when we browse through them? Oh, and the date, time and whoever created that concept note.

DH: Right, so you'd want to select a campaign and then see all the titles of notes that are associated with that campaign, so you could select one to view it?

(Thinking about the interaction between the user and the system.)

PB: Yes, that sounds about right

...
From this information, Dave Harris is going to be able to develop the use case descriptions for two use cases:

Create concept notes

Create concept note

The use case diagram is shown in Fig. 6.16. The use case descriptions will be as follows:

Create concept note

Create concept note
A member of staff working on a campaign can create a concept note, which records ideas, concepts and themes that will be used in an advertising campaign. The note is in text form. Each note has a title. The person who created the note, the date and time are also recorded.

Browse content notes

Browse concept notes
Any member of staff may view concept notes for a campaign. The campaign must be selected first. The titles of all notes associated with that campaign will be displayed. The user will be able to select a note and view the text on screen. Having viewed one note, others can be selected and viewed.

The interaction here is quite straightforward, so we shall not need a more detailed breakdown of the interaction between user and system.

Note that in Fig. 6.16, because *Campaign Staff* is a specialization of *Staff*, we do not need to show a communication association between the *Campaign Staff* actor and the *Browse* concept notes use case.

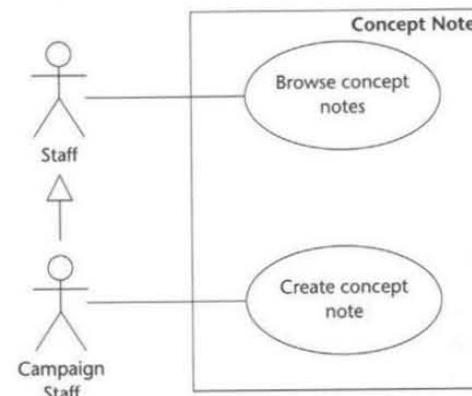


Figure 6.16 Use cases for Concept Notes subsystem

Review Questions

- 6.1** Read the following description of a requirement for FoodCo, and decide which parts of it are functional requirements and which are non-functional requirements.

The allocation of staff to production lines should be mostly automated. A process will be run once a week to carry out the allocation based on the skills and experience of operatives. Details of holidays and sick leave will also be taken into account. A first draft Allocation List will be printed off by 12.00 noon on Friday for the following week. Only staff in Production Planning will be able to amend the automatic allocation to fine-tune the list. Once the amendments have been made, the final Allocation List must be printed out by 5.00 pm. The system must be able to handle allocation of 100 operatives at present, and should be capable of expansion to handle double that number.

6.2 Name the five main fact-finding techniques and list one advantage and one disadvantage of each.

6.3 Imagine that you will be interviewing one of the three staff in Production Planning at FoodCo. Draw up a list of ten questions that you would want to ask him or her.

6.4 What is the purpose of producing use cases?

6.5 Describe in your own words the difference between the «extend» and «include» relationships in use case diagrams.

6.6 What is the difference between an ‘essential’ and a ‘real’ use case?

6.7 Write a use case description in the extended form, used for the Assign staff to work on a campaign example in Section 6.6.2, for either Create concept note or Browse concept notes.

Agate Ltd Case Study— Requirements Model

Agate Ltd

A2.1 Introduction

In this chapter we bring together the models (diagrams and supporting textual information) that constitute the Requirements Model. In Chapters 5 and 6 we have introduced the following UML diagrams:

- use case diagram
- activity diagram
- package diagram.

There is not the space in this book to produce a complete Requirements Model. However, in this chapter we have included a sample of the diagrams and other information. This is done to illustrate the kind of material that should be brought together in a Requirements Model. We have also tried to illustrate how iteration of the model will produce versions of the model that are elaborated with more detail.

A2.2 Requirements List

The Requirements List includes a column to show which use cases provide the functionality of each requirement. This Requirements List includes some use cases not in the first iteration through the use case model.

No.	Requirement	Use Case(s)
1	To record names, address and contact details for each client	Add a new client
2	To record the details of each campaign for each client. This will include the title of the campaign, planned start and finish dates, estimated costs, budgets, actual costs and dates, and the current state of completion	Add a new campaign

No.	Requirement	Use Case(s)
3	To provide information that can be used in the separate accounts system for invoicing clients for campaigns	Record completion of a campaign
4	To record payments for campaigns that are also recorded in the separate accounts system	Record client payment
5	To record which staff are working on which campaigns, including the campaign manager for each campaign	Assign staff to work on a campaign
6	To record which staff are assigned as staff contacts to clients	Assign a staff contact
7	To check on the status of campaigns and whether they are within budget	Check campaign budget
8	To allow creative staff to record notes of ideas for campaigns and adverts (concept notes)	Create concept note
9	To provide other staff with access to these concept notes	Browse concept notes
10	To record details of adverts, including the progress on their production	Add a new advert to a campaign. Record completion of an advert
11	To schedule the dates when adverts will be run	Add a new advert to a campaign
12	To maintain staff records for creative and administrative staff	Add a new member of staff
13	To maintain details of staff grades and the pay for those grades	Add a new staff grade. Change the rate for a staff grade
14	To record which staff are on which grade	Change the grade for a member of staff
15	To calculate the annual bonus for all staff	Calculate staff bonuses
16	To enable data about clients, campaigns, adverts and staff to be shared between offices	Not applicable
17	To allow the system to be modified to work in different languages	Not applicable

A2.3 Actors and Use Cases

Actor	Description
Accountant	The accountant works in the Accounts department and is responsible for the major resourcing issues for campaigns including staffing and related financial matters.
Campaign Manager	Either a Director or an Account Manager (job titles), who is responsible for estimating the campaign cost and agreeing it with the client. They are responsible for assigning staff to the team and supervising their work, managing the progress of the campaign, conducting any further budget negotiations and authorizing the final invoices.

Actor	Description
Staff Contact	Member of staff who is the contact for a particular client. They provide a first point of contact for the client when the client wants to contact Agate.
Staff	Any member of staff in Agate.
Campaign Staff	Member of staff working on a particular campaign.

Figures A2.1–A2.3 show the use cases from the first iteration, with use case descriptions in the tables.

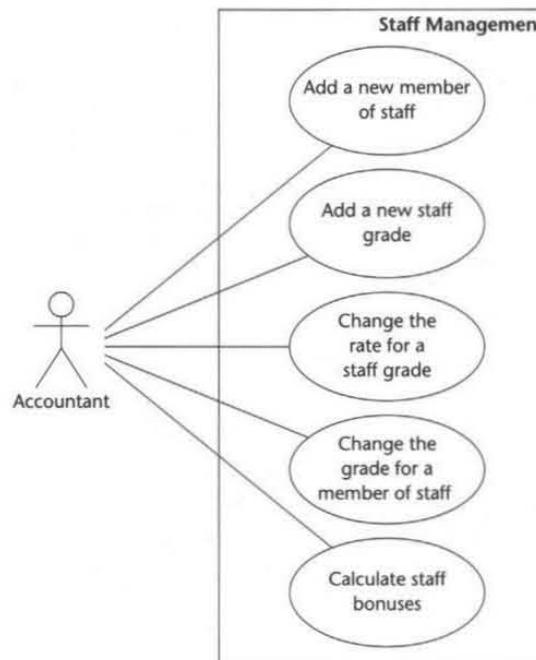


Figure A2.1 Staff Management use cases.

Use Case	Description
Add a new staff member	When a new member of staff joins Agate, his or her details are recorded. He or she is assigned a staff number, and the start date is entered. Start date defaults to today's date. The starting grade is entered.
Add a new staff grade	Occasionally a new grade for a member of staff must be added. The name of the grade is entered. At the same time, the rate for that grade and the rate start date are entered; the date defaults to today's date.
Change the rate for a staff grade	Annually the rates for grades are changed. The new rate for each grade is entered, and the rate start date set (no default). The old staff grade is retrieved and the rate finish date for that grade rate set to the day before the start of the new rate.
Change the grade for a staff member	When a member of staff is promoted, the new grade and the date on which they start on that grade are entered. The old staff grade is retrieved and the finish date set to the day before the start of the new grade.
Calculate staff bonuses	At the end of each month staff bonuses are calculated. This involves calculating the bonus due on each campaign a member of staff is working on. These are summed to give the total staff bonus.

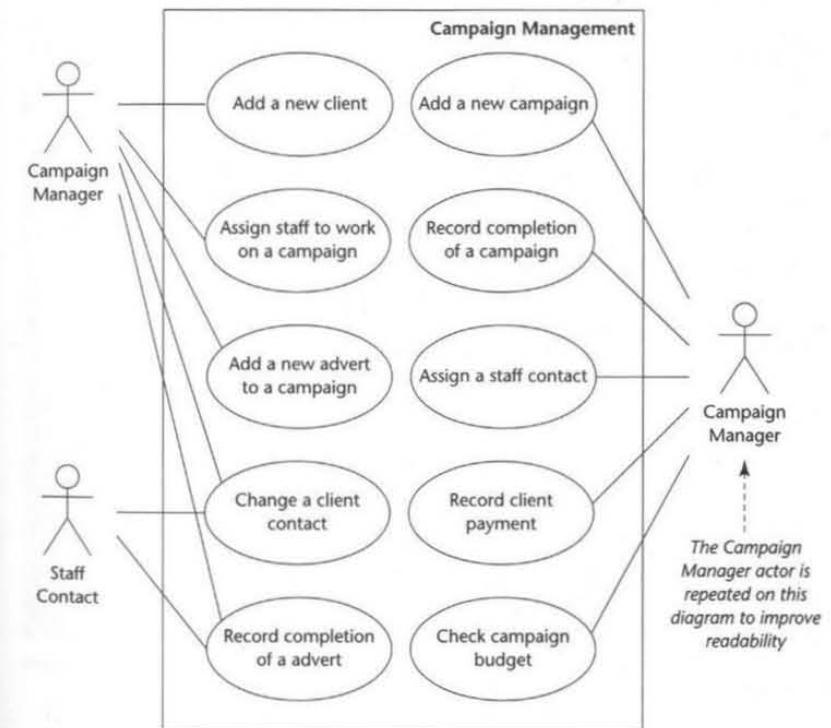


Figure A2.2 Campaign Management use cases.

The Campaign Manager actor is repeated on this diagram to improve readability

Use Case	Description
Add a new client	When Agate obtains a new client, the full details of the client are entered. Typically this will be because of a new campaign, and therefore the new campaign will be added straight away.
Assign staff to work on a campaign	The campaign manager selects a particular campaign. A list of staff not already working on that campaign is displayed, and he or she selects those to be assigned to this campaign.
Add a new advert to a campaign	A campaign can consist of many adverts. Details of each advert are entered into the system with a target completion date.
Change a client contact	Records when the client's contact person with Agate is changed.
Record completion of an advert	The actor selects the relevant client, campaign and advert. The selected advert is then completed by setting its completion date.
Add a new campaign	When Agate gets the business for a new campaign, details of the campaign are entered, including the intended finish date and the estimated cost. The manager for that campaign is the person who enters it.
Record completion of a campaign	When a campaign is completed, the actual completion date and cost are entered. A record of completion form is printed out for the Accountant as the basis for invoicing the client.
Assign a staff contact	Clients have a member of staff assigned to them as their particular contact person.
Record client payment	When a client pays for a campaign, the payment amount is checked against the actual cost and the date paid is entered.
Check campaign budget	The campaign budget may be checked to ensure that it has not been exceeded. The current campaign cost is determined by the total cost of all the adverts and the campaign overheads.

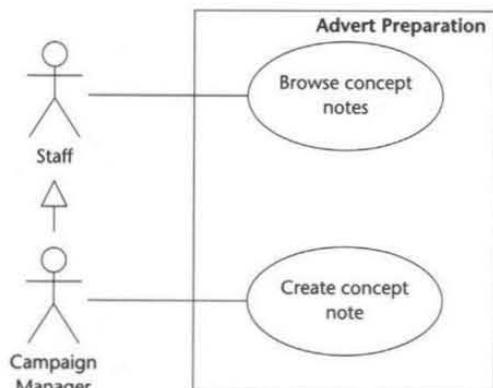


Figure A2.3 Advert Preparation use cases.

Use Case	Description
Browse concept notes	Any member of staff may view concept notes for a campaign. The campaign must be selected first. The titles of all notes associated with that campaign will be displayed. The user will be able to select a note and view the text on screen. Having viewed one note, others can be selected and viewed.
Create concept note	A member of staff working on a campaign can create a concept note, which records ideas, concepts and themes that will be used in an advertising campaign. The note is in text form. Each note has a title. The person who created the note, the date and time are also recorded.

As part of the second iteration of use case modelling, it is suggested that all the use cases that require the user to select a client, a campaign or an advert should have include relationships with use cases called Find client, Find campaign and Find advert. An example of this is shown in Fig. A2.4.

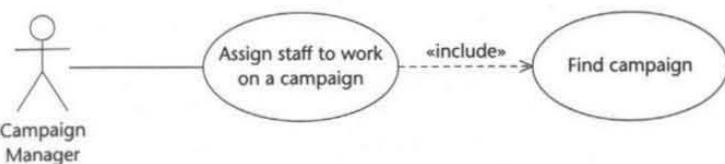


Figure A2.4 Inclusion of Find campaign use case.

In order to test out this idea, prototypes of the user interface were produced in the second iteration. The first prototypes used a separate user interface for these included use cases, as shown in Fig. A2.5.

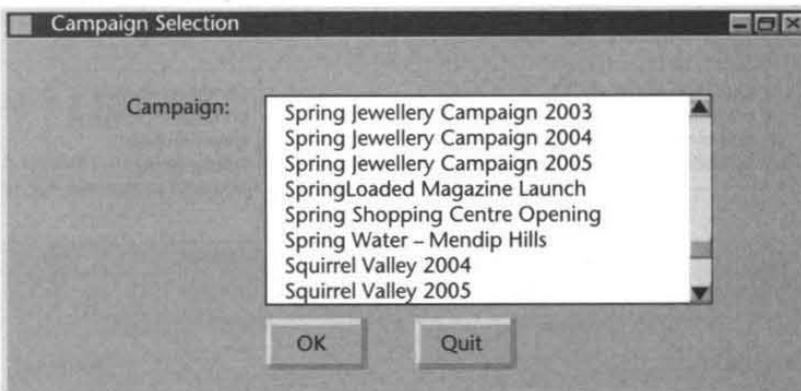


Figure A2.5 Prototype interface for the Find campaign use case.

However, feedback from the users indicated that this approach was not acceptable. They did not want to have to keep opening extra windows to find clients, campaigns and adverts. The users expressed the view that they should be able to

select these from listboxes or dropdown lists that were part of the interface for whatever use case they were in at the time.

In the third iteration of use case modelling, a set of prototypes was produced that uses listboxes. Figure A2.6 shows an example.

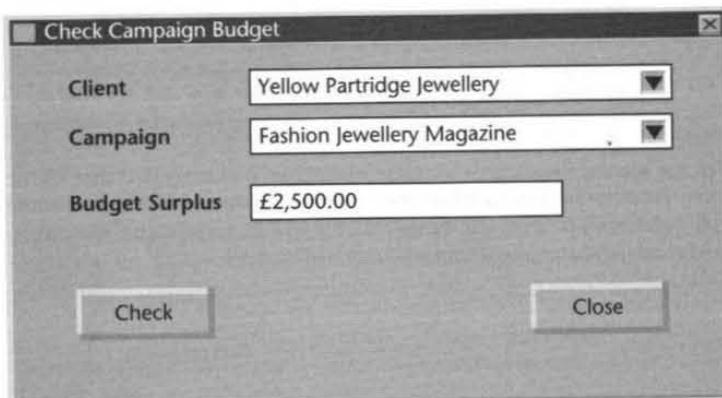


Figure A2.6 Prototype interface for the use case Check campaign budget.

In the third iteration, some additional functionality was identified and added to the use case diagrams. As an example of this, Fig. A2.7 shows the use case Check campaign budget extended by the use cases Print campaign summary and Print campaign invoice. This additional functionality will also require a change to the prototype interface in Fig. A2.6. Two additional buttons, Print Summary and Print Invoice, need to be added to the row of buttons at the bottom of the window.

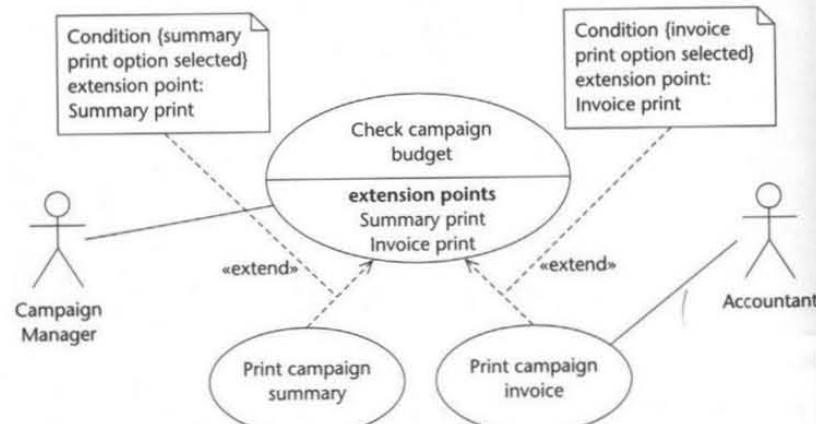


Figure A2.7 Modified use case Check campaign budget with extensions.

Also in the third iteration, the use case descriptions are elaborated to provide more detail about the interaction between the actors and the system. Two examples of these use case descriptions are provided below.

Use case description: Check campaign budget

Actor Action	System Response
1. None	2. Lists the names of all clients
3. The actor selects the client name	4. Lists the titles of all campaigns for that client
5. Selects the relevant campaign. Requests budget check	6. Displays the budget surplus for that campaign
Extensions	
After step 6, the campaign manager prints a campaign summary.	
After step 6, the campaign manager prints a campaign invoice.	

Use case description: Assign staff to work on a campaign

Actor Action	System Response
1. None	2. Displays list of client names
3. The actor selects the client name	4. Lists the titles of all campaigns for that client
5. Selects the relevant campaign	6. Displays a list of all staff members not already allocated to this campaign
7. Highlights the staff members to be assigned to this campaign	8. Presents a message confirming that staff have been allocated
Alternative Courses	
None.	

A2.4 Glossary

A glossary of terms has been drawn up, which lists the specialist terms that apply to the domain of this project—advertising campaigns.

Term	Description
Admin Staff	Staff within Agate whose role is to provide administrative support that enables the work of the creative staff to take place, for example secretaries, accounts clerks and the office manager
Advert	An advertisement designed by Agate as part of a campaign. Adverts can be for TV, cinema, websites, newspapers, magazines, advertising hoardings, brochures or leaflets. Synonym: Advertisement
Agate	An advertising agency based in Birmingham, UK, but with offices around the world. The customer for this project

Term	Description
Campaign	An advertising campaign. Adverts are organized into campaigns in order to achieve a particular objective, for example a campaign to launch a new product or service, a campaign to rebrand a company or product, or a campaign to promote an existing product in order to take market share from competitors
Campaign Staff	Member of staff working on a particular campaign
Client	A customer of Agate. A company or organization that wishes to obtain the services of Agate to develop and manage an advertising campaign, and design and produce adverts for the campaign
Concept Note	A textual note about an idea for a campaign or advert. This is where creative staff record their ideas during the process of deciding the themes of campaigns and adverts. Synonym: Note
Creative Staff	Staff with a creative role in the company, such as designers, editors and copy-writers; those who are engaged in the work of the company to develop and manage campaigns and design and produce adverts
Grade	A job grade. Each member of staff is on a particular grade, for example 'Graphic Artist 2' or 'Copywriter 1'
Grade Rate	The rate of pay for a particular grade, for example the Grade 'Graphic Artist 2' is paid £23,100 per year in the UK from 1/1/2005 to 31/12/2005
Staff	Any member of staff in Agate. Synonyms: Staff member, member of staff

A2.5 Initial Architecture

The initial architecture of the system is based on the packages into which the use cases have been grouped. These use cases have been grouped into three subsystem packages: Campaign Management, Staff Management and Advert Preparation.

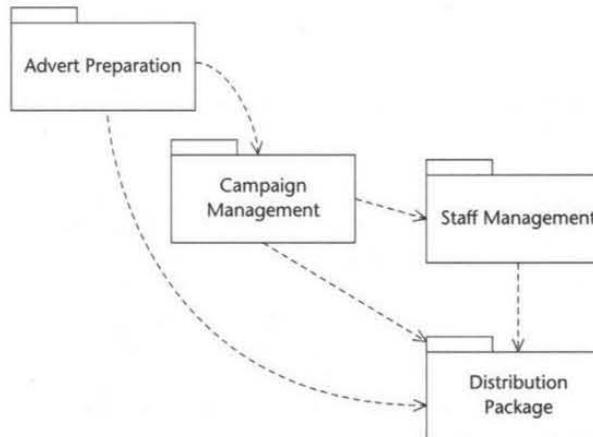


Figure A2.8 Initial package architecture.

Figure A2.8 shows the initial architecture of these three packages, and a package that will provide the mechanisms for the distribution of the application. At this early stage in the project, it is not clear what this will be, but something will be necessary to meet Requirement 16.

A2.6 Activities of Requirements Modelling

In Chapter 5, we outlined the phases and activities of the iterative lifecycle, and in Chapter 6 we included an activity diagram to show the activity Requirements capture and modelling. Figure A2.9 shows the same diagram. This activity can be broken down into other activities, and these are shown in Figs A2.10, A2.11 and A2.12.

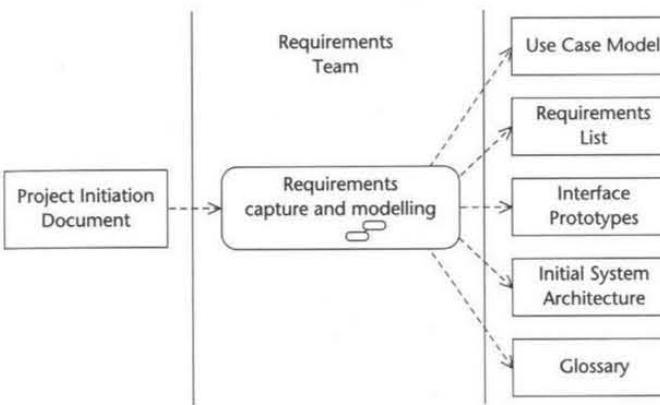


Figure A2.9 Activity diagram for Requirements capture and modelling.

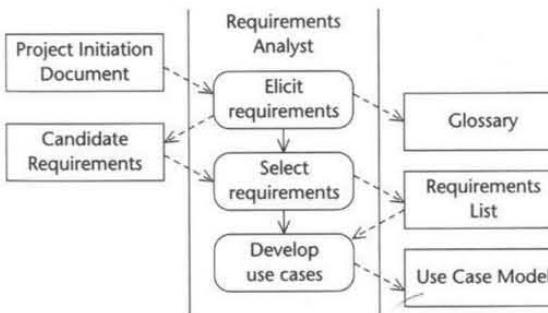


Figure A2.10 Activity diagram to show the activities involved in capturing requirements.

It is important to remember that in a project that adopts an iterative lifecycle, these activities may take place over a series of iterations. In the first iteration, the emphasis will be on requirements capture and modelling; in the second, it will shift to analysis, but some requirements capture and modelling activities may still take place. Refer back to Fig. 5.16. You may also want to look at Fig. 5.8, which illustrates the development of the use case model through successive iterations.

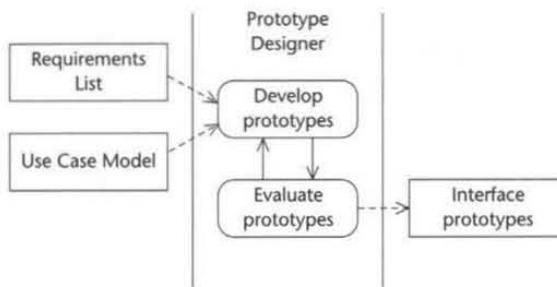


Figure A2.11 Activity diagram to show the activities involved in developing prototypes.

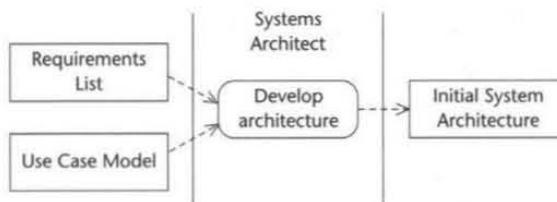


Figure A2.12 Activity diagram to show the activities in developing an initial architecture.

Requirements Analysis

Learning Objectives

In this chapter you will learn

- why we analyse requirements
- the technical terms used when working with class diagrams
- how the UML class diagram expresses a detailed model of user requirements
- how to realize use cases with communication diagrams and class diagrams
- how the CRC technique helps to identify classes and allocate their responsibilities.

7.1 Introduction

Not everyone agrees that it is necessary to build models that analyse the requirements for a new system. For example, some proponents of XP and other Agile Methods argue that most analysis documentation is unnecessary. (Chapter 21 and the additional material on the book website give more information about Agile methods.) In addition, experienced developers may follow familiar patterns and frameworks without building much in the way of tangible models. However, we believe that it is useful to analyse requirements for all but the simplest of new software systems, and this is particularly true for developers who are new to the profession. In this chapter, we describe how to produce analysis models that help to reduce errors and inconsistencies during software design and construction.

In Chapter 6 we saw how use case diagrams are used to build an initial model based on users' requirements for a new system. In this chapter we analyse those requirements in more detail. In Section 7.2, we begin by considering what a requirements model must do, and how it differs from models used at other stages in the lifecycle. Then, in Section 7.3, we discuss how to move from use cases, which are rather high level and concentrate on a user-centred view of the system, to the analysis class diagram, which is the primary model for describing the internal structure and behaviour of the proposed system. This class diagram also

forms a basis for the later design class diagram, from which the executable code will be developed. Use cases and class diagrams are linked by an activity called use case realization, which uses communication diagrams to help with the transition from use cases to class diagrams. In this section we briefly introduce the notation for collaborations and communication diagrams, but the topic is covered in more depth in Chapter 9. In Section 7.4, we illustrate the notation of the class diagram at some length, in order to show just how it helps to analyse in greater detail the model of requirements that is initially expressed as use cases. This involves elaborating the class definitions using attributes and operations, and analysing the logical structure and relationships between classes using associations. In Section 7.5, we give practical guidance on one approach to realizing a use case as an analysis class diagram. In Section 7.6, we introduce a non-UML technique called Class–Responsibility–Collaboration (CRC) cards that predates the UML by many years, but is still widely used to complement UML techniques in requirements analysis. Finally, in Section 7.7 we briefly discuss the derivation of an overall analysis class diagram from the various partial class diagrams that result from the CRC and use case realization techniques.

7.2 What Must a Requirements Model Do?

The most influential factor for the success of an IS project is whether the software product fulfils its users' requirements. Models constructed from an analysis perspective aim at determining these requirements. This means much more than just gathering and documenting facts and requests. The use case model introduced in Chapter 6 gives a perspective on many user requirements and models them in terms of what the software system can do to help the user perform their work (or play). In order to design software that will deliver this help, we must analyse the logical structure of the problem situation and the ways that its logical elements interact. We must go beneath the surface and examine the way in which different, possibly conflicting, requirements affect each other. Only by doing this can we be sure that we understand what is really happening. We must then communicate this understanding clearly and unambiguously to those who will design and build the software. This is usually done by building further models, and these must meet several objectives.

- They must contain an overall description of what the software should do.
- They must represent any people, physical things and concepts that are important to the analyst's understanding of what is going on in the application domain.
- They must show connections and interactions among these people, things and concepts.
- They must show the business situation in enough detail to evaluate possible designs.
- Ideally, they should also be organized in such a way that they will be useful later for designing the software.

The last point gives a general measure of usefulness for a requirements model. The software developed to help meet a need should have a structure that reflects that

of the situation in which the need arises. Our description of the application domain must include its conceptual structure. This is one of the conceptual foundations of the object-oriented approach, and the UML class diagram is designed to do just this. Its structure records a model of requirements that can later be directly translated into software components.

7.3 Use Case Realization

7.3.1 From use case to communication and class diagrams

To move from an initial use case ultimately to the implementation of software that adequately fulfils the requirements identified by the use case involves at least one iteration through all of the development activities, from requirements modelling to implementation. The development of an abstract model or element into another model or element that is closer to its implementation is known as *realization*, and in this section we look at the first steps in the activity that realizes a use case. We will illustrate this with the use case Add a new advert to a campaign, introduced in Fig. 6.9. Figure 7.1 repeats the use case diagram, and the series of figures that follow show some of the alternative ways that the use case can be represented, viewing it both from different perspectives and at different levels of abstraction. Some of the notation used in the diagrams may not mean much yet, but do not worry about this. It will be explained step by step later in the chapter.

Among other things, use case realization involves the identification of a possible set of classes, together with an understanding of how those classes might interact to deliver the functionality of the use case. The set of classes is known as a *collaboration*. The simplest representation of a collaboration is shown in Fig. 7.2.

You can see immediately that this tells us little except that the collaboration has a relationship with a use case. However, this can be useful in itself. In particular, the *dependency* arrow signifies that the specification of the collaboration (together with the specifications of any classes or other components that it includes) must maintain a reference to the use case. Later, we will see many other examples of dependencies between one model element and another (this notation is particularly useful when the elements are in different packages).

Figure 7.3 gives a more detailed view of the collaboration, which shows the objects that participate in the collaboration and the links between them. These are the objects that will interact, when implemented as software, in such a way as to achieve the result described by the use case. This view of the collaboration still doesn't tell us how they interact, nor does it show how they relate to other parts



Figure 7.1 Use case diagram for Add a new advert to a campaign.

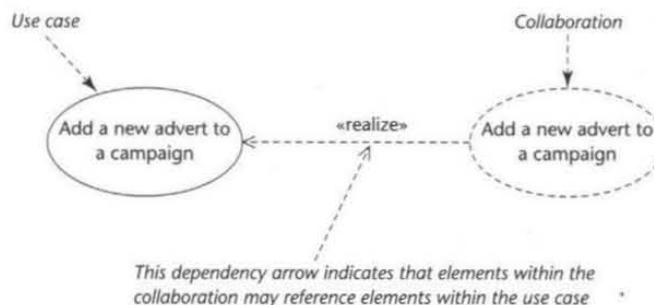


Figure 7.2 A collaboration can realize a specific use case.

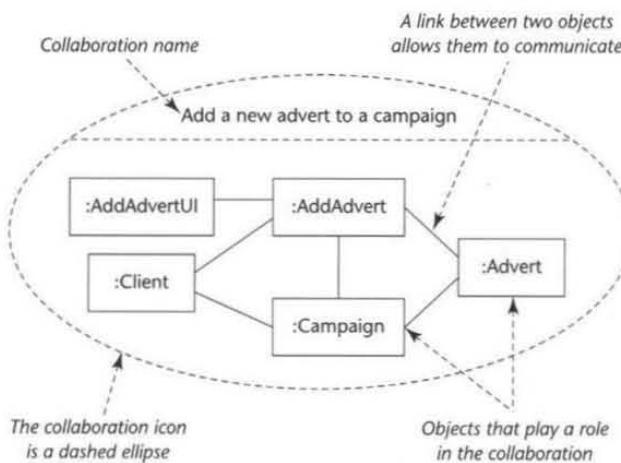


Figure 7.3 Collaboration for Add a new advert to a campaign.

of the model. All details that are not directly relevant to the collaboration are suppressed.

A *communication diagram* is one of the most useful views of the internal details of a collaboration, since it shows explicitly the interaction between objects. Figure 7.4 adds interaction to the collaboration in Fig. 7.3. Don't be too concerned about making full sense of this diagram for the moment. The important thing to notice is that it shows how the objects that take part in the collaboration may communicate with each other.

Interaction shown in a communication diagram takes the form of messages between objects, and the example shown here assumes quite a lot of understanding about the behaviour of the individual objects shown. Initial communication diagrams will not often be this detailed, as we will see later in this chapter.

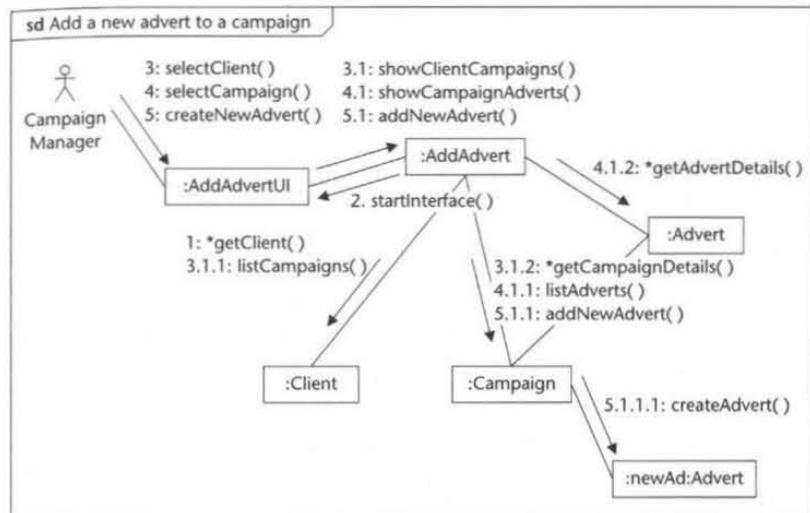


Figure 7.4 Communication diagram for Add a new advert to a campaign.

Finally (for the time being, at any rate), a collaboration can be represented as a class diagram. Figure 7.5 shows a class diagram for this example (and, during a first iteration through requirements analysis, this is as far as we need to go). This has structural and notational similarities to the collaboration in Fig. 7.3 and the communication diagram in Fig. 7.4. There is a class for each object, and some classes have associations that correspond to the links between objects.

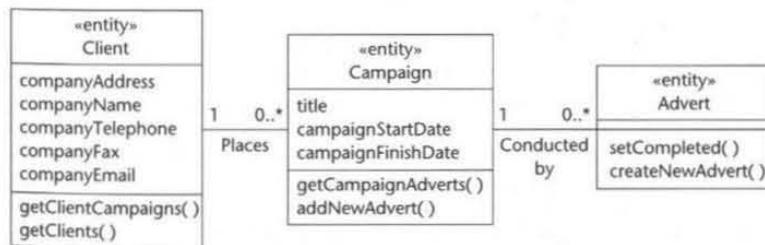
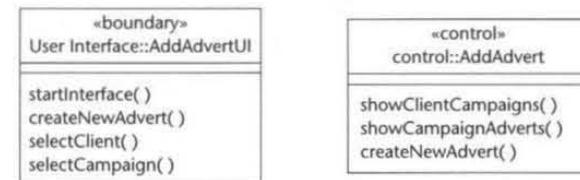


Figure 7.5 Analysis class diagram for Add a new advert to a campaign.

In this example, however, even the obvious differences are perhaps more apparent than real. For example, this class diagram shows a lot of the internal detail of the classes. Collaborations and communication diagrams can show some (though not all) of these details if this is desired. Also, some links between objects in Fig. 7.3 have no corresponding association in the class diagram. The reason for this will be explained a little later in this chapter.

At this point, it is worth saying a little more about the purposes of all these diagrams. The collaboration icon in Fig. 7.2 is in itself simply a high-level abstraction that can stand for any of the other forms. The diagrams in Figs 7.3–7.5 show some of the intermediate forms that realize a use case during the progressive and iterative development of the resulting software. Each form in this series is, in a sense, one step closer to executable code. Each also serves a particular modelling perspective. For instance, a collaboration identifies the participating objects and the links between them, but ignores messages and the internal details of objects. A communication diagram highlights interaction among a group of collaborating objects, although it can be hard to read the sequence of messages if the collaboration is complex. A class diagram ignores interaction, but shows the structure in more detail and usually shows a lot of the internal features of the classes. Collaborations can also be expressed in other ways that do not concern us so much from a requirements analysis perspective—for example, interaction sequence diagrams hide most of the structure but display the sequence of messages with greater clarity (sequence diagrams are explained in Chapter 9). For the most part, these other representations correspond to the design, test or implementation perspectives of the system model.

7.3.2 Analysis class stereotypes

Before we go any further, this is a suitable point to explain the concept of *analysis class stereotypes*. These represent particular kinds of class that will be encountered again and again during requirements modelling. (We have already encountered the idea of stereotypes in Chapter 5.)

Instances of a stereotyped class have a shared focus on certain kinds of task, which distinguishes them in a significant way from classes that are instances of another stereotype. It is often useful to identify this in our models. The everyday use of the word stereotype is not so very different. For example, if a friend says that the roles played by Arnold Schwarzenegger in his movies are quite stereotyped, you would probably understand that they think his characters are all similar to each other in certain ways, even though the plot and context may vary a lot from one film to another. Thus, if you know that Schwarzenegger is the star of a film that you are going to see, you already have some idea of what the film will be like, and also of some of the ways that it might differ from a Leonardo DiCaprio film.

UML is designed to be capable of extension, and developers can add new stereotypes where there is a clear need to do so. But this is only done when it is really necessary. We need only concern ourselves here with three widely used analysis class stereotypes: *boundary*, *control* and *entity* classes. These particular stereotypes date from the pre-UML work of Jacobson (Jacobson et al., 1992). They formed part of the notation set used in the Objectory methodology. Their use today is recommended by a number of other authors, including Rosenberg, whose approach is based partly on these stereotypes (Rosenberg and Scott, 1999).

It is not always necessary to stereotype classes. Nor, if classes are stereotyped, is it always necessary to show the stereotype of a class on diagrams. Stereotypes are shown where they add useful meaning to a model, but their use is not obligatory. In many diagrams shown later in this book, the stereotype is omitted, either because it can be assumed from the context or because it has no specific relevance to the purpose of the diagram.

Boundary classes

Boundary classes ‘model interaction between the system and its actors’ (Jacobson et al., 1999). Since they are part of the requirements model, boundary classes are relatively abstract. They do not directly represent all the different sorts of interface widget¹ that will be used in the implementation language. The design model may well do this later, but from an analysis perspective we are interested only in identifying the main logical interfaces with users and other systems. This may include interfaces with other software and also with physical devices such as printers, motors and sensors. Stereotyping these as boundary classes emphasizes that their main task is to manage the transfer of information across system boundaries. It also helps to partition the system, so that any changes to the interface or communication aspects of the system can be isolated from those parts of the system that provide the information storage or business logic.

The class `User Interface::AddAdvertUI` (shown in the communication diagram in Fig. 7.4 and in the class diagram in Fig. 7.5) is a typical boundary class. This style of writing the name shows that the class is `AddAdvertUI` (the UI is just an abbreviation for user interface) and it belongs to the `User Interface` package (the concept of packages was introduced in Chapter 5). When we write the package name in this way before the class name, it means that this class is imported from a different package than the one with which we are currently working. In this case, the current package is the `Agate` application package, which contains the application requirements model, and thus consists only of domain objects and classes.

We do not yet necessarily know what this user interface will look like or how it will behave. We may not have chosen the programming language or application package in which the software will be written, and this could restrict the choices of user interface. However, we know that some sort of interface will be needed to manage communication between the actor and the computerized information system, and we can already identify its main responsibilities, modelled here as operations. On class diagrams and communication diagrams, the stereotype of a class can be shown in different ways. Figure 7.6 shows the different symbols that are used to represent a boundary class.

Entity classes

The second analysis class stereotype is the entity² class, illustrated in Fig. 7.5 by the three classes `Client`, `Campaign` and `Advert` (note that these exist within the

¹ For instance, buttons, windows, list boxes and so on that will appear on the computer screen.

² Some readers may be tempted to confuse the concept of an entity class with the similarly named ‘entity’ in relational data modelling. There are similarities (e.g. entity classes may show the same kind of logical data structure that is revealed by relational data modelling), but there is also a crucial difference. Entity classes may have complex behaviour related to their information, whereas relational entities represent pure data structures with no behavioural aspect.

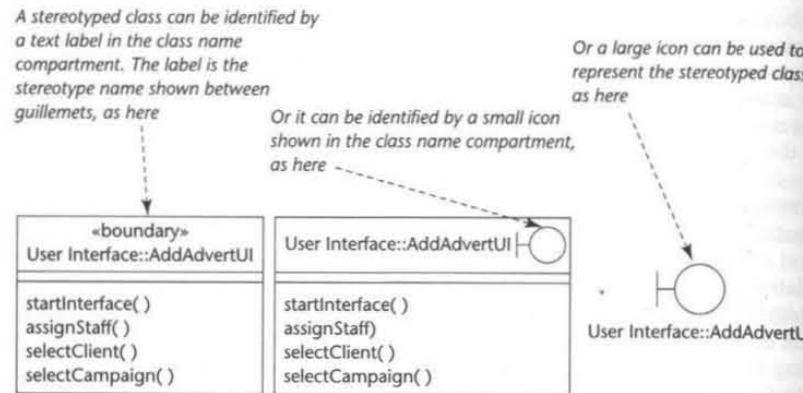


Figure 7.6 Alternative notations for the boundary class stereotype.

current package and so the package name does not need to be made explicit). Entity classes are used to model ‘information and associated behaviour of some phenomenon or concept such as an individual, a real-life object or a real-life event’ (Jacobson et al., 1999). As a general rule, entity classes represent something within the application domain, but external to the software system, about which the system must store some information. This thing might be quite abstract, for example, a campaign, or it may be quite concrete, for example a member of staff.

Instances of an entity class will often require persistent storage of information about the things that they represent. This can sometimes help to decide whether an entity class is the appropriate modelling construct. For example, an actor is often not represented as an entity class (although they *can* be when it is appropriate). This is in spite of the fact that all actors are within the application domain, external to the software system and important to its operation. But most systems have no need to store information about their users nor to model their behaviour. While there are some obvious exceptions to this (consider a system that monitors user access for security purposes), these are typically separate, specialist applications in their own right. In such a context, an actor would be modelled appropriately as an entity class, since the essential requirements for such a system would include storing information about users, monitoring their access to computer systems and tracking their actions while logged on to a network. But it is more commonly the case that the software we develop does not need to know anything about the people that use it, and so actors are not normally modelled as classes.

Figure 7.7 shows the symbols used for an entity class.

Control classes

The third of the analysis class stereotypes is the control class, illustrated in Fig. 7.5 by the class `Control::AddAdvert` (note the explicit package name). Figure 7.8 shows the symbols for a control class. Control classes ‘represent coordination, sequencing, transactions and control of other objects’ (Jacobson et al., 1999). In the USDP, as in the earlier methodology Objectory, it is generally recommended that there should be a control class for each use case (there are exceptions but

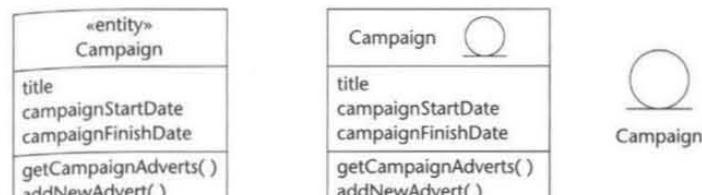


Figure 7.7 Alternative notations for an entity class.

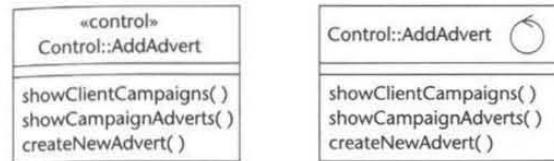


Figure 7.8 Alternative notations for a control class.

these need not concern us here). In a sense, then, the control class represents the calculation and scheduling aspects of the logic of the use case—at any rate, those parts that are *not* specific to the behaviour of a particular entity class and that *are* specific to the use case. Meanwhile the boundary class represents interaction with the user and the entity classes represent the behaviour of things in the application domain and storage of information that is directly associated with those things (possibly including some elements of calculation and scheduling).

7.4 The Class Diagram

7.4.1 Relative stability of classes and instances

Entity classes often represent the more permanent aspects of an application domain, while boundary and control classes represent relatively stable aspects of the way that the software is intended to operate. For example, as long as Agate continues to operate in the advertising business, its business activities are likely to involve campaigns, budgets and adverts, creative staff and campaign managers. And as long as the user requirements for the system do not change, the same boundary and control classes can model the operation of the software and its interaction with users. Thus the description of each class (its data and behaviour—i.e. what it knows, and what it can do) is also relatively stable, and will probably not change frequently.

By contrast, object instances often change frequently, reflecting the need for the system to maintain an up-to-date picture of a dynamic business environment. Instances are subject to three main types of change during system execution.

First, they are created. For example, when Agate undertakes a new campaign, details are stored in a new `Campaign` object. When a new member of staff is recruited, a corresponding `StaffMember` object is created.

Second, they can be destroyed. For example, after a campaign is completed and all invoices are paid, eventually there comes a time when it is no longer of interest to the company³. All information relating to the campaign is then deleted by destroying the relevant Campaign instance.

Instances of boundary and control classes are particularly volatile—that is, they have short lifetimes and are subject to frequent creation and destruction. For example, a control object and one or more boundary objects are typically instantiated at the start of each execution of a use case, and then destroyed again as soon as the execution is completed.

Finally, objects can be updated, which means a change to the recorded values of one or more characteristics. This is typically done to keep each object in step with the thing that it represents. For example, a client may increase the budget for a campaign, in order to cover a longer run of a TV commercial than was originally planned. To reflect this, the budget value set in the corresponding Campaign object must also be changed. Many entity objects are relatively long-lived in comparison with boundary and control objects, and some are updated frequently during their lifetime. Both boundary and control objects may be subject to frequent updates, but for the most part these updates record transient changes of state during the execution of the software, rather than lasting changes to stored information about the application domain.

7.4.2 Attributes

Attributes are part of the essential description of a class. They belong to the class, unlike objects, which instantiate the class. Attributes are the common structure of what a member of the class can ‘know’. Each object will have its own, possibly unique, value for each attribute (or values, if the attribute is an array).

Figure 7.9 shows some possible attributes of Client and StaffMember in the Agate case study. Note that this symbol for a class is subdivided into three compartments. The top compartment contains the class name while the second

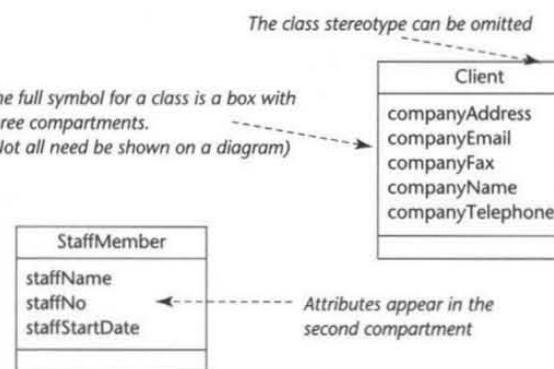


Figure 7.9 Including attributes on a class diagram.

³ For example, in the UK many financial records must be kept for six years for tax purposes before they may legally be destroyed.

contains the attribute names (which begin with a lower case letter). The third compartment remains empty for the moment.

In much the same way that a class is partly defined by its attribute structure, instances are described by the values of their attributes. To describe a client we give an appropriate value to each attribute. ‘FoodCo’ is the value given to the company name attribute for the instance of Client that represents the real-world client FoodCo. To describe an instance completely, we give a value to all its attributes, as in Fig. 7.10.

Some attribute values change during the life of an object. For example, FoodCo may change their email address, their telephone number or their address. They may even decide to change the company’s name. In each case, the corresponding attribute value should be updated in the object that represents FoodCo. Other attribute values may not change. For example, a client reference number may be assigned to each client. As long as this is correctly recorded in the first place, there is probably no need for its value ever to change.

7.4.3 Attributes and state

We now return to the idea of object state (first explained in Section 4.2.7). The current state of an object is partly described by the instance values of its attributes. When an attribute value changes, the object itself may change state. Some attribute value changes are not significant, which simply means they do not affect the behaviour of the object (i.e. the way that it responds to events), and hence of the system as a whole. Others have important implications for object and system behaviour. Changes to the state of an object affect how it responds to events, and these are modelled using statechart diagrams (see Chapter 11).

An example that may be familiar (to some people!) is when your cash card will not allow further cash withdrawals because you have reached your daily limit. We can understand this in terms of object states, as follows.

Let us say that, within the bank’s computer system, an object `yourAccount` has attributes `dayWithdrawn` and `dayLimit`, and three states: `CanWithdraw`, `Reset` and `WithdrawBarred`. (Naturally, this is greatly simplified – real bank systems are more complex.) The current value of `cardStatus` determines its state at any moment. When the object receives a `withdraw()` message, the `dayWithdrawn` and `dayLimit` attribute values are checked to see whether the withdrawal is permitted. Should `dayWithdrawn` reach the value stored in `dayLimit`, then

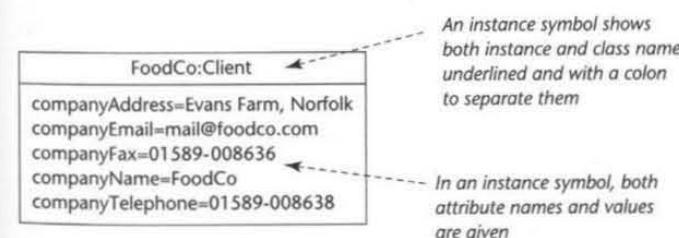


Figure 7.10 Including attribute values on an instance diagram.

`yourAccount` changes its state to `WithdrawalBarred`. Any `withdraw()` messages received while the object is in this state will now cause an error message to be displayed rather than cash being issued. At the start of each day, `dayWithdrawn` is reset to zero, returning the object to its `CanWithdraw` state. Figure 7.11 shows this as a statechart diagram.

This example is not intended as a presentation of statechart diagrams, but the notation is useful to illustrate what can happen when an object changes its state. At the instance level this is merely an update of attribute values, yet the consequences can extend beyond the boundaries of the software system into the user's daily life—as some bank customers know from bitter experience.

7.4.4 Link and association

The concepts of link and association, like those of object and class, are very closely related. A link is a logical connection between two or more objects⁴. In requirements analysis, this expresses a logical relationship in the application domain. An example for Agate is the connection between `FoodCo` and the 'World Tradition' TV campaign, described by: 'FoodCo is the client for the World Tradition campaign'. This is shown in Fig. 7.12.

Linked instances may be from different classes (as with `:Client`⁵ and `:Campaign`) or from the same class. An example of the latter might be the link `supervises` between a manager and another staff member who are both instances of `Staff Member`. Although this is uncommon, a link can even connect an instance to itself. An example might arise for the captain and players in a hockey team where the captain selects the players. Assuming that the captain is also a player, she may have a link to herself—for example 'the captain selects herself for the team'.

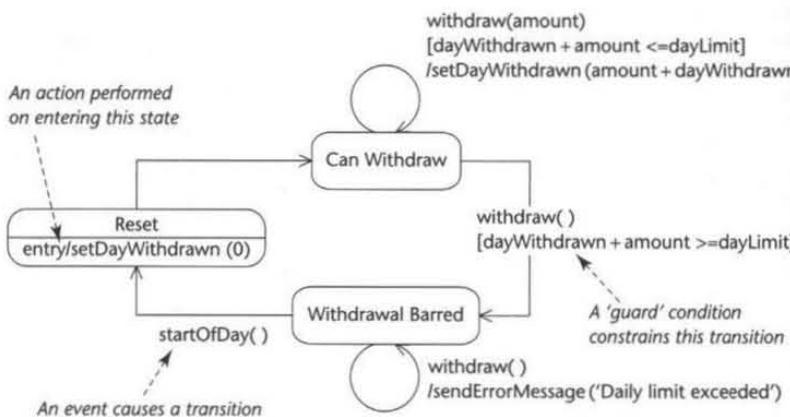


Figure 7.11 Some states and transitions for a bank account.

⁴ In most modelling situations, a link connects only two instances, as here. Rarely, a link connects three or more instances, but this is not described here.

⁵ This form of the notation refers to an anonymous instance of the class, e.g. `:Client` = any client.

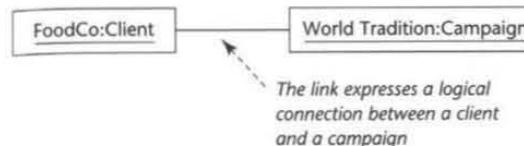


Figure 7.12 A link between instances.

Just as a link connects two instances, an association connects two classes and, just as a class describes a set of similar instances, an association describes a set of similar links (links are called *association instances* by some authors). The similarity is that those links that exist are all between objects of the associated classes. An association is a connection between two classes that represents the possibility of their instances participating in a link.

Some associations can be recognized easily without any initial awareness that there are any specific links. For example, it is probably obvious that there is an association between `Client` and `Campaign`. Staff at Agate need only record information about clients because they have won (or hope to win) business in the form of campaigns, and it is very unlikely that a campaign would be undertaken except on behalf of a specific client. Other associations are identified through the existence of links that are modelled on communication diagrams, as part of the activity of use case realization. As a general rule, wherever a link exists between two entity objects, there will be a corresponding association between their classes.

Like objects, links are not often modelled explicitly on class diagrams at this stage, although they do appear on communication diagrams.

7.4.5 Significance of associations

A link between two objects represents a real-world connection, while an association between two classes represents the possibility of links. For example, at Agate a member of staff is assigned to each client as a staff contact. This can be mirrored by links between a `:Client` and a corresponding `:StaffMember`, but a link is modelled only if it supports a specific requirement. We know from the use cases that campaign managers need to be able to assign and change a client contact. Therefore the requirements model must permit this link, otherwise it will not be possible to design software that meets these needs.

Figure 7.13 shows some links, but this is not a very economic way of modelling them. To show every link would be unnecessarily complex. There may be many staff and hundreds of clients, not to mention many thousands of links between instances of other classes.

Figure 7.14 shows an association that includes all possible links between clients and members of staff. Rather than modelling every link, this models a general ability to create a link between a member of staff and a client⁶.

7.4.6 Associations and state

We can now return briefly to the topic of object state. In Section 7.4.3 we saw that state is partly defined by an object's current attribute values. The second aspect of

⁶ But knowing that two classes have an association does not tell us which instances are linked, if any. An association is abstract and general, not particular.

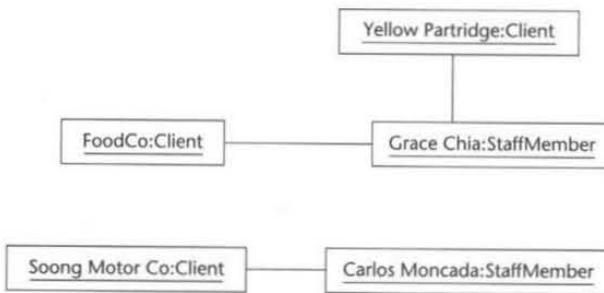


Figure 7.13 Links between instances of StaffMember and Client.

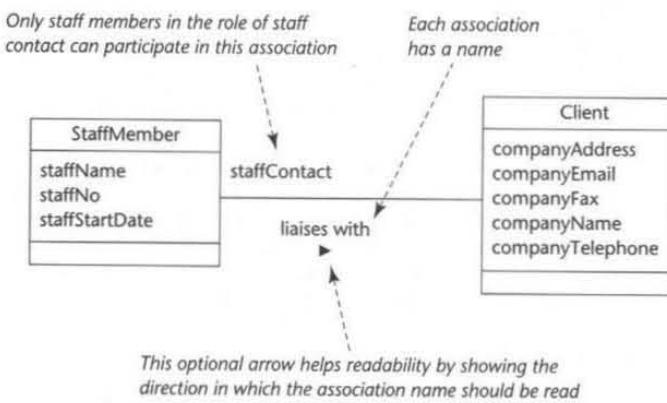


Figure 7.14 Liaises with association between StaffMember and Client.

an object's state is defined by its current set of links to other objects. Whenever a link to another object is created or destroyed, the object changes state. As with state changes that depend on attribute updates, some of these are significant and others are less so. An example of link creation is when a campaign manager assigns a member of staff to be staff contact for a new client. If, instead, an existing staff contact is changed, this involves destroying one link and creating another in its place.

7.4.7 Multiplicity

Associations represent the possible existence of links between objects. It is often important to define a limit on the number of links with objects of another specific class in which a single object can participate. Multiplicity (the range of allowed *cardinalities*) is the term used to describe constraints on the number of participating objects. Multiplicity reflects *enterprise* (or *business*) rules, which are constraints on the way that business activities can take place.

A familiar example is the number of people allowed to be the designated account-holder for a bank account. A sole account has *one and only one* accountholder, a

joint account may have *exactly two* accountholders and a business partnership account may have *two or more* accountholders. It is important to model these constraints correctly, as they may determine which operations will be permitted by the software. A badly specified system might incorrectly allow an unauthorized second person to draw money from a sole account. Alternatively, it might prevent a legitimate customer from being able to draw money from a joint account. In practice, the multiplicity of an association defines upper and lower limits on the number of other instances to which any one object may be linked.

As we saw in the previous section, it is not usually necessary for a requirements model to show the exact names of the members of staff who are staff contacts for every client. This is inappropriate, not only because it may make the diagram so complex it would become unreadable, but also because links may change frequently. By the time we reached the stage of designing our software, such a detailed model would already be out of date. But a requirements model should be able to tell us something about the variety of ways that individual instances can be linked, and in particular any constraints that apply.

We already know that in the Agate case study, each client has one member of staff assigned as *staffContact*, while each member of staff may be assigned to zero or more clients. This is modelled by adding a little more information to the simple class diagram in Fig. 7.14. Figure 7.15 shows this, with each end of the association now qualified by its multiplicity. Multiplicity must be read separately in the opposite directions for each association. Thus the possible number of clients allocated to a staff member ranges from 'zero' to 'any number', while the possible number of staff members allocated to a client must be exactly one. This type of knowledge always emerges, directly or indirectly, from users.

To be useful, the notation should be able to show all multiplicities, not just 'exactly one' and 'zero or more', and Figs 7.16–7.18 show some of the variations. These examples are not exhaustive, and there are many variations on the form, allowing any range of values to be specified, for example 0..3, 1..5, 2..10, 3..*, or discrete values, such as 3, 5 or 19, or combinations of the two, for

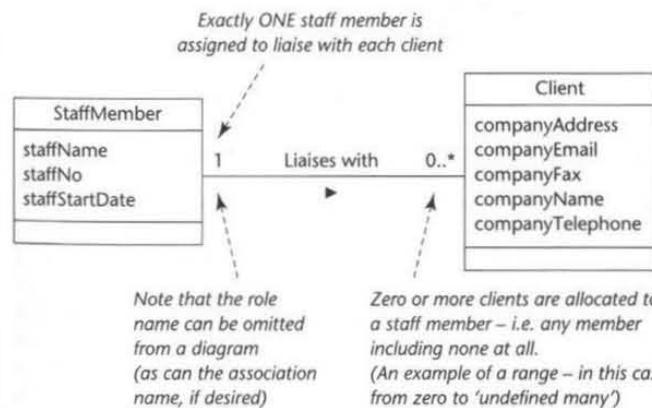


Figure 7.15 Multiplicity of the association between Client and StaffMember.



Figure 7.16 A Campaign is conducted by zero or more Adverts, while each Advert belongs to exactly one Campaign.

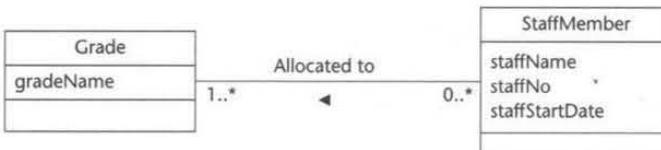


Figure 7.17 Every StaffMember must be allocated to one or more Grades, while a Grade may have zero, one or more staff allocated to it.



Figure 7.18 A Poker Hand contains up to 7 Cards. Each Card dealt must be in only one Hand (although a card may still be undealt in the pack). This assumes no cheating!

example 1, 3, 7...*. Normally, however, it is best not to restrict the multiplicity unnecessarily.

To summarize, association multiplicity conveys important information about the structure of the problem domain. Different assumptions about multiplicity have significant effects on the software design. If association multiplicities are modelled incorrectly, this may later make it impossible for the software to do things that users want it to do.

7.4.8 Operations

Operations are the elements of common behaviour shared by all instances of a class. They are actions that can be carried out by, or on, an object. The classes modelled during requirements analysis represent real-world things and concepts, so their operations can be said to represent aspects of the behaviour of the same things and concepts. However, as the basic idea of an object-oriented system is that it should consist of independent, collaborating objects, it is probably easier to understand operations as aspects of behaviour required to *simulate* the way that the application domain works (see Box 7.1). Another way of putting this is that operations are services that objects may be asked to perform by other objects. For example, in the Agate case study, StaffMember has an operation that calculates the amount of bonus pay due for a staff member. And, since staff bonus is partly based on the profit of each campaign a member of staff has worked on, Campaign has an operation to calculate the profit for each campaign.

An operation is a specification for some aspect of the behaviour of a class. Operations are thus defined for the whole class and are valid for every instance of the class. Figure 7.19 shows some examples of operations for the Agate case study. At this stage we do not need to be concerned with the details of how each operation will work, nor with when they are permitted to work or constrained from working. But we do need to know which operations to include in the requirements model, and we need to make a first guess at their appropriate location within classes. Operation names are always placed in the third compartment of the rectangular class symbol. As for attributes, operations are written beginning with a lower case letter. There is no separate notation for showing the operations of an object instance. In this respect, operations are unlike attributes and associations, in that they have exactly the same significance for instances as they do for classes.

Operations are eventually implemented by *methods*, and what a method actually does on any given occasion may be constrained by the value of object attributes and links when the method is invoked. In the bank cash card example described earlier, the object `yourAccount` has an operation `makeWithdrawal()` (this can also be written as `yourAccount.makeWithdrawal()`) and an attribute `dailyWithdrawn` (`yourAccount.withdrawnToday`). How the operation will respond at any time depends on the value of `yourAccount.withdrawnToday`. If

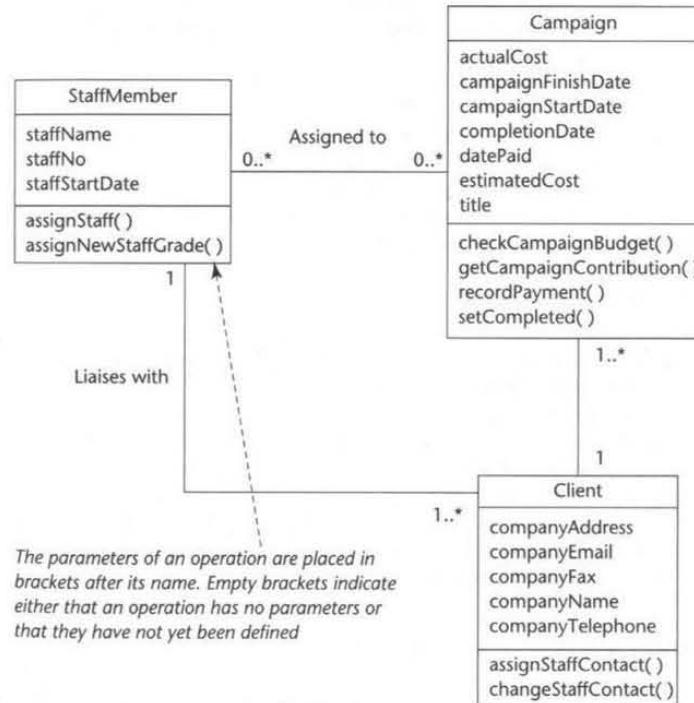


Figure 7.19 Partial class diagram for Agate, with some attributes and operations.

Box 7.1 Operations as responsibilities

There is a world of difference between saying, on the one hand, that `aStaffMember` has an operation to calculate its own bonus and, on the other hand, saying that a real member of staff has responsibility for calculating her own bonus. The former is an appropriate way of simulating the real-world operation, but there is certainly no implication of identity between the two. Nor do we wish to imply that real (but abstract) campaigns are capable of calculating their own profit—or anything else, for that

matter. What *is* implied is this. First, the ability to carry out these tasks is a requirement of the system. Second, we must make a judgement about where to locate these operations within the model. Third, our judgement should be based on the idea of object responsibilities. Later in this chapter we introduce CRC cards and in Chapter 9 we introduce interaction diagrams, two techniques that can be used to help with the allocation of responsibilities to classes in a coherent manner.

the limit has not yet been reached, a withdrawal is permitted. Otherwise, the operation may cause a message to be returned that says you have exceeded your limit. Alternative courses of action, and the basis for choosing between them, are included in the detailed definition of an operation (we describe this in Chapter 10).

The effects of an operation can include changing the characteristics of an object, for example, updating an attribute value. Another effect may be to change an object's links with other objects, for example assigning `:StaffMember` to work on `:Campaign`. (The creation of this link will be necessary for that member of staff to earn a bonus from the campaign profits.)

Some operations delegate part of their work to other objects. One example might be `aStaffMember.calculateBonus()`, since this operation must call `aCampaign.getCampaignContribution()` for each associated campaign before it can arrive at an answer. Thus each individual operation may represent only a small part of a larger task that is distributed among many objects. A relatively even distribution of computational effort is regarded as highly desirable—which is a big step on the way to building a modular system.

7.4.9 Operations and state

We have seen that state depends on attribute values. There is also a relationship between an object's operations and its states. No object can change its state except through the execution of an operation. In fact, this is just another way of saying that no attribute can store or update its own values, and no link can make or break itself. This illustrates an important point about the way that object encapsulation assists modularity in a system. The services that an object provides are only accessible through the object's interface, and this consists entirely of operation signatures, which are defined at the class level. In order to get an object to do anything at all—change its data, create or destroy links, even respond to simple queries—another object must send a message that includes a valid call on an operation.

7.5 Drawing a Class Diagram

In this section we examine the practical aspects of drawing a class diagram. This involves suggesting where to look for the necessary information, and also a

recommended sequence for carrying out the various tasks. It should be stressed that neither the information sources nor the sequence of tasks are intended to be prescriptive. Nor do we cover every eventuality on every project. In practice, one project differs greatly from another, and some steps may be omitted altogether or carried out at a completely different stage of the lifecycle. Experienced analysts will always use their own judgement on how to proceed in a given situation.

7.5.1 Identifying classes

The class diagram is fundamental to object-oriented analysis. Through successive iterations, it provides both a high-level basis for systems architecture and a low-level basis for the allocation of data and behaviour to individual classes and object instances, and ultimately for the design of the program code that implements the system. It is important to identify classes correctly. However, given the iterative nature of the object-oriented approach, it is not essential to get this right on the first attempt.

As we saw in Section 7.3.1, one of the first steps in creating a class diagram is to derive from a use case, via a collaboration (or communication diagram), those classes that participate in realizing the use case. Through further analysis, a class diagram is developed for each use case and the various use case class diagrams are then usually assembled into a single, combined analysis class diagram. This can be drawn first for a single subsystem or increment, but class diagrams can be drawn at any scale that is appropriate, from a single use case instance to a large, complex system.

In parallel with analysing the static structure shown on class diagrams, we are also interested in the dynamic interaction among classes. This, too, can be derived from the use cases and is shown on the communication diagrams. Interaction is often further explored using sequence diagrams, especially where the interaction is complex. In practice, these are often developed side-by-side with the class diagrams. In this chapter we will concentrate mainly on the notation and development of the class diagram (object interaction is covered in Chapter 9).

Identifying the objects involved in a collaboration can be difficult at first, and takes some practice before the analyst can feel really comfortable with the process. In our view, many texts—including the authoritative text on the USDP (Jacobson et al., 1999)—give little guidance that would help the novice learn how to carry out the task. These authors suggest that a collaboration (i.e. the set of classes that it comprises) can be identified directly for a use case, and that, once the classes are known, the next step is to consider the interaction among the classes and so build a communication diagram. We believe that it is usually easier to identify classes through considering their interaction together with their static structure. But it is worth stressing again that first-cut models are often tentative, and may be refined and modified more than once during later iterations.

Our starting point is the use case, and an extended version of a use case description is repeated below (for simplicity, we ignore alternative courses).

The task is to find a set of classes that can interact to realize the use case. This means, above all, thinking about those things and concepts in the application domain that are important to the goals of the system that is being developed. In this case, we know from the use case diagram that the campaign manager is the actor for this use case. The use case description tells us that when the manager

Use case description: Assign staff to work on a campaign

Actor Action	System Response
1. None	2. Displays list of client names
3. The actor (a campaign manager) selects the client name	4. Lists the titles of all campaigns for that client
5. Selects the relevant campaign	6. Displays a list of all staff members not already allocated to this campaign
7. Highlights the staff members to be assigned to this campaign	8. Presents a message confirming that staff have been allocated

selects a client name, the system should respond by displaying all campaigns for that client. When the manager then selects a campaign, the system should display all staff not yet allocated to it. The manager can select staff to assign to the campaign and the system should confirm the result. The objective of this use case is simple: to allow the manager to assign staff to a campaign.

Let's begin by picking out from the description all the important things or concepts in the application domain. A first list might include: campaign manager, client name, campaigns, client, staff. But we are only interested in those about which the system must store some information or knowledge in order to achieve its objectives. The campaign manager will be modelled as an actor. For the purposes of this particular use case, it is unlikely the system will need to encapsulate any further knowledge about the actor. We can also eliminate client name, since this is really just part of the description of a client. This leaves Client, Campaign and StaffMember in the collaboration.

Next, we can try to draw a communication diagram using these classes. This will help us to see whether they are all needed, and also whether any less obvious classes are needed as well. It will also help us to identify their structure. (Note that our use of communication diagrams in this chapter is limited to identifying classes for a use case. They are covered in greater depth in Chapter 9.)

Figure 7.20 shows an initial collaboration for this use case⁷. This is only part of the way towards becoming a communication diagram (hence the differences in both detail and structure from the more highly-developed diagram for Add advert to a campaign in Fig. 7.4. That, as we shall see over the next few pages, was the result of further analysis). So far, we have done no more than identify the participating entity objects, their classes and some probable links that might carry suitable messages.

The object symbols in a collaboration represent the lifelines of individual object instances, not classes. This is shown in the diagram by the convention of writing a colon before the class name, which indicates that this is an anonymous instance of the class, rather than the class itself.

⁷ Hand-drawn diagrams are not necessarily rough-cut or transient, although it happens that this one is both. Most UML diagrams are drawn on a CASE tool, but this does not automatically lend authority. As long as a diagram adheres to the standard, any medium is as good as any other.

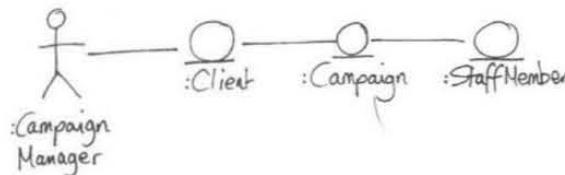


Figure 7.20 Initial collaboration for Assign staff to work on a campaign (not yet a communication diagram, as no messages are shown).

Class names are always written in the singular, although we know there are many staff, campaigns, adverts, etc. This convention reinforces the view of a class as a *descriptor* for a collection of objects, rather than the collection itself. Another convention (derived from object-oriented programming style) is that most names are written in lower case, but classes are capitalized at the beginning of the name. Multiple words are run together, punctuated by upper case letters at the start of each new word to improve readability, for example, the control class *AssignStaff*. Note, however, that use case names, such as *Assign staff to work on a campaign*, are written in sentence case, while operation names, such as *getClients*, are run together with initial capitals after the first word.

In Fig. 7.20, we have shown only entity object icons and links. This is partly because we not yet have thought about other details that will be added later. Once a class diagram has been derived from this relatively simple communication diagram, it will eventually grow quite complex. Nevertheless it is a good start at abstracting some useful details from a description that may be cluttered with many irrelevant facts. As more detail is added to the class diagram, it will soon be much easier to absorb and less ambiguous than the corresponding text—subject, of course, to the modeller's skill and comprehension. In real life, of course, even first-cut models are often more complicated than this, and a great deal more effort may be needed to arrive at a preliminary understanding.

The diagram in Fig. 7.20 does not yet show any boundary or control objects, and these must be added. It is also based on certain assumptions about how the interaction between objects would take place, and we must make these assumptions explicit and question them. The diagram implies that there will be a linear flow of messages, along the following lines. An initial message could be directed to a Client, which is assumed to know its Campaigns. Each Campaign is also assumed to know which StaffMembers are currently assigned to it, and which are not.

Although we are primarily concerned with analysis questions at present, this scenario raises some serious design issues. In particular, it effectively locates control of the use case within the client object, which would give this class responsibility for tasks that are not directly relevant to the responsibilities of a Client. The introduction of a control object would allow this responsibility to be encapsulated separately from the application domain knowledge that the entity classes represent. Figure 7.21 shows the collaboration after this refinement. Also, all links have now been routed centrally through the control object. This would mean that no entity class could find out anything directly from any other entity class. Instead, each entity object would need to ask the control object for any information it needed about an object of a different class, since the control object is the only one

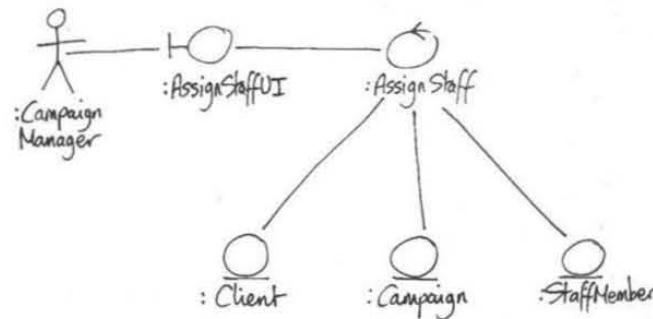


Figure 7.21 Boundary and control objects added to the collaboration, giving a different view on how the interaction might work.

with a link that would enable it to get the information. For example, the control object would need to keep track of which Client, Campaign and StaffMember objects participate in the current interaction. A boundary object has been added too; this will be responsible for the capture of input from the user and the presentation and display of results. In a limited sense, we have begun to design a software architecture that will allow a great deal of flexibility as to how the system will be implemented.

This collaboration is still drawn using the stereotype icon symbols for objects, but it could equally well be represented using rectangular object symbols, as shown in Fig. 7.22.

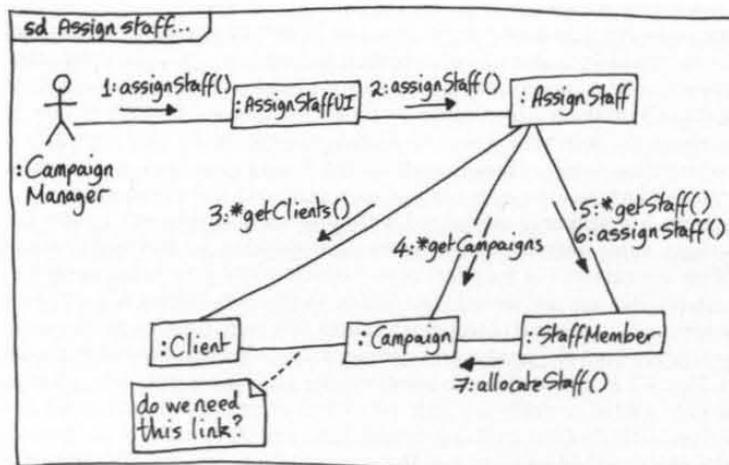


Figure 7.22 This communication diagram shows a more detailed version of the collaboration.

This version also adds the communication diagram frame, some messages and links between :Client, :Campaign and :StaffMember. These links address the question of how a Client can be responsible for knowing its own Campaigns, or how a Campaign can know the StaffMembers who are assigned to it. Taken together, the sequence of messages in Fig. 7.22 shows how the interaction might work.

We can expand this as follows. First, the user interface starts and then the control object is instantiated. The control object then obtains a list of Clients, and we can assume that it asks the boundary object to display them, although this message is not yet shown. It then asks a Client for a list of its Campaigns. The link suggests that the Client may be able to obtain some information directly from the Campaign objects themselves (perhaps which ones are associated with the Client, along with their names or status), but this message, too, is not yet shown. The control object then directly asks a selected Campaign for information about itself. Next, it asks StaffMembers for some of their details (perhaps to find out which ones are already assigned to a Campaign). At each point, since much of the information flow is routed through the control object, we can assume that it asks the boundary object to display a chunk or set of information. Finally, the control object instructs the selected StaffMember to assign itself, which it does by sending a message to the Campaign.

Figure 7.23 shows almost the same collaboration using the stereotype icon notation. Other changes are minor. All message names are omitted for clarity, but we have added messages between Client and Campaign and between Campaign and StaffMember. We have arrived at an initial judgement about how to distribute the responsibility for this use case among the various collaborating objects (this is shown by the sequence and labelling of the messages).

The final diagram (for the moment) in this set is shown in Fig. 7.24. Even this is still a simplified version of the full interaction, with many details left to be determined by further analysis. For example, we will still need to think about the

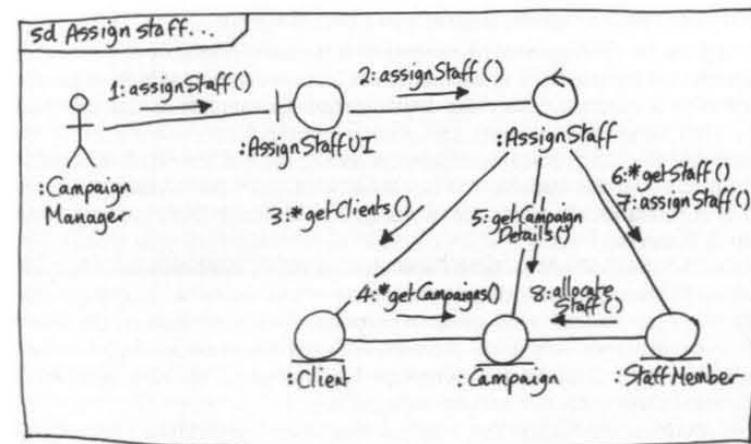


Figure 7.23 Alternative notation for a slightly refined version of the diagram.

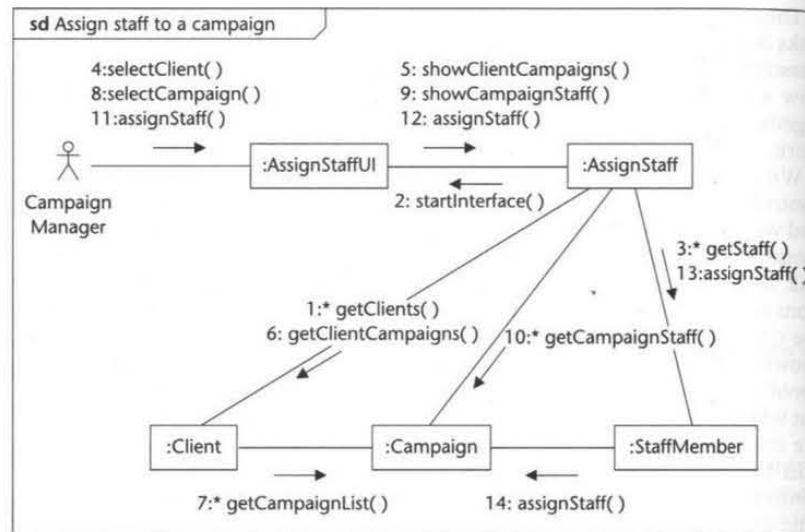


Figure 7.24 Near-final communication diagram for Assign staff to work on a campaign.

signatures of the operations that the messages will call. This will be important to understanding the interaction as a whole. But, while some issues remain to be clarified, this diagram approaches the level of understanding that we need in order to develop a robust class model capable of fully supporting the use case. But remember, too, that no decisions made at this stage are necessarily final, and we may well need to make several iterations through this activity before we achieve a full understanding.

7.5.2 From communication diagram to class diagram

The next step in the development of a requirements model is usually to produce a class diagram that corresponds to each communication diagram, which in turn is a realization of a use case. The class diagram that corresponds to the use case *Assign staff to work on a campaign* is shown in Fig. 7.25.

Provided that the communication diagrams are themselves the result of reasonably careful analysis, the transition is not usually difficult. For a start, there are some obvious similarities between the two diagrams, although there are also some important differences.

First, consider the similarities. Both show class or object symbols joined by connecting lines. In general, a class diagram has more or less the same structure as the corresponding communication diagram. In particular, both should show classes or objects of the same types. Any of the three analysis stereotype notations for a class can be used on either diagram, and stereotype labels (if used) can also be omitted from individual classes, or from an entire diagram.

Next, we examine the differences, some of which are less obvious than others. Perhaps the most obvious difference is that an actor is often shown on a communication diagram, but rarely shown on a class diagram. This is because the

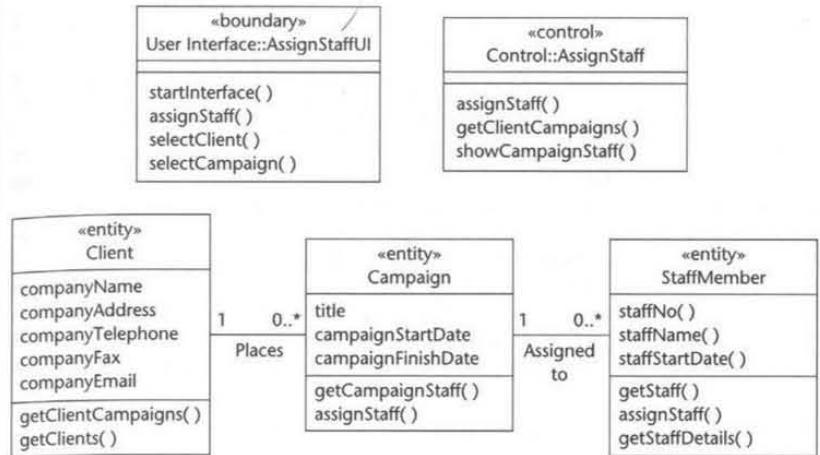


Figure 7.25 Class diagram for the use case *Assign staff to work on a campaign*.

communication diagram represents a particular interaction (for example, one that supports a specific path through a single use case) and the actor is an important part of this interaction. Actors *can* be shown on a class or object diagram when necessary, and they usually *are* if the actor is to be represented by a class—we mentioned this possibility earlier, in the discussion about entity classes in Section 7.3.2. However, a class diagram shows the more enduring structure of associations among the classes and frequently supports a number of different interactions that may represent several different use cases.

Some subtler details are associated with this change in conceptual emphasis. First, a communication diagram contains only object instances, while a class diagram (as its name suggests) usually contains only classes. This is visible (but maybe not obvious) in the names given to the class and object symbols on each diagram. A class diagram can contain instances too, but this is relatively unusual in practice. Another difference is that connections between object symbols on a communication diagram symbolize links, while connections on a class diagram stand for associations between classes. This is why some links have not been shown on the class diagram—those between AssignStaffUI and AssignStaff and between AssignStaff and Client. Transient boundary and control objects like these are created only when needed during the execution of the software, while entity objects and their links normally endure beyond one cycle of execution, and probably therefore require persistent storage. But the classes that boundary and control objects instantiate are important aspects of the requirements, and so they are still included in the class diagram. As the model develops, we can anticipate that these classes will be located in separate packages. But, since the class diagram is essentially a model of static structure, we take the view that their transient links do not need to be modelled in an analysis class diagram; hence their omission at this point. In contrast to this, a communication diagram shows the dynamic interaction of a group of objects and thus every link needed for message passing is shown. Next, the labelled arrows alongside links represent messages

between objects. On a class diagram, associations are usually labelled, but messages are not shown.

Finally, there are differences in the class and object symbols. Although any stereotype symbol can be used on either diagram, there are differences in this notation. When the rectangular box variant of the notation is used in a communication diagram it represents the lifeline of an object instance rather than a class, is normally undivided and contains only the class name (optionally, together with the object name). On a class diagram, the symbol is usually divided into three compartments that contain in turn the class name (optionally, together with its stereotype), its attributes and its operations (but all except the class name can be omitted if desired). The style of instance names also differs slightly. On a class or object diagram, instance names are underlined, while on a communication diagram instance names are not underlined (presumably because there is no need to distinguish an object from a class on a diagram that can only show objects).

7.5.3 Other approaches to finding objects and classes

So far, we have said that use cases are the best place to look for entity objects, and the best way to find them is through thinking about interactions between them that support the use case. But there are other approaches. One approach is to first develop a *domain model*—an analysis class model that is independent of any particular use cases. For example, the domain model is a significant feature of the ICONIX method (Rosenberg and Scott, 1999). In the approach that we follow in this book, the development of a domain model is considered primarily as a step that follows the development of class diagrams for each use case (see Section 7.7 and Chapter 8). However, this is largely to make it easier to understand. We do not believe that any one approach is necessarily the best for all situations (nor, indeed, do most other writers, including Jacobson and Rosenberg). Sometimes a domain model already exists, and in other cases it will make sense to produce a domain model before producing any use cases. The key to success is iterative refinement of the models, however they are produced in the first place.

It is worth reviewing any background documentation gathered during the fact-finding stage. A second reading, after an initial attempt at class modelling, can discover more classes, as a result of your clearer understanding of the problem.

Ideally, user representatives will be closely involved in discussing and developing the class diagram. Nowadays users often work alongside professional analysts as part of the project team. Most projects are a learning experience for everyone involved, so it is not unusual for users' understanding of their own business activity to grow and develop, and it is likely that users will identify a number of additional classes that were not apparent at first.

Your own intuition is another useful source, together with that of colleagues. And you can look for analysis patterns, but this is a more advanced technique that will be introduced in Chapter 8.

With experience these can all give guidance, but always check your intuitions with someone who knows the business intimately. If you let yourself be overwhelmed by similarities to other projects, you may overlook important differences. As an analyst, you should remember at all times that users are the experts on the business and on what the software system should do. Your role is to make users aware of the possibilities offered by modern information systems, and to

Category	Examples
People	Mr Harmsworth (a campaign manager), Dilip (a copywriter)
Organizations	Jones & Co (a forklift truck distributor), the Soong Motor Company, Agate's Creative Department
Structures	Team, project, campaign, assembly
Physical things	Fork-lift truck, electric drill, tube of toothpaste
Abstractions of people	Employee, supervisor, customer, client
Abstractions of physical things	Wheeled vehicle, hand tool, retail goods
Conceptual things	Campaign, employee, rule, team, project, customer, qualification
Enduring relationships between members of other categories	Sale, purchase, contract, campaign, agreement, assembly, employment

Figure 7.26 Looking for objects.

help translate their requests into a system that meets as many of their needs as possible.

However you approach the identification of classes, it helps to have a general idea of what you are looking for. Some pointers have been developed over the years that help to discriminate between likely classes and unlikely ones. Rumbaugh et al. (1991) usefully categorized the kinds of things and concepts that are more likely than others to need representation as an entity object or class.

The main categories shown in Fig. 7.26 are based on their categories. It is best to keep a list of potential classes, with a brief description for each (a rough list is fine; it will grow over time, but many items will also be crossed out and removed). When you enter your models into a CASE tool *repository* (a term for the model storage within a CASE tool environment), these textual descriptions and definitions will be an important supplement to the diagrams. Check your list carefully as it grows. Even the most experienced analyst will probably include at first some potential classes that may cause confusion later if they are retained in the model.

Next, there are some guidelines to help you to prune out unsuitable candidate classes. For each item on the list, ask yourself the following questions.

Is it beyond the scope of the system?

You may include people, things or concepts that are not strictly necessary to describe the application domain that you are investigating. Remove these from your list. They may become clear from use case descriptions or from collaborations, but do not worry if the odd one slips through. There will be lots of opportunities to catch them later on. Remember, too, that only the users can finally set the system boundary.

Beginners often include classes that represent the people who operate the current system, perhaps because their names or job titles appear in a use case description. It is rarely necessary to model the operators of the system as classes. But the exception is when they are themselves the subjects of one of the business

But the best answer at this stage is not to spend too long on making the distinction. The important thing during requirements analysis is to make sure all significant relationships are modelled, whether as classes or associations. We can review our judgements later when we understand more about the situation. Indeed, following the transition from requirements modelling to software design, it is often the case that certain types of association may be changed into classes, or further classes may be added to help implement the association effectively (this is covered in Chapter 14).

7.5.4 Adding and locating attributes

Many attributes will already appear in the use case descriptions. Others will become apparent as you think about your model in more detail. The simple rule is that attributes should be placed in the class they describe. This usually presents few problems. For example, the attributes `staffNo`, `staffName` and `staffStartDate` all clearly describe a member of staff, so should be placed in the `Staff` class.

Sometimes it is more difficult to identify the correct class for an attribute. The attribute may not properly belong to any of the classes you have already identified. An example will help to illustrate this. Consider this extract from an interview with Amarjeet Grewal (Agate Finance Director):

Amarjeet Grewal: Agate's pay structure is based on distinct grades. Directors and managers negotiate their own salaries, but other staff are placed on a grade that determines their basic salary. You can only change grade as a result of an appraisal with your line manager.

A member of staff has one grade at a time, but it sounds like they may have several previous grades, and several members of staff may be on the same grade at the same time. Staff and Grade are probably classes with an association between them.

The basic salary for each grade is fixed, usually for a year at a time. Every year after the final accounts are closed, I review the grade rates with the Managing Director, and we increase them roughly in line with inflation.

A grade has only one rate at a time, though it can change, and each rate has a money value. Grade may have a rate attribute.

If the company has performed well, we increase the rates by more than the rate of inflation. In case there are any queries, either from an employee or from the Tax Office, it is most important that we keep accurate records of every employee's grades; that is, the rates for all present and all past grades, and the dates these came into force.

There's quite a lot in this bit. A grade may have several previous rates, which suggests either that Grade has multiple rate attributes, or that Rate and Grade are distinct classes. If the latter, then Rate must have a date attribute, since we need to know when it took effect. We must also record when a member of staff changes to a grade, and possibly also when they change from a grade, which suggests one or two more date attributes. Each grade has a date it came into force—another attribute.

It's actually quite complicated, because you can have an employee who changes to several different grades, one after the other, and then the rate for each grade also changes each year. So, for each employee, I have to be able to tell exactly what grade they were on for every day they have worked for the company, and also what the rate for each grade was when they were on it. This is all quite separate from bonus, which is calculated independently each year. For creative staff, bonus is based on the profits from each campaign they have worked on, and for other staff we use an average profit figure for all campaigns.

This is necessarily tentative, but a preliminary analysis yields the following list of classes and attributes:

- classes: `StaffMember`, `Grade`, `Rate`
- attributes: `gradeStartDate`, `gradeFinishDate`, `rateStartDate`, `rateFinishDate`, `rateValue`.

In order to be reasonably lifelike, we can assume some other attributes not given above, such as `staffName` and `gradeDescription`, and also some other operations, such as `assignNewStaffGrade` and `assignLatestGradeRate`. An initial, though incomplete, class diagram might then look like the one in Fig. 7.27. One problem is where to put the attributes `gradeStartDate` and `gradeFinishDate`. These could be placed in `Grade`, but this would commit it to recording multiple start and finish dates. There may be also many members of staff associated with a grade. The computer system must be able to identify the member of staff to which each date applies, so the structure of dates that might need to be stored could grow quite complex. A similar problem occurs if date attributes are placed in `Staff`. The explanation for this difficulty is that these attributes do not describe either a member of staff or a grade in isolation. They only have meaning as a description of the link

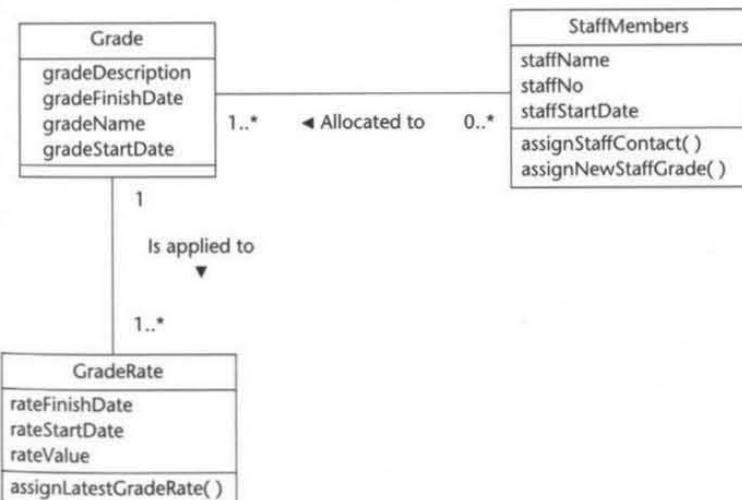


Figure 7.27 Partially completed Agate class diagram.

between a specific member of staff and a specific grade. Thus, the clearest answer is to create an additional class (called an association class) specifically to provide these attributes with a home. This is shown in Fig. 7.28.

Some readers may be familiar with the relational database technique known as normalization, a technique that provides a rigorous guide to placing attributes in tables (or relations) and ensures minimum redundancy of data. The case illustrated is an example of normalization in practice, but a full treatment of the underlying theory is beyond the scope of this book. Normalization is used in object-oriented design (this is discussed in Chapter 18) but on the whole, object-oriented approaches concentrate on capturing the structure of the world as perceived by the system's users. Unnormalized relations are therefore often acceptable in an object model, provided that they correspond accurately to users' intuitions about how their business activities are organized.

7.5.6 Adding associations

Find associations by considering logical relationships among the classes in the model. Associations may be found in use case descriptions and other text descriptions of the application domain, as stative verbs (which express a permanent or enduring relationship) or as actions that need to be remembered by the system. 'Customers are responsible for the conduct of their account' is an example of the first, while 'purchasers place orders' is an example of the second.

But this is not a very reliable way of finding associations. Some will not be mentioned at all, while others may be too easily confused with classes, attributes or

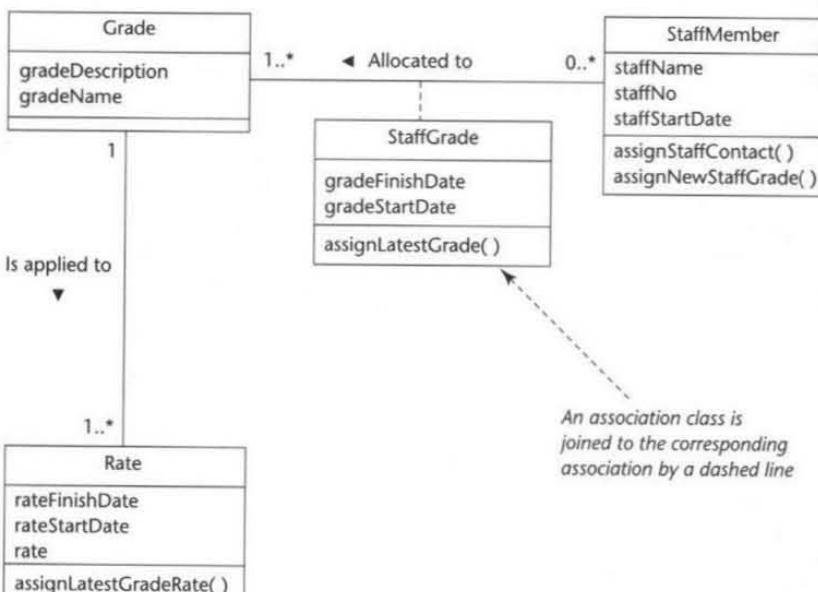


Figure 7.28 An association class gives a home to attributes that properly belong to a link between two objects.

operations. A full understanding of the associations in a class model can only be reached later by analysing the interaction between different classes. With practice, the most important ones will be found fairly easily, and for the moment it is not important if some are missed.

7.5.7 Determining multiplicity

Since association multiplicities represent constraints on the way users carry out their business activities, it is important to get these right, and the only way to do this is to question users about each association in turn. This is true even when the existence and character of the association have been inferred from user documents. An analyst should always check what is said in documents, in case it is ambiguous, erroneous or out of date.

Rosanne Martel: So let me be clear about this. A client must have exactly one staff contact, but a member of staff can be contact for no clients, one client or several clients. Is there an upper limit on that? A campaign must have one client, but a client can have many campaigns. Can you have a client with no campaigns—say, a new client who hasn't given you any business yet?

7.5.8 Finding operations

Operations are really a more detailed breakdown of the high-level system responsibilities already modelled as use cases. An operation can be thought of as a small contribution of one class to achieving the larger task represented by a whole use case. They are sometimes found as action verbs in use case descriptions, but this picture is likely to be fairly incomplete until the interaction between classes has been understood in more depth. Chapter 9 describes how to model class interaction. So, as with associations, do not worry if your first attempt has gaps.

7.5.9 Preliminary allocation of operations

Before attempting to allocate operations to specific classes, it is worth remembering that each entity class is only a representation of something in the application domain. As an analyst, you are trying to build a logical model that helps to understand the domain, not necessarily a replica that is perfect in every detail. Two guidelines help in deciding which class to locate each operation in, but there is not a single answer—only a satisfactory fit.

1. Imagine each class as an independent actor, responsible for doing or knowing certain things. For example, we might ask 'What does a staff member need to know or need to be able to do in this system?'
2. Locate each operation in the same class as the data it needs to update or access. However, this is often problematic, as you may not have identified all the attributes yet.

As a general comment on this stage, the most important thing is not to expect to get things right at the first attempt. You will always need to revise your assumptions and models as your understanding grows.

7.6 CRC (Class Responsibility Collaboration) Cards

We have recommended use cases as the starting point for the identification of classes. A further examination of the use cases also helps in identifying operations and the messages that classes need to exchange. However, it is often easier to think first in terms of the overall responsibilities of a class rather than its individual operations. A *responsibility* is a high-level description of something a class can do. It reflects the knowledge or information that is available to that class, either stored within its own attributes or requested via collaboration with other classes, and also the services that it can offer to other objects. A responsibility may correspond to one or more operations. It can be difficult to determine the most appropriate choice of responsibilities for each class as there may be many alternatives and all appear to be equally justified.

Class Responsibility Collaboration (CRC) cards provide an effective technique for exploring the possible ways of allocating responsibilities to classes and the collaborations that are necessary to fulfil the responsibilities. They were invented by Beck and Cunningham (1989) while they were working together on a Smalltalk development project. CRC cards can be used at several different stages of a project for different purposes. For example, they can be used early in a project to aid the production of an initial class diagram and to develop a shared understanding of user requirements among the members of the team. Here we concentrate on their use in modelling object interaction. The format of a typical CRC card is shown in Fig. 7.29.

CRC cards are an aid to a group role-playing activity that is often fun to do. A useful spin-off is that this can support team building and help a team identity to emerge. Index cards are used in preference to pieces of paper because of their robustness and the limitations that their size (approximately 15 cm × 8 cm) imposes on the number of responsibilities and collaborations that can be effectively allocated to each class. A class name is entered at the top of each card and responsibilities and collaborations are listed underneath as they become apparent. For the sake of clarity, each collaboration is normally listed next to the corresponding responsibility.

Wirfs-Brock et al. (1990) and others recommend the use of CRC cards to enact a system's response to particular scenarios. From a UML perspective, this corresponds to the use of CRC cards in analysing the object interaction that is triggered

Class Name:	
Responsibilities	Collaborations
<i>Responsibilities of a class are listed in this section</i>	<i>Collaborations with other classes are listed here, together with a brief description of the purpose of the collaboration</i>

Figure 7.29 Format of a CRC card.

by a particular use case scenario. The process of using CRC cards is usually structured as follows.

- Conduct a brainstorming session to identify which objects are involved in the use case.
- Allocate each object to a team member who will play the role of that object.
- Act out the use case. This involves a series of negotiations among the objects (played by team members) to explore how responsibility can be allocated and to identify how the objects can collaborate with each other.
- Identify and record any missing or redundant objects.

Before beginning a CRC session it is important that all team members are briefed on the organization of the session. Some authors (Bellin and Simone, 1997) recommend that a CRC session should be preceded by a separate exercise that identifies all the classes for that part of the application to be analysed. The team members to whom these classes are allocated can then prepare for the role-playing exercise by considering in advance a first-cut allocation of responsibilities and identification of collaborations. Others prefer to combine all four steps into a single session and perform them for each use case in turn. Whatever approach is adopted, it is important to ensure that the environment in which the sessions take place is free from interruptions and conducive to the free flow of ideas (Hicks, 1991).

During a CRC card session, there must be an explicit strategy that helps to achieve an appropriate distribution of responsibilities among the classes. One simple but effective approach is to apply the rule that each object (or role-playing team member) should be as lazy as possible, refusing to take on any additional responsibility unless persuaded to do so by its fellow objects (the other role-playing team members). During a session conducted according to this rule, each role-player identifies the object that they feel is the most appropriate to take on each responsibility and attempts to persuade that object to accept the responsibility. For each responsibility that must be allocated, one object (one of the role-players) is eventually persuaded by the weight of rational argument to accept it. This process can help to highlight missing objects that are not explicitly referred to by the use case description.

An alternative strategy is for each object to be equally keen to take on a responsibility, with the final choice determined by negotiation. Irrespective of the strategy chosen, it is important that all team members understand the need for an effective distribution of responsibilities. When responsibilities can be allocated in several different ways, it is useful to role-play each allocation separately to determine which is the most appropriate. The aim normally is to minimize the number of messages that must be passed and their complexity, while also producing class definitions that are cohesive and well focused.

We illustrate how a CRC exercise might proceed by considering the use case Add a new advert to a campaign. The use description is repeated below for ease of reference.

The campaign manager selects the required campaign for the client concerned and adds a new advert to the existing list of adverts for that campaign. The details of the advert are completed by the campaign manager.

This use case involves instances of Client, Campaign and Advert, each role played by a team member. The resulting CRC cards are shown in Fig. 7.30, and in the discussion that follows we explain how they can be derived from the use case.

The first issue is how to identify which client is involved. In order to find the correct Client the Campaign Manager (an actor and therefore outside the system boundary from the perspective of this use case) needs access to the client's name. Providing a client name and any other details for that client is clearly a responsibility of the Client object.

Next, the Campaign Manager needs a list of the campaigns that are being run for that client. This list should include the title, start date and finish date for each campaign. Although a Campaign object holds details of the campaign, it is not clear which object (and hence which class) should be responsible for providing a list of campaigns for a client. The team member playing the Campaign object argues that although it knows which Client object commissioned it, it does not know which other Campaign objects have been commissioned by the same

Class Name Client	
Responsibilities	Collaborations
Provide client information.	
Provide list of campaigns.	Campaign provides campaign details.

Class Name Campaign	
Responsibilities	Collaborations
Provide campaign information.	
Provide list of adverts.	Advert provides advert details.
Add a new advert.	Advert constructs new object.

Class Name Advert	
Responsibilities	Collaborations
Provide advert details.	
Construct adverts.	

Figure 7.30 CRC cards for the use case Add a new advert to a campaign.

Client. After some discussion, the Client object is persuaded to accept responsibility for providing a list of its campaigns and the Campaign object is persuaded that it should provide the information for this list. Once the Campaign Manager has obtained details of the campaigns for that client she requests that the Campaign object provide a list of its adverts, to which list the new advert will be added. Since the Campaign object already has responsibility for looking after the list of adverts, it is reasonable for it to add the new advert to its list. In order to do this it must collaborate with the Advert class which, by definition, has responsibility for creating a new Advert object. This completes the analysis of the use case interaction, and the new responsibilities and collaborations that have been identified are added to the cards, as shown in Fig. 7.30. We have already seen a preliminary communication diagram in Fig. 7.4 and a class diagram developed from this use case in Fig. 7.5. The reader is invited to refer back to these to see how CRC cards relate to the development of a requirements model.

During a CRC session, the team can keep track of the relationships between classes by sticking the index cards on a large board and attaching pieces of thread or string to represent collaborations. This is particularly useful when CRC cards are used early in the development cycle to help produce a class diagram. The cards and pieces of thread can be a very effective prototype of the class diagram. CRC cards can also be extended in various ways. For example, superclasses and subclasses can be shown beneath the class name and some users of the technique also like to list attributes on the back of each card.

7.7 Assembling the Analysis Class Diagram

The final step that we look at in this chapter is to assemble the various class diagrams that result from use case realization into a single analysis class diagram. This may consist of a single package of entity classes (the domain model), with boundary and control classes typically located in separate packages. With large systems, the domain model alone may comprise several distinct packages, each representing a different functional subsystem of the overall system.

There is usually little conceptual or technical difficulty in this step. All we really have to do is to place the various entity classes into a single class diagram. Where we find that we have defined the same class in different ways to meet the needs of different use cases, we simply assemble all of the operations and attributes into a single class definition. For example, consider the Campaign class as seen in relation to Add a new advert to a campaign and Assign staff to work on a campaign. Different use cases have suggested different operations. Putting these together results in a class that is capable of meeting the needs of both use cases. When we consider other use cases too, a more complete picture of the class emerges. The stages are illustrated in Fig. 7.31.

Integrating the various associations derived from different use cases can seem a little more problematic, but it is actually quite straightforward. The general rule is that if *any* use case requires an association, it should be included. Where there is an apparent conflict in the multiplicity values for an association, then clarification should be sought from users who understand the business rules of the organization.

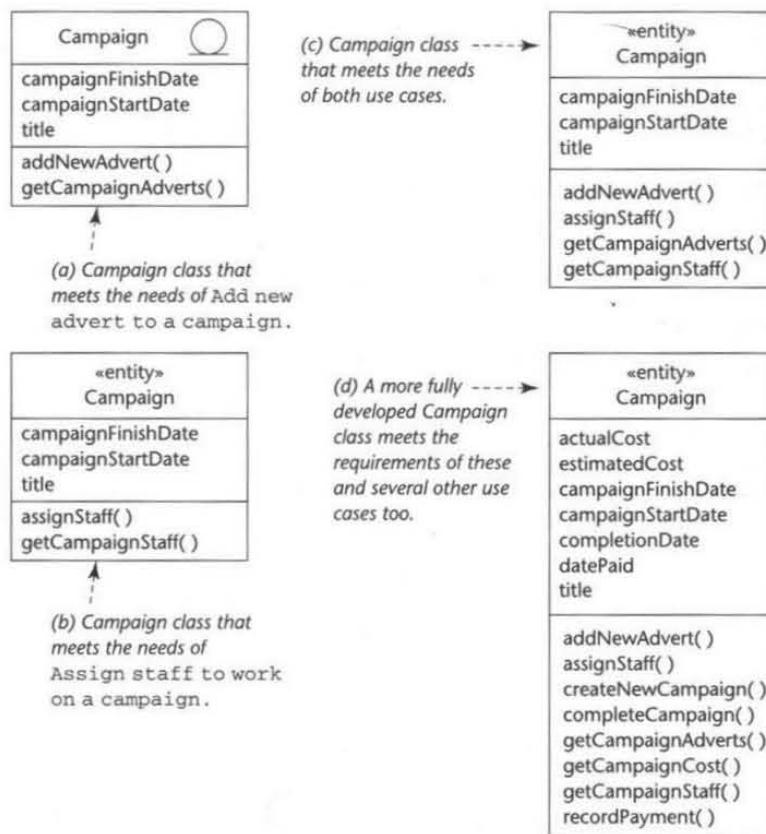


Figure 7.31 Putting together different partial definitions of a class.

Figure 7.31 illustrates the process of consolidation. A full analysis class diagram can be found in case study Chapter A3. This includes the requirements identified from many of the use cases modelled in Chapter 6.

7.8 Summary

In this chapter we have seen how to build an initial version of the class diagram for a requirements model, following the process known as use case realization. We have also seen how the CRC technique can help with the preliminary allocation of attributes and operations to classes. The important elements of the analysis model at this stage are classes, with their attributes and operations, and associations, which show the relationships between classes, together with any multiplicity constraints. Once this task has been completed to a reasonable first-cut standard, the model embodies a good understanding of the main functional requirements for the system, in terms of its responsibilities for providing services. It has also defined

a logical architecture that is a basis for the design work that follows. However, before design can be successfully undertaken, we must refine the requirements model by identifying any latent generalization or aggregation associations, together with opportunities for the application of patterns. This can simplify the structure, and may also suggest opportunities for reusing some of the requirements modelling work that has already been carried out on other projects. In the next chapter, we look at the most important concepts and techniques involved in refining the requirements model.

Review Questions

- 7.1 Explain what is meant by 'use case realization'.
- 7.2 Why are requirements models in IS development neither entirely graphical nor entirely textual?
- 7.3 Distinguish between attribute and value.
- 7.4 In what sense are classes generally more stable than their instances, and why is this usually the case?
- 7.5 Distinguish between link and association.
- 7.6 What is multiplicity, and why can it be called a constraint?
- 7.7 What is an operation?
- 7.8 How are operations related to messages?
- 7.9 What is an attribute?
- 7.10 Explain why Section 7.4.6 makes no mention of updating a link when it is changed, but instead only discusses the creation and destruction of links.
- 7.11 What is a collaboration?
- 7.12 How does a communication diagram differ from a class diagram?
- 7.13 Outline the main steps in developing a class diagram for a use case.
- 7.14 What are the advantages of team members acting the parts of objects when they are developing a set of CRC cards?

Case Study Work, Exercises and Projects

The following transcript gives the first part of an interview that Rosanne Martel conducted with Hari Patel, the Factory Manager in charge of FoodCo's Beechfield factory. Read this through carefully, and then carry out the exercises that follow.

Rosanne Martel: Hari, for the benefit of the tape, I'd be grateful if you could confirm that you're the manager responsible for all production at Beechfield.

Hari Patel: Yes, that's right.

RM: Good. Now the purpose of this interview is for me to find out about operations on the production lines. Can you tell me how this is organized?

HP: Sure. How much detail do you want?

RM: Can we start with those aspects that are common to all lines? That will give me a general feel, then if there are differences we can go into more detail later.

HP: OK, there are quite a few similarities. First, there are two main grades of shop-floor staff: operatives and supervisors. Different operatives have a range of skills, of course, but that doesn't affect the way the line works.

RM: How many operatives work on a line, and what do they actually do?

HP: There might be anything from around six operatives to over twenty, depending on the product. They really do all the actual work on the line, either by hand or operating a machine. This could be a semi-skilled labourer feeding in the different kinds of lettuce for salad packs, or a more skilled operator running one of the automatic mixing machines. In this factory, unlike Coppice and Watermead, the work is mostly quite unskilled.

RM: How many supervisors are there to each line?

HP: Just one. They are on full-time supervision duties, and they each look after one production line.

RM: Always the same line?

Rosanne is trying to find out what possible classes there are. What else do you think her questions seek to discover?

HP: Well, let's just say nobody has changed line in the last couple of years.

RM: How about the operatives—are they always on the same line too?

HP: No, we swap them around quite a bit. But it doesn't really matter what line an operative works on. They get paid piecework rates depending on the production run, and the rates are based on the job numbers that appear on their timesheets. There's a separate job number for each run.

RM: I'd like a copy of a timesheet please—preferably a real one with some data, if that's all right. We can blot out the name and staff number on the copy for confidentiality.

A sensible request. Real documents with live data are an invaluable source of information. Figure B1.4 shows the timesheet that Rosanne collected.

HP: Sure. Remind me when we finish, and I'll get you one.

RM: Thanks. Now, does one line always produce the same product?

HP: No, that changes from one day to the next. The production planners produce a new schedule every Friday, and this lists all the production runs for each line for the following week.

RM: I'll take a copy of a production schedule too, please. So the supervisor finds out on Friday what their line is working on over the next week?

Here Rosanne is checking where the inputs come from, as well as what they contain.

HP: That's right.

RM: Good, I think I've got that clear. Now let's talk about what happens when people come in to work. Do all the lines start up first thing in the morning?

HP: Usually. Production runs generally last for a whole day if possible, or sometimes a half-day. Production Planning try to keep the change-overs simple, so they tend to schedule changes during breaks to avoid wasting productive time.

RM: The lines don't keep running all the time?

HP: No, they stop for coffee and meal breaks.

RM: What role does the line supervisor play in this?

HP: Well, they make sure the lines have enough raw materials, and they deal with minor emergencies. They also monitor output, liaise with production control, keep track of employee absences, and so on.

RM: Can we go through what a supervisor does on a typical run, please, step-by-step?

Another sensible request. Asking someone to go over things again in more detail will often reveal aspects of the situation that are not obvious from a brief description.

HP: First, they make sure everything is ready before the run starts. They check the storage area to see there is enough of each ingredient. If a long run is planned, you don't need all the ingredients ready at the beginning, but there has to be enough to keep the line running smoothly until the next supply drop. They also have to check if the staff allocated to that run have turned up. A line can usually run for a little while with one or two staff missing, but it's best to have everyone there from the start.

RM: How does a supervisor know what ingredients are required, and how many staff?

A good analyst always probes to find out how, what, why, when, where and who.

HP: Every run has a job card, with this information on it. The warehouse gets a copy of the job card too, so in theory they know what supplies to deliver, to which line and when they will be needed.

RM: Does that usually work?

HP: (Laughs) Sometimes!

RM: What if there aren't enough staff?

HP: Sometimes the supervisor can find a spare body on another line. Or they can run the line slower. You can manage with fewer staff if necessary, but productivity is a lot lower.

RM: Let's say the ingredients are all ready, and all the staff are there waiting to go. What next?

HP: The supervisor switches on the line, and then it's mostly troubleshooting and paperwork.

RM: What does the paperwork involve?

HP: Well, they start by taking the names of all the staff at the start of the run. They copy the job number from the job card to the production record sheet and all the timesheets. If it is the first time that operative has worked that week, then the supervisor makes out a new timesheet. When they start the line, they note the time on the production record sheet. Then they keep a rough note of anyone who leaves the line during a run, and how long they're absent.

RM: What kind of problems does the supervisor deal with?

HP: The main problem is if something goes wrong with the run. Say the line breaks down. They would have to call in maintenance, record the downtime while the line's not running, and try to find useful things for the staff to do while they're waiting for it to be repaired. If an ingredient runs out this could also halt the line, and might mean chasing the warehouse, or contacting the farm or an outside supplier. Sometimes people go missing, or leave early because they're sick. The supervisor has to find a replacement as quickly as possible.

RM: Right, now let's go to the end of a run. What information is formally recorded, and by whom?

HP: First the supervisor notes the finish time on the production record sheet.

RM: I'll have one of those too, please.

HP: OK, no problem.

Figure B1.3 shows a blank production record sheet.

HP: Next the supervisor phones for someone to come over from Production Control to verify the quantity produced and note this on the production record sheet. Then the supervisor totals all the absences, because if anyone has more than 15 minutes absence, it's deducted from their total unless they have a good reason, say a medical certificate. Then they work out the total hours for each operative. If someone joined the line in mid-session they might not have a timesheet, so one is made out now and their hours are added in. By the time all that has been done, Production Control has usually checked out the total quantity produced, and this goes on the production record sheet. After that, it's just returning unused ingredients to the warehouse, tidying up the line ready for the next run, that kind of thing.

RM: Thanks, that was really helpful. Now I'd like to ask about how the piecework formula works. Can you tell me what the calculation is?

HP: To be honest, I can never remember the exact formula. You'd do better asking a supervisor or someone from payroll ...

Now carry out the following exercises, based on the information given in the interview transcript.

7.A Write descriptions for the following use cases:

- Start line run
- Record employee joining the line
- Record employee leaving the line
- Stop line
- Record line problem
- End line run

7.B From your use case descriptions, produce communication diagrams and then class diagrams.

7.C Produce a draft analysis class diagram, initially showing only classes and associations.

7.D Review your analysis class diagram together with the various intermediate models, and add any attributes and operations that you think are justified by your use cases. Make reasonable assumptions and add others that you think might be justified by other use cases not directly derived from the transcript.

Further Reading

The natural source for all readers interested in this subject is the 'Three Amigos' book on the USDP (Jacobson et al., 1999). But this text, while authoritative, is not (in our view, at any rate) ideally suited to the novice requirements analyst.

Rosenberg (1999) describes in a very accessible way a process that uses UML in object-oriented requirements modelling. Rosenberg's process differs in many respects from the one followed in this book, but is very much in sympathy with our aim of producing a robust class model.

Larman (2005) also describes a process for using UML in object-oriented requirements modelling. Larman's approach is very different again from the one taken in this book, and also from that recommended by Rosenberg.

Agate Ltd Case Study—Requirements Analysis

Agate Ltd

A3.1 Introduction

In this chapter we analyse the Requirements Model described in Chapter A2 and produce a number of use case realizations. The activities involved in use case realization are described in Chapter 7 and involve the production of the following UML diagrams:

- communication diagrams
- class diagrams that realize individual use cases
- analysis class model.

Use cases are initially analysed as collaborations and as communication diagrams. This helps to identify classes involved in their realization. After individual use case realizations have been developed, a combined analysis class model is produced from them. A more detailed analysis class diagram is also included to indicate how the model develops as the use cases are analysed.

A3.2 Use Case Realizations

The first use case analysed here is Add a new campaign (all the use cases are specified in Chapter A2). Figure A3.1 shows a collaboration that realizes the use case. Figure A3.2 shows the communication diagram, with boundary and control classes added.

Note that the initiation of the dialogue described by the use case is not modelled explicitly (apart from the inclusion of a startInterface message). Details such as this will be added later for this system, though in some projects it may be important to model them early on. The class diagram that supports this use case (and its collaboration) is shown in Fig. A3.3. Notice that the class Campaign includes only attributes that are required for the use case. The requirements analyst

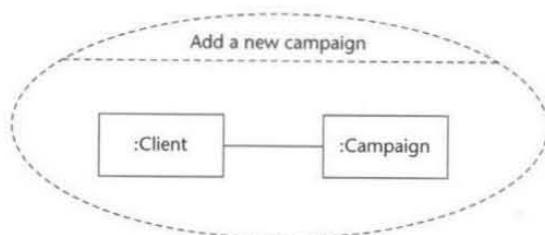


Figure A3.1 Collaboration for the use case Add a new campaign.

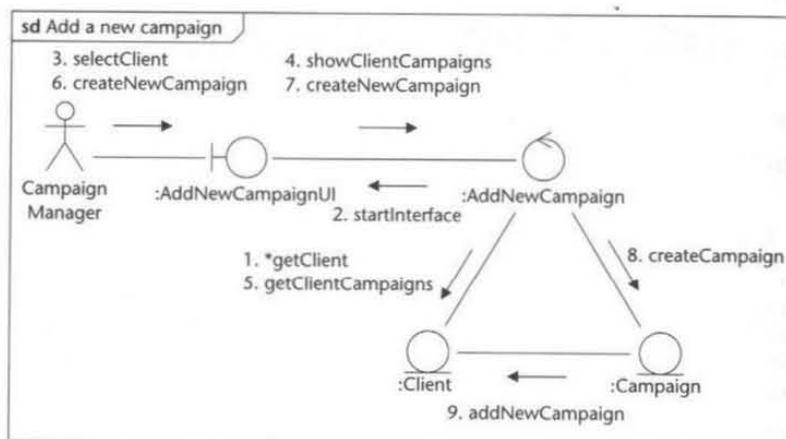


Figure A3.2 Communication diagram for the use case Add a new campaign.

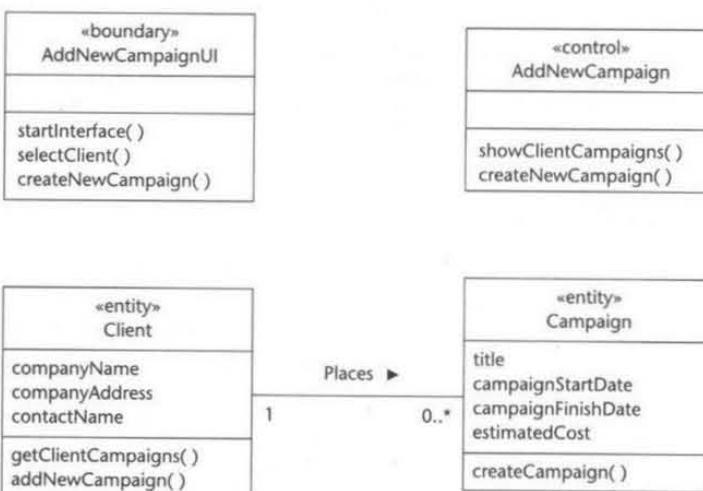


Figure A3.3 Class diagram for the use case Add a new campaign.

may identify the need for additional attributes (or functionality) while the use case is being analysed, but it is important to confirm any changes with the stakeholders. In these models we have named the constructor operation `createCampaign` to make it clear where in the interaction a new campaign object is created. If we were preparing a design model, the naming conventions used in object-oriented programming languages would be more appropriate.

Figures A3.4 to A3.12 show the development of the use case realizations for the use cases Assign staff contact, Check campaign budget and Record completion of a campaign. The use case Record completion of a campaign involves the production of a completion note. The boundary class Completed CampaignPI (we use the suffix PI to stand for Printer Interface) is responsible for printing the completion note.

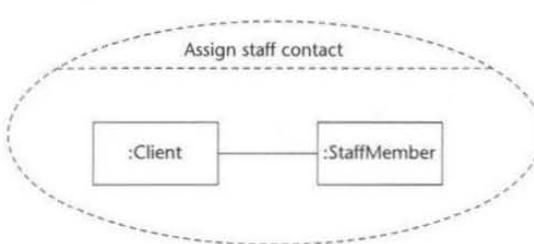


Figure A3.4 Collaboration for the use case Assign staff contact.

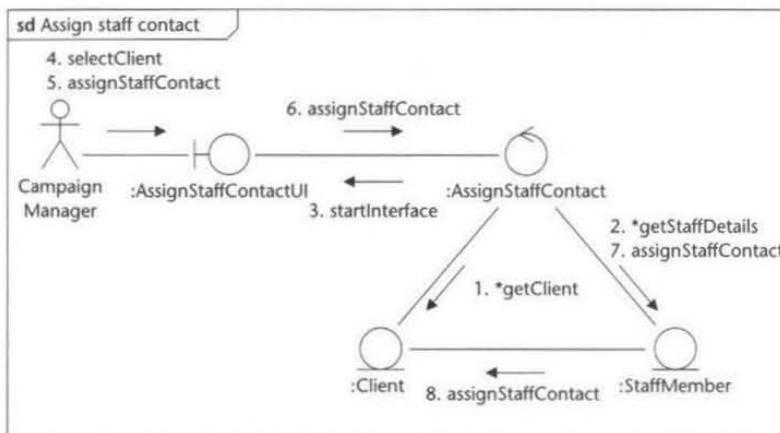


Figure A3.5 Communication diagram for the use case Assign staff contact.

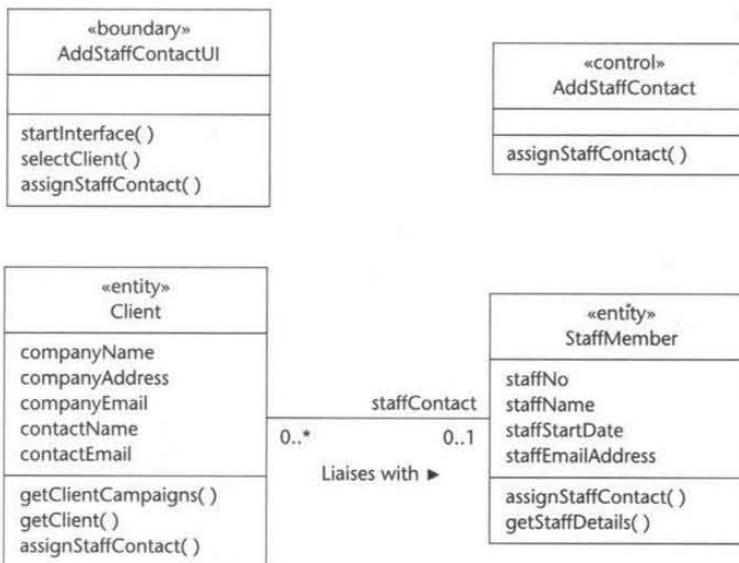


Figure A3.6 Class diagram for the use case Assign staff contact.

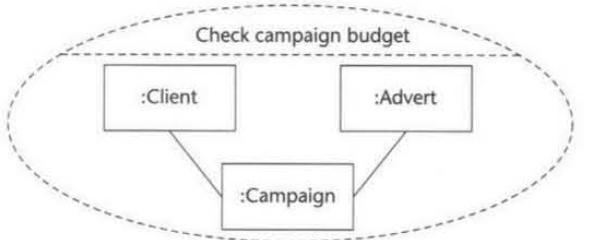


Figure A3.7 Collaboration for the use case Check campaign budget.

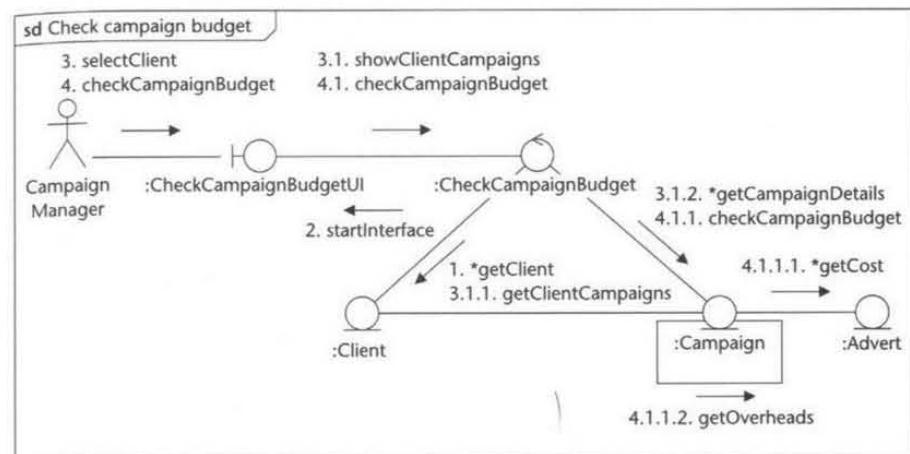


Figure A3.8 Communication diagram for the use case Check campaign budget.

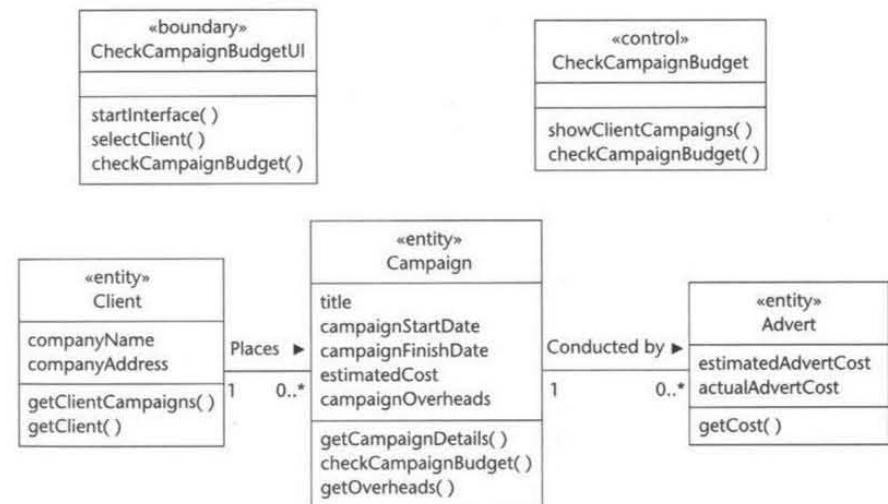


Figure A3.9 Class diagram for the use case Check campaign budget.

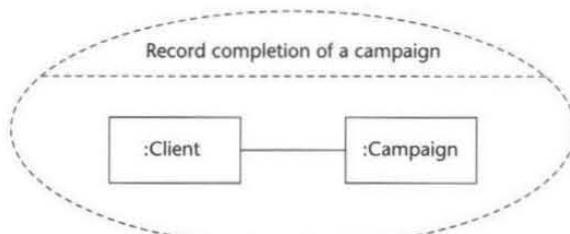


Figure A3.10 Collaboration for the use case Record completion of a campaign.

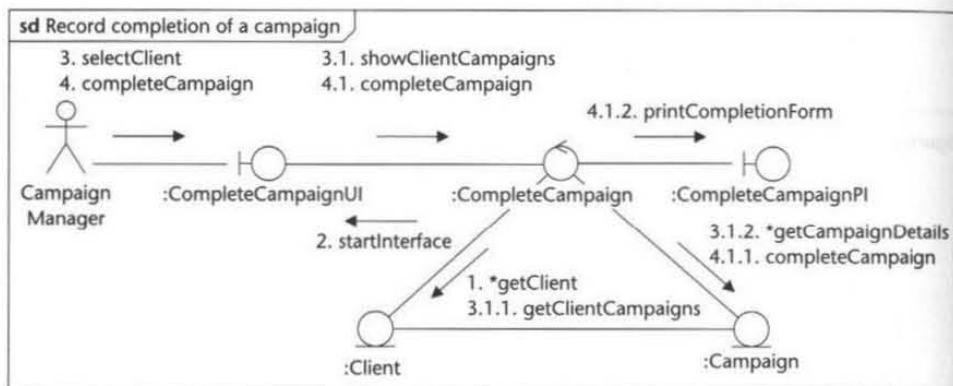


Figure A3.11 Communication diagram for the use case Record completion of campaign.

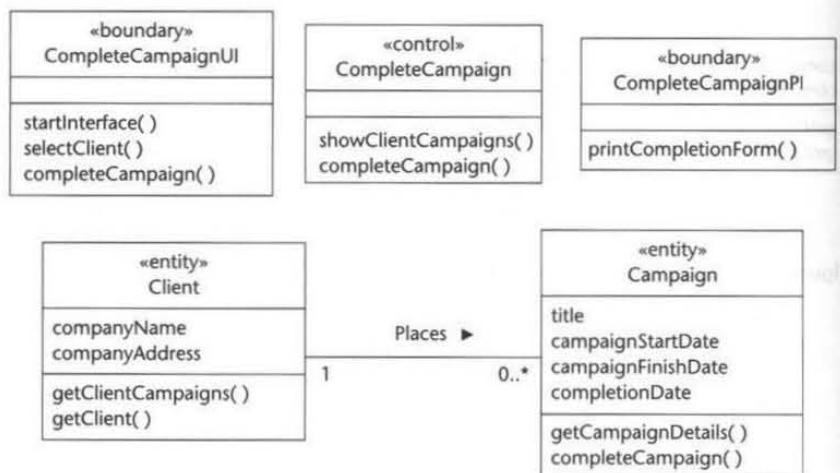


Figure A3.12 Class diagram for the use case Record completion of campaign.

A3.3 Assembling the Analysis Class Diagram

The class diagram in Fig. A3.13 has been assembled from the realizations for Add a new campaign, Assign staff contact, Check campaign budget and Record completion of a campaign. Figure A3.14 shows a more fully developed class diagram that includes classes, attributes, operations and associations that have been identified from the other use cases in the Campaign Management package. This illustrates how a more detailed and complete picture of the analysis model is developed as the use cases are analysed. The use cases Add a new advert to a campaign and Assign staff to work on a campaign are analysed in Chapter 7. Their realizations are shown in Figs 7.5 and 7.25 respectively.

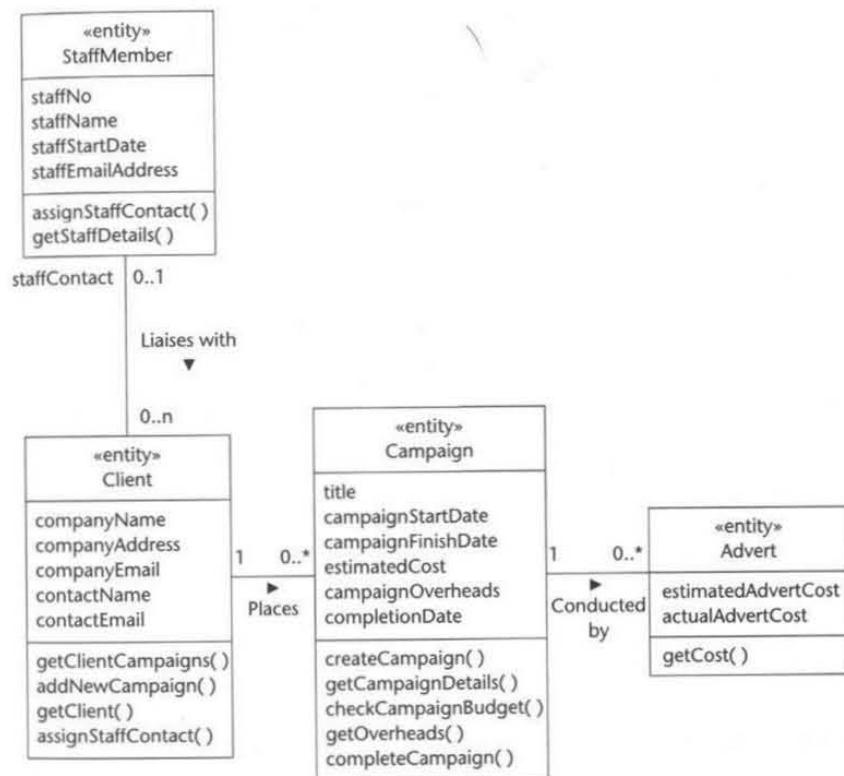


Figure A3.13 Combined class diagram for four use cases.

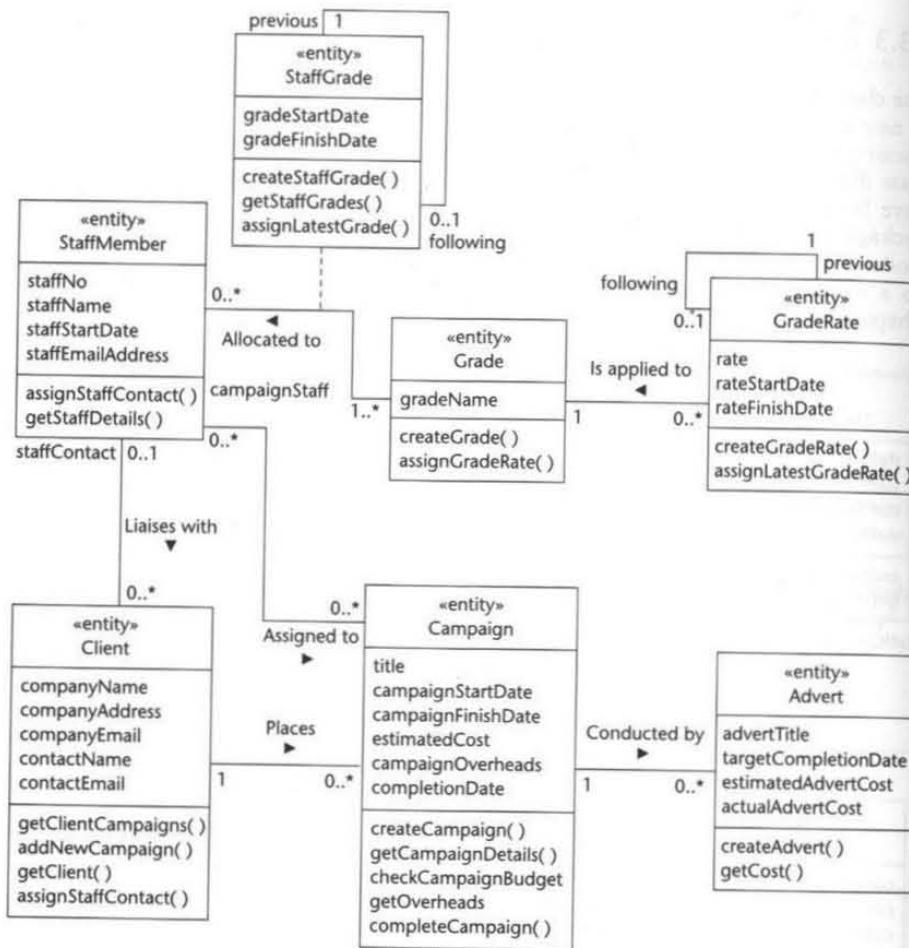


Figure A3.14 Combined class diagram after further requirements analysis.

A3.4 Activities of Requirements Analysis

Figure A3.15 shows an activity diagram that illustrates the relationship between the requirements models and the products of use case realization. The activity diagram in Fig. A3.16 shows the main activities involved in requirements analysis.

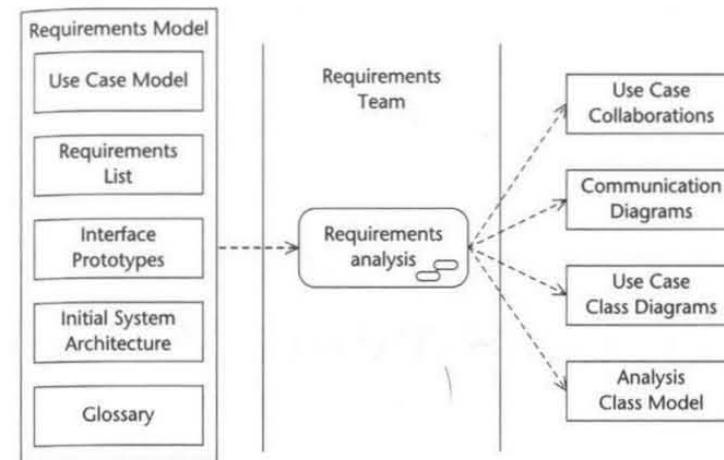


Figure A3.15 High-level activity diagram for Requirements analysis.

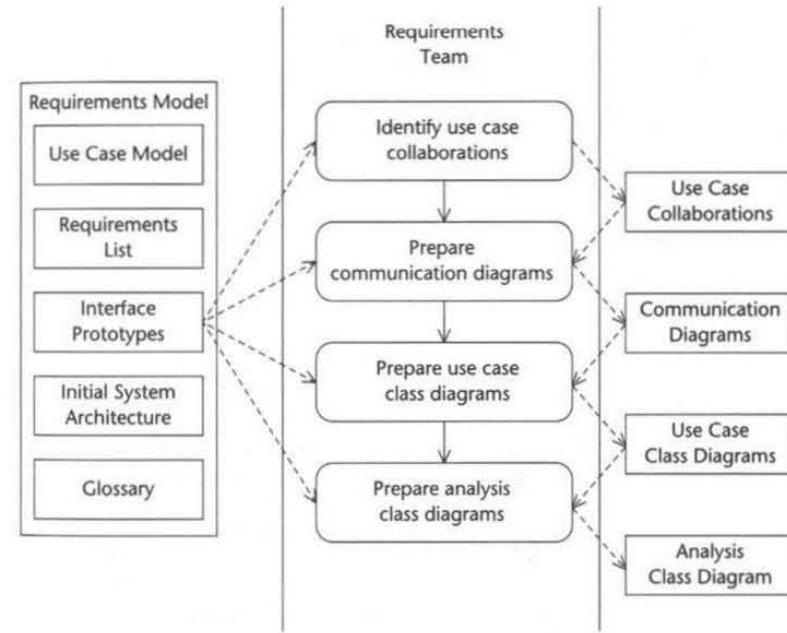


Figure A3.16 Activity diagram describing analysis use case realization.

CHAPTER 8

Refining the Requirements Model

Learning Objectives

In this chapter you will learn

- the significance of reuse in software development
- how object-oriented principles contribute to reuse
- how to identify and model generalization and composition
- one approach to modelling reusable components
- what the term *pattern* means in software development
- how analysis patterns can be used to help structure a model.

8.1 | Introduction

Once the requirements have been identified and documented in a single analysis model that shows all relevant classes and associations, we should refine and structure this model. This is done with a view to creating the conditions for reuse: either we may be able to reuse existing specifications or software originally written for an earlier system, or the new specifications and software that we create may be useful for other systems in the future. In general, reuse is achieved either through abstraction or through the encapsulation of composite structures or components. In this chapter we explain the role of abstraction and introduce the idea of a reusable component.

The principal abstraction mechanisms in UML are generalization and encapsulation (both explained in Chapter 4). In turn, encapsulation implies the use of interfaces, behind which the internal details of an object, structure or component are hidden. UML 2.0 includes new notation for composite structures, components and the interfaces that they present to other parts of the overall system. We introduce and explain the main elements in this chapter. We also show how these and other refinements can be used to encourage reuse and to build software

components. We introduce one approach to component-based development and illustrate the thinking behind this and some of the models used.

In practice, most benefits of reuse are gained during the design and construction activities, rather than during requirements analysis. Software and specification reuse requires careful management, and we discuss this further in later chapters (particularly in Chapter 20). Here we concentrate on laying the groundwork.

Another approach to abstraction is found in the 'Patterns' movement, which in recent years has offered new ways of capturing and communicating generalizable knowledge, particularly during analysis and design activities. We introduce and explain the concept of patterns and illustrate this with some examples of analysis patterns. Later (in Chapters 12 and 15 respectively), we will discuss architecture and design patterns.

8.2 | Software and Specification Reuse

Over the last few years, a small revolution has been occurring in the software development world. For many people who work in the sector, the emphasis has shifted from the development of entire new software systems to the assembly of systems from pre-existing components. It has long been a principal aim of object-oriented development to minimize the amount of new specification, design and programming work that must be done when a new system is built. In theory, object-oriented approaches made this possible through well-established principles of abstraction and encapsulation, yet for many years large-scale reuse proved elusive. It is now at last becoming a reality. This is partly due to the familiar object-oriented principles mentioned above, but also due to newer ideas that have emerged from the patterns movement, and to the rise of component-based development and service-oriented architecture.

8.2.1 Why is it sometimes hard to achieve reuse?

Generally speaking, it is a waste of time and effort to produce from scratch an artefact that has already been produced elsewhere to a satisfactory standard. If you need a new light bulb for your room, it would not seem very sensible to invent and build your own. Even if you have the knowledge, skill and equipment, the cost would be prohibitive.

This applies just as much to software development as to any other field of production. Good professionals have always learned as much as they could from their own experience, and from that of their colleagues. Programmers have built up extensive libraries that range from personal collections of useful subroutines to commercially distributed products that contain large numbers of industry-standard components. Examples of the latter include the .DLL (Dynamic Link Library) files used in Microsoft Windows and the class libraries available to Java programmers. Designers have built up corresponding libraries that consist of fragments of designs, templates, patterns and frameworks. In most professional situations it makes very little sense to repeat work that others have done before.

Till now, most authors agree that the promised benefits of software reuse have not been fulfilled in practice. Why is this so?

Reuse is not always appropriate

There are some exceptions to the general rule that reuse is good. For example, students are frequently asked to solve, as if they were the very first person ever to do so, problems that have been completely solved by others before them. This makes good educational sense, since it is often the process of enquiry that brings about understanding, and not just knowledge of the solution. For this reason, among others, educators have a deep hostility to copying: a student who passes off another's work as his or her own has learned nothing from its creation.

It is also desirable to start a new project without preconceptions. For example, at first we probably know nothing about the requirements for a new system. Analysts should take account of the unique characteristics of a proposed system and its environment. Thus we should begin to investigate a new situation with as few preconceptions as possible. But it would be madness to assume that we know nothing about how to solve any of the problems encountered along the way. Wherever we can, we should capitalize on successful past work, provided that it is relevant to the current problem.

The 'Not Invented Here' syndrome

Some software developers (even occasionally whole departments) seem to ignore the accumulated wisdom of their profession. Why? One reason is the NIH ('Not Invented Here') syndrome, which is said mainly to afflict programmers. This describes the attitude of someone who thinks: 'I don't trust other people's widgets—even those that appear to work, suit my purpose and are affordable—I want to invent my own widgets anyway.' This may be understandable in someone who enjoys a technical challenge, or has reasons not to trust the work of others, but it usually does not make good commercial sense.

Reuse can be difficult to manage

One key to successful reuse is the management of the process. First, from the point of view of a developer who wishes to find artefacts for possible reuse (models, templates, program subroutines, entire programs, etc.), a catalogue must be available. This catalogue needs to be comprehensive, up to date, and organized in such a way that it is easy to find artefacts that meet a present need. Second, the artefacts themselves have to be designed for reuse, and this can make their construction more difficult and more costly. But any artefact that is too specific in its design is likely to need extensive adaption before it can be reused in another context. The adaptation may prove more trouble than the creation of a new artefact that precisely suits the context.

The difficulties of managing this process well sometimes mean that reuse is difficult to achieve in practice, although few people doubt its benefits.

Requirements are harder to reuse than software

At its simplest, software reuse can be very easy to achieve. For example, to use a library function within a program, you may need to import the library, call the function by name and pass to it any required parameters. Your program can then carry on to use the result. For the programmer, all that is required is a copy of the library, knowledge of its signature and some understanding of what the function does. These things are typically learned in the fairly early stages of a programming course.

When we speak of a 'reusable requirement', however, we really mean a reusable model of a requirement. To date, this is one of the least developed areas of software reuse because it is complex by its nature. First, only parts of any model are likely to be reusable. It is also necessary to organize a model so that it abstracts out (hides) those features of a requirement that are not necessary for a valid comparison with a similar requirement on another project. Next, the whole point of reuse is to save work, so it should also not be necessary to develop a full model of a new requirement in order to make the comparison. Finally, any relevant differences between the two requirements being compared should be clearly visible—so it should also not be necessary to develop a full model of the new requirement in order to see these. We shall see later that patterns are one way of overcoming these difficulties.

8.2.2 How object-orientation contributes to reuse

Object-oriented software development relies on two main forms of abstraction that help to achieve reuse: generalization and encapsulation. Not surprisingly, their use resembles the practice in many other industries.

Generalization

First, let us consider generalization. This is a form of abstraction, which (as we saw in Chapter 4) means focusing on things that are relevant to our purpose while ignoring things that are irrelevant. Generalization concentrates on identifying those aspects of a design or specification that are relevant to more than one situation, while ignoring those that are only relevant to a specific situation. It is often possible to identify some element of a design, or of a solution to a problem, that has general application to a variety of situations or problems. At the risk of labouring a metaphor, the wheel is a good example. Wheels can be made almost any size and of many different designs and materials, depending on their intended use. The little plastic wheels inside a mouse register its movement over the mousemat. The rubber-tyred wheels on a bicycle push and steer it along its track, and provide some cushioning to the rider. The huge iron balance wheel on an old-fashioned steam engine smooths the jerky motion of the pistons. Despite the obvious differences in their size, material and method of construction, all share the same general circular form and rotate around a central shaft.

The engineer who designs a mouse must consider specific aspects of the situation, such as the need for mouse wheels to be non-conducting, small, light and cheap to make. But the basic principle of circular movement requires no special thought, since this is a long-understood solution to a well-known family of engineering problems. In this example, the particular size, weight, etc. of each wheel are specific to its application. The circular form is a general principle that has been abstracted from all the wheels made over the years, and can be applied again and again to the design of many other wheels to come.

Generalization in software is a lot like this. The aim is to identify features of a specification or design that will be useful in systems for which they were not specifically developed. In Chapter 4, we saw how the introduction of an abstract Employee superclass generalized the common aspects of the descriptions of different kinds of real employee (hourly, weekly and monthly-paid). Later in this chapter, we explain how to find generalization using further examples from the Agate case study.

Encapsulation

Encapsulation is a kind of abstraction that, as we saw in Chapter 4, focuses on the external behaviour of something and ignores the internal details of how the behaviour is produced. The assembly of modern desktop computer systems provides many examples. For example, PC mice can use very different technologies, but still behave in a similar way from a user's perspective. On my desk as I write are three different mice: two are wired, but one of these is a wheel mouse while the other is optical; the third is optical and wireless. You can tell the difference if you look, since there are some external clues (presence or absence of wire or red light). But most of the time whether a mouse is wheeled or optical, wired or wireless, makes little practical difference. More important, encapsulation allows one component to be replaced by another with no need for an exact match. If a wired wheel mouse fails, it can be replaced with a wireless optical mouse. The key here is a standard definition of the interface between one system (or subsystem) and others with which it interacts. You can only replace one mouse with another that works in a different way because they share a common interface with the computer. In this case, the interface is defined by the plug type and the pin connections (what type of signal each carries, and at what voltage). Encapsulation of software aims to achieve the same ease of replacement, and software interfaces also need to be defined in a standard way for this to work. The definition of a software interface is usually in terms of services provided and the message signature required to call each service.

Composition involves encapsulating a group of classes that collectively have the capacity to be a reusable subassembly. The idea is that a complex whole is made of simpler components. These, while less complex than the whole, may themselves be made of still less complex subassemblies, elementary components or a mixture of the two. Later in this chapter, we consider the use of composition using examples from the Agate case study.

Components

Composition structures are, in a sense, components, as described above. However, the term is now generally reserved for relatively complex structures that are developed separately from each other, often at different times or in different organizations, and then simply 'plugged together' to achieve the desired functionality.

It is hard today to think of an industry where the use of standard components, and even standard designs, is not common. To take a familiar example, an architect may design many different houses, each unique in its own particular way. But houses are typically designed and assembled using a standard range of bricks, roofing timbers, tiles, doors, window frames, electrical components, water pipes, floorboards, etc., and each component is chosen from a catalogue. Thus, while one house may be very unlike another in overall appearance, floorplan and number of rooms, the difference is in the way that standard components have been assembled.

A simple example of the usefulness of the idea can be seen in a house-builder fitting a window to a new house. Like many other components, windows are delivered to site as ready-assembled units. All internal details of the subassembly are 'hidden' from both architect and builder, in the sense that they do not need to

think about them. The architect needs only to know the size, appearance and function of a window (e.g. how it opens). The builder needs to know the size and how the window should be fitted into a wall. For this to work, the window must first be identified as a component that can be encapsulated, and thus treated by the architect and builder as a single, simple thing, even though really it is not. In another part of the building industry, specialist designers tackle the problems of window design and construction (what materials, structure, etc. make a good window) and specialist constructors are busy building windows to the designers' specifications.

Similarly, the internal details of a software component are encapsulated, and other components requesting their service need not know how the request will be met. This allows different parts of the software to be effectively isolated in operation and greatly reduces the problems in getting different subsystems to interact with each other, even when the subsystems have been developed at different times or in different languages, and even when they execute on different hardware platforms. Subsystems that have been constructed in this way are said to be *decoupled* from each other.

The approach can be scaled up to any arbitrarily complex level. Any part of a software system—or, by extension, a model of one—can be considered for reuse in other contexts, provided certain criteria are met.

- A component meets a clear-cut but general need (in other words, it delivers a coherent service or set of services).
- A component has one or more simple, well-defined external interfaces.

In theory, reusable components can be designed following any development approach, but object-orientation is well suited to the task. Well-chosen objects already meet both criteria, since an object requesting a service knows only the message protocol and the identity of an object that can provide it. Object-oriented models, and hence also code, are also organized in a way that is helpful for reuse. For example, Coleman et al. (1994) point out that generalization hierarchies are a very practical way of organizing a catalogue of components. This is because they encourage the searcher to begin first with a general category, then progressively refine their search to more and more specialized levels. This mirrors the way that the Dewey decimal system is used to catalogue books in a library.

Software developers have one significant advantage over architects and builders. Using inheritance, a 'software architect' has a way of spawning new products from old ones with minimal effort. There is nothing properly analogous to this in the manufacturing activities of house building (although there may be a closer comparison between design activities). For example, each time a brick is made, whether or not it is a new type, the manufacturing effort is similar to that for any other brick. But inheritance provides a way of designing and building part of some new software components without effort, since only the specialized details need be completed at a later stage. In a generalization hierarchy, those characteristics that are shared by subclasses are maintained at the superclass level, and are instantly available to any subclass when required.

Patterns

In everyday speech, a pattern refers to a kind of design that is used to reproduce images or products in a repetitive manner (on wallpaper or fabric, for example).

This suggests that patterns are a sort of generalization, which is not too far removed from the use of the term in software development. This particular usage is traced back to the architect Christopher Alexander, who first used the term *pattern* to describe solutions to recurring problems in architecture. Alexander identified many related patterns for the development of effective and harmonious architectural forms in buildings. Alexander's patterns address many architectural issues—for example the best place to site a door in a room, or how to organize and structure a waiting area in a building so that waiting can become a positive experience. Alexander argued that his patterns became a design language within which solutions to recurring architectural problems could be developed and described. Alexander's definition of a pattern is as follows.

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of a solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Alexander et al. (1977)

A pattern provides a solution that may be applied in different ways depending upon the specific problem to which it is being applied. Alexander's definition makes no reference to buildings or architecture but only to 'our environment'. Although Alexander intended 'environment' to be interpreted as the physical environment in which we live, there is clearly an analogous concept appropriate to software development. For example, Riehle and Zullighoven (1996) describe a pattern as a generalized abstraction from concrete forms that recur in certain situations. It is not clear how tangible 'concrete' is intended to be but a reasonable interpretation might be 'specific' or 'particular'. Gabriel's (1996) definition is more detailed and expresses the structure of a pattern:

Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.

Gabriel (1996)

This definition focuses on three elements—a *context* that can be understood as a set of circumstances or preconditions, *forces* that are issues that have to be addressed and a software configuration that addresses and resolves the forces.

In fact, patterns are applied widely in systems development, and not just to 'software configurations'. The Boundary, Control and Entity object architecture introduced in Chapter 7 is in fact a pattern widely applied during requirements analysis and systems design. Other architectures that we will consider later are also examples of the application of patterns to the activity of systems design.

Coplien (1996) identifies the critical aspects of a pattern as follows.

- It solves a problem.
- It is a proven concept.
- The solution is not obvious.
- It describes a relationship.
- The pattern has a significant human component.

The human component Coplien refers to is not simply a good user interface to a working application; it is concerned with the nature of the software constructs

used to build the application. Software patterns should result in structures that are sympathetic to the human perspective. It is argued that a good software pattern offers not only a solution that works but a solution that has an aesthetic quality, that is in some way elegant. This aesthetic quality of patterns is sometimes termed Quality Without A Name (QWAN for short) to avoid the overloading of current terminology. As you might imagine, QWAN is the subject of much controversy. Our discussion of patterns will not address issues of elegance, though readers may judge for themselves whether the solutions developed from the patterns have a sense of elegance.

In the same way that a pattern captures and documents proven good practice, *antipatterns* capture practice that is demonstrably bad. It is sensible to do this. We should ensure not only that a software system embodies good practice but also that it avoids known pitfalls. Antipatterns are a way of documenting attempted solutions to recurring problems that proved unsuccessful. An antipattern can also include reworked solutions that proved effective (Brown et al., 1998).

For example, Mushroom Management (Brown et al., 1998) is an example of an antipattern in the domain of software development organizations. It describes a situation where there is an explicit policy to isolate systems developers from users in an attempt to limit requirements drift. In such an organization, requirements are passed through an intermediary such as the project manager or a requirements analyst. The negative consequence of this pattern of development organization is that inadequacies in the analysis documentation are both inevitable and not resolved. Furthermore, design decisions are made without user involvement and the delivered system may not address users' requirements. The reworked solution suggested by Brown et al. (1998) is to use a form of spiral process development model (see Chapter 3). Other reworked solutions include the involvement of domain experts in the development team, as recommended by the Dynamic Systems Development Method (DSDM) (we introduce DSDM in Chapter 21).

Patterns have been applied to many different aspects of software development. Beck and Cunningham (1989) documented some of the earliest software patterns in order to describe aspects of interface design in Smalltalk environments. Subsequently Coplien (1992) catalogued a set of patterns specifically for use in C++ programming (patterns that are related to constructs in a specific programming language are now known as *idioms*). Coad et al. (1997) greatly widened the field by describing a series of analysis and design patterns. Patterns have been applied to software development approaches other than object-oriented ones. For example, Hay (1996) identified a series of analysis patterns for data modelling. Hay discusses patterns relating to concepts such as Party and Contract, which appear commonly in information systems.

Design patterns were popularized by Gamma et al. (1995) in their book *Design Patterns: Elements of Reusable Object-Oriented Software*¹. The publication of this book gave significant impetus to the use of patterns in software development, but other authors have identified patterns that are concerned with analysis (Fowler, 1997), organizational issues (Coplien, 1996) and systems architecture (Buschmann et al., 1996).

¹ The four authors of the book are known as the 'Gang of Four' (GOF) and the book is known as the GOF book.

Later in this chapter, we apply the Accountability analysis pattern (Fowler, 1997) to the Agate case study.

8.3 Adding Further Structure

This section shows how to add structure to the class diagram that will help with reuse at later stages of development. First, in Section 8.3.1 we concentrate on generalization. Then, in turn we examine the use of composition, components and patterns.

8.3.1 Finding and modelling generalization

Figure 8.1 shows a note of an interview carried out by an analyst in the Agate case study. Her main objective was to understand more about different types of staff. In her haste, the analyst gathered only a handful of facts, but these highlight some useful information that must be modelled appropriately.

- There are two types of staff.
- Bonuses are calculated differently.
- Different data should be recorded for each type of staff.

Figure 8.2 shows a partial class diagram that corresponds to this (for clarity, only relevant classes are shown).

Redefined operations

Why is there a `calculateBonus()` operation in all three classes in the hierarchy in Fig. 8.2? Has the analyst made a mistake? Or has she failed to take advantage of the economy of representation offered by the generalization notation?

17 March - brief interview with Amarjeet Grewal (Finance Director)
Purpose - clarification of points from last Thursday's interview

Asked about staff types

- only two types seem relevant to system -
creative staff (C) and admin staff (A)

How do they differ?

- main difference is bonus payment...
 1. (C) bonus calculated on basis of campaign profits
(only those campaigns they worked on)
 2. (A) paid rate based on average of all campaign profits

Any other diffs? Amarjeet says -

- C qualifications need to be recorded
- C can be assigned as contact for a client
- A are not assigned to specific campaigns

No other significant differences.

(NOTE - at next interview, get details of both algorithms)

Figure 8.1 Analyst's note of the differences between Agate staff types.

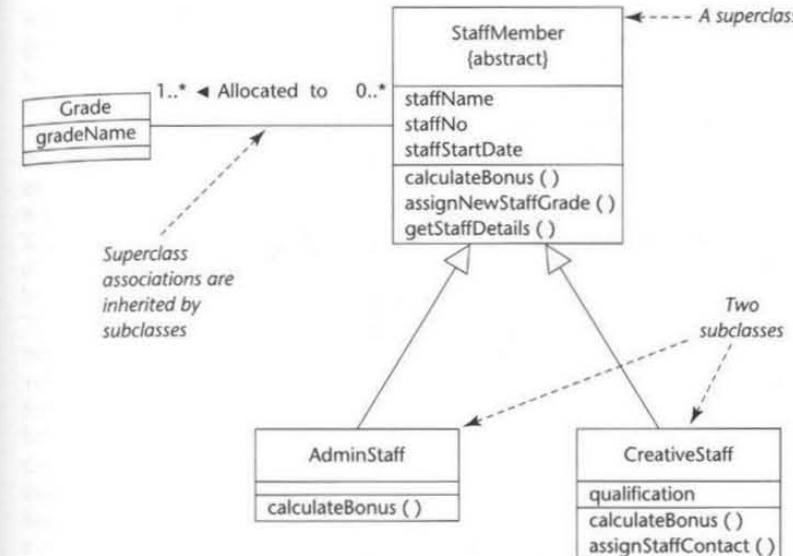


Figure 8.2 A generalization hierarchy for Agate staff roles.

The explanation is that, while both `AdminStaff` and `CreativeStaff` require an operation `calculateBonus()`, it works differently in each case. Since the precise logic for the calculation differs between these two groups of staff, the two operations need to be treated separately when each algorithm is designed, and also when code is written to implement them. This justifies the separate appearance of a superficially similar operation in both subclasses.

However, if the operation `calculateBonus()` is to be redefined in all subclasses, why include it in the superclass? The answer is that it is an attempt at 'future-proofing'. A superclass may later acquire other subclasses that are as yet unknown. Here, the analyst has recognized—or assumed—that objects belonging to *all* subclasses of `StaffMember` are likely to need an operation to calculate bonus. For this reason, at least a 'skeleton' of the operation is included in the superclass. This may consist of no more than the signature, but since the interface is all that matters to other classes, this alone justifies its inclusion in the superclass definition. Even when the superclass operation is defined in full, some subclasses may choose not to use it because the logic for their version of the operation is different. For all these cases, the operation is said to be *redefined* or *overridden* in the subclass.

Abstract and concrete classes

Now consider the `{abstract}` annotation above the `StaffMember` class name in Fig. 8.2. `StaffMember` is abstract in the sense that it has no instances, and this is shown by the `{abstract}` stereotype (an alternative notation is to write the class name in italics). This stereotype can only be applied to a superclass in a generalization hierarchy. All other classes may have at least one instance, and are said to be *concrete* or *instantiated*. Applying the stereotype to `StaffMember`

means that no staff exist at Agate who are 'general' members of staff, and not members of a particular subgroup. All staff members encountered so far (among those that are relevant to the model) are defined as either *AdminStaff* or *CreativeStaff*. If we later discover another group of staff that is distinct in behaviour, data or associations, and if we need to model this new group, it will be represented in the diagram by a new subclass. The point of a superclass is that it sits at a higher level of abstraction than its subclasses. This generality allows it to be adapted for use in other systems. While this in itself is not always enough for it to be declared as an abstract class, this is usually the case, and for the moment we can safely ignore exceptions to the rule.

The usefulness of generalization

We can now consider the contribution of generalization hierarchies to reuse. Imagine that the Agate system is completed and in regular use. Some time after installation, the Directors reorganize the company, and one result is that Account Managers are to be paid bonuses related to campaign profits. This bonus will be calculated in a different way from both administrative and other creative staff, and will include an element from campaigns that they supervise and an element from the general profitability of the company. It is easy to add another subclass to cater for this new behaviour. This can be seen in Fig. 8.3.

Note also the alternative notation styles. In Fig. 8.2 each subclass is joined to the superclass by its own generalization association, while in Fig. 8.3 the three subclasses are organized into a tree structure with a single triangle joining this to the superclass. The single tree structure is known as the *shared target* form of the notation. Both forms are acceptable, but the shared target notation is used only where the generalization relationships belong to the same *generalization set*. In the example shown in Fig. 8.3 this is appropriate, since the types of employee shown represent a single coherent way of partitioning the superclass. However, suppose

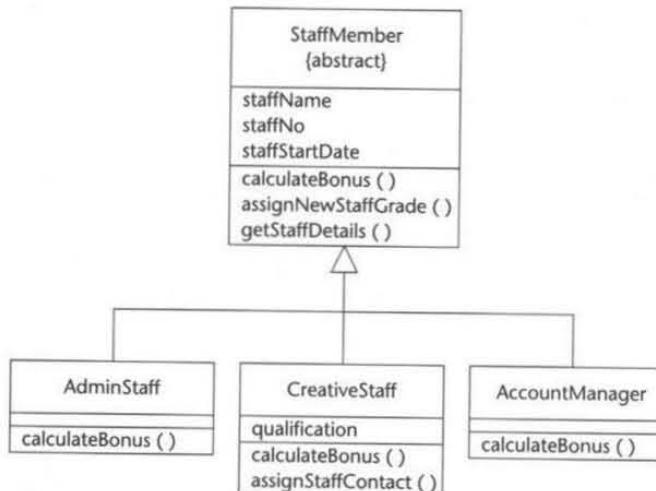


Figure 8.3 A new subclass is easy to add.

that for some reason we need also to specialize employees as Male and Female: the new generalization relationships for this belong to a different generalization set. The name of the applicable generalization set can optionally be shown on the diagram next to the relationship. This is illustrated in Fig. 8.4.

Adding a new *AccountManager* subclass has little or no impact on the rest of the class model. The reuse derives from basing *AccountManager* on the existing abstract class *StaffMember*. The new class inherits all attributes and operations that are not part of its own specialism—in this case roughly 85% of its specification. This is for only one class; over the many classes of a large system, the saving in design and coding effort can clearly be significant. But the opportunity is available because we have previously identified the generalized aspects of a staff member. This is the main benefit of generalization, that hierarchies can usually be extended without significant effects on existing structures.

A top-down approach to finding generalization

It is relatively easy to discover generalization where both superclasses and subclasses have already been identified. If an association can be described by the expression *is a kind of*, then it can usually be modelled as generalization. Sometimes this is so obvious that you may wonder if that is all there is to it. For example, 'administrative staff are a kind of staff'. In just the same way 'a helicopter is a type of aircraft and so is a fixed-wing jet' and 'a truck is a type of vehicle and so is a buffalo cart' imply generalizations with similar structures.

It is not uncommon to find multiple levels of generalization. This simply means that a superclass in one relationship may be a subclass in another. For example,

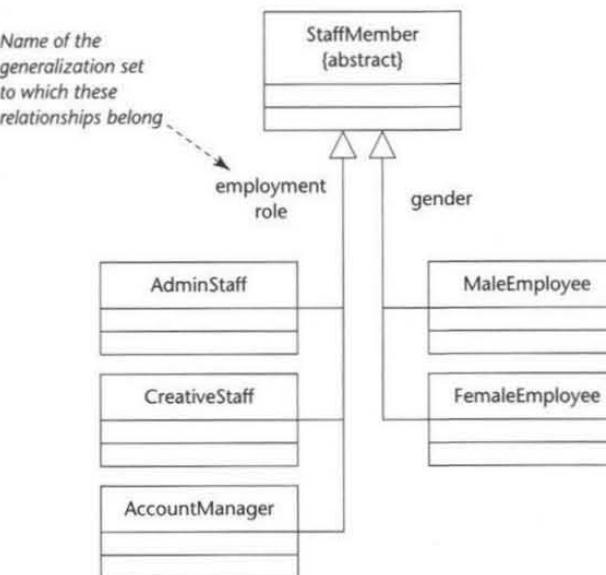


Figure 8.4 Generalization sets partition a superclass in different ways.

Aircraft is both a superclass of Helicopter and a subclass of Vehicle. In practice, more than about four or five levels of generalization in a class model is too many (Rumbaugh et al., 1991), but this is primarily for design reasons.

A bottom-up approach to finding generalization

An alternative approach is to look for similarities among classes in your model, and consider whether the model can be 'tidied up' or simplified by introducing superclasses that abstract out the similarities. This needs to be done with some care. The purpose of doing this is to increase the level of abstraction of the model, but any further abstraction introduced should still be 'useful'. The guiding principle is still that any new generalization must meet all the tests described in Chapter 4.

When not to use generalization

Some judgement is needed to determine the likely future usefulness of generalization, as it can be overused.

We can illustrate this best with an example of inappropriate modelling. At Agate, staff and (some) clients are people. This is perhaps stretching a point, since many clients are in fact companies, but for the sake of illustration we will ignore this and assume all clients are individuals. An inexperienced analyst might feel that this justifies the creation of a Person superclass, to contain any common attributes and operations of Client and StaffMember. But it may quickly become apparent that the new class definition contains little but the attribute `personName`. This is really an attempt to force a generalization hierarchy to include subclasses that are too dissimilar.

Second, we should not anticipate subclasses that are not justified by currently known requirements. For example, at Agate AdminStaff and CreativeStaff are distinct classes based on differences in their attributes and operations. We also know about other kinds of staff in the organization, e.g. directors. But we should not automatically create another subclass of StaffMember called Director-Staff. Even if it were to turn out that directors have some relevance for the system, there is no reason yet to suppose they will be a distinct class. They may be adequately modelled by an existing class (say, AdminStaff) unless we find that their behaviour or information structure differs in some way.

There is a tension here. On the one hand, generalization is modelled to permit future subclassing in situations that the analyst cannot reasonably anticipate. The ability to take advantage of this is one of the main benefits of constructing a generalization hierarchy. On the other hand, if generalization is overdone, it adds to the complexity of the model for little or no return. There is no simple answer to this problem, save the judgement that comes with experience. But it seems probable that over the next few years, organizations will grow much better at managing the activity of reuse, and part of this will involve clear guidance to project teams on when and when not to use generalization.

Multiple inheritance

It is possible, and often appropriate, for a class to be simultaneously the subclass of more than one superclass. This is familiar in everyday classification. For example, if we classify some household items according to their use, a coffee mug is a

drinking vessel. If we classify the same items according to their aesthetic qualities, the mug might be an attractive craft item. If we classify the same items according to their health risk, the mug might be a hazard (because it is cracked). The mug can belong at one time to various categories derived from different classification schemes without any logical conflict.

In object-oriented modelling, especially during design, it can be useful to define classes that inherit features from more than one superclass. In each case, all features are inherited from every superclass.

8.3.2 Finding and modelling composition

Composition (or *composite aggregation*) is based on the concept of *aggregation*, which is a feature of many object-oriented programming languages. Put at its simplest, aggregation represents a whole–part relationship between classes, while composition expresses a similar relationship but differs in showing a stronger form of ownership by the whole.

One application of composition is familiar to users of any common computer drawing package. For example, many drawings in this text were prepared and edited with a widely used drawing package. This application allows the user to select and group several objects. Grouped objects behave as a single object, and can be sized, rotated, copied, moved or deleted with a single command. Figure 8.5 illustrates this, while Fig. 8.6 models the composition as a class diagram. Note that

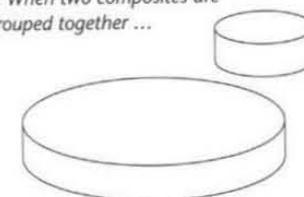
1. A collection of separate drawing objects ...



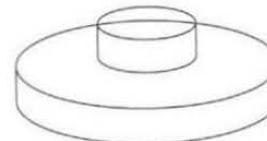
2. ... when grouped together, make a single composite:



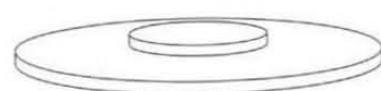
3. When two composites are grouped together ...



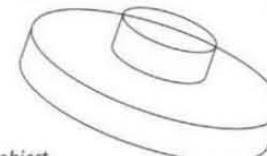
4. ... they make a further composite ...



5. ... that can be scaled ...



6. ... or rotated ...



... as if it were a single object.

Figure 8.5 Composition of objects in a drawing package.

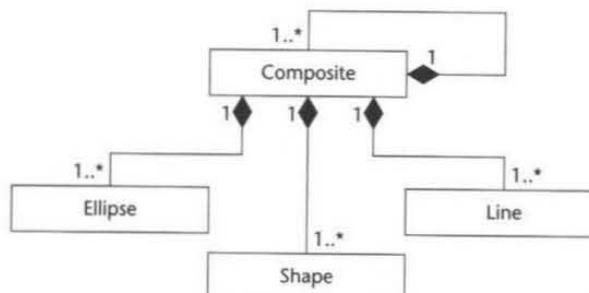


Figure 8.6 The same composite structure, expressed as UML classes.

the composite structure is nested—a composition can itself contain a further composition. In the same way that a composite drawing object can only be handled as a single drawing component, the ‘part’ objects in a composition structure cannot be directly accessed, while the whole object presents a single interface to other parts of the system.

Composition and aggregation may both be identified during requirements analysis, but their main application is during design and implementation activities, where they can be used to encapsulate a group of objects as a potentially reusable subassembly. This is more than just a matter of labelling the group of objects with a single name. Their encapsulation is much more important. The external interface for a composition is the interface of the single object at the ‘whole’ end of the association. Details of the internal structure of the composition—that is, what other objects and associations it contains—remain hidden from the client.

The notation is similar to a simple association, but with a diamond at the ‘whole’ end. The diamond is filled with solid colour to indicate composition, and left unfilled for aggregation.

An alternative notation for a composite structure is illustrated in Fig. 8.7. This explicitly shows the composite object as a container for its parts.

Use of composition and aggregation in business-oriented applications is more problematic than in the drawing package example, but it is still worth modelling where it conveys useful information about the structure of the business domain. An example from the Agate case study is used in the next Section to illustrate how aggregation is identified.

8.3.3 Combining generalization with composition or aggregation

Now let us examine the Agate case study material for further generalization and composition. One possibility is revealed in the statement in Section A1.3 that ‘adverts can be one of several types’. For each type, it is certainly true that, for example, a newspaper advert is *a kind of* advert. For simplicity, we assume that ‘advert’ refers to a *design*, rather than an *insertion*—so an advert that appears five times in one newspaper is one advert appearing five times, and not five adverts each appearing once. This suggests that advert could be a superclass, with newspaper advert, etc. as subclasses. Would this meet the tests described in Section 4.2.4? An authoritative answer would require a detailed examination of attributes and operations for each class, but the answer appears likely to be yes.

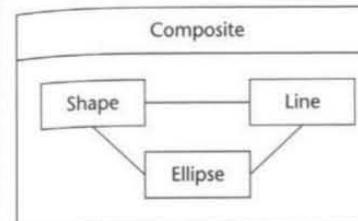


Figure 8.7 Alternative notation for a composite structure.

Next, a newspaper advert consists of a particular set of parts. The precise composition of each type of advert is different, and so this structure of associations could not be defined at the superclass level (the attributes, operations and composition structure of television adverts may resemble newspaper adverts in some respects, but will differ in others).

We can see possible composition in the association between Campaign and Advert, and in turn between Advert and its associated parts. A campaign includes one or more adverts. A newspaper advert includes written copy, graphics and photographs.

Is this really composition rather than aggregation? First, can an advert belong to more than one campaign? This is not stated in the case study, but it seems unlikely that an advert can simultaneously be part of more than one campaign. Second, has each Advert a coincident lifetime with its Campaign? Again, this is not given explicitly, but a client might wish to use an expensive advert again on another campaign. This point needs to be clarified, but in the meantime it does not appear justified to model this as composition. Third, can copy, graphics or photographs belong to more than one newspaper advert? Perhaps graphics or photographs may be reused, but this seems very unlikely to be the case for copy. Finally, has each of these components a coincident lifetime with the advert? Probably some do and some do not. This would all need further clarification, but in the meantime composition seems justified only in the case of NewspaperAdvertCopy, and aggregation has therefore been used elsewhere. Figure 8.8 shows the partial class model that results from this analysis.

8.3.4 Organizing the analysis model—packages and dependencies

An analysis model may include completely free-standing components, which are discussed in the following section. Even if it does not, it must still be organized in such a way that it will remain robust in the face of changing requirements. This demands skill and judgement on the part of the analyst. To some extent this depends on defining analysis packages that are relatively independent of each other while still internally highly cohesive (see Chapter 14). The package (introduced in Chapter 5) is a means by which a developer can ‘factor out’ classes or structures that have potential use in a wider context than one project alone. But when a model is partitioned into packages (or when pre-existing components are used to support a current project) it is important to keep track of the dependencies between different classes and packages.

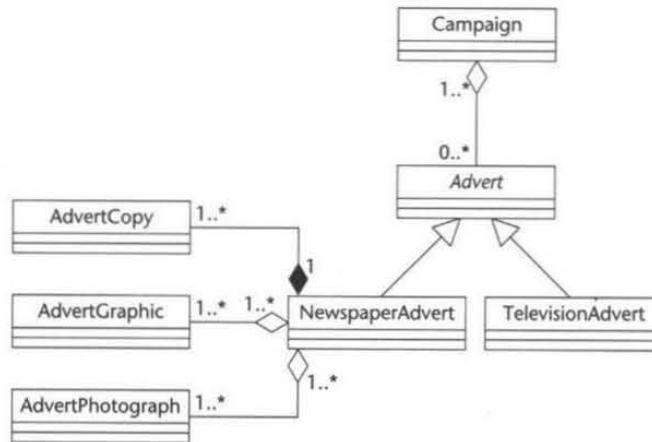


Figure 8.8 Aggregation, composition and generalization in the Agate model.

We saw in Chapter A2 that the Agate case study suggests three related but distinct application areas: advert preparation, campaign management and staff management. Even if this model proves to be only one part of a larger domain, it probably still makes sense to model these as separate packages. It is likely that some entity objects will prove to be common to more than one package.

Based on our analysis so far, the *StaffMember* entity class plays a role in both the *Campaign Management* and the *Staff Management* packages. This leads to an architectural decision. Figure 8.9 illustrates some of the variations described below.

- We could decide to locate this class within the *Staff Management* package. In this case, we need to model a dependency from *Campaign Management* to *Staff Management*, since the *Client* and *Campaign* classes need an association with *StaffMember* (diagram variation (i)).
- We could remove *StaffMember* to a separate package. This would be justified if it appeared to have more widespread use. For example, there may also be wages, personnel, welfare and pension applications that need this class. In this case, we need to model dependencies from all the corresponding packages to the package that contains the *StaffMember* class (diagram variation (ii)).
- Further analysis may reveal that in fact *StaffMember* is not a single class. In this case, the derivative classes may remain within their respective packages, but it is likely that there will be an association between them and the dependency must still be documented.
- In addition, we have already made an implicit judgement (in Chapter 7) to separate all boundary objects into a *User Interface* package and all control objects into a *Control* package. Objects in these specialized packages will certainly have dependencies on objects in other packages (diagram variation (iii)).

The significance of documenting package dependencies is perhaps not obvious when dealing, as we are here, with a relatively simple model. But it will become

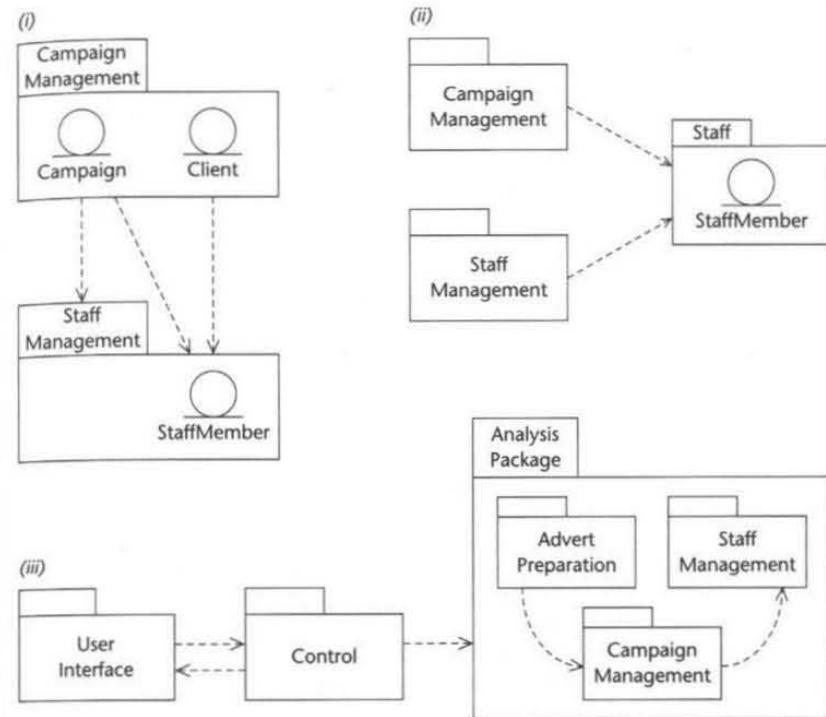


Figure 8.9 Alternative dependencies among packages and objects.

more important when the models are large, complex and, above all, when there is a substantial element of reuse involved, whether this occurs at the level of individual classes or at a component scale.

8.4 UML Support for Modelling Components

The basic notation for modelling components in UML is illustrated in Fig. 8.10. This shows two components whose interfaces are joined by a ball-and-socket assembly connector. Component A has a *provided interface*, which offers services to other components that know how to request those services. Meanwhile, component B has a *required interface*, which requests services from a provided interface on another component. The form of the protocol for requesting a service is the same as that for calling an object operation. The requesting component sends a message that contains an operation name and any necessary parameters.

Each interface may offer a range of services, and each service has its own specific protocol or signature. These are modelled independently of any decisions about the internal class structure of the component. The interface operations that result from this analysis imply what Cheesman and Daniels (2001) call ‘an

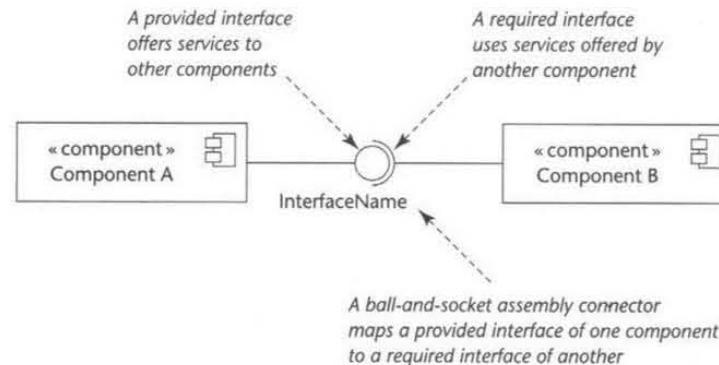


Figure 8.10 Basic notation in UML 2.0 for a component structure diagram.

information model' of the interface, and this can give a good starting point for modelling the internal design of a component.

In this section we introduce some elements of the component notation and show how it can be used to model a component-based architecture and to specify important aspects of component interfaces. However, this is certainly not intended as a complete description of how to analyse software components for component-based development. Readers who are interested in learning more should consult a specialist work on the subject, such as Cheesman and Daniels (2001), which in our view is one of the clearest books on modelling components.

8.4.1 Component-based development

Many activities involved in component-based development (CBD) are much the same as they would be for the development of any other software. The classes that comprise an individual component must be identified, modelled, specified, designed and coded. These activities can be carried out in the ways described elsewhere in this book.

One aspect of CBD that differs from 'simple' object-oriented analysis and design is the specification of component architectures and component interactions. Much of this activity relates to interfaces and behaviour at a higher level of abstraction than single classes or the kind of small-scale composition structures that we have described up to this point.

Components can themselves be specified at several different levels of abstraction. Cheesman and Daniels (2001) discuss the different forms that a single component can take. First, components must follow some sort of common standard if they are to work together with each other when they are assembled. An everyday example is the way that many electrical devices follow a standard that defines the operating voltage and the size and shape of the plug. Voltage and plug designs vary from one country to another; hence the need for the kind of travel adaptors sold at airports. The standard for software components is effectively realized as the environment within which a given component is designed to work.

Component function is also important. The speaker and headphone output sockets on many electric guitar amplifiers may be designed to accept exactly the

same size and shape of jack plug connector. But if you plug a pair of headphones into the speaker output socket on a powerful amplifier, you could end up blowing the headphones. If it is a transistor amplifier, you may destroy the power transistors too. Plugging in a speaker just allows you to hear the guitar. A component's behaviour is described by its specification, much of which will consist of definitions of its interfaces with other components.

Each specification may have many implementations. It should be possible to replace one implementation of a specification with another, just as a guitarist can unplug one guitar and plug in another. That, after all, is the whole point of using components.

It is possible to imagine a component implementation that will be installed only once, but it is much more likely that there will be multiple installations of each implementation. Consider a web browser (Mozilla Firefox, say) as an example. We can assume that there is a single specification for the software, but there are different implementations for each of the main operating systems (Windows, Mac and Linux). Each implementation is installed on many millions of computers around the world. On each of these computers there is an installed copy of the implementation, and for this to work properly it needs to be registered with its operating system environment. Finally, each time that the software is launched, a new component 'object' starts. This brings us finally to the level where actual working components are realized—the level where data is stored and processes are executed.

Another way in which CBD differs from conventional systems development is that it involves not only the creation of new components, but also the reuse of existing ones. These may be components that have been developed in-house during earlier projects, or they may be supplied commercially by an external third party. In either case, there is a need for catalogues that describe components in a standardized way, so that developers can find components that will be useful, or—equally important—can determine that none yet exist. For this reason, CBD life-cycles typically have a pattern that differentiates between the development of components and their use, and also recognize the need for life-long management of components, from specification through to end use. We return to the topic of managing components in Chapter 20. In the following section, we concentrate on describing some examples that show how UML can be used in their specification.

8.4.2 Modelling components

An airline seat booking system provides a typical scenario for the use of software components. This is partly because legacy systems (see Section 6.2.1) are still in widespread use in the industry. These must interoperate with newer systems, themselves often developed using object-oriented methods and also (increasingly often) interacting over the web. New systems components developed for such an environment must be capable of integrating with components that are already in use—some quite antiquated in their design and implementation—and also with new components still to come, some of which have not yet been specified.

Let us assume that bookings are to be taken from various sources, including traditional travel agents, online travel agents and the company's own website. A component architecture makes sense, so that individual components can be replaced in a plug-and-play fashion without disrupting the operation of other

systems. This will permit the upgrading of older components with minimum fuss. It will also allow the substitution of different booking processes and platforms while using a single component to manage all the transactions.

Figure 8.11 shows part of a possible component-based architecture for an airline. The main components in this simplified view are systems to handle bookings, to handle payments, to keep records of customers, to check in passengers at the airport, and to manage flight information (including the allocation of passengers to seats on the aircraft). A real airline would, of course, require many other systems to support its operations.

The Bookings component provides an interface called MakeBooking. This is available to any other system that understands how to use it, which really just means knowing the services that are provided by the interface and the protocol or signature for each service. Implementation details of a component are encapsulated in just the same way that the attributes and operations of an object are encapsulated.

In this case, other systems that might use the interface include those of travel agents and an e-commerce system on the airline's website. The interface decouples client systems that wish to make a seat booking from the actual operation of the booking system component, allowing bookings to be made remotely from other networks or over the Web. The architecture shown also permits new clients to be implemented for other platforms. For example, if the airline wishes to enable bookings from mobile devices, the booking system will need no modification. All that is required is for any new system using this interface to be designed so that it can send properly formed requests to the MakeBooking interface.

The Bookings component provides an interface to the Check-in component, which needs details of bookings to ensure that the correct customers are checked

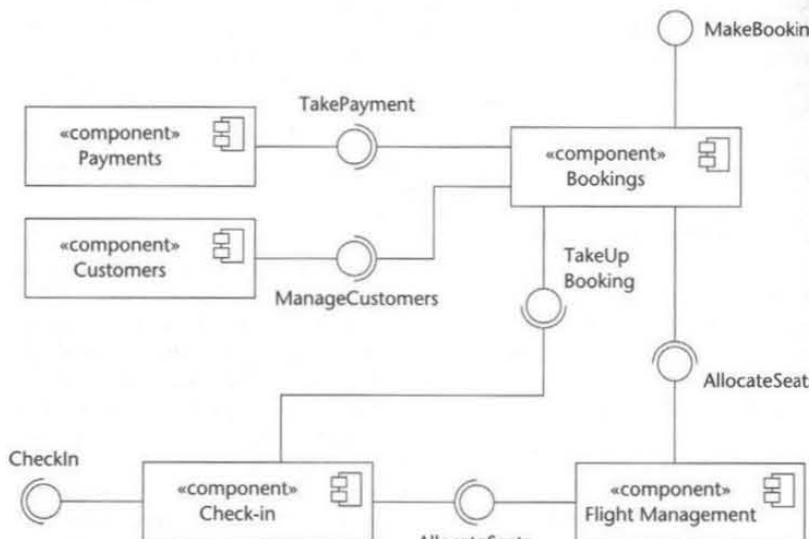


Figure 8.11 Component structure diagram for airline systems.

in. It uses an interface provided by the Payments component, which handles the processing of credit cards and any other types of payment. Both Bookings and Check-in components use an interface provided by the Flight Management component to obtain (and to update) details of seat availability on flights. Finally, the Bookings component uses an interface provided by the Customers component to obtain and update details of the airline's customers.

A communication diagram can be used to model the interaction of these interfaces in more detail (communication diagrams are explained fully in Chapter 9). Figure 8.12 shows the interaction for checking in passengers for a booking. Note that individual messages are shown with their parameters, allowing the analyst to specify in some detail the operations and thus, by extension, the classes needed to implement each interface. Once a model has been built up that gives a complete picture of the interactions, the specification of a component can be represented as the totality of all operations on all its interfaces.

Figure 8.13 shows the dependency relationship between the Flight Management component and its *AllocateSeats* interface. The component itself is realized by a package (or packages) of classes analysed as described elsewhere in this chapter.

Note that this diagram, even if it were complete and showed all interfaces offered by the Flight Management component, still represents the specification of a component, rather than its implementation. Component specifications concentrate on the external view of operations and attributes, in just the same way that a class specification does. A component implementation (in other words, the specification of classes and associations that exist within the component) would be

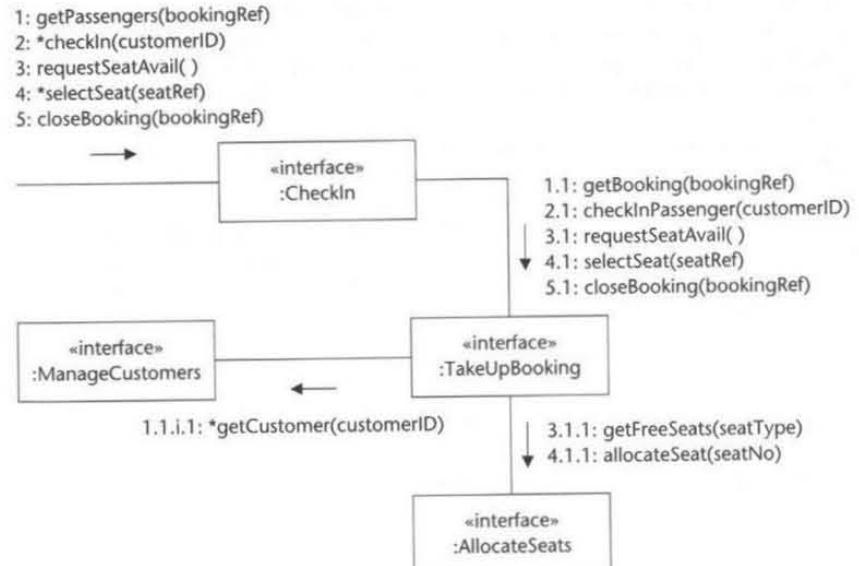


Figure 8.12 A communication diagram can show interaction between instances of component interfaces.

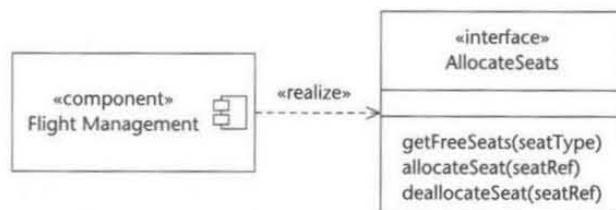


Figure 8.13 Dependency relationship between an interface and a package that implements it.

much more detailed than we have shown in these few examples, but would also be beyond the scope of this chapter.

8.5 Software Development Patterns

Coad et al. (1997) distinguish a *strategy*, which they describe as a plan of action intended to achieve some defined purpose, from a *pattern*, which they describe as a template that embodies an example worth emulating. This is slightly different from the views of a pattern described earlier as it does not emphasize contextual aspects to the same extent. One example of a Coad et al. strategy is 'Organize and Prioritize Features', which relates to the need to prioritize requirements (discussed in Chapter 3). A simple example of an analysis pattern from Coad et al. (1997) is the Transaction–Transaction Line Item pattern shown in Fig. 8.14.

Figure 8.15 shows that pattern as it might be applied to a sales order processing system. Here the Transaction suggests a SalesOrder class and the Transaction Line Item suggests a SalesOrderLine class. Note that we have modelled the association as a composition. This differs from the published pattern, but appears to be justified here.

Very similar structures are used in a wide variety of circumstances (e.g. shipment and shipment line item, payment and payment line item). A novice software developer has to learn this structure, or to reinvent it—but the latter is much less efficient. The act of describing it as a pattern highlights it as a useful piece of development expertise and makes it readily available to the novice. An example of a principle that makes this pattern advantageous is the desirability of low interaction coupling (see Chapter 14).

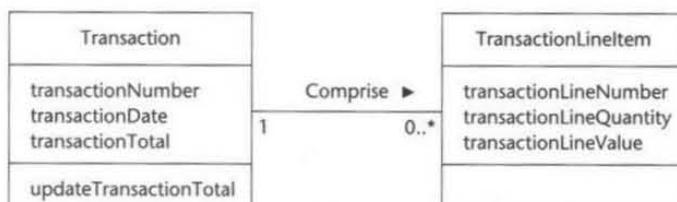


Figure 8.14 Transaction–Transaction Line Item pattern (adapted from Coad et al., 1997).

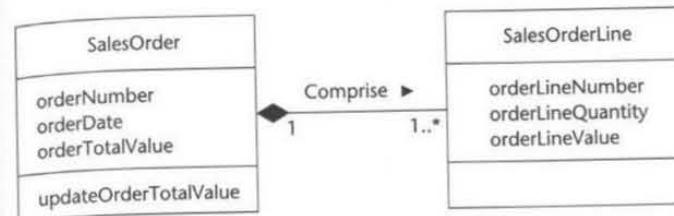


Figure 8.15 Simple application of the Transaction–Transaction Line Item pattern.

8.5.1 Analysis patterns

In this section we offer a brief introduction to analysis patterns. The use of analysis patterns is an advanced approach that is principally of use to experienced analysts, and we leave the interested reader to follow this up through further reading. They are closely related to design patterns, which we cover in some detail in Chapter 15.

An analysis pattern is essentially a structure of classes and associations that is found to occur over and over again in many different modelling situations. Each pattern can be used to communicate a general understanding about how to model a particular set of requirements, and therefore the model need not be invented from scratch every time a similar situation occurs. Since a pattern may consist of whole structures of classes, the abstraction takes place at a higher level than is normally possible using generalization alone. Fowler (1997) describes a number of patterns that recur in many business modelling situations, such as accounting, trading and organization structure.

Figure 8.16 shows Fowler's Accountability pattern as an illustration of an analysis pattern in practice. For the sake of simplicity, we will discuss only the class structure, although patterns are normally documented in more detail than this (see Chapter 15). An accountability structure may be of many kinds, such as management or contract supervision. In the case of Agate, this pattern could apply to several different relationships: that between a manager and a member of staff they supervise, that between a client and a client contact, or that between a client and

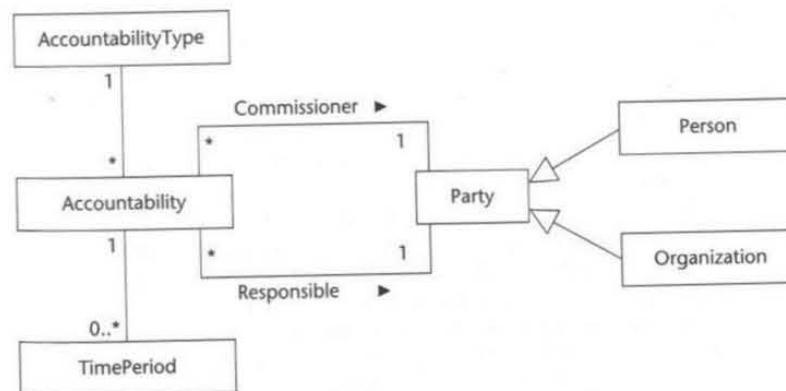


Figure 8.16 The Accountability analysis pattern (adapted from Fowler, 1997).

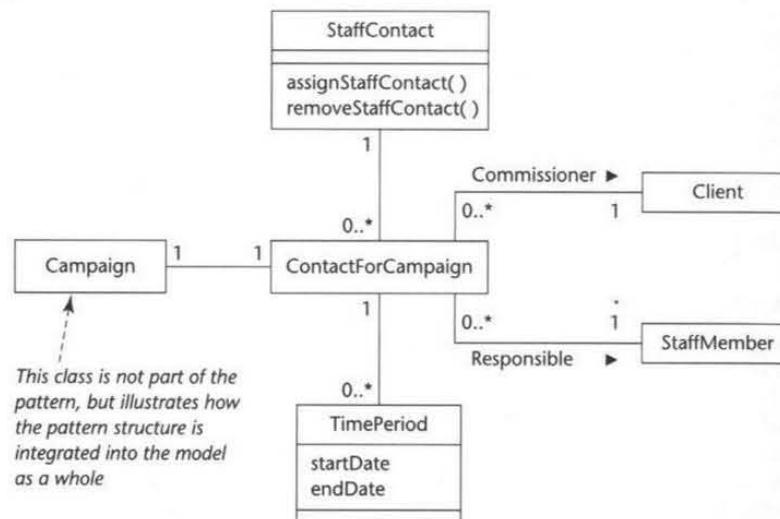


Figure 8.17 Accountability pattern applied to Agate's StaffContact relationship.

a campaign manager. Since the details of the relationship itself have been abstracted out as *AccountabilityType*, this one class structure is sufficiently general to be adapted to any of these relationships, given an appropriate set of attributes, operations and associations to other classes specific to the application model. The generalization of Person and Organization as Party similarly allows the pattern to represent relationships between individuals, organizations, or a mixture of the two.

Figure 8.17 shows a possible application of this analysis pattern to the StaffContact relationship at Agate.

Another pattern that might be useful in developing an analysis model for Agate is the Composite pattern (Gamma et al., 1995). However, since this pattern is useful from both analysis and design perspectives, we will defer its introduction until Chapter 15.

8.6 Summary

The primary purpose of refining the requirements model is to maximize opportunities for reuse. This can take various forms. Software components can reduce the effort involved in coding and implementation. Design components and patterns can reduce the time required to develop and test a design, and also act as a store of knowledge that embodies best known practice. Analysis components and patterns can reduce the time taken to develop a deep understanding of the system. In principle, whatever form they take, opportunities for reuse occur in essentially only three ways. Existing components or structures may be imported from other sources beyond the project boundaries. This requires careful evaluation of the degree of commonality between the requirements of the current project and the

features of the available components and structures. Second, new reusable components or structures may be developed for use on more than one part of the current project. For this to be possible, the requirements model must be at a sufficient level of abstraction for the common features between different aspects of the project to become evident. Finally, new reusable components or structures may be developed for export to other projects. Here, too, it is necessary for the requirements modelling to identify those aspects of the project that can be generalized for use in other contexts.

The abstraction mechanisms of the object-oriented approach, particularly generalization, composition and encapsulation, can make a significant contribution to all three approaches to reuse in systems development. So far, these benefits have been mainly apparent during the design and coding activities, but the starting point for finding reuse opportunities is during requirements analysis.

The explicit modelling of substitutable software components is a relatively new approach in software development and we expect that there will be more evolution in this area in the future. Patterns are now a more established approach to documenting, sharing and reusing useful insights into many aspects of the software development activity. This chapter has considered how components and patterns can be used in software development, in particular from an analysis perspective. Later chapters discuss in more detail the application of patterns and components to address issues of architecture and design.

Review Questions

- 8.1 What are the advantages of components?
- 8.2 Explain why the NIH syndrome occurs.
- 8.3 What does object-orientation offer that helps to create reusable components?
- 8.4 Distinguish composition from aggregation.
- 8.5 Why are operations sometimes redefined in a subclass?
- 8.6 What is an abstract class?
- 8.7 Why is encapsulation important to creating reusable components?
- 8.8 Why is generalization important to creating reusable components?
- 8.9 When should you not use generalization in a model?
- 8.10 What does the term pattern mean in the context of software development?
- 8.11 How do patterns help the software developer?
- 8.12 What is an antipattern?

Case Study Work, Exercises and Projects

- 8.A Find out from your library about the coding system that is used for classifying books, videos, etc. Draw part of the structure in UML notation as a generalization hierarchy. Think up some attributes for 'classes' in your model to show how the lower levels are progressively more specialized.

8.B Choose an area of commercial activity (business, industry, government agency, etc.) with which you are familiar. Identify some ways in which its products show the use of generalization, and some ways that components used as inputs show the use of generalization.

8.C In Section 8.3.4, we saw that generalization probably was an appropriate way of modelling the association between *Advert* and *NewspaperAdvert*. Identify the other possible subclasses for *Advert* from Section A1.3 and repeat the checks for each. Which of them pass (if any)? Do you think there are really only two levels to the hierarchy? Explain your reasoning. Redraw the Agate class diagram to include all the generalizations that you feel are justified.

8.D For each of your new *Advert* subclasses, suggest appropriate attributes, operations and aggregation or composition structures.

8.E Reread the case study material for FoodCo and identify possible generalizations or compositions. Add these to the class diagram you drew for Exercises 7.C and 7.D.

8.F Consider your class diagram for FoodCo. Try to identify possible applications of the Transaction–Transaction Line Item pattern or of the Accountability pattern and redraw your diagram as appropriate.

Further Reading

Ambler (2003) discusses the role of generalization in the reuse of classes and use cases. However, this topic has received little detailed attention in recent books. Thus, Rumbaugh et al. (1991) remains one of the clearest summaries. This book pre-dates UML, but Rumbaugh was later one of the three amigos (along with Jacobson and Booch) who devised UML.

Jacobson et al. (1999) are very clear on the architectural and reuse issues related to composition, generalization and the identification of packages in requirements analysis.

There are now a number of books on developing reusable software components. Cheesman and Daniels (2001) give a very succinct and thorough introduction to the use of UML for the specification of components. (This book predates UML 2.0, so the notation differs in some minor details from the adopted specification.)

Apperley et al. (2003) give a higher-level view of component development. They concentrate less on modelling techniques and more on managing the process. However, despite its relatively recent publication, many techniques used in this book are not UML.

For a broad-ranging introduction to patterns, Gamma et al. (1995) and Buschmann et al. (1996) are still essential reading. The *Pattern Languages of Program Design* (known as the PLOP) books (Coplien and Schmidt, 1995; Vlissides et al., 1996; Martin et al., 1998) catalogue a wide range of patterns for all aspects of software development.

Fowler (1997) describes a number of analysis patterns that may be applied at this stage of modelling, although these are not presented in UML. Coad et al. (1997) also gives a number of analysis and design patterns; most are presented in Coad notation, but some are also shown in OMT notation and in Unified notation (an early version of UML).

The patterns home page can be found at <http://hillside.net/patterns/>. Further useful patterns are stored in the Portland Pattern Repository at www.c2.com/ppr.

Object Interaction

Learning Objectives

In this chapter you will learn

- how to develop object interaction from use cases
- how to model object interaction using an interaction sequence diagram
- how to model object interaction using a communication diagram
- how to model object interaction using interaction overview diagrams
- how to model object interaction using timing diagrams
- how to cross-check between interaction diagrams and a class diagram.

9.1 Introduction

Communication and collaboration between objects is fundamental to object-orientation. This mirrors the world in which we live, where most human endeavour involves communication, interaction and collaboration between individuals. For example, each employee in a manufacturing organization has specialized tasks. Different employees interact and work with each other in order to satisfy a customer request. This involves communicating to request information, to share information and to request help from each other. In a similar way an object-oriented application comprises a set of autonomous objects, each responsible for a small part of the system's overall behaviour. These objects produce the required behaviour through interaction, by exchanging messages that request information, that give information or that ask another object to perform some task. In Section 9.2 we explore the concept of interaction in more detail.

We have already analysed use cases (in Chapter 7) to determine some aspects of the interaction and collaboration between the objects involved. This results in the identification of classes, their attributes and their associated responsibilities¹. As we move to design these object interactions have to be specified more precisely.

¹ This process is known as use case realization in the Unified Software Development Process.

This involves deciding how to represent responsibilities as operations. CRC cards were suggested in Section 7.6 as a supporting technique to aid the analysis and the resulting identification and allocation of responsibilities. CRC cards can also be used effectively when designing object interaction in more detail to identify and allocate operations. UML 2.0 provides several diagram types and a rich syntax for modelling interaction between objects: interaction sequence diagrams, communication diagrams, interaction overview diagrams and timing diagrams². Communication diagrams were known as interaction collaboration diagrams in UML 1.X and have been introduced in Chapter 7. Interaction sequence diagrams, which for simple interactions provide equivalent models as communication diagrams, are introduced in Section 9.3. Communication diagrams are explored further in Section 9.4. The roles of the new diagram types, interaction overview diagrams and timing diagrams are explored in Sections 9.5 and 9.6 respectively. In Section 9.7 we examine the issues concerned with interaction diagrams of all types and model consistency.

9.2 Object Interaction and Collaboration

When an object sends a message to another object, an operation is invoked in the receiving object. For example, in the Agate case study there is a requirement to be able to determine the current cost of the advertisements for an advertising campaign. This responsibility is assigned to the Campaign class. For a particular campaign this might be achieved if the Campaign object sends a message to each of its Advert objects asking them for their current cost. In a programming language, sending the message `getCost()` to an Advert object, might use the following syntax.

```
currentAdvertCost = anAdvert.getCost()
```

Note that in this example the Advert object is identified by the variable name `anAdvert` and the response to message, known as the return value, is stored in the variable `currentAdvertCost`.

The cost of each advert returned by the operation `getCost` is totalled up in the attribute `currentActualCost` in the sending object, Campaign (see Fig. A3.14 in Chapter A3 to see the attributes so far identified for the class). In order to calculate the sum of the costs for all adverts in a campaign, the statement above that gets the cost for each advert must be executed repeatedly. However, rather than think in terms of operation invocation we use the metaphor of message passing to describe object interaction, as this emphasizes that objects are encapsulated and essentially autonomous. Message passing can be represented on a class diagram, as in Fig. 9.1 where the message is shown as an arrow between two objects. This notation is rather like that of the communication diagram (Section 9.5) and the similarity demonstrates that the latter is a natural extension of the class diagram.

It can be difficult to determine what messages should be sent by each object. In this case, it is clear that the `getCost` operation should be located in the Advert class. This operation uses data that is stored in the `advertCost` attribute, and this has been placed in Advert. We can also readily see that an operation that

² These last two diagram types have been introduced in UML 2.0.



Figure 9.1 Object messaging.

calculates the cost of a Campaign must be able to find out the cost of each Advert involved. But this is a simple interaction and the allocation of these operations is largely dictated by the presence of particular attributes in the classes. More complex requirements may involve the performance of complex tasks where an object receiving one message must itself send messages that initiate further interactions with other objects, but it may not be straightforward as to how these other objects should be involved in the interaction.

It is an aim of object-oriented analysis and design to distribute system functionality appropriately among its classes. This does not mean that all classes have exactly equal levels of responsibility but rather that each class should have appropriate responsibilities. Where responsibilities are evenly distributed, each class tends not to be unduly complex and, as a result, is easier to develop, to test and to maintain. A class that is relatively small and self-contained has a much greater potential for reuse than one that is large, one that is complex or one that has responsibilities or functionality that are not clearly focused.

An appropriate distribution of responsibility among classes has the important side-effect of producing a system that is more resilient to changes in its requirements. When the users' requirements for a system change it is reasonable to expect that the application will need some modification, but ideally the change in the application should be of no greater magnitude than the change in the requirements. An application that is resilient in this sense costs less to maintain and to extend than one that is not. Fig. 9.2 illustrates this.

The aim of modelling object interaction is to determine the most appropriate scheme of messaging between objects in order to support a particular user requirement. As we saw in Chapter 6, user requirements are first documented by use cases. Each use case can be seen as a dialogue between an actor and the system that results in objects performing tasks so that the system can respond in the way that is required by the actor. For this reason many interaction diagrams explicitly include objects to represent the user interface (boundary objects) and to manage the object interaction (control objects). When such objects are not shown explicitly, it can be assumed in most cases that they will need to be identified at a later stage. The identification and specification of boundary objects is in part an analysis activity and in part a design activity. When we are analyzing requirements, our concern is to identify the nature of a dialogue in terms of the user's need for information and his or her access to the system's functionality. Deciding how an interaction should be realized in software will involve the detailed design of boundary objects that manage the dialogue and the introduction of other objects to enable the effective execution of the interaction. This is discussed in detail in Chapters 16 and 17.

UML (OMG, 2004c) discusses object interaction within the context of a collaboration. Collaborations have already been introduced in Section 7.3.1. Essentially a collaboration is a group of objects or classes that work together to provide an element of functionality or behaviour. The behaviour mentioned above can be that

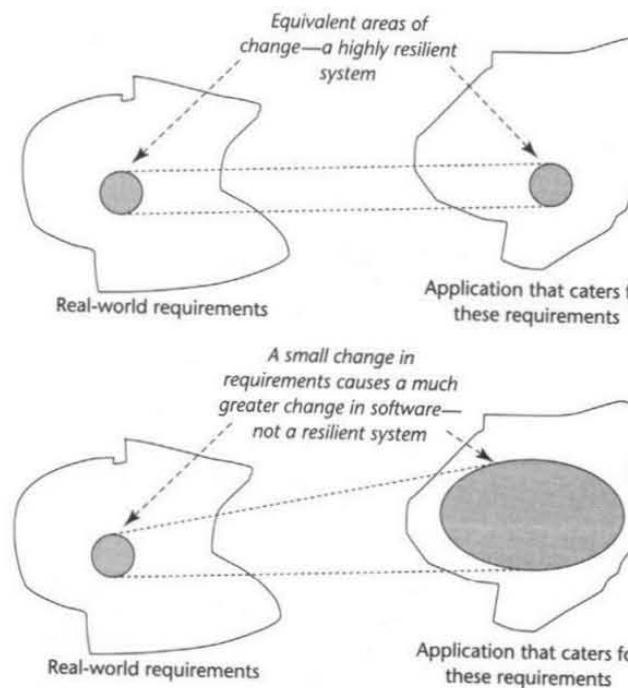


Figure 9.2 Resilience of a design.

of an operation or a use case (or any other behavioural classifier in UML). A particular object instance may play different roles in different contexts or interactions and may even play more than one role in a given interaction. Figure 9.1 illustrates a very simple collaboration showing the two participating objects and the link between them. Strictly speaking a collaboration does not show the messages that are being sent.

In order to illustrate the preparation of interaction diagrams we build on the CRC card analysis of the use case Add a new advert to a campaign that was discussed in Section 7.6. The use case description used in Chapter 7 is repeated here.

The Campaign Manager selects the required Campaign for the Client concerned and adds a new Advert to the existing list of adverts for that campaign. The details of the Advert are completed by the Campaign Manager.

The resulting CRC cards are shown in Fig. 7.30. These form the basis for the interaction sequence diagrams that are developed in the next two sections.

9.3 Interaction Sequence Diagrams

An *interaction sequence diagram* (or simply a *sequence diagram*) is one of several kinds of UML interaction diagram. The sequence diagram is semantically equivalent

to a communication diagram for simple interactions. An interaction specifies the communication patterns amongst a set of objects or systems that are participating in a collaboration. The interaction is described by a partially ordered³ sequence of the messages between objects or communicating roles. This definition of an interaction is in terms of communicating roles and highlights the fact that these concepts can be applied in various contexts. Commonly, during requirements analysis or interaction design, object instances are modelled in terms of the roles they play and communicate by message passing.

A sequence diagram shows an interaction between objects arranged in a time sequence. Sequence diagrams can be drawn at different levels of detail and to meet different purposes at several stages in the development lifecycle. The commonest application of a sequence diagram is to represent the detailed object interaction that occurs for one use case or for one operation. When a sequence diagram is used to model the dynamic behaviour of a use case it can be seen as a detailed specification of the use case. Those drawn during analysis differ from those drawn during design in two major respects. Analysis sequence diagrams normally do not include design objects; nor do they usually specify message signatures in any detail.

9.3.1 Basic concepts and notation

Figure 9.3 shows a sequence diagram for the use case Add a new advert to a campaign. The vertical dimension represents time and all objects⁴ involved in the interaction are spread horizontally across the diagram. (The horizontal ordering of objects is arbitrary and has no modelling significance, though it improves readability if the interaction tends to proceed from left to right.) Time normally proceeds down the page. Each object (or Classifier Role) in a sequence diagram is represented by a *lifeline*. This is a vertical dashed line with an object symbol at the top. The names of lifelines are not underlined even if the lifeline refers to an object. A message is shown by a solid horizontal arrow from one lifeline to another and is labelled with the message name. Each message name may optionally be preceded by a sequence number that represents the sequence in which the messages are sent, but this is not usually necessary on a sequence diagram since the message sequence is already conveyed by their relative positions along the time axis.

When a message is sent to an object, it invokes an operation⁵ of that object. The message name is usually the same as the operation that is being invoked. We will give examples of detailed specifications later. Once a message is received, the operation that has been invoked begins to execute. The period of time during which an operation executes is known as an activation or an execution occurrence and is shown on the sequence diagram by a rectangular block laid along the lifeline. The activation period of an operation includes any delay while the operation waits for a response from another operation that it has itself invoked as part of its execution.

Figure 9.3 shows a straightforward sequence diagram drawn without boundary or control objects. The sequence diagram is drawn within a rectangle known as a

³ Partially ordered means that the messages may be placed in a time sequence and two or more messages may be sent at the same time.

⁴ More generally an interaction may involve *subsystems* or *other connectable elements*.

⁵ Though not explicitly specified in the UML specification, interaction sequence diagrams may be used to model interaction at the level of responsibilities as well.

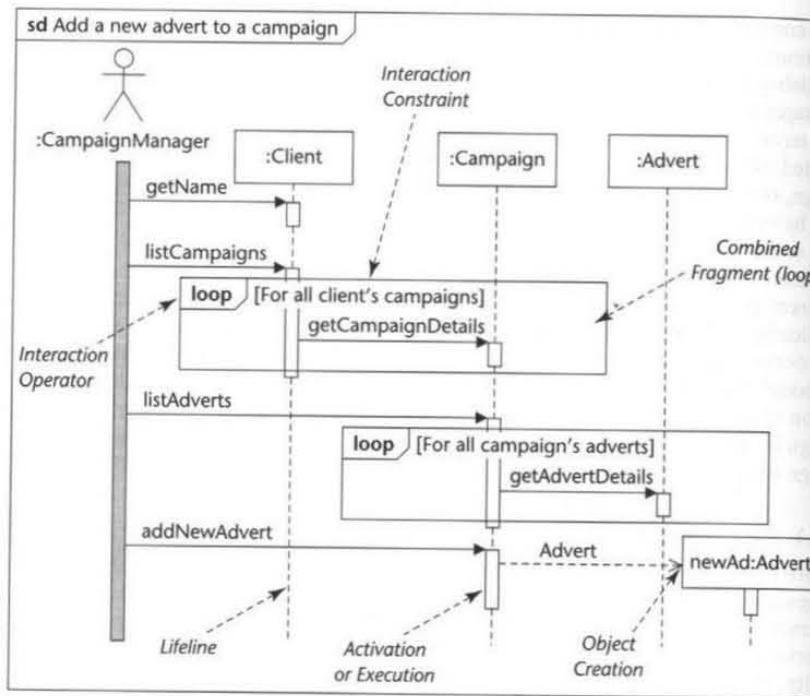


Figure 9.3 Sequence diagram for the use case Add a new advert to a campaign.

frame. The sequence diagram has been given the same name as the use case whose interaction it models, namely Add a new advert to a campaign.

The general format for the heading of the frame is

```
[kind] name [ parameters ]
```

Terms in square brackets [...] are optional.

In this example the kind of frame is a sequence diagram⁶ and we are using the shortened form **sd**. The name field is the name of the use case that the interaction represents. There are no parameters for this sequence diagram.

The `getName` message is the first message received by the Client and is intended to correspond to the Campaign Manager requesting the name of the selected Client. The Client object then receives a `listCampaigns` message and a second period of operation activation begins. This is shown by the tall thin rectangle that begins at the message arrowhead. The Client object now sends a message `getCampaignDetails` to each Campaign object in turn in order to build up a list of campaigns. This repeated action is called an iteration. It is represented by enclosing the repeated messages inside a frame with the heading `loop`. The interaction in this type of frame is known as a *combined fragment*. The keyword

⁶ A frame may be used for all UML diagram types. If the diagram has a naturally implied boundary then the frame may be omitted. Frames are used for interaction diagrams to make it clearer how they are combined.

loop is an example of an *interaction operator* that specifies the type of the combined fragment. We will introduce other interaction operators later. The conditions for continuing or ending an iteration may be shown beside the frame's heading.

For the loop that retrieves all the campaign details for a client the guard condition is

[For all client's campaigns]

This is an example of an interaction constraint. A combined fragment with an interaction constraint will only execute if the constraint is evaluated as true.

The Campaign Manager next sends a message to a Campaign object asking it to list its advertisements. The Campaign object delegates responsibility for getting the advertisement title to each Advert object, although the Campaign object retains responsibility for the list as a whole (indicated by the continuation of the activation bar beyond the point where the message is sent).

When an advertisement is added to a campaign an Advert object is created. This is shown by the Advert construction arrow (this invokes the constructor⁷ operation) drawn with its arrowhead pointing directly to the object symbol at the top of the lifeline. Where an object already exists prior to the interaction the first message to that object points to the lifeline below the rectangle at the top of the lifeline. For example, this is the case for the Campaign object, which must exist before it can receive an `addNewAdvert` message. The sequence diagram in Fig. 9.3 corresponds directly to the interaction suggested by the CRC cards in Fig. 7.30. In effect, the sequence diagram is a more formal representation of the same interaction, but with the messages and the sequence of interaction both made explicit.

Figure 9.4 shows the basic notation for sequence diagrams. The messages that we have been considering so far are synchronous messages or blocking calls. The sending operation is suspended while the operation invoked by the message is executing. This is essentially a nested flow of control where the complete nested sequence of operations is completed before the calling operation resumes execution. This may be because the invoking operation requires data to be returned from the destination object before it can proceed. Formally sending the message is an example of a *send message event* and receiving the message is a *receive message event*. The execution occurrence starts with an *execution occurrence start event* and ceases with an *execution occurrence end event*. A reply message (with a dashed line) representing a return of control after the execution occurrence has ended is shown in the diagram.

This diagram also shows that the state of a lifeline can be shown on the lifeline in a sequence diagram. This is used to represent the constraint that the lifeline must be in the Active state before the message msg a can be accepted.

Most use cases imply at least one boundary object that manages the dialogue between the actor and the system. In Chapter 17 we show a number of sequence diagrams that explicitly include interface classes, but these have been prepared from a design perspective. For example, Fig. 17.15 shows a further revision to the

⁷ A constructor operation creates an object instance. In the executable system a constructor typically allocates memory for the new object and initializes attribute values. It is conventional in object-oriented programming languages to name constructors with the class name; hence the constructor operation name begins with a capital letter as does the class name.

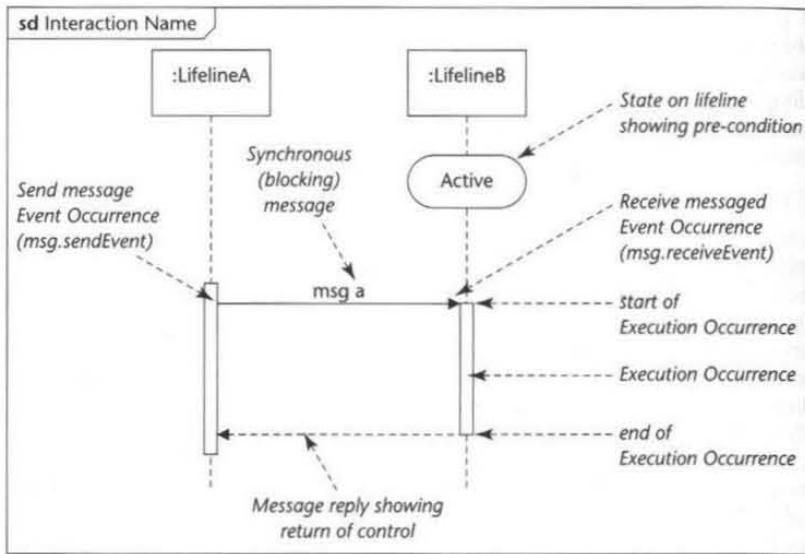


Figure 9.4 Sequence diagram showing message and execution events and states.

sequence diagram for the use case Check campaign budget. If this is compared with the version shown in Fig. 9.7, it can be seen that the management of the interaction between the user and the system is managed explicitly by the :CheckCampaignBudgetUI class in Fig. 17.15, whereas in Fig. 9.7 the dialogue management is not shown at all. Some methodologies (for example USDP, which is described in Chapter 21) suggest that boundary classes should be identified during the analysis, while other approaches take the view that this is primarily a feature of design.

Figure 9.5 shows an alternative sequence diagram for the use case Add a new advert to a campaign but drawn this time with boundary and control objects. Essentially this is in the style of the Unified Software Development Process. The boundary object, representing the user interface, is :AddAdvertUI. We have used the suffix UI to mean user interface. The control object is :AddAdvert and this manages the object communication. Although not shown in Fig. 9.5, the interaction is initiated by the creation of the control object :AddAdvert. This then gets the client details before initiating the dialogue by creating the boundary class :AddAdvertUI.

Objects may be created or destroyed at different stages during an interaction⁸. On a sequence diagram the destruction of an object is indicated by a large X on the lifeline at the point in the interaction when the object is destroyed. An object may either be destroyed when it receives a message or it may self-destruct at the end of an execution occurrence if this is required by the operation that is being executed (this is shown in Fig. 9.6).

⁸ Destructor operations are discussed in more detail in Chapter 14.

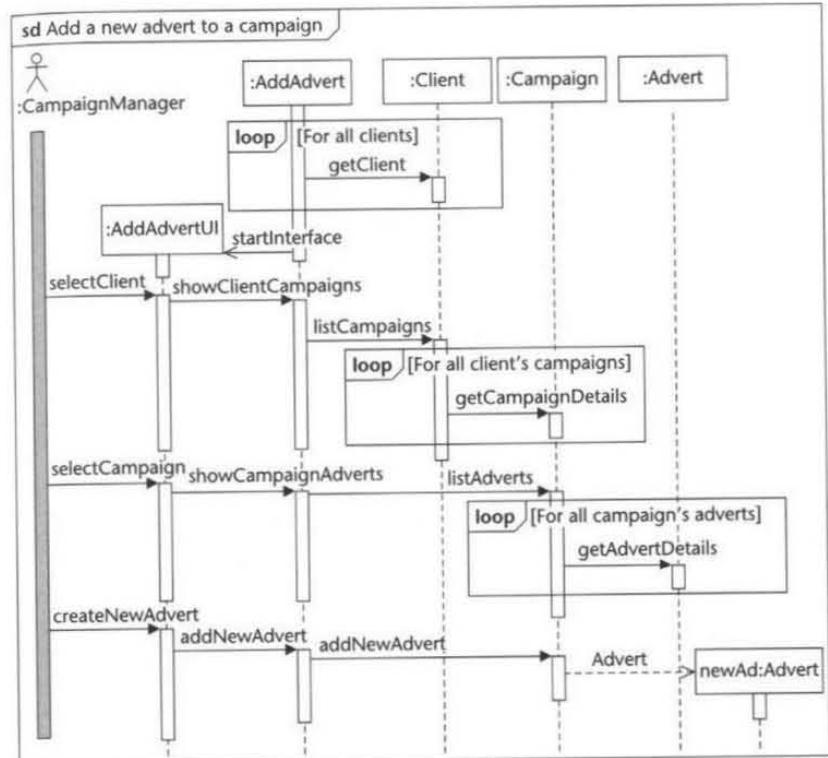


Figure 9.5 Sequence diagram for the use case Add a new advert to a campaign with boundary and control classes.

An object can send a message to itself. This is known as a *reflexive message* and is shown by a message arrow that starts and finishes at the same object lifeline. The use case Check campaign budget includes an example of this. For ease of reference the use case description is repeated below.

The campaign budget may be checked to ensure that it has not been exceeded. The current campaign cost is determined by the total cost of all the adverts and the campaign overheads.

The corresponding sequence diagram is shown in Fig. 9.7 and this includes a reflexive message getOverheads sent from a Campaign object to itself.

In this case the reflexive message invokes a different operation from the operation that sent the message and a new activation symbol is stacked on the original execution occurrence. (This is the shorter second rectangle shown offset against the first execution occurrence.) In certain circumstances an operation invokes itself on the same object; this is known as recursion and can be similarly represented but is not illustrated here⁹.

⁹ If a limit has been set to the depth of recursion this can be recorded in a note on the diagram.

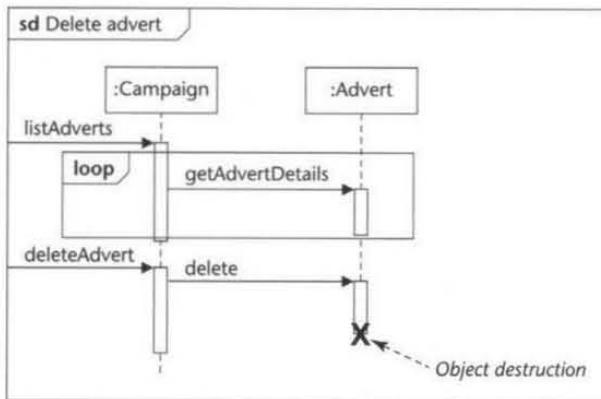


Figure 9.6 Object destruction.

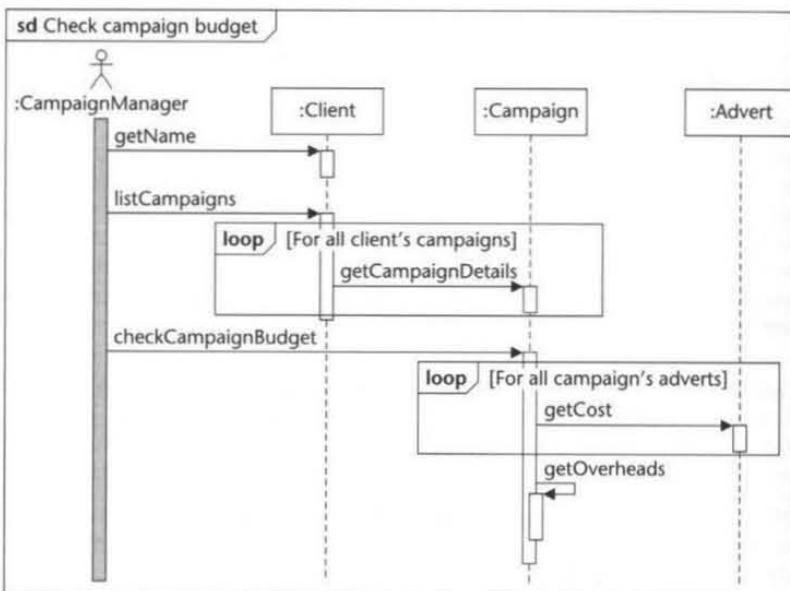


Figure 9.7 Sequence diagram for the use case Check campaign budget.

Until this point our discussion has centred on simple use cases and correspondingly simple interactions. These are typical of many modelling situations, but more complex interactions also occur in many systems. It is sometimes also necessary to represent in more detail the synchronization of messages. Figure 9.8 illustrates some variations in the UML notation.

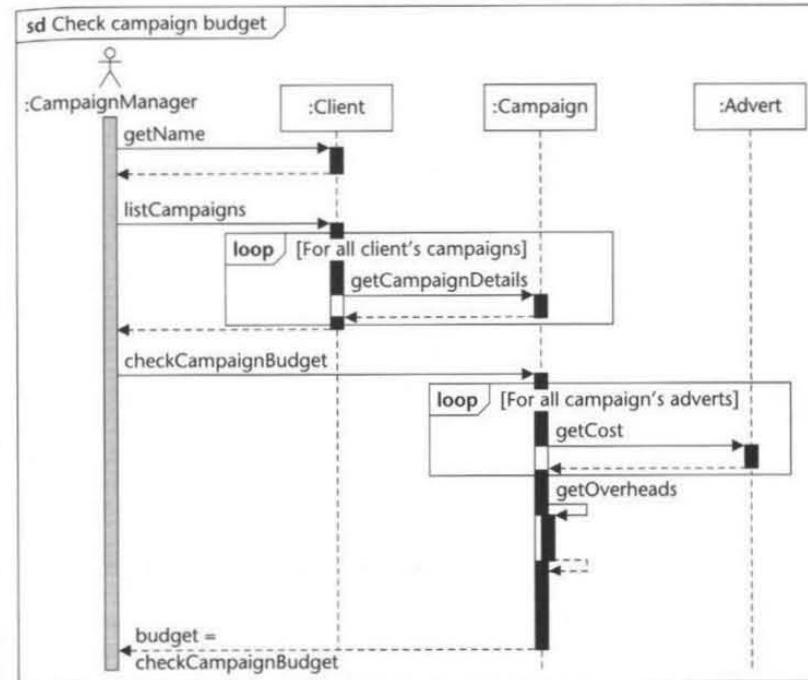


Figure 9.8 Sequence diagram for Check campaign budget with foci of control and explicit returns shown.

The *focus of control* indicates times during an execution occurrence when processing is taking place within that object. Parts of an execution occurrence that are not within the focus of control represent periods when, for example, an operation is waiting for a return from another object. The focus of control may be shown by shading those parts of the activation rectangle that correspond to active processing by an operation. In Fig. 9.8 the Check campaign budget use case is redrawn with foci of control shaded. The focus of control for the *checkCampaignBudget* operation is initially with the Campaign object, but is then transferred to the Advert object and the activation rectangle in the Campaign object is now unshaded while the Advert object has the focus of control. The *checkCampaignBudget* activation is also unshaded while the *getOverheads* operation is activated by the reflexive message *getOverheads*.

A *reply* is a return of control to the object that originated the message that began the activation. This is not a new message, but is only the conclusion of the invocation of an operation. Replies are shown with a dashed arrow, but it is optional to show them at all since it can be assumed that control is returned to the originating object at the end of the activation in a destination object (asynchronous messages—see Section 9.3.4—are exceptions). Replies are often omitted, as in Fig. 9.7. Figure 9.8 explicitly shows all replies for the same interaction.

A *return-value* is the value that an operation returns to the object that invoked it. These are rarely shown on an analysis sequence diagram, and are discussed further in Chapter 10. For example, in Fig. 9.8 the operation invoked by the message `getName` would have return-value of `clientName` and no parameters. In order to show the return-value the message could be shown as

```
clientName = getName
```

where `clientName` is a variable of type `Name`. The formal message label syntax is as follows:

```
[attribute=] signal-or-operation-name [(argument-list)]  
[return-value] /*
```

where return-value and attribute assignment are only used on reply messages. The alternative message label of '*' is used to represent a message of any type being sent.

Lifelines in sequence diagrams may represent different types of UML element depending upon the scope and purpose of the sequence diagram; the choice of lifelines is very much up to the systems modeller. Although a lifeline may represent an object, the lifeline name is not underlined¹⁰. In most of the examples used so far the lifeline name has been that of the object it is representing, for example `:Campaign` in Fig. 9.8. In Fig. 9.5 the lifeline representing the newly created advertisement is named `newAd:Advert`. This could have been named simply `newAd` if we felt that it was not necessary to show explicitly that this was an object of the class `Advert`. In Fig. 9.9 the lifeline representing the `Campaign` class objects is named `campaign[i]:Campaign` to show explicitly that a set of `Campaign` objects will one after the other be involved in the interaction. In Fig. 9.15 the lifeline that is representing the subsystem `ClientCampaignAds` is named `:ClientCampaignAds ref ClientCampaignAds`. Here the keyword `ref` is referring the reader to another sequence diagram `ClientCampaignAds`.

Interaction constraints may be represented also in various formats. In Fig. 9.7 the first loop combined fragment has its interaction constraint specified as

```
[For all client's campaigns]
```

This is a straightforward statement of the condition that must be true for the combined fragment to execute. Alternatively, as in Fig. 9.9 the interaction constraint for the same combined fragment is more formally stated as

```
[i=1;i=<campaigns.count;i++ ]
```

This is stating that the loop will iterate from 1 to the value of `campaign.count`, which holds the number of campaigns associated with that particular client. This links in with the object selector naming for the lifeline `campaign[i]:Campaign`, where the first iteration involves the first object and so on.

In Fig. 9.10 the loop interaction operator is shown with the parameters. The first parameter is the minimum number of iterations and the second is the maximum number of iterations. This interaction constraint for the first combined fragment is stated as

```
[i=<campaign.count]
```

¹⁰ This is a change from UML 1.X.

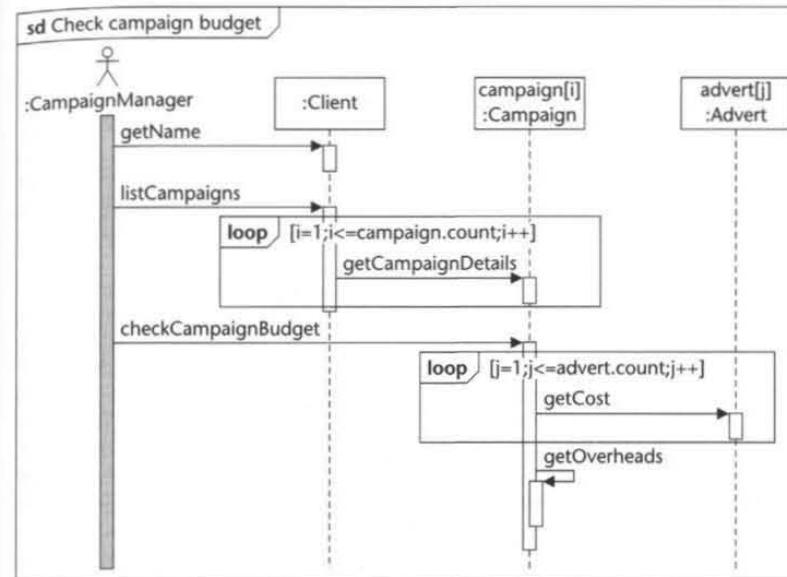


Figure 9.9 Sequence diagram for Check campaign budget with object selector notation shown.

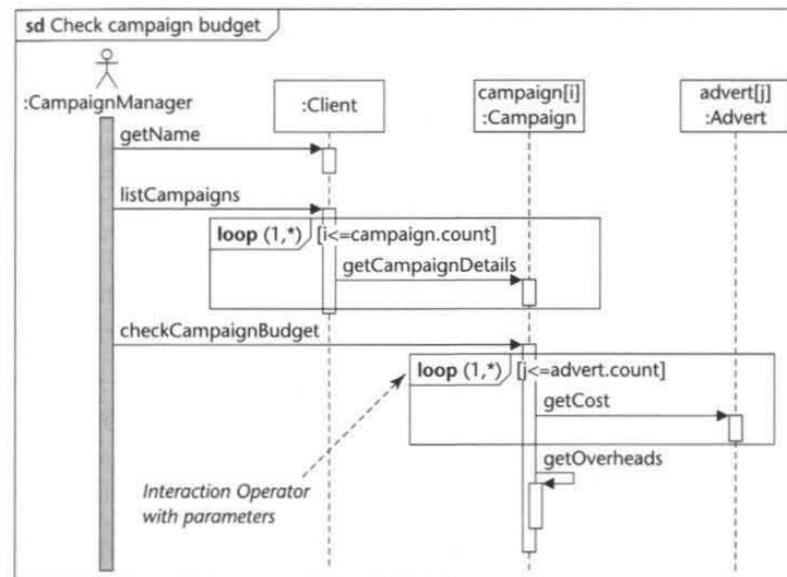


Figure 9.10 Sequence diagram for Check campaign budget with an interaction operator with parameters.

The loop operand will stop executing when the interaction constraint is not true and it has executed at least the minimum number of times. These three styles of notation are all valid.

A *synchronous message or procedural call* is shown with a full arrowhead (see Fig. 9.4) and is one that causes the invoking operation to suspend execution until the focus of control has been returned to it. In Fig. 9.8 the Check campaign budget interaction is shown with procedural calls and explicit returns. Procedural calls are appropriate for the interaction since each operation that invokes another does so in order to obtain data and cannot continue until that data is supplied.

9.3.2 Branching

The interactions that we have considered so far have only one execution path, although some have iterations during their execution. Some interactions have two or more alternative execution pathways. Each reflects a branch in the possible sequence of events for the use case it represents. The notation for branching is illustrated in Fig. 9.11. This shows a sequence diagram for the use case Add a new advert to a campaign if within budget. The relationship between this use case and the use case Add a new advert to a campaign will result in a change to

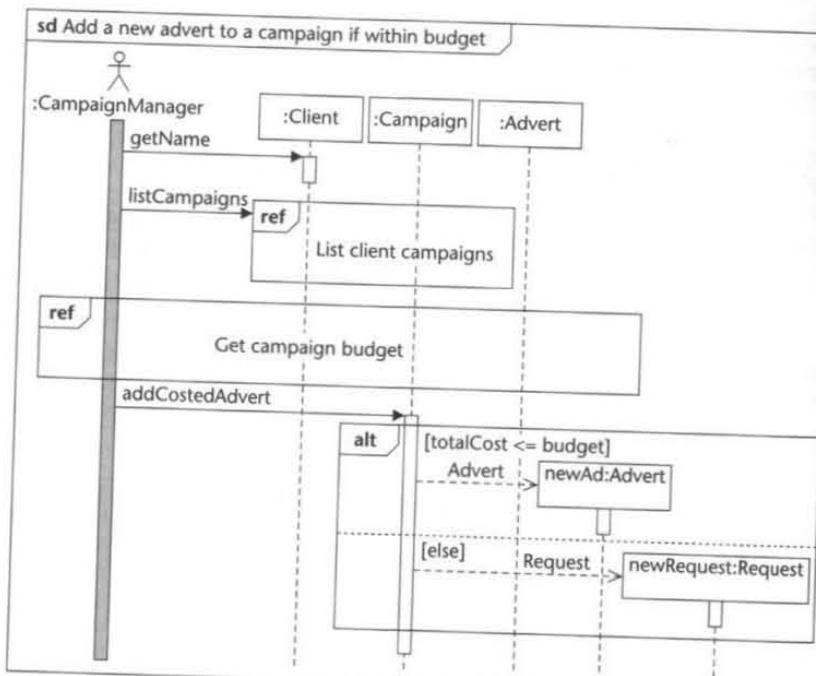


Figure 9.11 Sequence diagram for Add a new advert to a campaign if within budget showing branching.

the use case diagram in Fig. A2.2, where Add a new advert to a campaign if within budget needs to be added and shown with an «include» relationship with Check campaign budget. The use case description is as follows.

A new advertisement is added to a campaign by the campaign manager only if the campaign budget is not exceeded by adding the new advert. If adding the advertisement would cause the budget to be exceeded then a campaign budget extension request is generated. This will be recorded for later reference. The budget extension request is printed and sent to the client at the end of the day.

The first part of this sequence diagram uses *interaction occurrences* which refer to the other sequence diagrams List client campaigns and Get campaign budget. The use of interaction occurrences is explained in the next section.

The branching is shown in the combined fragment named with the keyword *alt* which is a short form of *alternatives*. The combined fragment has two (or more) compartments known as *interaction operands*. Each operand corresponds to one of the alternatives in the combined fragment and each operand should have an interaction constraint to indicate under what conditions it executes. The sequence of the operands is not significant. In this example there are only two operands. The interaction constraint *[else]* can be used as the default in the second or last operand if there are more than two. The first operand deals with the case where the campaign is within budget and a new object of the class Advert is created by the message *Advert*. The second operand represents the case when the budget is spent or exceeded and a request to add a new advert has to be created. This is shown by the message *Request* creating the *newRequest* instance.

The branching notation can be used at a generic level to create a sequence diagram that represents all possible sequences of interaction for a use case. Such a generic diagram will typically be showing communication between anonymous objects or roles rather than particular instances. In general, looping and branching constructs correspond respectively to iteration and decision points in the use case. When drawn at an instance level a sequence diagram shows a specific interaction between specific objects. The two kinds of sequence diagram (generic and instance level) are equivalent to one another if the interactions implied by the use case contain no looping or branching constructs.

9.3.3 Managing sequence diagrams

On occasions it is necessary to represent complex or large interactions using two or more sequence diagrams. It may be that a single interaction is too complex to represent in a single sequence diagram. Perhaps there are interaction fragments that are common to several interactions and it is more effective to model these common interaction fragments only once. Another possibility is that part of the interaction involves complex messaging between members of a group of objects and that this part of the interaction is best shown separately.

The first approach to modelling an interaction with more than one sequence diagram is to use interaction occurrences as shown in Fig. 9.12. Here there are two interaction occurrences List client campaigns and Get campaign budget. The keyword *ref* indicates that they are interaction occurrences and that they are referring to the sequence diagrams List client campaigns and Get campaign

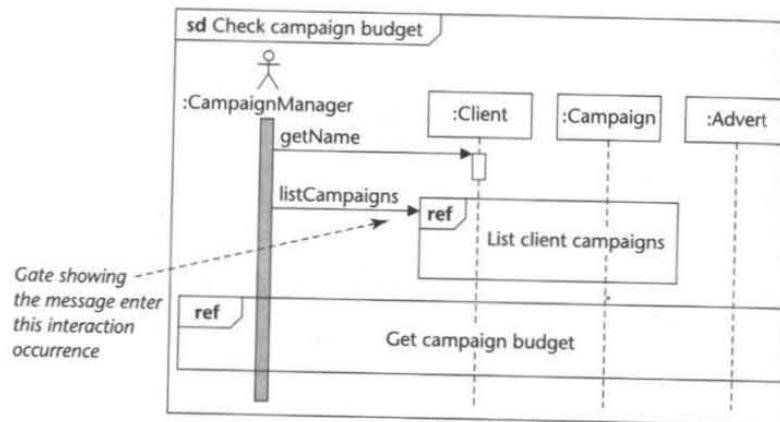


Figure 9.12 Sequence diagram for the interaction Check campaign budget with interaction occurrences.

budget. These two sequence diagrams are shown in Figs 9.13 and 9.14 respectively. Each of these is an example of an *interaction fragment*, that is, they can be used as part of one or more larger interactions. The interaction fragment *List client campaigns* in Fig. 9.13 could clearly be reused in any interaction that requires a list of the campaigns for a specific client.

In Fig. 9.12 the message *listCampaigns* is being sent to the interaction occurrence *List client campaigns*. This same message enters the corresponding interaction fragment in Fig. 9.13 and is shown coming from the edge of the frame. The point where a message goes to an interaction occurrence and from where a message enters an interaction fragment is known as a *gate*.

The interaction occurrence *Get campaign budget* in Fig. 9.12 does not have any messages going to it so there are no gates shown on the interaction occurrence or in the interaction fragment in Fig. 9.14. This interaction fragment has been chosen for illustrative purposes and has one clear disadvantage. It contains the

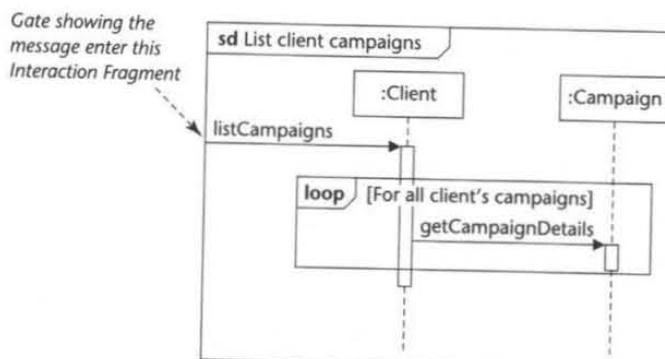


Figure 9.13 Sequence diagram for the Interaction Fragment List client campaigns.

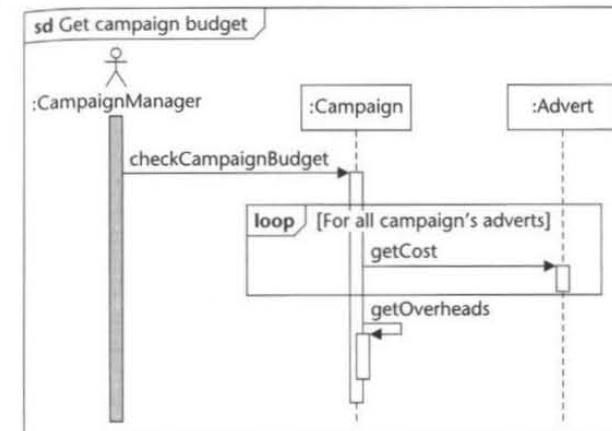


Figure 9.14 Sequence diagram for the Interaction Fragment Get campaign budget.

actor *:CampaignManager* and consequently can only be used in interactions where the campaign manager is getting the campaign budget. If the interaction did not contain *:CampaignManager*, it could be used for interactions where, for example, the account manager wished to know the campaign budget.

Complex interactions can also be split up using lifelines to represent groups of objects and their interaction or to represent subsystems. Figure 9.15 shows the interaction *Add a new advert to a campaign* where the messaging between the

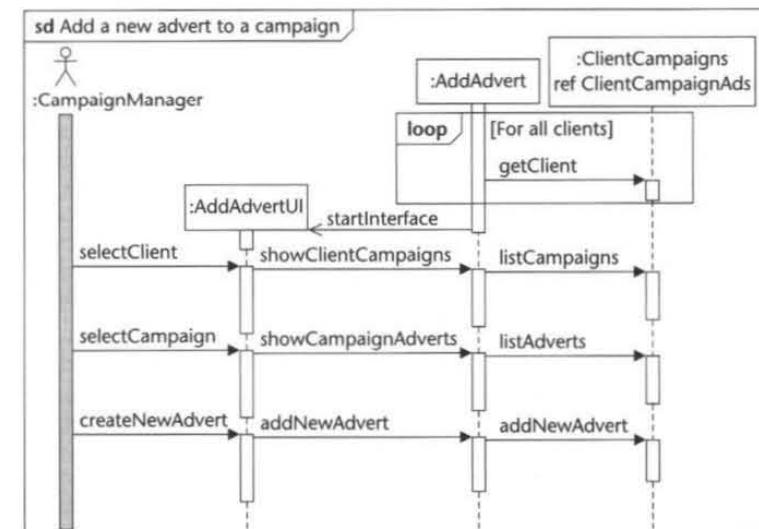


Figure 9.15 Sequence diagram with the detail of the interaction in the *ClientCampaignAds* subsystem not showing.

:Client, :Campaign and :Advert lifelines are hidden within the :ClientCampaigns lifeline. The :ClientCampaigns lifeline references the sequence diagram ClientCampaignAds for the hidden detail. This is shown in Fig. 9.16. Note that the messages that are received by the lifeline :ClientCampaigns enter the interaction fragment ClientCampaignAds through gates.

UML 2.0 provides further notation to allow sequence diagrams to be linked. *Continuations* can be specified within an alt combined fragment to allow a link back to a referring sequence diagram. In Fig. 9.17 the continuations Within budget and Budget spent are used to link the two sequence diagrams. The sequence diagram Authorize expenditure includes the interaction occurrence Calculate costs. At this point in Authorize expenditure the detail of the interaction is shown in Calculate costs. The interaction in Calculate costs proceeds with the two getCost messages. If the cost is within budget the interaction constraint in the first operand of the alt combined fragment is true and the interaction occurrence Identify under spend is followed. (This interaction fragment is just illustrative and is not specified.) The flow of the interaction now comes to the continuation Within budget. The interaction now moves back to sequence diagram Authorize expenditure, continues from the point of the continuation with the same name and the message authorize is sent. In a similar way if the cost is not within budget the [else] operand within Calculate costs is satisfied and control is returned to Authorize expenditure at the Budget

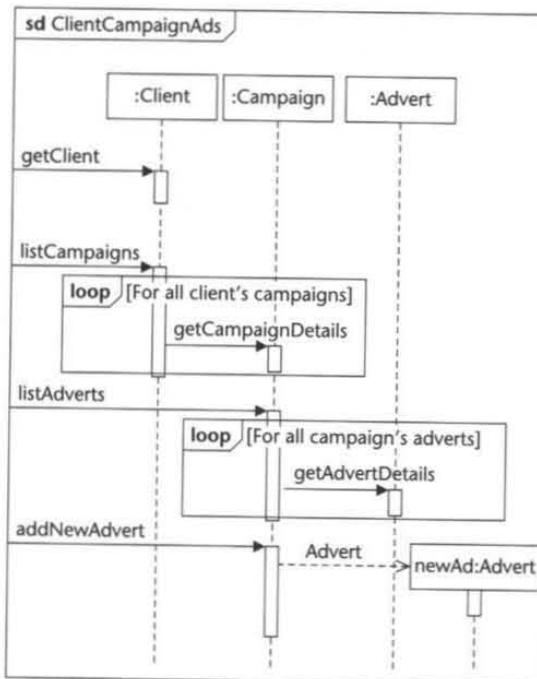


Figure 9.16 Sequence diagram showing the interaction for ClientCampaignAds subsystem.

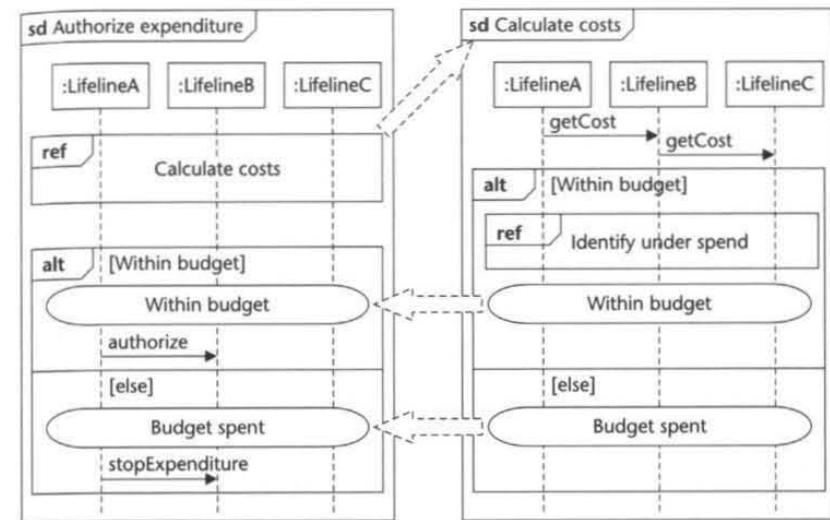


Figure 9.17 An example showing the use of continuations.

spent continuation. The dashed arrows between the two sequence diagrams are not part of UML and are only intended to illustrate the transfer of execution between the diagrams.

9.3.4 Asynchronous messages

So far we have been dealing only with synchronous messages. An *asynchronous message*, or *signal* drawn with an open arrowhead as in Fig. 9.18, does not cause the invoking operation to halt execution while it awaits a return. When an asynchronous message is sent, operations in both objects may carry out processing at the same time. Asynchronous messages are frequently used in real-time systems where operations in different objects must execute concurrently, either for reasons of efficiency or because the system simulates real-world activities that also take place concurrently. It may be necessary for an operation that has been invoked asynchronously to notify the object that invoked it when it has terminated. This is done by explicitly sending a message (known as a *callback*) to the originating object.

9.3.5 Time constraints

A sequence diagram can be labelled to document it in various ways. Labels may be included using the note notation in UML with, for example, descriptions or explanations of aspects of the diagram or model. Time constraints may apply to the execution of operations or other elements of interaction diagrams. In Fig. 9.18 each of the messages is simply named with signalE, signalF and so on. Time expressions may be associated with the name of the message so that duration constraints can be specified for the execution of an operation or the transmission of a message. Construction marks may also be used to show a time interval with a

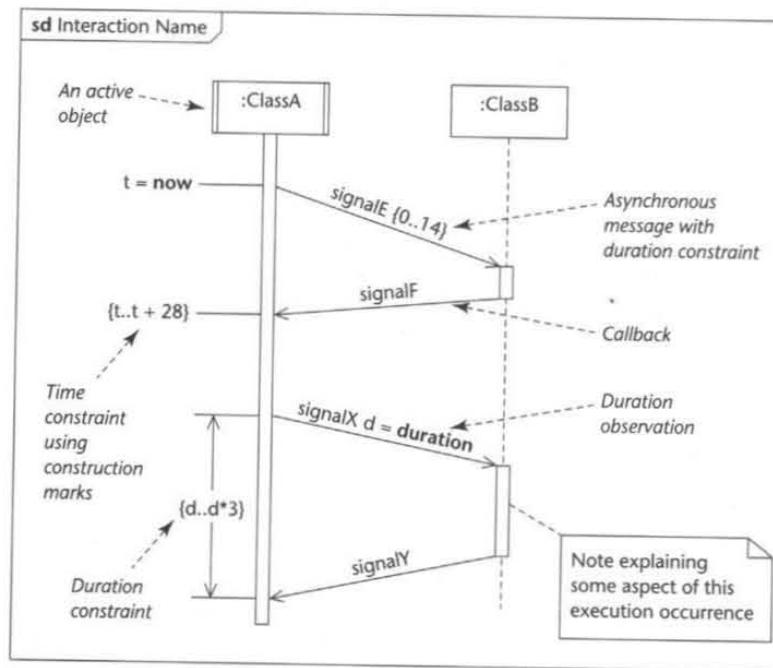


Figure 9.18 Sequence diagram showing different message types and time constraints.

constraint. This is illustrated in Fig. 9.18 to show the interval between the sending `signalE` and receiving `signalF`. Time constraints are frequently used in modelling real-time systems where the application must respond within a certain time, typically for reasons of safety or efficiency. For most other information systems the sequence of the messages is the most important issue.

So far we have only considered message arrows that have been drawn horizontally across the sequence diagram and at right angles to the object lifelines. Drawing a message arrow in this fashion indicates that the time taken to send a message is not significant in comparison to the time taken for operation execution. There is consequently no need to model another activity during the period while a message is in transit. In some applications the length of time taken to send a message is itself significant. For example, in distributed systems messages are sent over a network from an object on one computer to another object on a different computer. If the transit time for a message is significant, the message arrow is slanted downwards so that the arrowhead (the arrival of the message) is below (later than) the tail (the origination of the message). The asynchronous messages (e.g. `signalE`) shown in Fig. 9.18 illustrate this. The message `signalE` is constrained to take between 0 and 14 time units by the duration constraint shown in Fig. 9.18. The duration of `signalX` is shown as taking d time units and a duration constraint has been specified for the interval from sending `signalX` and receiving `signalY`. This duration constraint states that the interval must be between d and $d \times 3$ time units.

9.3.6 Modelling real-time systems and concurrency

Real-time systems are broadly characterized by the need to respond to external events within tight time constraints. Partly for this reason, they frequently exhibit concurrent behaviour in the form of simultaneous execution pathways or *threads of control*. An application that has concurrent execution always includes some objects that co-ordinate and initiate threads of control; these are *active objects*. In addition, real-time applications usually include many other objects that work only within a thread of control; these are known as *passive objects*. Active objects or classes are shown with a double line at each side of the head of the lifeline in an interaction diagram. The lifeline `:ClassA` in Fig. 9.18 is an example of an active object. Active objects can continue to operate without the invocation of operations from other objects. They have their own thread of control. Active objects are frequently composites with embedded parts (the interface or boundary class as a composite is discussed in Chapter 17).

It is important for a sequence diagram of a concurrent system to show clearly which threads of control are active at any time. The features that have been added in UML 2.0 provide better capability for modelling real-time systems using interaction diagrams. For example, the introduction of combined fragments to show parallel (keyword `par`), optional (keyword `opt`) and critical activity offers a much richer syntax than was available in previous versions of UML. Figure 9.19 shows a complete list of the interaction operators defined in UML with a short explanation of each. The introduction of timing diagrams also enables clear diagrammatic representation of the timing constraints, state changes and messaging between lifelines.

9.3.7 Guidelines for preparing sequence diagrams

Modelling interaction is an important activity during the information systems development process. The following are some general guidelines for the preparation of sequence diagrams in particular and are adapted from Bennett et al. (2005).

1. Decide at what level you are modelling the interaction. Is it describing an operation, a use case, the messaging between components or the interaction of subsystems or systems?
2. Identify the main elements involved in the interaction. If the interaction is at use case level the collaborating objects may already have been identified through the use of CRC cards and their responsibilities partly allocated. Of course, CRC cards can be used at different levels of granularity to explore the behaviour of any group of lifelines.
3. Consider the alternative scenarios that may be needed. Again a CRC card exercise may be helpful in exploring these.
4. Identify any existing interactions that have already been modelled as sequence diagrams or that will be so that they can be included as interaction occurrences.
5. Draw the outline structure of the diagram.
 - a. Create a frame with a suitable name.

Interaction Operator	Explanation and use
alt	Alternatives represents alternative behaviours, each choice of behaviour being shown in a separate operand. The operand whose interaction constraint is evaluated as true executes.
opt	Option describes a single choice of operand that will only execute if its interaction constraint evaluates as true.
break	Break indicates that the combined fragment is performed instead of the remainder of the enclosing interaction fragment.
par	Parallel indicates that the execution operands in the combined fragment may be merged in any sequence once the event sequence in each operand is preserved.
seq	Weak Sequencing results in the ordering of each operand being maintained but event occurrence from different operands on different lifelines may occur in any order. The order of event occurrences on common operands is the same as the order of the operands.
strict	Strict Sequencing imposes a strict sequence on execution of the operands but does not apply to nested fragments.
neg	Negative describes an operand that is invalid.
critical	Critical Region imposes a constraint on the operand that none of its event occurrences on the lifelines in the region can be interleaved.
ignore	Ignore indicates the message types, specified as parameters, that should be ignored in the interaction.
consider	Consider states which messages should be considered in the interaction. This is equivalent to stating that all others should be ignored.
assert	Assertion states that the sequence of messaging in the operand is the only valid continuation.
loop	Loop is used to indicate an operand that is repeated a number times until the interaction constraint for the loop is no longer true.

Figure 9.19 Interaction operators that may be used with combined fragments.

- b. Add the appropriate lifelines starting with the lifeline that is first involved with the interaction and then placing the others from left to right. This can improve the layout of the sequence diagram. If an actor lifeline is being modelled, then this should be placed first followed by the boundary lifeline, if this is being modelled.
6. Add the detailed interaction.
 - a. Starting from the top of the frame add the first message. Lay out the subsequent messages from top to bottom, showing the appropriate level of detail in the message labels.
 - b. Use combined fragments with appropriate interaction operators to describe, for example, looping, branching and optional paths. The full list of interaction operators is shown in Fig. 9.19. Add interaction constraints to these as necessary.

- c. Identify any interaction fragments that are or will be used in other interactions and place these in separate sequence diagrams. Prepare the sequence diagrams for these interaction fragments so that they are as reusable as possible. Place the corresponding interaction occurrences in the diagram being drawn.
- d. Annotate the diagram with comments where this is necessary, for example to include pre- and post-conditions or to improve readability.
- e. Add state invariants to diagram as required.
7. Check for consistency with linked sequence diagrams and modify as necessary. Sequence diagrams may be linked in various ways that have been described in Section 9.3.3. If the interaction is at the level of a use case it is useful to consider any other use cases that are linked by extend or include dependencies.
8. Check for consistency with other UML diagrams or models, in particular, with the relevant class diagrams (and state machine diagrams if they have been prepared at this point).

Once a first-cut sequence diagram has been produced, it is important to work through these steps again from step 2 to refine the model. For complex interactions it takes several iterations to produce a model that describes the required behaviour unambiguously and clearly.

9.4 Communication Diagrams

Communication diagrams are the second kind of interaction diagram in the UML notation set. They have already been introduced in Chapter 7, where they have been used to represent the collaboration that realizes a use case. We will examine the notation for communication diagrams in more detail here.

9.4.1 Basic concepts and notation

Communication diagrams have many similarities to sequence diagrams. For straightforward interactions they express the same information in a different format and, like sequence diagrams, they can be drawn at various levels of detail and during different stages in the system development process. The most significant difference between the two types of interaction diagram is that a communication diagram explicitly shows the links between the lifelines that participate in a collaboration. (Collaborations have been discussed in Chapter 7.) Unlike sequence diagrams, there is no explicit time dimension and lifelines are represented only by rectangles. Figure 9.20 shows an example of a communication diagram.

In a communication diagram the interaction is drawn on what is essentially a fragment of a class or object diagram, as can be seen in Fig. 9.20. This example is drawn at quite a simple level of detail (but note that it includes a lifeline `:AddAdvertUI` representing a boundary object and the lifeline `:AddAdvert` representing a control object). This level of detail is often sufficient to capture the nature of an interaction. Since the diagram has no time dimension the order in which messages are sent is represented by sequence numbers. In this diagram the sequence numbers are written in a nested style (for example, 3.1 and 3.1.1) to

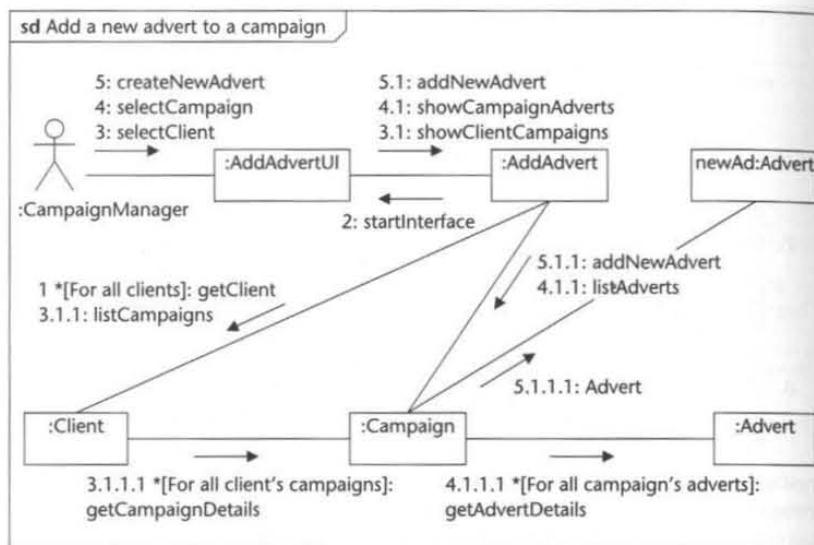


Figure 9.20 Communication diagram for the use case Add a new advert to a campaign.

indicate the nesting of control within the interaction that is being modelled. Thus the operation `showCampaignAdverts` passes control to the operation `listAdverts`, which has one deeper level of nesting. A similar style of numbering is used to indicate branching constructs.

The notation for iteration is different in communication diagrams and for message 3.1.1.1 is

* [For all client's campaigns]

where the * indicates iteration. The message label syntax is discussed in more detail in the next section.

Typically, there is more than one possible interaction for a particular use case and each of the alternative interactions will have different strengths and weaknesses. The alternatives arise because of the different possible allocations of responsibility. For example, although feasible, the interaction in Fig. 9.20 may have some undesirable features. The message `getCampaignDetails` from `:Client` to `:Campaign` requires the `:Client` lifeline to return these details to the `:AddAdvert` lifeline. If the campaign details only include the campaign names, then a relatively small amount of data is being passed from `:Campaign` to `:Client` and then on to `:AddAdvert`. This may be acceptable. On the other hand, if the campaign details also include the start and finish dates for each campaign and the campaign budget then much more data is being passed through `:Client`. In these circumstances `:Client` is now responsible for providing significant amounts of data about the campaigns; arguably this should be the responsibility of the `:Campaign` lifelines themselves. An alternative interaction can pass data about campaigns directly from `:Campaign` to `:AddAdvert`. This alternative interaction

is shown in Fig. 9.21, where :AddAdvert takes the responsibility for getting the campaign details directly from the :Campaign lifelines. In this interaction the Client object is only responsible for providing :AddAdvert with a list of its campaigns¹¹. This is an appropriate responsibility for :Client.

9.4.2 Message labels in communication diagrams

Messages on a communication diagram are represented by a set of symbols that are the same as those used in a sequence diagram, but with some additional elements to show sequencing and recurrence as these cannot be inferred from the structure of the diagram. Each message label includes the message signature and also a sequence number that reflects call nesting iteration, branching and concurrency within the interaction. The formal message label syntax is as follows:

sequence-expression [attribute=] signal-or-operation-name
[(argument-list)] [return-value]

Guard conditions may be written in Object Constraint Language (OCL) (see Chapter 10), and are only shown where the enabling of a message is subject to the defined condition.

A *sequence-expression*¹² is a list of integers separated by dots ('.') optionally followed by a *name* (a single letter), optionally followed by a *recurrence term* and terminated by a colon. A *sequence-expression* has the following syntax:

```
integer [x] integer} [name] [recurrence] ::
```

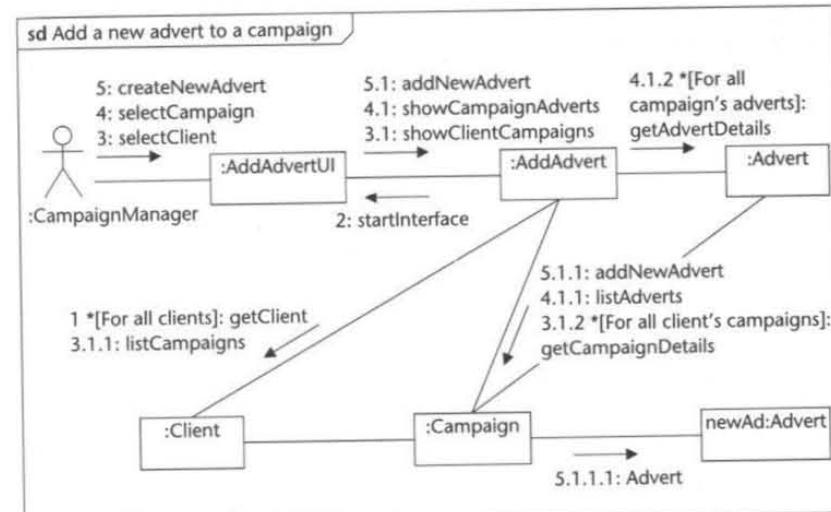


Figure 9.21 Alternative communication diagram for the use case Add a new advert to a campaign.

11 This will be a list of object identifiers that is then used by the `AddAdvert` to navigate to each `Campaign` object in turn.

12 Sequence-expressions are commonly omitted from interaction sequence diagrams as the sequence is normally implied by the relative position of the messages one after the other or by using combined fragments with interaction operators.

In this expression *integer* represents the sequential order of the message. This may be nested within a loop or a branch construct, so that, for example, message 5.1.3 occurs after message 5.1.2 and both are contained within the activation of message 5.1. In Fig. 9.21 messages 4.1.1 and 4.1.2 are nested within the activation of message 4.1. The *name* of a sequence-expression is used to differentiate two concurrent messages since these are given the same sequence number. For example, messages 3.2.1a and 3.2.1b are concurrent within the activation of message 3.2. *Recurrence* reflects either iterative or conditional execution and its syntax is as follows:

Branching: '['guard']'
Iteration: '*'['iteration-clause']'

Some further sample message labels are listed in Fig. 9.22.

Figure 9.23 illustrates the use of a communication diagram to show the interaction for a single operation, in this case *checkCampaignBudget*, which is one of the operations shown in the sequence diagrams in Fig. 9.7.

Type of message	Syntax example
Simple message	4: addNewAdvert
Nested call with return-value. The return value is placed in the variable name	3.1.2: name = getName
Conditional message. This message is only sent if the condition [balance > 0] is true	5 [balance > 0]: debit(amount)
Iteration	4.1 * [For all adverts]: getCost

Figure 9.22 Examples of the syntax for various types of message label.

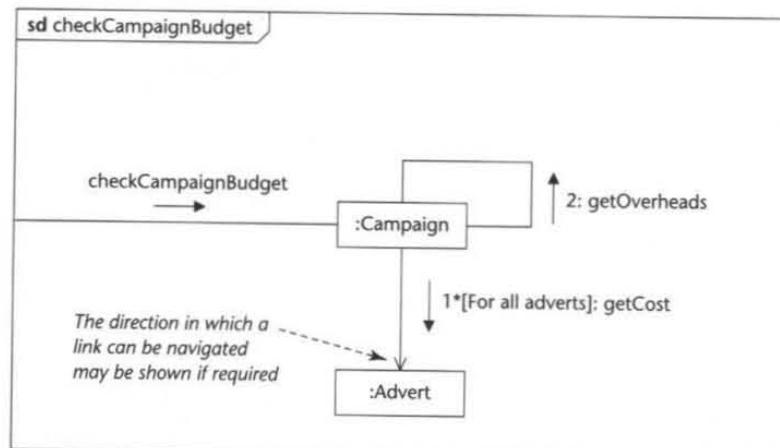


Figure 9.23 Communication diagram for the operation *checkCampaignBudget*.

Communication diagrams are preferred to sequence diagrams by some developers as they offer a view of object interaction that is easy to relate to the underlying collaboration because of the visibility of links between the lifelines. However, they do not provide the same level of syntax as sequence diagrams and are not suitable for complex interaction. Generally, communication diagrams are probably useful during analysis activities while sequence diagrams are better at representing design detail. Communication diagrams are used to describe analysis use case realizations because typically the messages are not fully specified at this stage. However, when there are many messages between two objects in one interaction then a collaboration diagram is more difficult to read than the equivalent sequence diagram. In particular, sequence diagrams also offer more explicit visual cues to the duration of each activation and show detailed design interactions more clearly. We believe that it is seldom useful to draw both kinds of interaction diagram for the same purpose. Some developers use sequence diagrams for the generic model of the interaction and collaboration diagrams for specific scenarios. The choice of which is the more appropriate usually depends on the nature of the interaction and the purpose of the diagram. Often neither offers a clear advantage and it should be a matter of organizational policy as to which is normally prepared.

9.5 Interaction Overview Diagrams

Interaction overview diagrams have been introduced in UML 2.0. They are variants of activity diagrams (explained in Chapter 5) and incorporate interaction diagrams. An interaction overview diagram focuses on the overview of flow of control in an interaction where the nodes in the diagram are interactions or interaction occurrences. As a result the detailed messaging of the interaction is hidden in the diagram. The syntax for activity diagrams is used including decision and merge nodes.

In order to produce an interaction overview diagram the interaction needs to be broken down into its key elements. The interaction Add a new advert to a campaign if within budget (Fig. 9.11) is already broken down to some extent through the use of the two interaction occurrences. However, the alt combined fragment could be broken down further to show the flow of control. Two more interaction occurrences, Create advert and Create request are introduced and this is shown in Fig. 9.24. The detail of these interaction fragments is shown in Figs 9.25 and 9.26 respectively.

The interaction overview diagram is shown in Fig. 9.27. As it is a variant of an activity diagram, it starts with an initial node leading to the interaction fragment List campaigns for client. The next node is an interaction occurrence (its detail is shown in Fig. 9.13). This is followed by another in-line interaction fragment, Add Costed Advert. It is a matter of judgement as to when it is best to have interaction fragments or interaction occurrences as nodes. It depends upon the level of detail that is most appropriate for the diagram. It is useful to take advantage of interaction fragments where they have already been specified, but it does depend upon the purpose of the interaction overview diagram. In this case the interaction fragment could just as easily have been represented as an interaction occurrence and the detail of the interaction fragment could be specified in

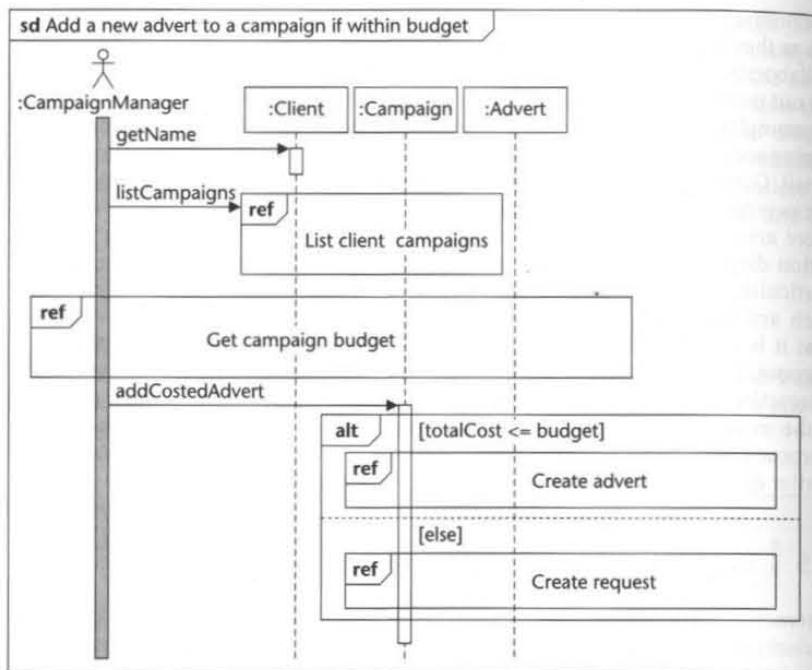


Figure 9.24 Alternative version of the sequence diagram for the interaction Add a new advert to a campaign if within budget.

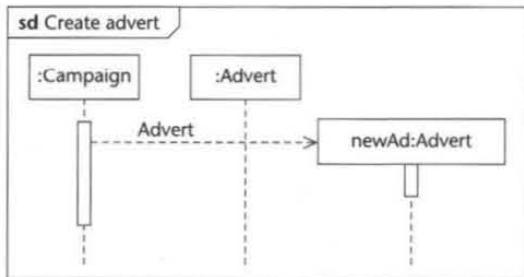


Figure 9.25 Sequence diagram for the interaction fragment Create advert.

a separate sequence diagram. The alt combined fragment in the original sequence diagram is represented in the interaction overview diagram as a decision node with two output paths, each with a guard condition, that lead to the appropriate interaction occurrences. The end of the interaction is indicated by the final node notation.

An interaction overview diagram is useful when describing a complex interaction, particularly when it comprises a series of interaction fragments, some of

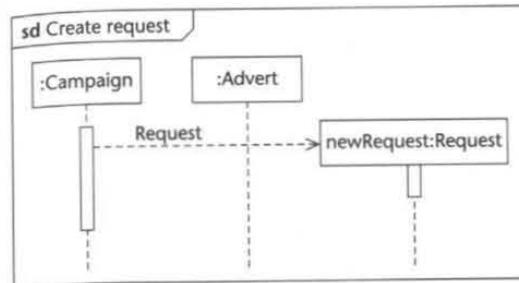


Figure 9.26 Sequence diagram for the interaction fragment Create request.

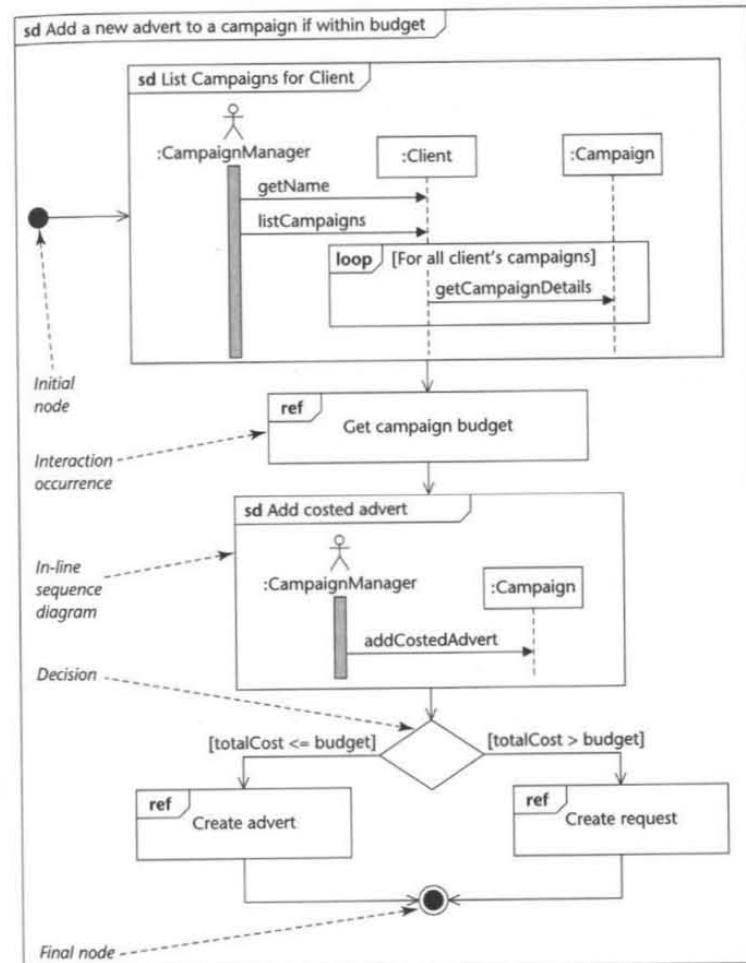


Figure 9.27 Interaction overview diagram.

which may be used in several interactions. In most circumstances it is not going to be helpful to produce both an interaction sequence diagram and an interaction overview diagram for the same interaction. Interaction overview diagrams also provide a useful notation to describe high-level system interactions.

9.6 Timing Diagrams

Timing diagrams are another new feature in UML 2.0, but were previously included in the book by Douglass (1999) on real-time systems development using UML. They are of particular value when specifying how time constraints affect the interaction between lifelines. It is useful to build the state machines for the key lifelines at the same time. State machines are discussed in Chapter 11.

Timing diagrams may be prepared at different levels of detail. They include the state changes for one or more lifelines; typically those with important time dependent behaviour are modelled. Where more than one lifeline is included in a timing diagram the messages between the lifelines may be shown.

Figure 9.28 shows an interaction fragment that describes part of the interaction that occurs when a car enters a car park. Access to the car park is controlled by a barrier which is only raised after a ticket has been requested and issued to the driver. There is a weight sensor just before the barrier that activates the ticket machine when a car is detected. When the car moves to go under the barrier the ticket machine is deactivated. There is another weight sensor after the barrier that detects when the car has passed under the barrier so that it can be lowered.

The timing diagram containing the lifelines `:TicketMachine` and `:Barrier` is shown in Fig. 9.29. The diagram is divided into two instances, one for each lifeline. The `:Barrier` lifeline starts off in the `Lowered` state and moves to the `Raised`

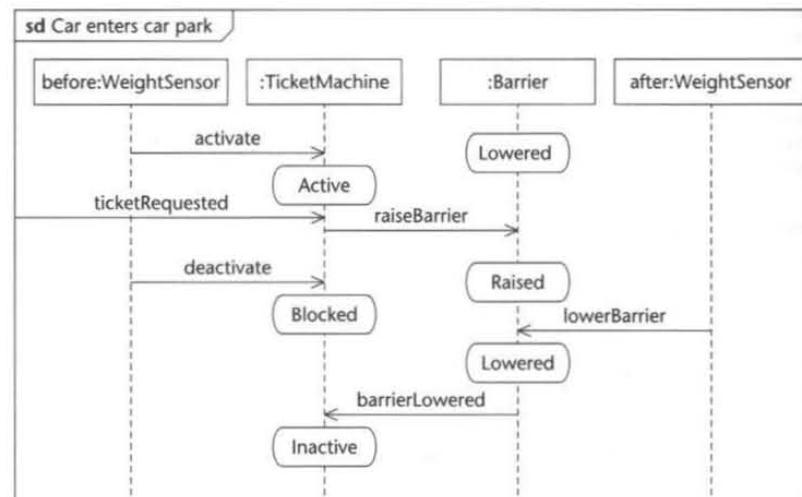


Figure 9.28 Sequence diagram for the interaction fragment Car enters car park.

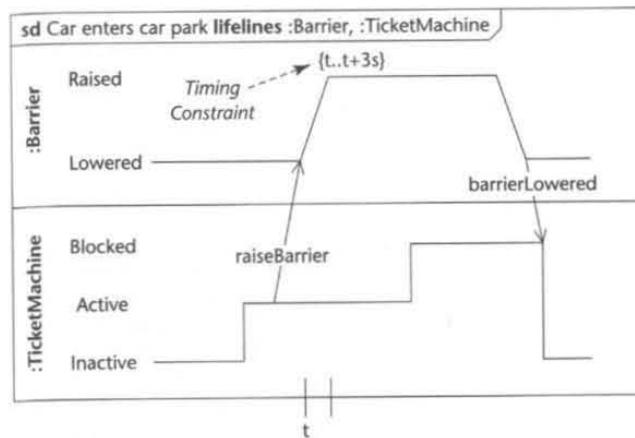


Figure 9.29 Timing diagram for the interaction Car Enters Car Park.

state after the lifeline has received the signal `raiseBarrier`. The time t is set at the time the `raiseBarrier` message is received by the lifeline `:Barrier` and the timing constraint specifies that `:Barrier` should change state within 3 seconds of time t . The sloped line between the two states models the duration of the state change.

9.7 Model Consistency

The preparation of sequence and communication diagrams involves the allocation of operations to classes. These operations should be listed against the correct classes in the class diagram and, if operation signatures have been specified in full, these must be consistent. The sequence and communication diagrams and the class diagrams should be mutually consistent. A good CASE tool enforces this consistency at a syntactic level, usually by prompting the developer with a list of operations currently allocated to a class when he or she adds a message that is sent to an object of that class. If a corresponding operation has not been defined in the destination class, the appropriate operation is added automatically to the class definition.

But to ensure full consistency between a class diagram and a set of related interaction diagrams requires more than the simple syntactic check described above. Every sending object in an interaction diagram must have the ability to send its message to the destination object and this requires it to know the identity or *object reference* of the destination object. There are only two ways that a sending object can know the reference of a destination object. It may already be known by the sending object via a direct link, which really means that an association exists between the respective classes to which the objects belong. Alternatively, a sending object may obtain the reference it needs indirectly from another object (usually of a different class) that has a link with the destination object. The representation and placement of the object references that represent associations is a design issue that we discuss in detail in Chapter 14. At this stage it is sufficient to ensure that there

is some possible pathway via object links (deduced from associations on the class diagram) that connects a sending object to a destination object. Any inconsistency between an interaction diagram and the corresponding class diagram indicates a flaw in one or the other. There may be a necessary association missing from the class diagram, for instance. Note that the existence of an association does not by itself guarantee the existence of any particular link. Where the minimum multiplicity of an association is zero there may not be any object that has a link for that association. If the multiplicity is one (or more), then each object must have at least one link. All message pathways should be analysed carefully.

When an interaction occurrence is used within a more complex interaction (described in either a sequence diagram or an interaction overview diagram) it is important that this use is consistent with the description of the interaction occurrence in the associated sequence diagram. Of course an interaction occurrence may well be used in several other interactions. Thus the use of interaction occurrences requires careful management to ensure consistency.

State machines (described in Chapter 11) document information about messages from the perspective of an object rather than an interaction, and it is also important to check for consistency between the state machine diagram for a class and all interaction diagrams that involve objects of that class.

9.8 Summary

Object interaction is a crucial feature of the object-oriented approach to systems development. When discussing object interaction we use the metaphor of message passing to describe the mode of collaboration between objects. Developing interaction diagrams requires a careful analysis of the use cases and may involve the use of CRC cards (see Chapter 7). UML provides a range of modelling techniques to describe interactions—sequence, communication, interaction overview and timing diagrams—which provide a rich notation. Communication diagrams are more commonly used when analysing use cases to prepare analysis use case realizations. Sequence and interaction overview diagrams may be used effectively to represent detailed design specifications of interactions. Timing diagrams are particularly useful for real-time systems. When working with complex interactions it may be necessary to use several linked diagrams. UML provides various notational alternatives for this. It is common to have more than one possible interaction for a use case and it is a pragmatic judgement as to which is most appropriate. The design considerations that inform such judgements are discussed in detail in Chapter 14. An integral part of the process of developing interaction diagrams is ensuring that they and the class diagrams for an application are mutually consistent.

Review Questions

- 9.1** List two specific features of bad object-oriented modelling that are discouraged by the use of communication diagrams.
- 9.2** What are the benefits of keeping all classes reasonably small and self-contained?
- 9.3** What are the main differences between sequence diagrams and communication diagrams?

9.4 What are the essential parts of a message label (i) in a sequence diagram and (ii) in a communication diagram?

9.5 What is a lifeline?

9.6 What is meant by execution occurrence?

9.7 How are combined fragments used in sequence diagrams?

9.8 How do asynchronous messages differ from synchronous messages (i) in their behaviour and (ii) in their notation?

9.9 In what circumstances are sequence numbers in a communication diagram written in nested style (e.g. 3.2.1)?

9.10 What consistency checks should be applied to interaction diagrams?

9.11 Describe three ways in which complex interactions may be represented using UML.

9.12 When are interaction occurrences used?

9.13 What is the difference between an interaction occurrence and an interaction fragment?

9.14 What is the purpose of interaction overview diagrams?

9.15 What can be represented by the nodes in an interaction overview diagram?

9.16 How are timing diagrams used and when are they most useful?

Case Study Work, Exercises and Projects

Exercises 9.A–9.C are based on the use cases listed in Exercise 7.A and the use case realizations developed in Exercise 7.B.

9.A For each of the use cases prepare a sequence diagram.

9.B For the use case Start line run identify an alternative interaction and prepare a sequence diagram for this interaction.

9.C Critically compare the two interactions that you have identified for the use case Start line run and with suitable justification determine which is the more appropriate.

9.D Using a CASE tool with which you are familiar, enter several use case realizations including at least one communication and one sequence diagram (e.g. the FoodCo models).

9.E Critically evaluate the extent to which the CASE tool supports UML and the consistency check that is necessary between the different diagrams.

Further Reading

Rumbaugh et al. (1991) and Booch (1994) discuss their variants of message modelling in detail. Buschmann et al. (1996) provide interesting examples of system sequence diagrams using a notation from which the UML notation for sequence diagrams has developed. Douglass (1999) discusses the use of this notation for real-time systems.

Specifying Operations

Learning Objectives

In this chapter you will learn

- why operations need to be specified
- the difference between algorithmic and non-algorithmic methods
- how to interpret different ways of specifying operations
- how to specify operations using one method.

10.1 Introduction

Operation specifications play a similar role in the project repository to that of other entries, such as attribute specifications. They support a graphical model by adding precision so that users can confirm the correctness of the model, and designers can use them as a basis for software development. But they are potentially the most complex of all entries in the repository, since they explain the detailed behaviour of the system.

We consider the need for specifying operations (Section 10.2) and introduce the ‘contract’ (Section 10.3) as a kind of black box specification. If the behaviour of an operation is simple, a contract that describes only its external interface may be all that is required and, if its behaviour is not yet understood in any detail, a black box specification may be all that is possible. Often there is also a need to describe an operation’s logic, or internal behaviour.

The two general ways of doing this are respectively called ‘algorithmic’ (or ‘procedural’) and ‘non-algorithmic’ (or ‘declarative’). A non-algorithmic approach is generally preferred in object-oriented development, but in some situations only an algorithmic approach is sufficiently expressive. We review some established techniques, in particular decision tables and pre- and post-condition pairs (Section 10.4.1) and Structured English (Section 10.4.2). None of these are specific either to UML or to an object-oriented approach, but all can be used to specify

operations in a UML model. A full description is beyond the scope of this book, so we introduce them at an overview level.

UML does not require any specific techniques or notations for specifying operations, but activity diagrams (first introduced in Chapter 5) can be used to express the logic of an operation in a graphical form (Section 10.4.2). The UML has also a formal language known as the Object Constraint Language (OCL), which was originally intended for specifying general constraints on a model. In its latest version (2.0), OCL also permits the writing of queries, business rules and other expressions that can apply to a model. We introduce OCL (Section 10.5) and show how it can be used in specifying operations (Section 10.6).

10.2 The Role of Operation Specifications

Each operation specification is a small but necessary step on a path that begins with a user’s idea of a business activity and leads ultimately to a software system made up of collaborating objects with attributes and methods. From an analysis perspective, an operation specification is created at a point when the analyst’s understanding of some aspect of an application domain can be fed back to users, ensuring that the proposals meet users’ needs. From a design perspective, an operation specification is a framework for a more detailed design specification, which later guides a programmer to a method that is an appropriate implementation of the operation in code. An operation specification can also be used to verify that the method does indeed meet its specification, which in turn describes what the users intended, thus checking that the requirements have been implemented.

Novice programmers often do not appreciate the need to design, still less specify, an operation before beginning to write it in program code. This is partly because beginners are given such simple tasks, e.g. to write a program that can calculate and display the area of a rectangle. More importantly, the student is shielded from the activity of requirements analysis. In effect, the teacher has already carried this out, and the student is presented with its results as a starting point: ‘There is a need for a program to calculate the area of a rectangle.’ Why? An answer given to a student will be put in educational terms, such as: ‘This will help you to develop important basic skills in ...’.

Of course, the situation just described is quite artificial, and most students know this perfectly well. But it is only once the complexity or scale of a software system reaches a certain threshold that the production of code too early becomes extremely inefficient, and very possibly disastrous. To code a relatively small subtask in a large system requires some understanding of the ways in which that subtask will interact with other subtasks. If this understanding has not yet been achieved, assumptions must be made, and these may later turn out to be inappropriate, even disastrous, for the system as a whole.

Object-oriented programming is generally more immune to this kind of problem than other programming approaches, but it is still important to describe the logical operation of the planned software as early as possible. Modelling object interaction (see Chapter 9) is part of this description process, as it helps to determine the distribution of behaviour among the various classes. A detailed description of individual operations must also now be provided.

There are differences of opinion on how much specification should be done. Many proponents of XP—‘Extreme Programming’, that is, not the Microsoft operating system—argue that conversations among users and developers can effectively replace much of the documentation that has been traditionally created during systems development, including operation specifications (see Chapter 21 for more on XP). The moderate view is held by Rumbaugh et al. (1991), who suggest that only operations that are ‘computationally interesting’ or ‘non-trivial’ need be specified. ‘Trivial’ operations (e.g. those that create or destroy an instance, and those that get or set the value of an attribute) need not be specified at all. Further, operation specifications should be kept simple in form and should consist only of the operation signature and a description of its ‘transformation’ (i.e. its logic). On the other hand, Allen and Frost (1998), recommend the specification of all operations, although the level of detail may vary according to the anticipated needs of the designer. We recommend the latter approach, mainly because of the problems that can arise later in a project if full documentation is not maintained. It is important to keep to at least a minimal documentation standard, even for operations that are very simple.

Each operation has a number of characteristics, which should be specified at the analysis stage. Users must confirm the logic, or rules, of the behaviour. The designer and the programmer responsible for the class will be the main users of the specification, as they need to know what an operation is intended to do: does it perform a calculation, or transform data, or answer a query? Designers and programmers of other parts of the system also need to know about its effects on other classes. For example, if it provides a service to other classes, they need to know its signature. If it calls operations in other classes or updates the values of their attributes, this may establish dependencies that guide how these classes should be packaged during design or implementation.

Defining operations should be neither begun too early nor left too late. For Allen and Frost (1998), this task should be left until the class diagram has stabilized. In a project where the development activity has been broken down at an early stage to correspond to separate subsystems, this may refer only to that part of the class diagram that relates to a particular subsystem. But for any given part of the model, it is important to create all operation specifications before moving into the object design activity.

10.3 Contracts

The term ‘contract’ is a deliberate echo of legal or commercial contracts between people or organizations. Signing (or becoming a party to) a contract involves making a commitment to deliver a defined service to an agreed quality standard. For example, a small ground-care company has a contract to mow the grass on the lawn in front of the Agate headquarters building. The contract stipulates how often the grass must be cut (every two weeks from April to October), the maximum height of the grass immediately after it is cut (no more than 3 centimetres) and how much Agate will pay for the service (£80 per cut). The contract does *not* spell out how the work will be done—for example, what type of mower should be used (electric or petrol, cylinder or rotary), how many staff or mowers should be involved, or in which direction the lawn should be cut.

In the language of system theory, a contract is an interface between two systems. In this example, Agate is a business system and the ground-care company is a system for mowing Agate’s grass. The contract defines inputs and outputs, and treats the grass-mowing system to some extent as a black box, with its irrelevant details hidden. Which details are deemed irrelevant is always a matter of choice, and any contract can specify that some details of the implementation should be visible to other systems. For example, Agate’s directors might not wish to permit the ground-care contractor to use toxic pesticides or weedkillers. This can be included as a constraint in the contract.

Meyer (1988, 1991) was one of the first to draw an analogy between commercial contracts and service relationships between objects. The use of the term is now widespread in object-oriented development since it stresses the encapsulation of classes and subsystems in a model. Cook and Daniels (1994) used the concept extensively in the Syntropy methodology; it is applied in the SELECT Perspective methodology (Allen and Frost, 1998; Apperley et al., 2003); and design by contract is an important part of the thinking behind OCL (Warmer and Kleppe, 2003).

One of Meyer’s principal arguments for using the analogy of a contract is that design by contract helps to achieve a software design that is correct in terms of its requirements. During requirements analysis, we do not yet need the full technical rigour that is required of a design specification, but there is still a clear advantage in adopting an approach that can later be extended seamlessly through design into implementation. Specification by contract means that operations are defined primarily in terms of the services they deliver, and the ‘payment’ they receive (usually just the operation signature).

Contracts can also be applied at a much higher level of abstraction than individual operations. Larman (2005) describes the use of contracts to define services provided by a system as a whole. Whether written for a single operation, for the behaviour of the system as a whole or for some intermediate packaged component, the structure of a contract is very similar. A commercial contract usually identifies the parties, the scope (i.e. the context in which it applies), the agreed service, and any performance standards that apply. In just the same way, in object-oriented modelling we identify the nature of the service provided by the server object, and what must be provided by the client object in order to obtain the service. These various aspects can be summarized as follows:

- the intent or purpose of the operation
- the operation signature including the return type (probably established during interaction modelling)
- an appropriate description of the logic (the following sections present some alternative ways of describing the logic of an operation)
- other operations called, whether in the same object or in other objects
- events transmitted to other objects
- attributes set during the operation’s execution
- the response to exceptions (e.g. what should happen if a parameter is invalid)
- any non-functional requirements that apply.

This list of features is adapted from Larman (2005) and Allen and Frost (1998). Most of it is self-explanatory, but the critical part of an operation specification is the logic description, and it is to this that we turn in the next section.

There are differences of opinion on how much specification should be done. Many proponents of XP—‘Extreme Programming’, that is, not the Microsoft operating system—argue that conversations among users and developers can effectively replace much of the documentation that has been traditionally created during systems development, including operation specifications (see Chapter 21 for more on XP). The moderate view is held by Rumbaugh et al. (1991), who suggest that only operations that are ‘computationally interesting’ or ‘non-trivial’ need be specified. ‘Trivial’ operations (e.g. those that create or destroy an instance, and those that get or set the value of an attribute) need not be specified at all. Further, operation specifications should be kept simple in form and should consist only of the operation signature and a description of its ‘transformation’ (i.e. its logic). On the other hand, Allen and Frost (1998), recommend the specification of all operations, although the level of detail may vary according to the anticipated needs of the designer. We recommend the latter approach, mainly because of the problems that can arise later in a project if full documentation is not maintained. It is important to keep to at least a minimal documentation standard, even for operations that are very simple.

Each operation has a number of characteristics, which should be specified at the analysis stage. Users must confirm the logic, or rules, of the behaviour. The designer and the programmer responsible for the class will be the main users of the specification, as they need to know what an operation is intended to do: does it perform a calculation, or transform data, or answer a query? Designers and programmers of other parts of the system also need to know about its effects on other classes. For example, if it provides a service to other classes, they need to know its signature. If it calls operations in other classes or updates the values of their attributes, this may establish dependencies that guide how these classes should be packaged during design or implementation.

Defining operations should be neither begun too early nor left too late. For Allen and Frost (1998), this task should be left until the class diagram has stabilized. In a project where the development activity has been broken down at an early stage to correspond to separate subsystems, this may refer only to that part of the class diagram that relates to a particular subsystem. But for any given part of the model, it is important to create all operation specifications before moving into the object design activity.

10.3 Contracts

The term ‘contract’ is a deliberate echo of legal or commercial contracts between people or organizations. Signing (or becoming a party to) a contract involves making a commitment to deliver a defined service to an agreed quality standard. For example, a small ground-care company has a contract to mow the grass on the lawn in front of the Agate headquarters building. The contract stipulates how often the grass must be cut (every two weeks from April to October), the maximum height of the grass immediately after it is cut (no more than 3 centimetres) and how much Agate will pay for the service (£80 per cut). The contract does *not* spell out how the work will be done—for example, what type of mower should be used (electric or petrol, cylinder or rotary), how many staff or mowers should be involved, or in which direction the lawn should be cut.

In the language of system theory, a contract is an interface between two systems. In this example, Agate is a business system and the ground-care company is a system for mowing Agate’s grass. The contract defines inputs and outputs, and treats the grass-mowing system to some extent as a black box, with its irrelevant details hidden. Which details are deemed irrelevant is always a matter of choice, and any contract can specify that some details of the implementation should be visible to other systems. For example, Agate’s directors might not wish to permit the ground-care contractor to use toxic pesticides or weedkillers. This can be included as a constraint in the contract.

Meyer (1988, 1991) was one of the first to draw an analogy between commercial contracts and service relationships between objects. The use of the term is now widespread in object-oriented development since it stresses the encapsulation of classes and subsystems in a model. Cook and Daniels (1994) used the concept extensively in the Syntropy methodology; it is applied in the SELECT Perspective methodology (Allen and Frost, 1998; Apperley et al., 2003); and design by contract is an important part of the thinking behind OCL (Warmer and Kleppe, 2003).

One of Meyer’s principal arguments for using the analogy of a contract is that design by contract helps to achieve a software design that is correct in terms of its requirements. During requirements analysis, we do not yet need the full technical rigour that is required of a design specification, but there is still a clear advantage in adopting an approach that can later be extended seamlessly through design into implementation. Specification by contract means that operations are defined primarily in terms of the services they deliver, and the ‘payment’ they receive (usually just the operation signature).

Contracts can also be applied at a much higher level of abstraction than individual operations. Larman (2005) describes the use of contracts to define services provided by a system as a whole. Whether written for a single operation, for the behaviour of the system as a whole or for some intermediate packaged component, the structure of a contract is very similar. A commercial contract usually identifies the parties, the scope (i.e. the context in which it applies), the agreed service, and any performance standards that apply. In just the same way, in object-oriented modelling we identify the nature of the service provided by the server object, and what must be provided by the client object in order to obtain the service. These various aspects can be summarized as follows:

- the intent or purpose of the operation
- the operation signature including the return type (probably established during interaction modelling)
- an appropriate description of the logic (the following sections present some alternative ways of describing the logic of an operation)
- other operations called, whether in the same object or in other objects
- events transmitted to other objects
- attributes set during the operation’s execution
- the response to exceptions (e.g. what should happen if a parameter is invalid)
- any non-functional requirements that apply.

This list of features is adapted from Larman (2005) and Allen and Frost (1998). Most of it is self-explanatory, but the critical part of an operation specification is the logic description, and it is to this that we turn in the next section.

10.4 Describing Operation Logic

Rumbaugh et al. (1991) suggest an informal classification of operations that is a useful starting point in considering the various ways of describing their logic. First, there are operations that have side-effects. Possible side-effects include the creation or destruction of object instances, setting attribute values, forming or breaking links with other objects, carrying out calculations, sending messages or events to other objects, or any combination of these. A complex operation may do several of these things, and, where the task is at all complex, an operation may also require the collaboration of several other objects. It is partly for this reason that we identify the pattern of object collaboration before specifying operations in detail. Second, there are operations that do not have side-effects. These are pure queries; they request data but do not change anything within the system.

Like classes, operations may also have the property of being either {abstract} or {concrete} (although this decision is often the result of design considerations, and is therefore not always made when an operation is first specified). Abstract operations have a form that consists of at least a signature, sometimes a full specification, but they will not be given an implementation (i.e. they will not have a method). Typically, abstract operations are located in the abstract superclasses of an inheritance hierarchy. They are always overridden by concrete methods in concrete subclasses.

A specification may be restricted to defining only external and visible effects of an operation, and we may choose either an algorithmic or a non-algorithmic technique for this. A specification may also define internal details, but this is effectively a design activity.

10.4.1 Non-algorithmic approaches

A non-algorithmic approach concentrates on describing the logic of an operation as a black box. In an object-oriented system this is generally preferred for two reasons. First, classes are usually well-encapsulated, and thus only the designers and programmers responsible for a particular class need concern themselves with internal implementation details. Collaboration between different parts of the system is based on public interfaces between classes and subsystems implemented as operation signatures (or message protocols). As long as the signatures are not changed, a change in the implementation of a class, including the way its operations work, has no effect on other parts of the system¹. Second, the relatively even distribution of effort among the classes of an object-oriented system generally results in operations that are small and single-minded. Since the processing carried out by any one operation is simple, it does not require a complex specification.

Even in non object-oriented approaches, a declarative approach has long been recognized as particularly useful where, for example, a structured decision is made, and the conditions that determine the outcome are readily identified, but the actual sequence of steps in reaching the decision is unimportant. For situations

like this, structured methods make use of non-algorithmic techniques such as decision tables and pre- and post-condition pairs (described in the following sections).

Decision tables

A decision table is a matrix that shows the *conditions* under which a decision is made, the *actions* that may result and how the two are related. They cater best for situations where there are multiple outcomes, or actions, each depending on a particular combination of input conditions. One common form shows conditions in the form of questions that can be answered with a simple yes or no. Actions are listed, and check-marks are used to show how they correspond to the conditions. The following is an example of a possible application in the Agate case study. Figure 10.1 shows a corresponding decision table.

When a campaign budget is overspent, this normally requires prior approval from the client, otherwise Agate is unlikely to be able to recover the excess costs. A set of rules has been established to guide Campaign Managers when they identify a possible problem. If the budget is expected to be exceeded by up to 2%, a letter is sent notifying the client of this. If the budget is expected to be exceeded by more than 2%, a letter is sent and the staff contact also telephones the client to invite a representative to a budget review meeting. If the campaign is not thought likely to exceed its budget, no action is taken.

The vertical columns with Y, N and X entries are known as *rules*. Each rule is read vertically downwards, and the arrangement of Ys and Ns indicates which conditions are true for that rule. An X indicates that an action should occur when the corresponding condition is true (i.e. has a Y answer). We can paraphrase the table into text as follows.

Rule 1. If the budget is not overspent (clearly in this case the scale of overspend is irrelevant, indicated by a dash against this condition), no action is required.

Rule 2. If the budget is overspent and the overspend is not likely to exceed 2%, a letter should be sent.

Rule 3. If the budget is overspent and the overspend is likely to exceed 2%, a letter should be sent and a meeting set up.

A single rule may have multiple outcomes that overlap with the outcomes of other rules. Decision tables are very useful for situations that require a non-algorithmic

Conditions and actions	Rule 1	Rule 2	Rule 3
Conditions			
Is budget likely to be overspent?	N	Y	Y
Is overspend likely to exceed 2%?	-	N	Y
Actions			
No action	X		
Send letter		X	X
Set up meeting			X

Figure 10.1 A decision table with two conditions and three actions, yielding three distinct rules.

¹ In practice, the situation is more complex than this might suggest. Some object-oriented languages, e.g. C++, allow a lot of flexibility as to how much of the internal implementation of a class is visible to other classes.

specification of logic, reflecting a range of alternative behaviours. But this is relatively unusual in an object-oriented system, where thorough analysis of object collaboration tends to minimize the complexity of single operations.

Pre- and post-conditions

As its name suggests, this technique concentrates on providing answers to the following questions.

- What conditions must be satisfied before an operation can take place?
- What conditions should apply (i.e. what states may the system be in) after an operation is completed?

Let us consider an example from Agate. The operation `Advert.getCost()`² was first discussed in Section 8.2. Let us suppose that it has the following signature.

```
Advert.getCost():Money
```

This operation has no pre-condition. (We may note that the object sending the message must know the identity of the object that contains the operation, but this is not in itself a pre-condition for the operation to execute correctly when invoked.) The post-conditions should express the valid results of the operation upon completion. In this case, a money value is returned (for simplicity, we ignore the question of valid values for an advert cost, but we should note that in reality this attribute may be able to take only a limited range of values, depending on business constraints).

Pre-condition: none

Post-condition: a valid money value is returned

More complex examples can easily be constructed from the use case descriptions, or by consulting users if existing descriptions are not sufficiently detailed. Consider the use case *Assign staff to work on a campaign*. This involves calling the operation `Campaign.assignStaff()` for each member of staff assigned. Let us assume that the signature of this operation is as follows.

```
Campaign.assignStaff(creativeStaff)
```

This example has one pre-condition: a calling message must supply a valid `creativeStaff` object. There is one post-condition: a link must be created between the two objects.

Pre-condition: `creativeStaffObject` is valid

Post-condition: a link is created between `campaignObject` and `creativeStaffObject`

Let us look at one more example from Agate, with more complex conditions. This is taken from the use case *Change the grade for a member of staff* (we assume that the use case is being invoked for a member of creative staff). This use case involves several operations including:

```
CreativeStaff.changeGrade()
StaffGrade.setFinishDate()
```

² We have suffixed brackets to the names of operations to distinguish them from attributes. This does not mean that they are operations with no parameters, only that the parameters are not shown.

`StaffGrade()` (*the constructor operation that creates a new instance of this class*)

We examine only one of these in detail, `CreativeStaff.changeGrade()`, but our specification must still recognize calls made to other operations during execution. Let us assume that the operation signature is as follows.

```
CreativeStaff.changeGrade(gradeObject, gradeChangeDate)
```

The pre-conditions are straightforward, consisting only of a valid `gradeObject` and `gradeChangeDate`. The post-conditions are more involved, as once the operation is completed we should expect several effects to have taken place. A new instance of `StaffGrade` is created, and this is linked to the appropriate `creativeStaffObject` and `gradeObject` (by a `staffOnGrade` link). The new `staffGradeObject` is also linked to the previous `staffGradeObject` (by a `previousGrade` link). Attribute values in the new `staffGradeObject` are set by its constructor operation (including `gradeStartDate`, which is set equal to the supplied parameter `gradeChangeDate`). The attribute `StaffGrade.gradeFinishDate` in the previous instance is also set, through a message to invoke the operation `StaffGrade.setFinishDate`. A full logic description is thus as follows.

pre-conditions: `creativeStaffObject` is valid
`gradeObject` is valid
`gradeChangeDate` is a valid date

post-conditions: a new `staffGradeObject` exists
the new `staffGradeObject` is linked to the `creativeStaffObject`
the new `staffGradeObject` is linked to the previous one
the value of the previous `staffGradeObject.gradeFinishDate` is set equal to `gradeChangeDate - 1 day`

For many operations in an object-oriented model, such a specification would be sufficiently detailed.

In general, any operation specification must pass the following two tests.

- A user should be able to check that it correctly expresses the business logic.
- A class designer should be able to produce a detailed design of the operation for a programmer to code.

However, while a declarative approach to operation specification usually meets all the needs of object-oriented development, there is still sometimes a case for using an algorithm. One example would be a requirement that involves carrying out a calculation where the sequence of steps is significant, and neither a designer nor a programmer could reasonably be expected to come up with a formula that produces the correct result.

10.4.2 Algorithmic approaches

An *algorithm* describes the internal logic of a process or decision by breaking it down into small steps (the word derives from al-Kwarazmi, an Arab mathematician of the ninth century). The level of detail to which this is done varies

greatly, depending on the information available at the time and on the reason for defining it. An algorithm also specifies the sequence in which the steps are performed. In the field of computing and information systems, algorithms are used either as a *description* of the way in which a programmable task is currently carried out (this is their purpose in operation specification), or as a *prescription* for a program to automate the task. This dual meaning again reflects the differing perspectives of analysis (understanding a problem and determining what must be done to achieve a solution) and design (the creative act of imagining a system to implement a solution). An algorithmic technique is almost always used during method design, because a designer is concerned with the efficient implementation of requirements, and must therefore select the best algorithm available for the purpose. But algorithms can also be used with an analysis intention. A major difference here is that there is no need for the analyst to worry about efficiency, since the algorithm need only illustrate accurately the results of the operation.

Control structures in algorithms

Algorithms are generally organized procedurally, which is to say that they use the fundamental programming control structures of sequence, selection and iteration. We can illustrate this in the Agate case study by considering the operation that calculates the total cost of a campaign. This operation is invoked during the use case *Check campaign budget*. For ease of reference, the use case description is repeated below.

The campaign budget may be checked to ensure that it has not been exceeded. The current campaign cost is determined by the total cost of all the adverts and the campaign overhead costs.

Let us suppose that there is a precise (though simple) formula for this calculation, based on summing the individual total costs of each advert and adding the campaign overhead costs. For further simplicity, let us assume that the overhead cost part of the calculation simply involves multiplying the total of all other costs by an overhead rate (this approximates to normal accounting practice). To convey an understanding of the calculation, we can begin by representing it as a mathematical formula.

```
total_campaign_cost = (sum of all advert_costs) * overhead_rate
```

This does not explicitly identify all the steps, but a sequence can be deduced. In fact, several possible sequences can be deduced, but any sequence that always produces a correct result will do. One possible sequence, at a very coarse level of detail, would include the following steps.

1. Add up all the individual advert costs.
2. Multiply the total by the overhead rate.
3. The resulting sum is the total campaign cost.

For such a relatively simple calculation as this one, the formula itself would almost certainly serve better as a specification, but some are a lot more complex. When it is necessary to specify the sequence of calculation in more detail, we can use Structured English for this.

Structured English

This is a 'dialect' of written English that is about halfway between everyday non-technical language and a formal programming language. When it is necessary to specify an operation procedurally, this is the most useful and versatile technique. Its advantages include the possibility, with a little care, of retaining much of the readability and understandability of everyday English. It also allows the construction of a formal logical structure that is easy to translate into program code. Structured English is very easy to write iteratively, at successively greater levels of detail, and it is easily dismantled into components that can be reassembled in different structures without a lot of reworking. The logical structure is made explicit through the use of keywords and indentation, while the vocabulary is kept as close as possible to everyday usage in the business context. Above all, expressions and keywords that are specific to a particular programming language are avoided. The result ideally is something that a non-technical user is able to understand, alter or approve, as necessary, while it should also be useful to the designer. This means it must be capable of further development into a detailed program design without undue difficulty.

The main principles of Structured English are as follows. A specification is made up of a number of simple sentences, each consisting of a simple imperative statement or equation. Statements may only be combined in restricted ways that correspond to the sequence, selection and iteration control structures of structured programming. The very simplest specifications contain only sequences and differ little from everyday English except in that they use a more restricted vocabulary and style (many organizations have their own Structured English house style). Here are some statements that illustrate a typical style of Structured English:

```
get client contact name
sale cost = item cost * (1 - discount rate)
calculate total bonus
description = new description
```

Selection structures show alternative courses of action, the choice between them depending on conditions that prevail at the time the selection is made. For example, an *if-then-else* construct, which has only two possible outcomes, is shown in the following fragment.

```
if client contact is 'Sushila'
    set discount rate to 5%
else
    set discount rate to 2%
end if
```

If the two alternatives are not really different actions, but are rather a choice between doing something and not doing it, the 'else' branch can be omitted. The following fragment shows this simpler form.

```
if client contact is 'Sushila'
    set discount rate to 5%
end if
```

Note that in each case the end of the structure is marked by *end if*. This important marker cannot be omitted. It allows the entire structure to be treated logically as an element, as if it were a single statement in a sequence.

Multiple outcomes are handled either by a *case* construct or by a *nested if*. This fragment illustrates the case structure.

```
begin case
    case client contact is 'Sushila'
        set discount rate to 5%
    case client contact is 'Wu'
        set discount rate to 10%
    case client contact is 'Luis'
        set discount rate to 15%
    otherwise
        set discount rate to 2%
end case
```

The 'otherwise' branch of a case construct can be omitted if it is not required, although it is generally good practice to include a catch-all to ensure completeness. The next fragment shows the same selection specified using a *nested-if* construct.

```
if client contact is 'Sushila'
    set discount rate to 5%
else
    if client contact is 'Wu'
        set discount rate to 10%
    else
        if client contact is 'Luis'
            set discount rate to 15%
        else
            set discount rate to 2%
        end if
    end if
end if
```

This also illustrates how indentation can help the readability of a specification. For each corresponding set of control statements (lines beginning with 'if', 'else' and 'end if'), the indentation from the left margin is the same. This helps to show which sequence statements ('set discount rate to 10%', etc.) belong to each structure.

The third type of control structure is iteration. This is used when a statement, or group of statements, needs to be repeated. Typically this is a way of applying a single operation to a set of objects. Logically, once something has begun to occur repeatedly, there must be a condition for stopping the repetition (unless the repetition is to continue indefinitely). There are two main forms of control of iteration. These differ in whether the condition for ending the repetition is tested before or after the first loop. The next two examples show typical applications of each kind of structure. In the first, the test is applied before the loop is entered, so that if the list is empty no bonus is calculated.

```
do while there are more staff in the list
    calculate staff bonus
    store bonus amount
end do
```

In the second iteration example below, the test is applied after the loop is exited. This ensures that the action will be processed (or attempted) at least once. Note that the line at the end beginning until acts as an end-of-structure marker, just like the end do above.

```
repeat
    allocate member of staff to campaign
    increment count of allocated staff
    until count of allocated staff = 10
```

Complex structures in Structured English. Different types of structure can be nested inside each other, as in the next fragment.

```
do while there are more staff in the list
    calculate bonus for this staff member
begin case
    case bonus > £250
        add name to 'star of the month' list
    case bonus < £25
        create warning letter
    end case
    store bonus amount
end do
format bonus list
```

The operation mentioned near the beginning of this section (Check campaign budget) also illustrates the use of all three control structures, although in this case there is no nesting.

```
do while there are more adverts for campaign
    get next advert
    get cost for this advert
    add to cumulative cost for campaign
end do
set total advert cost = final cumulative cost
set total campaign cost =
    total advert cost + (total advert cost × overhead rate)
get campaign budget
if total campaign cost > campaign budget
    generate warning
endif
```

A Structured English specification can be made as complex as it needs to be, and it can also be written in an iterative, top-down manner. For example, an initial version of an algorithm is defined at a high level of granularity. Then, provided the overall structure is sound, more detail is easily added progressively. In refining the level of detail, structures can be nested within each other to any degree of complexity, although in practice it is unlikely that even the most complex operation would need more than two to three levels of nesting at most. It is in any case sensible to limit the complexity. One often-quoted guideline is that a Structured English specification should not be longer than one page of typed A4, or one screen if it is likely to be read within a CASE tool environment—although in practice the acceptable length of a section of text depends on the context.

The style in all the examples given above is based on that of Yourdon (1989), but this should not be taken as necessarily prescriptive. What passes for acceptable style varies widely from one organization to another, and in practice an analyst should follow the house style, whatever that happens to be.

Pseudo-code

Pseudo-code differs from Structured English in that it is closer to the vocabulary and syntax of a specific programming language. There are thus many different dialects of pseudo-code, each corresponding to a particular programming language. They differ from each other in vocabulary, in syntax and in style. Structured English avoids language specificity primarily to avoid reaching conclusions about design questions too early. Sometimes there seems no good reason to hold back, for example because the final implementation language has been decided early in the project. This can be misleading, as it may be desirable at a later stage to redevelop the system in a different programming language. If the operations have been specified in a language-specific pseudo-code, it would then be necessary to rewrite them.

However language-specific it may be, pseudo-code remains only a skeleton of a program, intended only to illustrate its logical structure without including full design and implementation detail. In other words, it is not so much a fully developed program as an outline that can later be developed into program code. The following pseudo-code for Check campaign budget can be compared with the Structured English version above.

```
{
  { while more adverts:
    next advert;
    get advertcost;
    cumulativecost = cumulativecost + advertcost;
    endwhile;
  }
  { campaigncost = cumulativecost + (cumulativecost * ohrate)
  get campaignbudget;
  case campaigncost > campaignbudget
    return warningflag;
  endcase
  }
}
```

Note that, while this pseudo-code resembles C in its syntax, it is not actually written in C. Pseudo-code requires further work to turn it into program code.

Activity diagrams

Activity diagrams can be used to specify the logic of procedurally complex operations. The notation of activity diagrams was introduced in Chapter 5; in this section we illustrate their role in operation specification. When used for this purpose, actions in the diagram usually represent steps in the logic of the operation. This can be done at any level of abstraction, so that, if appropriate, an initial high-level view of the operation can later be decomposed to a lower level of detail.

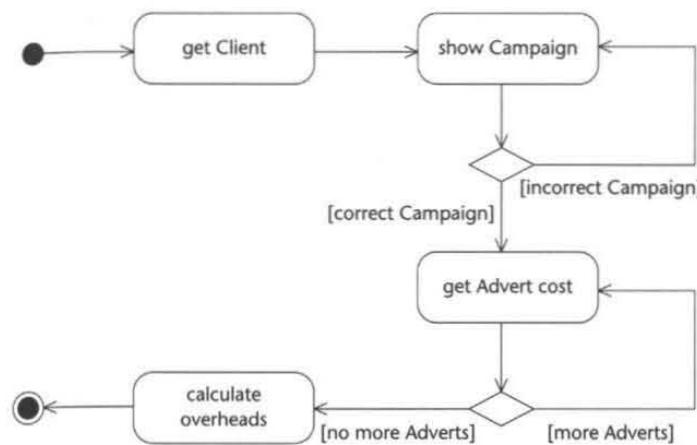


Figure 10.2 Activity diagram for the use case Check campaign budget.

Activity diagrams are inherently very flexible in their use, and therefore a little care should be exercised when they are employed in operation specification. An activity diagram can be drawn to represent a single operation on an object, but this is less common than using an activity diagram to represent a collaboration between several objects (for example, one that realizes a use case). Figure 10.2 illustrates this for the use case Check campaign budget (compare with the corresponding sequence diagram later in Fig. 10.4).

An activity diagram can also be drawn for a more abstract collaboration between larger components of a system, or between entire systems. A single diagram does not necessarily translate into a single operation; whether or not it does is essentially a design decision.

10.5 Object Constraint Language

In drawing any class diagram, much of the time and effort is spent in working out what constraints apply. For example, the multiplicity of an association represents a constraint on how many objects of one class can be linked to any object of the other class. This particular example can be adequately expressed in the graphical language of the class diagram, but this is not equally so for all constraints. Among those for which it is not true are many of the constraints within operation specifications. For example, many pre- and post-conditions in a contract are constraints on the behaviour of objects that are party to the contract. Sometimes the definition of such constraints can be done in a relatively informal manner (as in the examples in Section 10.4), but where greater precision is required, OCL provides a formal language.

OCL expressions are constructed from a collection of pre-defined elements and types, and the language has a precise grammar that enables the construction of unambiguous statements about the properties of model components and their relationships to each other. The latest version of OCL (2.0) (OMG, 2003) has been

extended so that it can also be used to define queries, to reference values and to state business rules (Warmer and Kleppe, 2003). It is beyond the scope of this book to give a detailed treatment, and this section is intended only to illustrate some of the ways that OCL can be used to support a model, in particular for operation specification.

Most OCL statements consist of the following structural elements.

- A *context* that defines a domain within which the expression is valid. This is often an instance of a specific type, for example an object in a class diagram. A link (i.e. an instance of an association) may also be the context for an OCL expression.
- A *property* of that instance which is the context for the expression. Properties may include attributes, association-ends and query operations.
- An OCL *operation* that is applied to the property. Operations include (but are not restricted to) the arithmetical operators *, +, - and /, set operators such as size, isEmpty and select and type operators such as *oclIsTypeOf*.

OCL statements can also include *keywords*, such as the logical operators **and**, **or**, **implies**, **if**, **then**, **else** and **not** and the set operator **in**, printed in bold to distinguish them from other OCL terms and operations. Together with the non-keyword operations mentioned above, these can be used to define quite complex pre- and post-conditions for an operation. OCL expressions can specify both initial and derived values of attributes, and can be used to check various operation results, such as whether an object has been created or a message sent during the execution of the operation.

OCL expression	Interpretation
context Person self.gender	In the context of a specific person, the value of the property 'gender' of that person, i.e. a person's gender.
context Person inv: self.savings >= 500	It is an invariant of the model that the property 'savings' of the person under consideration must be greater than or equal to 500.
context Person self.husband->notEmpty() implies self.husband.gender = Gender::male	If the set 'husband' associated with a person is not empty, then the value of the property 'gender' of the husband must be male. (This example assumes that <i>Gender</i> ::male is an enumerated value in a <i>Gender</i> «enumeration» type)
context Company inv: self.CEO->size() <= 1	The size of the set of the property 'CEO' of a company must be less than or equal to 1. That is, a company cannot have more than one Chief Executive Officer.
context Company self.employee-> select(age < 60)	The set of employees of a company whose age is less than 60.

Figure 10.3 Examples of some expressions in OCL.

Figure 10.3 gives some examples of expressions in OCL, some of which are adapted from the OCL 2.0 Specification (OMG, 2003). All have an object of some class as their context.

Since OCL can specify constraints that cannot be expressed directly in diagrammatic notation, it is useful as a precise language for pre- and post-conditions of operations. An OCL expression can also specify the result of a query operation. This differs from a post-condition, which specifies a side-effect of an operation. However, queries have no side-effects—that is, they do not change anything in the model or the system. The general syntax for operation specification is as follows:

```
context Type::operation(parameter1:type,parameter2:type):
    return type
pre: parameter1 operation
     parameter2 operation
body: -- an OCL expression that defines the query output
post: result = /* some OCL expression that defines the
          effect of the operation, for example in
          terms of attribute values, objects created
          or destroyed, and so on */
```

Note that the contextual type is the type (for our purposes, normally a class) that owns the operation as a feature. The **pre:** expressions are functions of operation parameters, while the **body:** and **post:** expressions are functions of self, of operation parameters, or of both. OCL expressions must be written with an explicit **context** declaration.

Note also the different styles of comment in the preceding example. A single-line comment is introduced with a double hyphen, thus: --, while a comment that spans more than one line is opened and closed thus: /* comment text */.

The following example illustrates the use of an **inv:** label to denote an invariant. (Invariants were first introduced in Box 5.1 in Section 5.2.2, in the discussion of the UML metamodel.)

```
context Person
inv: self.age >= 0
```

The invariant here is merely that a person's age must always be greater than or equal to zero—arguably, this should not need specification, but poorly specified computer systems often get the really obvious things wrong! For a complete list of keywords, see Section 8 of the OCL Specification (OMG, 2003).

This example shows an OCL expression that defines the initial value of an attribute in the Agate case study:

```
context Advert::actualAdvertCost
init: 0
```

Another useful feature of OCL is that you can define two values for a single property using the postfix @pre. As you might expect, this refers to the previous value of a property, and is only permitted in post-condition clauses. Thus it is possible to constrain the relationship between the values of an attribute before and after an operation has taken place. For example, the decision specified in Fig. 10.1 defines different actions depending on changes in the estimated cost of a campaign in comparison with its budget. If the new estimated cost is greater than the old

estimated cost, but exceeds the budget by no more than 2%, the value of this attribute is set to true, flagging a need to generate a warning letter to the client³. We could model this in a very simple way by adding an attribute `Campaign.clientLetterRequired`. We could then write part of the logic in OCL as follows:

```
context Campaign
post: if self.estimatedCost > estimatedCost@pre and
      self.estimatedCost > budget and
      self.estimatedCost <= budget * 1.02 then
        self.clientLetterRequired = 'true'
      else self.clientLetterRequired = 'false'
      endif
```

This expression will help to define tests that check if the system displays the correct behaviour when a campaign budget has changed. (However, note that this example is intended only to illustrate the notation. In practice, it is unlikely that this is really how we would model this requirement.)

Operation specifications frequently include invariants. An invariant that is associated with an operation specification describes a condition that always remains true for an object, and which must therefore not be altered by an operation side-effect. Formal definition of invariants is valuable because they provide rigorous tests for execution of the software.

For example, the value of `Campaign.estimatedCost` should always equal the sum of all associated `Advert.estimatedCost` values multiplied by the current overhead rate. In OCL, this might be written as follows⁴:

```
context Campaign
inv: self.estimatedCost = self.adverts.estimatedCost->
    sum() + (self.adverts.estimatedCost->sum() * ohRate)
```

In this example, the context is the `Campaign` class. To use an invariant within an operation specification, it can be written simply as an additional `inv:` clause.

```
context: Class::operation(parameter: type):return type
pre: parameter1...
      parameter2...
post: result1...
      result2...
inv: invariant1...
      invariant2...
```

For an example from the Agate case study, we revisit the operation `creativeStaff.changeGrade()`, for which we specified the logic in Section 10.4.1. To help make sense of this specification, it is also worth referring back to the analysis class diagram in Fig. A3.14. In particular, note the recursive association from `StaffGrade` to itself. However, remember also that, as we saw in Chapter 8,

³ If the budget is exceeded but the estimated cost has not increased, then we assume that a letter has already been sent and so the flag does not need to be set.

⁴ Note that this makes some assumptions about the way that the classes are designed and is really intended only to illustrate the style of this kind of statement when written in OCL.

`CreativeStaff` is a subclass of `Staff Member` and therefore inherits the same associations and roles (Chapter A4 includes a revised analysis class diagram that shows this specialization). Here is the main part of the operation specification rewritten in OCL:

```
context: CreativeStaff::changeGrade(grade:Grade,
                                    gradeChangeDate:Date)
pre: grade oclIsTypeOf(Grade)
      gradeChangeDate >= today
post: self.staffGrade->exists() and
      self.staffGrade[previous]->notEmpty() and
      self.staffGrade.gradeStartDate = gradeChangeDate and
      self.staffGrade.previous.gradeFinishDate =
      gradeChangeDate - 1 day
```

10.6 Creating an Operation Specification

Figure 10.4 shows the sequence diagram for the use case `Check campaign budget` first introduced in Chapter 9 (the use case description is repeated above in Section 10.4.2 and an activity diagram is shown in Fig. 10.6). In this particular example the message `checkCampaignBudget` invokes the operation `Campaign.checkCampaignBudget()`.

A specification for `Campaign.checkCampaignBudget()` is given below. We have used different fonts to signpost the specification as follows. This font (Arial)

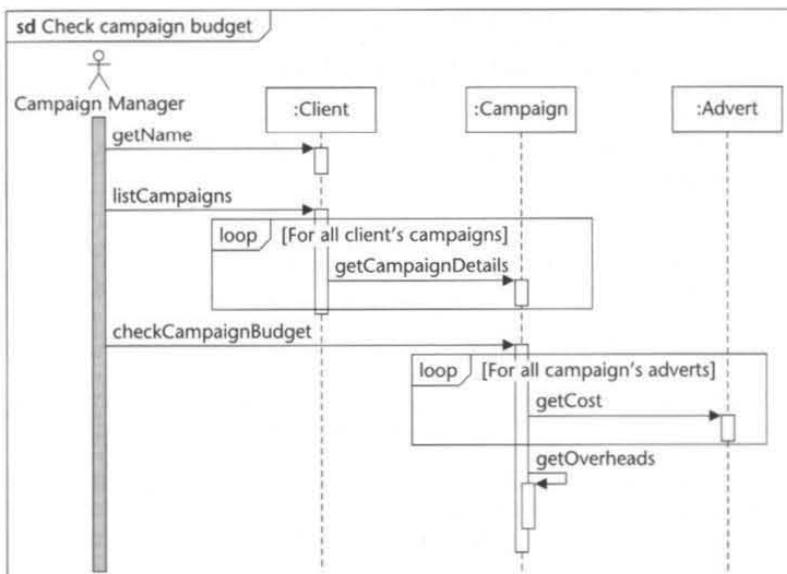


Figure 10.4 Sequence diagram for the use case `Check campaign budget`.

labels the specification structure, while this (Courier) highlights its content. Comments on the reasoning behind the specification are formatted like this (in Comic Sans).

Operation specification: checkCampaignBudget

Operation intent: return difference between campaign budget and actual costs.

The invocation appears not to require any parameters, but does have a return type that we can expect to contain a numerical value. Let us assume that there is a Money type available. The signature is shown below, followed by the pre- and post-conditions.

Operation signature: Campaign::checkCampaignBudget()

 budgetCostDifference:Money

Logic description (pre- and post-conditions):

```
context Campaign
pre: self->exists()
post: result = self.originalBudget - self.estimatedCost
      and self.estimatedCost = self.adverts.estimated
      Cost->sum()
```

As can be seen from the sequence diagram, this operation calls two other operations and these must be listed. In a full specification, full signatures would be recorded, but we omit this detail here.

Other operations called: Advert.getCost(), self.getOverheads()

Events transmitted to other objects: none

The only messages are those required to call the operations just mentioned, whose return values are required by this operation. An 'event' is a message that starts another distinct thread of processing (see Chapter 11).

Attributes set: none

This is a query operation whose only purpose is to return data already stored within the system.

Response to exceptions: none defined

Here we could define how the operation should respond to error conditions, e.g. what kind of error message will be returned if a calling message uses an invalid signature.

Non-functional requirements: none defined

Several non-functional requirements may be associated with the operation, but these need to be determined through discussion with users. They may include, for instance, response time under live conditions (enquiries that are made frequently typically require a faster response) or the format of the output (e.g. if there is a house standard that overspent budgets are displayed in red). However, these are really design issues, and would be noted at this stage only if the information happens to be available at the time.

10.7 | Summary

Operation specifications are the most detailed description of the behaviour of a system model. As such, they are also one of the more significant elements in the project repository. They provide an important link between the system's users, who typically possess a detailed understanding of the required system behaviour, and the designers and programmers who must implement this in software. Accurate specification of operations is essential if the software is to be coded correctly.

In this chapter we introduced the 'contract' as a framework for specifying operations, in terms of the service relationship between classes. Contracts are a particularly useful element of operation specification since they concentrate on the correctness of each object's behaviour.

We also described several techniques for describing operation logic. Non-algorithmic techniques, such as decision tables and pre- and post-condition pairs, take a black box approach and concentrate on specifying only the inputs to an operation (its pre-conditions) and the intended results of an operation (its post-conditions). In many cases, particularly where the operations themselves are simple, this is all the specification that a programmer needs to code the operation correctly.

Algorithmic techniques, such as Structured English, pseudo-code and activity diagrams, take a white box approach, and this means that they concentrate on defining the internal logic of operations. These techniques are particularly useful when an operation is computationally complex. They are also useful when we need to model some larger element of system behaviour, such as a use case, that has not yet been decomposed to the level of individual operations that can be assigned to specific classes.

Many elements of an operation specification can be written in OCL (UML's Object Constraint Language). OCL is intended for use as a formal language for specifying constraints and queries on an object model, and this includes operation pre- and post-conditions and invariants.

Review Questions

10.1 What are the two main purposes of an operation specification?

10.2 To what kinds of situation are decision tables particularly suited?

10.3 Why is it important to specify both pre- and post-conditions for an operation?

10.4 What are the main differences between algorithmic and non-algorithmic approaches to operation specification?

10.5 Why are non-algorithmic (or declarative) approaches generally preferred in object-oriented development?

10.6 Why are operation specifications in an object-oriented project likely to be small?

10.7 What are the three kinds of control structure in Structured English?

10.8 What is a sensible limit on the size of a Structured English specification?

10.9 What are the three components of most OCL expressions?

10.10 What is an invariant?

Case Study Work, Exercises and Projects

10.A Consider the first sequence diagram you drew for the use case Start line run from the FoodCo case study (in Exercise 9.A). Choose an operation in one of the classes involved in this ISD and write a contract for it. Make reasonable assumptions where necessary, and use a pre- and post-conditions approach for describing its logic.

10.B Consider the decision table in Fig. 10.1. Suppose you have learned that an extra condition must be taken into account: the rules in the current table actually apply only to campaigns with a total budget of £5,000 or over, but for smaller campaigns the thresholds for each action are different. Thresholds for smaller campaigns are as follows. For an expected overspend of less than 10%, no action is taken. For expected overspends of 10–19%, a letter is sent. For an expected overspend of 20% or more a letter is sent and a meeting is arranged. Draw a new version of the table that caters for small campaigns.

10.C Redraw the original decision table in Fig. 10.1 as an activity diagram. Do the same for your new decision table from Exercise 10.B.

10.D Consider the decision table in Fig. 10.1. Which of the three control structures are required to convert this into a Structured English specification? Rewrite the decision table in Structured English format.

10.E Find out how a decision tree differs from a decision table (e.g. from one of the books listed in Further Reading). Produce a decision tree that corresponds to the decision table in Fig. 10.1. What are the relative advantages and disadvantages of decision trees, decision tables and Structured English?

Further Reading

Decision tables and decision trees are both covered well in Kendall and Kendall (2005), while Avison and Fitzgerald (2003) and Hoffer et al. (2005) are both good sources on Structured English.

Larman (2005) describes a contract-based approach to object-oriented analysis and design, with examples taken through to Java code.

Meyer (1997) is perhaps the definitive text on design-by-contract in object-oriented software engineering. This book is very comprehensive, and quite technical in tone.

For a thorough introduction to OCL, Warmer and Kleppe (2003) have at the time of writing no real rivals.

For a statement of the school of thought (expressed chiefly by proponents of XP and other Agile Methods) that documentation, including operation specifications, is often unnecessary, see Jeffries (2001).

Further examples of operation specifications for Agate can be found in Case Study Chapter A4.

Specifying Control

Learning Objectives

In this chapter you will learn

- how to identify requirements for control in an application
- how to model object lifecycles using state machines
- how to develop state machine diagrams from interaction diagrams
- how to model concurrent behaviour in an object
- how to ensure consistency with other UML models.

11.1 Introduction

The UML notation that has been introduced so far enables us to model an application's requirements (use cases), the static structure of an application (class diagrams), the way in which objects interact (interaction diagrams) and the procedures or operations that objects perform (activity diagrams). A further important aspect of an application that must be modelled is the way that its response to events can vary depending upon the passage of time and the events that have occurred already. For an application such as a real-time system it is easy to understand that the response of the system to an event depends upon its state. For example, an aircraft flight control system should respond differently to events (for example, engine failure) when the aircraft is in flight and when the aircraft is taxiing along a runway. A more mundane example is that of a vending machine, which does not normally dispense goods until an appropriate amount of money has been inserted. This variation in behaviour is determined by the state of the vending machine—which depends on whether or not sufficient money has been inserted to pay for the item selected. In reality, of course, the situation is more complicated than this. For example, even when the correct amount of money has been inserted, the machine cannot dispense an item that is not in stock. It is important to model state-dependent variations in behaviour such as these since they represent constraints on the way that a system should act.

Objects can have similar variations in their behaviour dependent upon their state. The concepts of state and event are discussed in Section 11.2. These variations in behaviour represent important constraints on the way that an object acts, and are dictated by the requirements for the system. UML uses state machines to model states and state dependent behaviour for objects¹ and for interactions. The notation used in UML is based upon work by Harel (1987) and was adopted by OMT (Rumbaugh et al., 1991) and also in the second version of the Booch approach (Booch, 1994). We introduce the UML notation in Sections 11.3 and 11.4. The procedures for preparing state machines are explained in Section 11.5.

In UML 2.0 a distinction has been introduced between behavioural state machines and protocol state machines, which is sometimes useful. Behavioural state machines may be used to specify the behaviour of individual entities, for example class instances. Protocol state machines may be used to describe usage protocols for classes, interfaces and ports. This distinction is discussed further in Section 11.6.

There is an important link between interaction diagrams and state machines. A model of state behaviour in a state machine captures all the possible responses of a single object to all the use cases in which it is involved. By contrast, a sequence or a communication diagram captures the responses of all the objects that are involved in a single use case or other interaction. A state machine can be seen as a description of all the possible lifecycles that an object of a class may follow. It can also be seen as a more detailed view of a class. Some modelling tools handle state machines as a child diagram to a class on a class diagram. We consider issues regarding consistency between models in Section 11.7 and examine quality guidelines in Section 11.8.

It is not only object-oriented approaches that have recognized the importance of modelling the lifetime behaviour of entities. For example, SSADM (Skidmore et al., 1994) uses entity life histories for this purpose, and some approaches to the development of real-time systems, such as that of Yourdon, model the state behaviour of a system or subsystem.

The state machine is a versatile model and can be used within an object-oriented approach to describe the behaviour of other model entities. In Chapter 17 we show how to use state machines to build models of human-computer dialogues.

11.2 States and Events

All objects have a *state*. The current state of an object is a result of the events that have occurred to the object and is determined by the current value of the object's attributes and the links that it has with other objects. Some attributes and links of an object are significant for the determination of its state while others are not. For example, in the Agate case study the values of the *staffName* and *staffNo* attributes of a *StaffMember* object have no impact upon its state, whereas the date that a staff member started his or her employment at Agate determines when the probationary period of employment ends (after six months, say). A *StaffMember* object is in the *Probationary* state for the first six months of employment. While

¹ State machines, previously known as statecharts, may be used to show state changes for various UML elements including systems, subsystems, interfaces and ports.

in this state, a staff member has different employment rights and is not eligible for redundancy pay in the event that they are dismissed by the company.

A state describes a particular condition that a modelled element (e.g. object) may occupy for a period of time while it awaits some event or *trigger*. The possible states that an object can occupy are limited by its class. Objects of some classes have only one possible state. For example, in the Agate case study a *Grade* object either exists or it does not. If it exists it is available to be used, and if it does not exist it is not available. Objects of this class have only one state, which we might name *Available*. Objects of other classes have more than one possible state. For example, an object of the class *GradeRate* may be in one of several states. It may be *Pending*, if the current date is earlier than its start date, *Active*, if the current date is equal to or later than the start date but earlier than the finish date (we assume that the finish date is later than the start date), or *Lapsed*, if the current date is later than the finish date for the grade. If the current date is at least a year later than the finish date then the object is removed from the system. The current state of a *GradeRate* object can be determined by examining the values of its two date attributes. Alternatively, the *GradeRate* class might have a single attribute (an enumerated type—that has an integer value for each possible state) with values that indicate the current state of an object. An attribute that holds the value of the current state of an object is sometimes known as a *state variable*. It is important to note that movement from one state to another for a *GradeRate* object is dependent upon events that occur with the passage of time. Figure 11.1 shows a state machine for *GradeRate*. This state machine is presented in frame of kind state machine (*sm* could be used as shortened form). UML 2.0 is quite flexible about the use of frames, and states that they need not be used when the diagram boundary is clear².

Movement from one state to another is called a *transition*, and is initiated by a *trigger*. A trigger is an event that can cause a state change and that is relevant to the object (or to the modelled element). When its triggering event occurs a transition is said to *fire*. A transition is shown as an open arrow from the source state to the target state. For example, the cancellation of an advert at Agate is a trigger that will change the state of the *Advert* object being cancelled. Just as a set of objects is defined by the class of which they are all instances, events are defined by an *event type* of which each event is an instance. For example, the cancellation of an advert in the CheapClothes jeans campaign is one instance of an event, and the cancellation of an advert in the Soong Motor Co Helion campaign is another instance. Both are defined by the event type *cancellationOfAdvert*. This event is a trigger as it causes a state change. A trigger can have parameters and a return value, and is implemented by a message in an object-oriented system. Triggers can be grouped into several general types.

A *change trigger* occurs when a condition becomes true. This is usually described as a Boolean expression, which means that it can take only one of two values: true or false. This form of conditional event is different from a guard condition, which is normally evaluated at the moment that its associated event fires.

A *call trigger* occurs when an object receives a call for one of its operations, either from another object or from itself. Call triggers correspond to the receipt of a call message and are annotated by the signature of the operation as the trigger for the transition.

² Frames must be used for interaction diagrams.

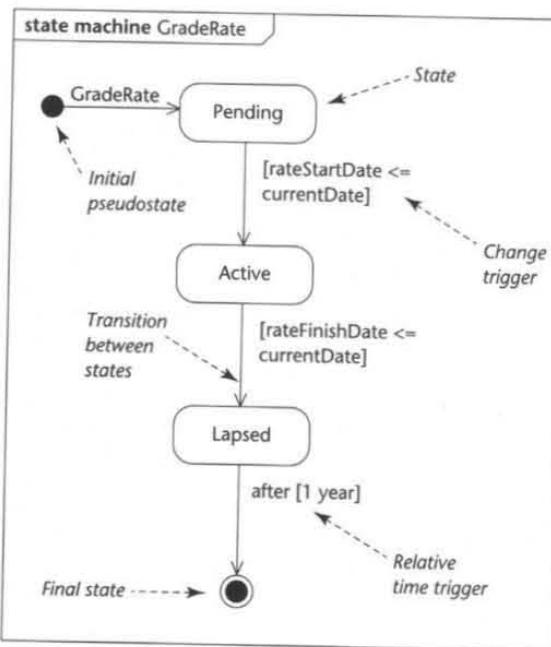


Figure 11.1 State machine for the class GradeRate.

A *signal trigger* occurs when an object receives a signal³. As with call triggers the event is annotated with the signature of the operation invoked. There is no syntactic difference between call triggers and signal triggers. It is assumed that a naming convention is used to distinguish between them.

A *relative time trigger* is caused by the passage of a designated period of time after a specified event (frequently the entry to the current state). Relative time triggers are shown by time expressions near the transitions. The time expression is placed in parentheses and should evaluate to a period of time. It is preceded by the keyword *after* and if no starting time is indicated it reflects the passage of time since the most recent entry to the current state.

11.3 Basic Notation

All state machines need to have some starting state (at least notionally). The *initial pseudostate* (in other words the starting point) of a state machine is indicated by a small solid filled circle. The initial pseudostate is a notational convenience and an object cannot remain in its initial state, but must immediately move into another named state. In Fig. 11.1 the GradeRate object enters the Pending state immediately on its creation. A transition from the initial state can optionally be labelled with the trigger that creates the object. The end point of a state machine (in other

³ An asynchronous message.

words its final state) is shown by a bull's-eye symbol. This too is a notational convenience and an object cannot leave its final state once the state has been entered. All other states are shown as rectangles with rounded corners and should be labelled with meaningful names. Each node in a state machine diagram is known as a vertex. In Fig. 11.1 all transitions except the first and the last transitions have change triggers. The state machine for a GradeRate object is very simple, since it enters each state only once. Some classes have much more complex lifecycles than this. For example, a BookCopy object in a library system may move many times between the states OnLoan and Available.

Figure 11.2 shows an alternative notation for a state, which may be used for composite states. Figure 11.3 illustrates the basic notation for a state machine with two states for the class Campaign and one transition between them. A transition should be annotated with a *transition string* to indicate the event that triggers it.

For call and signal events the format of the transition string is as follows:

```

trigger-signature '[' constraint ']' '/'
activity-expression
  
```

The trigger signature takes the following form:

```

event-name '(' parameter-list ')'
  
```

where the event-name may be the call or signal name and the parameter-list contains parameters of the form:

```

parameter-name ':' type-expression
  
```

separated by commas. (Characters in single quotes, such as '(', are *literals* that appear as part of the event.) Note that empty brackets '()' have not been used when no parameters are listed.

A constraint is a Boolean expression that is evaluated at the time the trigger fires. It is known as *guard condition* or just a *guard* ('it guards the transition'). The transition only takes place if the guard condition evaluates to true. A guard condition is a constraint that may involve parameters of the trigger, attributes or links of the object that owns the state machine. A guard is shown in square brackets—'[' ... ']'.⁴

In Fig. 11.3 the guard condition is a test on the contractSigned attribute in the class Campaign and since the attribute is Boolean it may be written as follows:

```

[contractSigned]
  
```

This expression evaluates to true only if contractSigned is true. A guard condition can also be used to test concurrent states of the current object or the state of some other reachable object. Concurrent states are explained later in Section 11.4.2.



Figure 11.2 Alternative notation for a state.

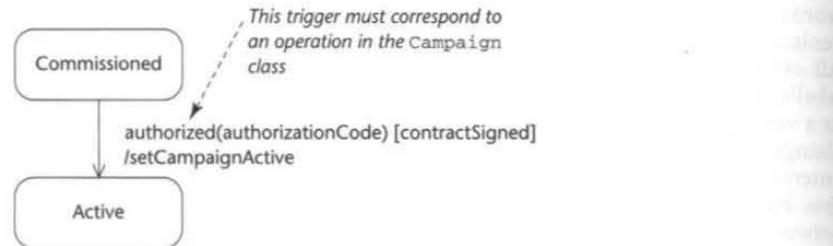


Figure 11.3 Fragment of the state machine for the class Campaign.

An *activity-expression* is executed when a trigger causes the transition to fire. Like a guard condition, it may involve parameters of the trigger and may also involve operations, attributes and links of the owning object or modelling element. In Fig. 11.3 the activity-expression begins with the '/' delimiter character and is the execution of the Campaign object's operation `setCampaignActive`.

An activity-expression may comprise a sequence of actions and include actions that may generate events such as sending signals or invoking operations. Each action in an action string is separated from its preceding action by a semi-colon. An example of an activity-expression with multiple actions is shown in the transition string below:

```
left-mouse-down(location) [validItemSelected] / menuChoice  
= pickMenuItem(location); menuChoice.highlight
```

The sequence of actions in an activity-expression is significant since it determines the order in which they are executed. In the example above, if the actions were in the reverse order, the value of `menuChoice` would be likely to be different when the `highlight` message is sent and the effect of the event would be different. Actions are considered to be atomic (that is, they cannot be subdivided) and cannot be interrupted once they have been started. Once initiated this action must execute fully before any other action is considered. This is known as '*run-to-completion*' semantics. An action will actually take time to execute in a live application but this is dependent only upon internal factors such as the complexity of the program code and the speed of the processor. The duration of a state is normally dependent upon external events in the application environment. The effect of a trigger can be deferred to another state. This *deferred event* is shown by listing the trigger in the state symbol with postfix `/defer`.

So far we have considered only action-expressions that are associated with a transition. It can also be useful to model internal activities associated with a state. These activities may be triggered by events that do not change the state, by an event that causes the state to be entered or by an event that results in exiting the state. A state vertex may be subdivided into separate compartments: name compartment, internal activities compartment and internal transitions compartment.

The *name compartment* holds the name of the state. States may be unnamed and anonymous. Normally a state is only represented on a state machine diagram once (except for the final state).

The *internal activities compartment* lists the *internal activities* or *state activities* that are executed in that state. Each internal activity has a label stipulating under

which circumstances the activity expression will be invoked. Three kinds of internal event have a special notation. Two of these are the *entry activities* and the *exit activities*, respectively indicated by the keywords `entry` and `exit`. These cannot have guard conditions as they are invoked implicitly on entry to the state and exit from the state respectively. Entry or exit *activity-expressions* may also involve parameters of incoming transitions (provided that these appear on all incoming transitions) and attributes and links of the owning object. It is important to emphasize that any transition into a state causes the entry activity to fire and all transitions out of a state cause the exit activity to fire.

```
'entry' // 'activity-name' ('parameter-list')  
'exit' // 'activity-name' ('parameter-list')
```

State activities are preceded by the keyword `do` and have the following syntax:

```
'do' // 'activity-name' ('parameter-list')
```

State activities may 'persist' for a period of time, perhaps the duration of the state. For example in Fig. 11.5 the state activity `playSoundClip` will last as long as the sound clip or as long as the object remains in the state `Menu.Visible`, whichever is the shorter.

The *internal transitions compartment* lists internal transitions. Each of these transitions is described in the same way as a trigger. Internal transitions do not cause a state change and do not invoke the exit or entry activities.

In Fig. 11.4 the state symbol is shown with two compartments, a name compartment and an internal activities compartment.

Figure 11.5 shows the `Menu.Visible` state for a `DropDownMenu` object with three compartments. On occasions the internal transitions and the internal activities are placed in a single compartment within a state. This is a matter of notational style. In this example, the entry activity causes the menu to be displayed. The entry activity runs to completion before the state `do` activity is invoked. While the object remains in the `Menu.Visible` state, the state `do` activity causes a sound clip to be played and, if the event `itemSelected` occurs, the action `highlightItem` is invoked. It is important to note that, when the event `itemSelected` occurs, the `Menu.Visible` state is not exited and entered and as a result the exit and entry activities are not invoked. When the state is actually exited the menu is hidden.

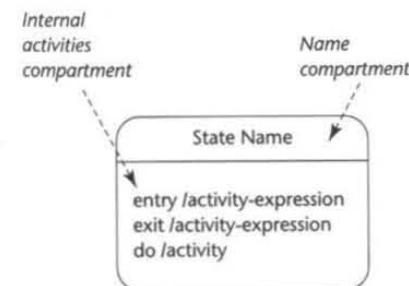


Figure 11.4 Internal activities for a state.

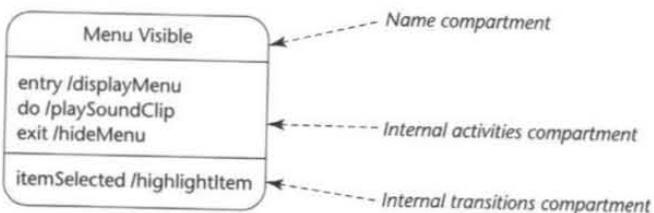


Figure 11.5 MenuVisible state for a DropDownMenu object.

Figure 11.6 shows a state machine for the class Campaign. The transition from the initial pseudostate to the Commissioned state has been labelled only with an activity-expression that comprises the operations assignManager and assignStaff. Execution of these operations ensures that when a campaign is created a manager and at least one member of staff are assigned to it⁴. The operations are triggered by the event that creates a Campaign object. The transition from the Completed state to the Paid state has a guard condition that only allows the transition to fire if total amount due (paymentDue) for the Campaign has been completely paid (note that this guard condition allows a Campaign to enter the Paid state when the client overpays).

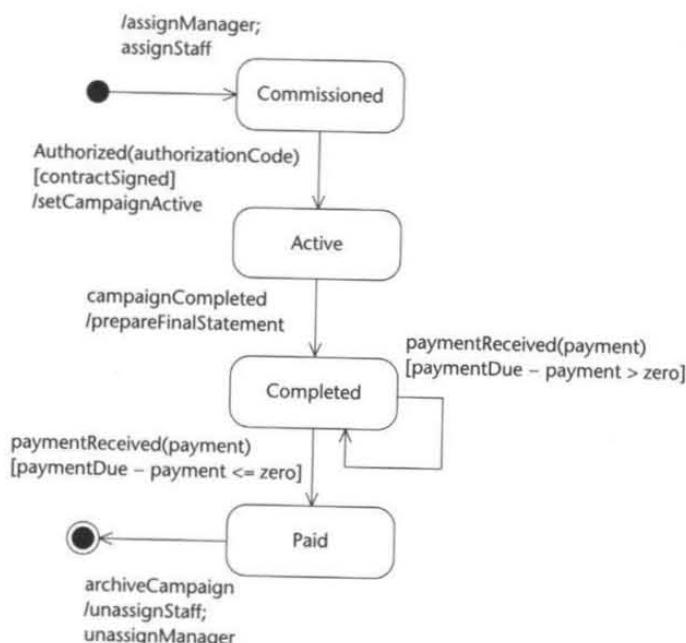


Figure 11.6 State machine for the class Campaign.

⁴ Unless the specifications for these operations permit a null option.

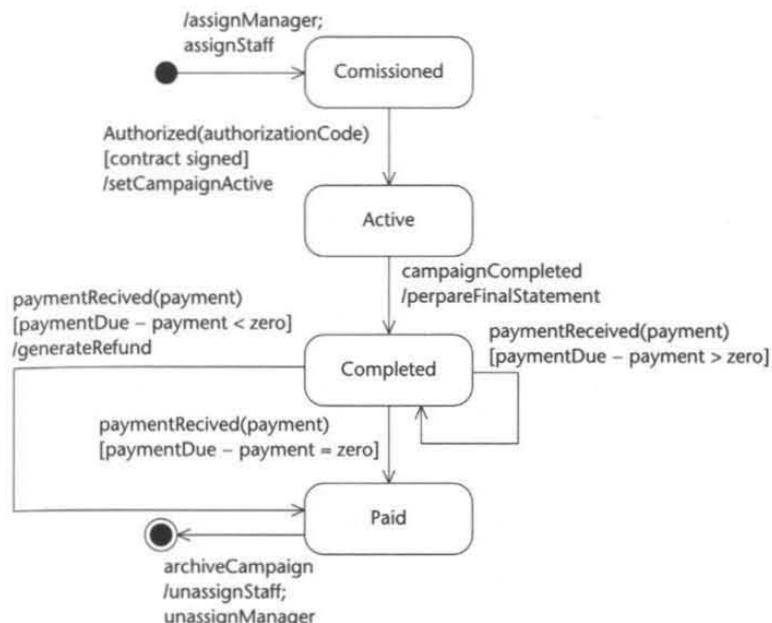


Figure 11.7 A revised state machine for the class Campaign.

The reflexive transition from the Completed state models any payment event that does not reduce the amount due to zero or less. Only one of the two transitions from the Completed state (one of which is recursive) can be triggered by the paymentReceived event since the guard conditions are mutually exclusive. It would be bad practice to construct a state machine where one event can trigger two different transitions from the same state. A state machine is only unambiguous when all the transitions from each state are mutually exclusive.

If the user requirements were to change, so that an overpayment is now to result in the automatic generation of a refund, the state machine could be changed. Since the action that results from an overpayment is different from the action that results from a payment that reduces paymentDue to zero, a new transition is needed from the Completed state to the Paid state. The guard conditions from the Completed state must also be modified. Figure 11.7 shows a state machine that captures this requirement. It is important to appreciate that the state machines in Figs 11.6 and 11.7 are not equivalent to each other, but capture different versions of the users' requirements.

11.4 Further Notation

The state machine notation can be used to describe highly complex time-dependent behaviour. Hierarchies of states can be nested and concurrent behaviour can also be represented.

11.4.1 Composite states

When the state behaviour for an object or an interaction is complex, it may be necessary to represent it at different levels of detail and to reflect any hierarchy of states that is present in the application. For example, in the state machine for Campaign the state Active encompasses several *substates*. These are shown in Fig. 11.8 where the Active state is seen to comprise three disjoint substates: Advert Preparation, Scheduling and Running Adverts, which are placed in the *decomposition compartment* of the state. This decomposition compartment has one region. This diagram now shows a single state that contains a nested state diagram within it known as *submachine*. In the nested state machine within the Active state, there is an initial pseudostate with a transition to the first substate that a Campaign object enters when it becomes active. The transition from the initial pseudostate to the first substate (Advert Preparation) should not be labelled with an event but it may be labelled with an action, though this is not required in this example. It is implicitly fired by any transition to the Active state⁵. A final pseudostate symbol may also be shown on a nested state diagram. A transition to the final pseudostate symbol represents the completion of the activity in the enclosing state (i.e. Active) and a transition out of this state triggered by the completion event unless overridden by a specific trigger. This transition may be unlabelled (as long as this does not cause any ambiguity) since the event that triggers it is implied by the completion event (see Fig. 11.11).

When a campaign enters the Active state in Fig. 11.8 it first enters the Advert Preparation substate, then if the adverts are approved it enters the Scheduling substate and finally enters the Running Adverts substate when the schedule is approved. If the campaign is deemed completed, the object leaves the Running

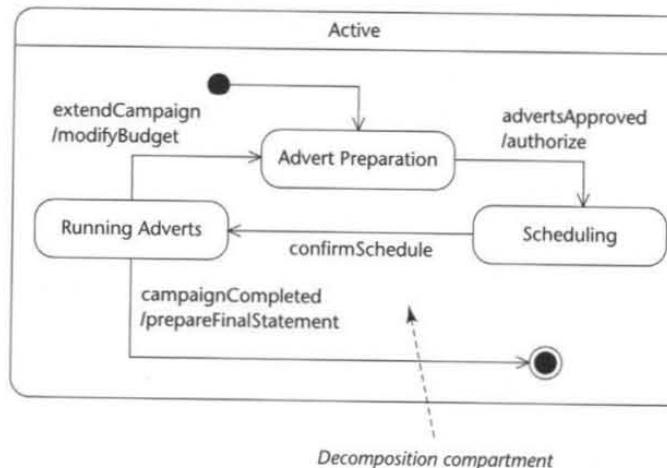


Figure 11.8 The Active state of Campaign showing nested substates.

⁵ Except transitions to entry or history pseudostates. These are explained in Sections 11.4.3 and 11.4.5 respectively.

Adverts substate and also leaves the Active enclosing state, moving now to the Completed state (see Fig. 11.7). If the campaign is extended while in the Running Adverts substate, the Advert Preparation substate is re-entered (Fig. 11.8). A high-level state machine for the class Campaign can be drawn to include within the main diagram the detail that is shown in the nested state machine for the Active state if so desired. If the detail of the submachine is not required on the higher-level state machine or is just too much to show on one diagram, the higher-level state machine can be annotated with the hidden decomposition indicator icon (two small state symbols linked together) as shown in Fig. 11.9. The sub-machine Running is referenced in the state name compartment using the syntax:

state name ':' reference-state-machine-diagram-name

The state Active is known as a *submachine state* because it contains a sub-machine.

11.4.2 Concurrent states

Objects can have *concurrent states*. This means that the behaviour of the object can best be explained by regarding it as a product of two (or more) distinct sets of substates, each state of which can be entered and exited independently of substates in the other set. Figure 11.10 illustrates this form with two submachines, Running and Monitoring.

Suppose that further investigation reveals that at Agate a campaign is surveyed and evaluated while it is also active. A campaign may occupy either the Survey substate or the Evaluation substate when it is in the Active state. Transitions between these two states are not affected by the campaign's current state in relation to the preparing and running of adverts. We model this by splitting the Active state into two concurrent nested submachines, Running and Monitoring, each in a separate region of the Active state machine decomposition compartment. This is shown in Fig. 11.10 by dividing the decomposition compartment with a dashed line. The concurrent states are described as orthogonal⁶, meaning that they are independent of each other.

A transition to a complex state such as this one is equivalent to a simultaneous transition to the initial states of each concurrent submachine. An initial state must be specified in both nested submachines in order to avoid ambiguity about which substate should first be entered in each concurrent region. A transition to the Active state means that the Campaign object simultaneously enters the Advert



Figure 11.9 The Active submachine state of Campaign with detail hidden.

⁶ Orthogonal literally means at right angles.

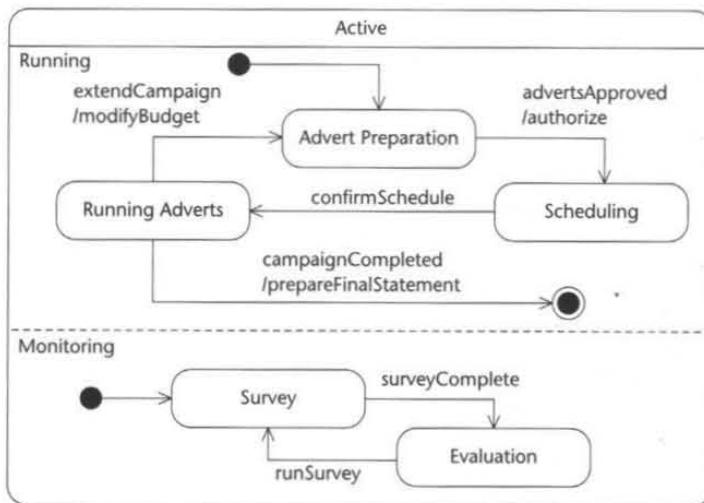


Figure 11.10 The Active state with concurrent substates.

Preparation and Survey substates after any entry activities defined for the Active state itself have been invoked. A transition may now occur within either concurrent region without having any effect on the substate in the other concurrent region. However, a transition out of the Active state applies to all its substates (no matter how deeply nested). In a sense, we can say that the substates inherit the campaignCompleted transition from the Active state (shown in Fig. 11.7) since the transition applies implicitly to them all. This is equivalent to

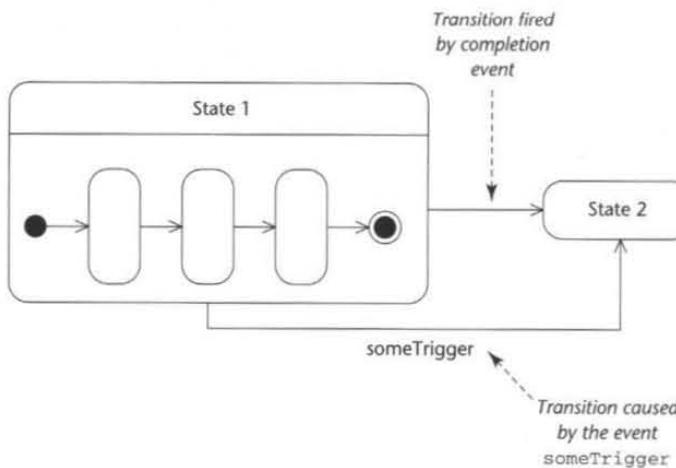


Figure 11.11 Completion event.

saying that an event that triggers a transition out of the Active state also triggers a transition out of any substates that are currently occupied. The submachine Monitoring does not have a final state and, when the Active state is exited, one of the two states Survey or Evaluation will be occupied. Inherited transitions can be masked if a transition with the same trigger is present in one of the nested state machines (as is the case for the campaignCompleted transition from the Running Adverts state in Fig. 11.10).

In general composite states may exit in several ways. Figure 11.11 illustrates two of these. State 1 will exit when the trigger someTrigger occurs irrespective of which substate the submachine occupies. Alternatively State 1 will exit when its submachine is completed, generating a completion event. The unlabelled transition from State 1 to State 2 represents the transition caused by this completion event.

Figure 11.12 shows the use of a *fork pseudostate* splitting the transition into two paths, each leading to a specific concurrent substate. It also shows that the containing state is not exited until both parallel nested submachines are exited with transitions that merge at the *join pseudostate*.

11.4.3 Entry and exit pseudostates

On occasions it is useful to model exceptional entry to and exit from a submachine state. This is achieved using an *entry pseudostate* and an *exit pseudostate*. Figure 11.13 shows an example of this notation in use. The state machine Advert has three states StoryBoard, AdvertPrep and AdvertRunning. AdvertPrep is a submachine state with entry and exit pseudostates defined. The state machine AdvertPrepSM also has the same entry and exit pseudostates defined (they have the same names—Advert Reworked and Advert Aborted respectively). In AdvertPrepSM these pseudostates are shown on the frame boundary. Alternatively they could have been placed inside the frame. The trigger abort causes the state machine AdvertPrepSM to exit via Advert Aborted and this then follows the transition to the final state in the state machine Advert. The transition from AdvertRunning to Advert Reworked causes the submachine to start from the state linked to that entry pseudostate.

11.4.4 Junction and choice pseudostates

UML offers notation to show decision points on state machine diagrams. Junction and choice pseudostates may both be used for this purpose, although they have

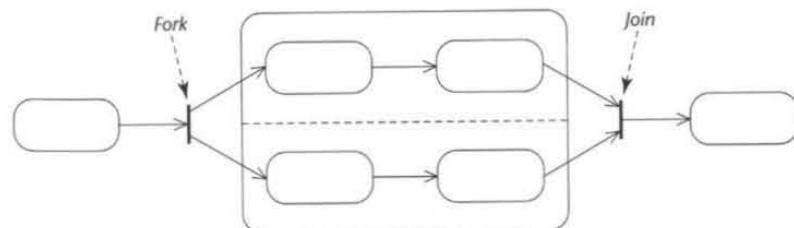


Figure 11.12 Parallel states using fork and join.

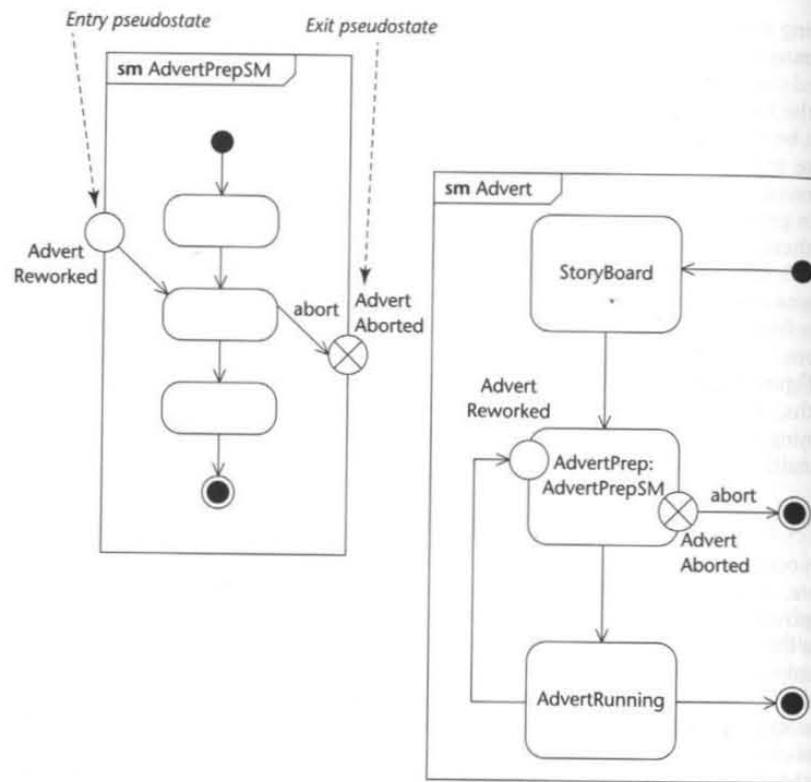


Figure 11.13 Entry and exit pseudostates.

subtly different semantics. Figure 11.14 illustrates the notation. The diagram shows transitions from StateA to StateB and from StateA to StateC, which illustrate a compound transition. In general a compound transition may have more than one source state and more than one target state and will use junction, choice, fork or join pseudostates.

A *junction pseudostate* (the notation is the same as that for the initial pseudostate) has one or more entry transitions and will have one or more exit transitions. When there are many entry transitions and one exit transition this is known as a *merge*. When there are several exit transitions and only one entry transition this is known as a *static conditional branch*. Each exit transition will have a guard condition. The transition whose guard condition evaluates as true fires. These guards are evaluated before the compound transition fires. If several guard conditions evaluate as true the transition that fires will be one of these.

Choice pseudostates allow the splitting of transitions into multiple exit transitions. Each exit transition will have a guard condition and, as can be seen in Fig. 11.14, if they all share a common argument it may be placed in the diamond-shaped choice symbol. The difference between choice and junction pseudostates is

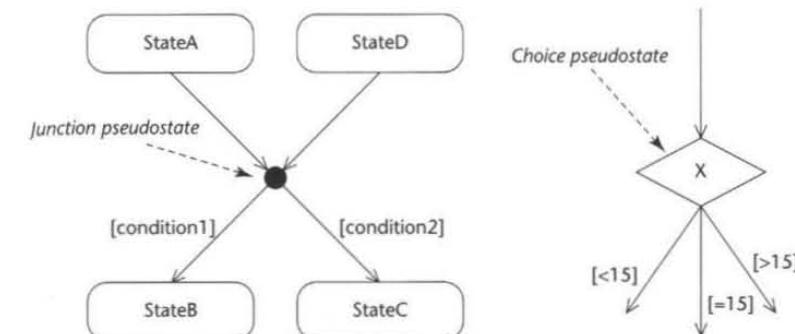


Figure 11.14 Junction and choice pseudostates.

that for choice pseudostates the guard conditions are evaluated at the time the choice point is reached. Any actions associated with the first transitions in the compound transition (before the choice state) will have been executed and may affect which of the guard conditions evaluates to true.

11.4.5 History pseudostates

Composite states can be entered in various ways. We have already discussed the use of an entry pseudostate, which allows the submachine of composite state to be entered at a pre-determined substate. If a composite state has been entered and then exited prematurely before its submachine has completed, it may be useful to resume the composite state at the substate that was last active. The *shallow history pseudostate* and *deep history pseudostate* in Figure 11.15 are used to represent this.

For example, in Figure 11.16, a Campaign in the Active state may be suspended by the suspendCampaign trigger and move into the Suspended state. At some point the issue that resulted in the suspension is resolved and the campaign resumes where it left off. In this example the state Active has two parallel submachines, both of which need to be resumed from their last active substate. This is shown by the transition resumeCampaign from Suspended. This goes to the *fork pseudostate* and splits into two pathways, one going to the *shallow history pseudostate* in Running and the other going to the shallow history pseudostate in Monitoring. (A region may have no more than one history pseudostate.) Each of these history pseudostates then activates the last substate that was active in its region. The unlabelled transition from the shallow history pseudostate indicates

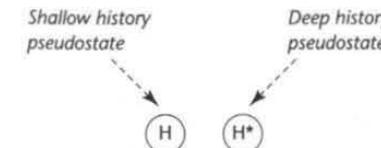


Figure 11.15 History pseudostates.

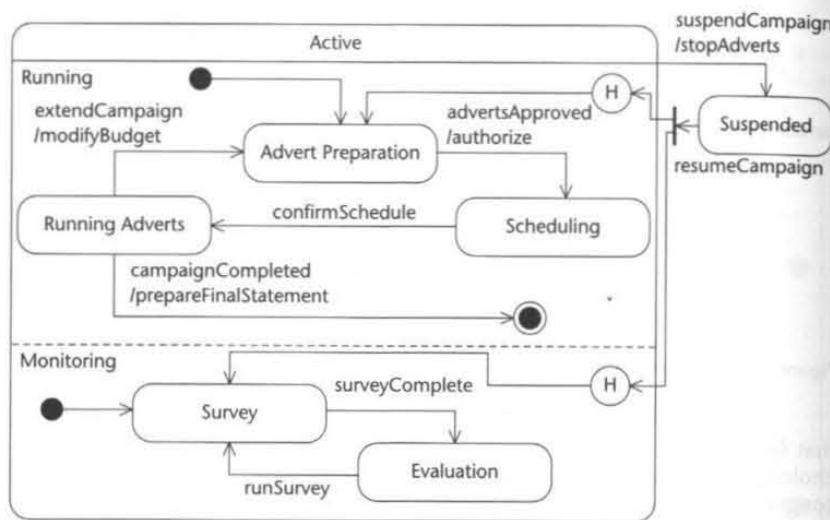


Figure 11.16 The Active state with history pseudostates.

the default shallow history substate. This state is activated if the last active substate was the final state or if the composite state had not been previously active. For Running the default shallow history substate is Advert Preparation and for Monitoring it is Survey.

Composite states may have substates that are in turn composite and so on. Thus, there may be an arbitrarily deep nesting of substates. The *deep history pseudostate* works in a similar way to the shallow history pseudostate but causes the composite state to resume at the last active state in each of the nested submachines within that region, no matter how deeply nested they are. In Fig. 11.16 shallow history and deep history pseudostates would have the same effect as there is only one level of nesting.

11.4.6 State machine specialization

UML 2.0 has also introduced the facility to specialize state machines in the same way that classes can be specialized. For example, a new class International Campaign might be identified for Agate. The state machine for objects of this new class would be a specialization of the Campaign state machine and may add new states and transitions.

11.5 Preparing a State Machine

State machines can be prepared from various perspectives. The state machine for a class can be seen as a description of the ways that use cases can affect objects of that class. Use cases give rise to interaction diagrams (sequence diagrams or communication diagrams) and these can be used as a starting point for the preparation of a state machine.

Interaction diagrams show the messages that an object receives during the execution of a use case. The receipt of a message by an object does not necessarily correspond to an event that causes a state change. For example, simple ‘get’ messages (e.g. getTitle) and query messages (e.g. listAdverts) are not events in this sense. This is because they do not change the values of any of the object’s attributes, nor do they alter any of its links with other objects. Some messages change attribute values without changing the state of an object. For example, a message receivePayment to a Campaign object will only cause a change of state to Paid if it represents payment of at least the full amount due.

11.5.1 A behavioural approach

Figure 11.17 shows a sequence diagram for the use case Record completion of a campaign. The receipt of the message campaignCompleted by a Campaign object is an event from the perspective of the Campaign object. In this example this event is a call trigger and causes the campaignCompleted() operation to be invoked, triggering a transition from the Active state to the Completed state. Incoming messages to an object generally correspond to a trigger causing a transition to fire. Allen and Frost (1998) describe the use of interaction diagrams to develop a state machine as a behavioural approach.

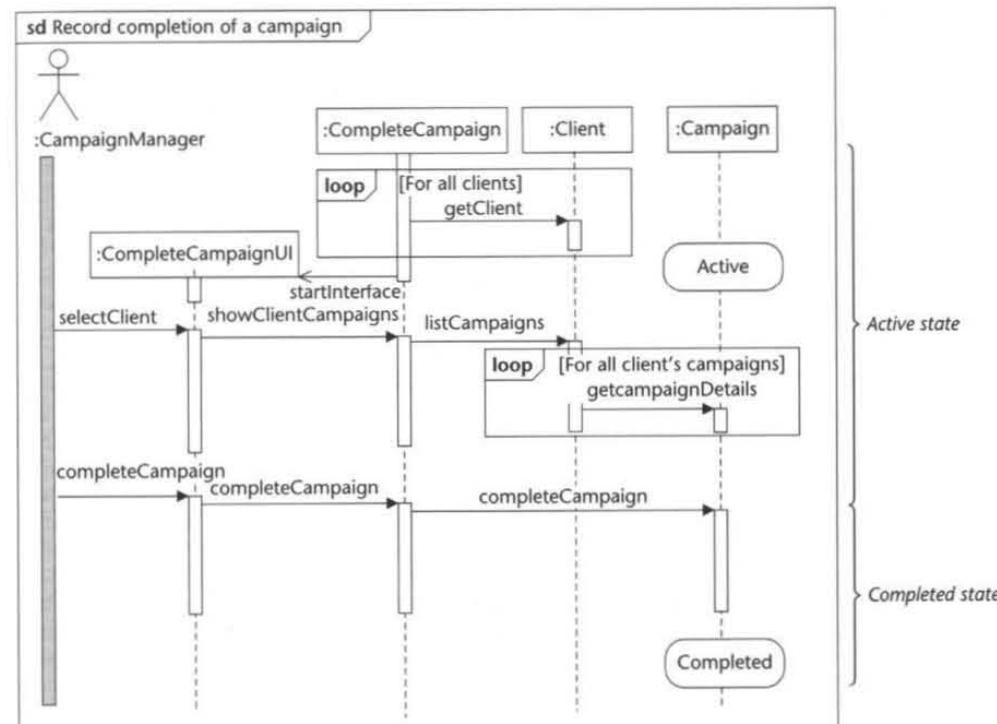


Figure 11.17 Sequence diagram for use case Record completion of a campaign.

The preparation of a state machine from a set of interaction diagrams using this behavioural approach has the following sequence of steps.

1. Examine all interaction diagrams that involve each class that has heavy messaging.
2. For each class for which a state machine is being built follow steps 3 to 9.
3. On each interaction diagram identify the incoming messages that may correspond to events for the class being considered. Also identify the possible resulting states.
4. Document these events and states on a state machine.
5. Elaborate the state machine as necessary to cater for additional interactions as these become evident, and add any exceptions.
6. Develop any nested state machines (unless this has already been done in an earlier step).
7. Review the state machine to ensure consistency with use cases. In particular, check that any constraints that are implied by the state machine are appropriate.
8. Iterate steps 4, 5 and 6 until the state machine captures the necessary level of detail.
9. Check the consistency of the state machine with the class diagram, with interaction diagrams and with any other state machines⁷.

The sequence diagram in Fig. 11.17 has been annotated to indicate the state change that is triggered by the event `campaignCompleted`. In order to identify all incoming messages that may trigger a state change for an object, all interaction diagrams that affect the object should be examined (sequence diagrams are probably easier to use for this purpose than communication diagrams, but this is a matter of personal preference). Analysis of the interaction diagrams produces a first-cut list of all events (caused by incoming messages) that trigger state changes, and also a first-cut list of states that the object may enter as a result of these events. If only major interactions have been modelled then the lists will not be complete, but they can still provide an effective starting point.

The next step is to prepare a draft state machine for the class. Figure 11.18 illustrates the level of detail that might be shown in a first-cut state machine for the `Campaign` class. This would need to be expanded in order to reflect any events that have not been identified from the interaction diagrams, and also to include any exceptions. Complex nested states can be refined at this stage. A review of the state machine in Fig. 11.18 results in the addition of the `Active` state to encompass the states `Advert Preparation`, `Scheduling` and `Running Adverts` (shown in the revised state machine in Fig. 11.19).

The state machine is then compared to use cases in order to check that the constraints on class behaviour shown in the state machine satisfy the requirements documented in the use case. In this example the states `Surveying` and

⁷ This step should also include checking of consistency with any class diagram constraints and constraints defined in operation specifications. In operation specifications this typically involves consideration of pre- and post-conditions and invariants.

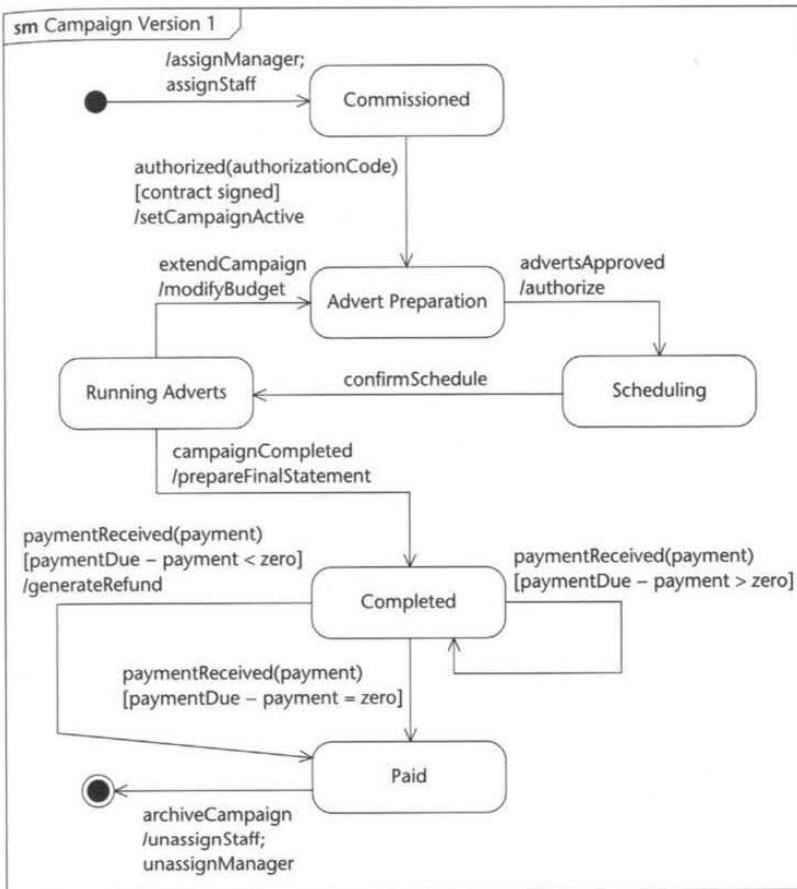


Figure 11.18 Initial state machine for the `Campaign` class—a behavioural approach.

Evaluating have not yet been included. These might be identified in a final sweep-up to check that the state machine is complete, and could then be added as concurrent states within the `Active` state.

Let us suppose that further investigation of behaviour that can affect a campaign reveals that in some circumstances a campaign can be cancelled. This is not permitted after a campaign has been completed but a campaign can be cancelled while it is in the `Commissioned` state or in the `Active` state. In either case cancellation costs are calculated for billing to the client. If the campaign is active then advertisement schedules are also cancelled. A final state machine that includes this additional requirement is shown in Fig. 11.20. In this version the transition `campaignCompleted` is shown explicitly from the nested concurrent substate `Running Adverts` to the state `Completed`. When a transition like this fires, any exit activities for the other concurrent substates that are occupied are performed.

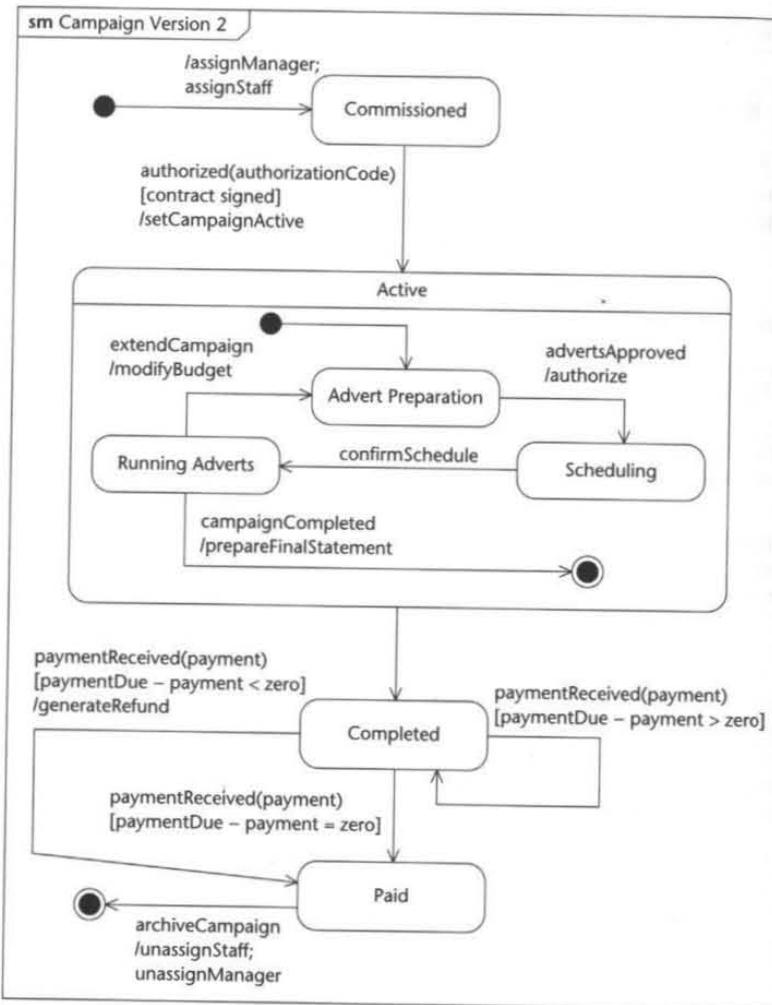


Figure 11.19 Revised state machine for the Campaign class.

11.5.2 A lifecycle approach

An alternative approach to the preparation of state machines is based on the consideration of lifecycles for objects of each class. This approach does not use interaction diagrams as an initial source of possible events and states. Instead, they are identified directly from use cases and from any other requirements documentation that happens to be available. First, the main system events are listed (at Agate 'A client commissions a new campaign' might be one of the first to consider). Each event is then examined in order to determine which objects are likely to have a state dependent response to it.

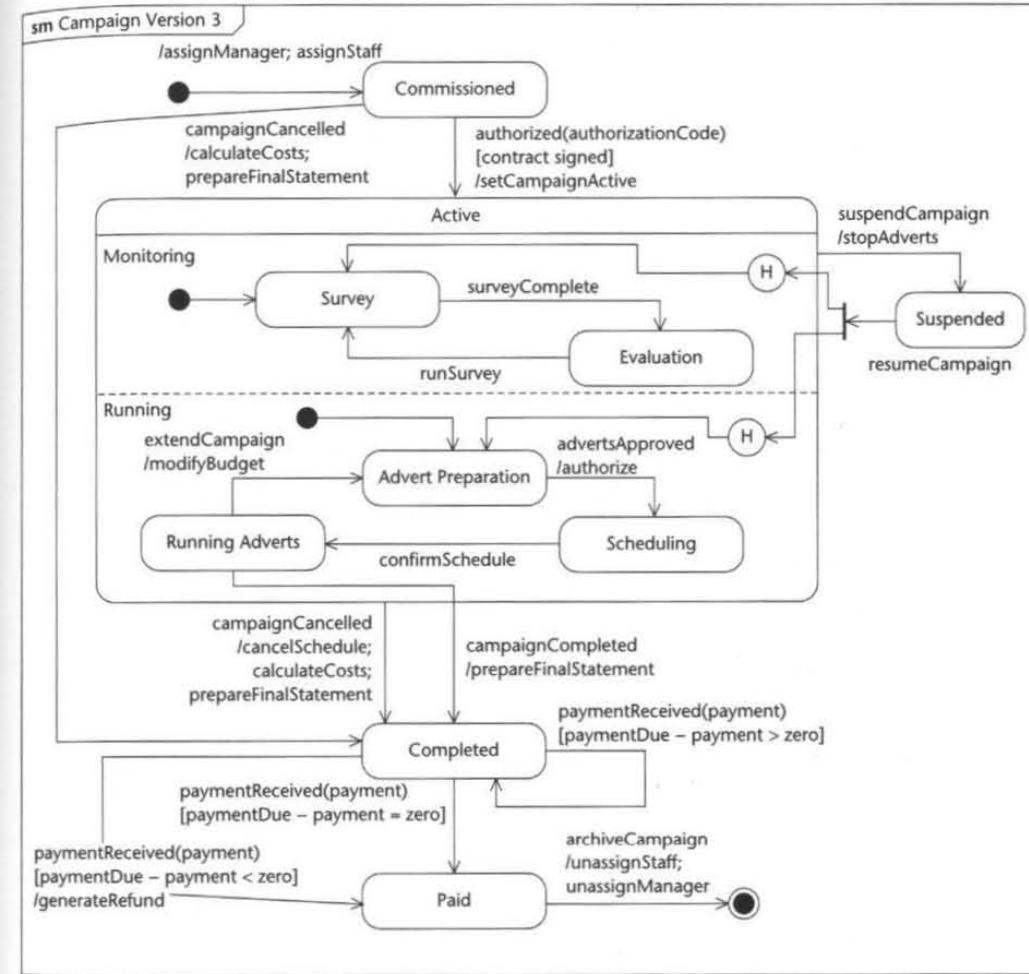


Figure 11.20 Final version of the Campaign state machine.

The steps involved in the lifecycle approach to state modelling are as follows.

1. Identify major system events.
2. Identify each class that is likely to have a state-dependent response to these events.
3. For each of these classes produce a first-cut state machine by considering the typical lifecycle of an instance of the class.
4. Examine the state machine and elaborate to encompass more detailed event behaviour.

5. Enhance the state machine to include alternative scenarios.
6. Review the state machine to ensure that it is consistent with the use cases. In particular, check that the constraints that the state machine implies are appropriate.
7. Iterate through steps 4, 5 and 6 until the state machine captures the necessary level of detail.
8. Ensure consistency with class diagram and interaction diagrams and other state machines.

The lifecycle approach is less formal than the behavioural approach in its initial identification of events and relevant classes. It is often helpful to use a combination of the two, since each provides checks on the other. A lifecycle approach might produce Fig. 11.6 as an initial first-cut state machine for the `Campaign` class, but further elaboration should still result in the state machine shown in Fig. 11.20.

11.6 Protocol and Behavioural State Machines

UML 2.0 introduces a distinction between protocol and behavioural state machines. So far in this chapter we have been developing behavioural state machines. Protocol state machines differ in that they only show all the legal transitions with their pre- and post-conditions. The states of a protocol state machine

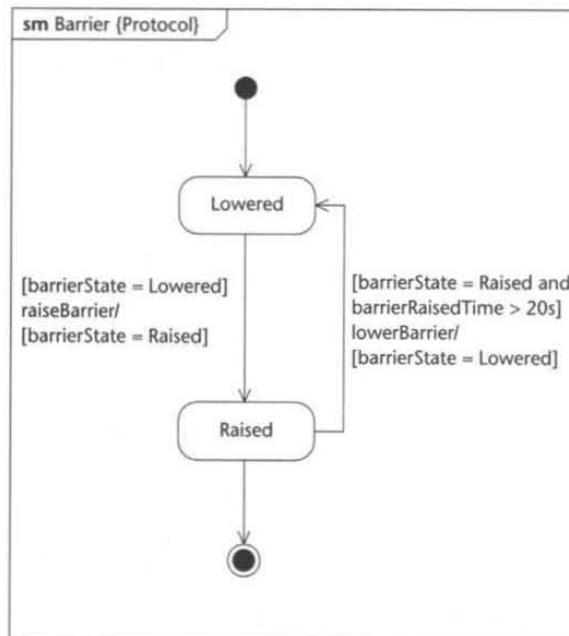


Figure 11.21 Protocol state machine for `Barrier`.

cannot have entry, exit or do activity sections, they cannot have deep or shallow history states and all transitions must be protocol transitions. The syntax for a protocol transition label is as follows:

`'[<pre-condition>]' trigger '/<*>'['post-condition']'`

Unlike behavioural transitions protocol transitions do not have activity expressions.

Figure 9.28 shows the sequence diagram for the use case `Car enters car park` and Fig. 9.29 shows the interaction timing diagram for the `:Barrier` and `:TicketMachine` lifelines. The corresponding protocol state machine for the class `Barrier` is shown in Fig. 11.21. This provides a description of all the transitions that are permitted for the `Barrier` class and the circumstances under which they are valid. For example, the barrier can be lowered only if the pre-condition

`[barrierState = Raised and barrierRaisedTime > 20s]`

is true. This states that before the barrier can be lowered it must be raised and that it has been raised for at least 20 seconds.

11.7 Consistency Checking

The need for consistency between different models was discussed in Chapter 9 in relation to interaction diagrams. State machines must also be consistent with other models.

- Every trigger should appear as an incoming message for the appropriate object on an interaction diagram.
- Every action should correspond to the execution of an operation on the appropriate class, and perhaps also to the dispatch of a message to another object.
- Every trigger should correspond to an operation on the appropriate class (but note that not all operations correspond to triggers).
- Every outgoing message sent from a state machine must correspond to an operation on another class.

Consistency checks are an important task in the preparation of a complete set of models. This process highlights omissions and errors, and encourages the clarification of any ambiguity or incompleteness in the requirements.

11.8 Quality Guidelines

Preparing state machines is an iterative process that involves refining the model until it captures the semantics of the object or model element behaviour. State machines are effective for communication among team members. Listed below are a series of general guidelines that aid the production of good quality state machines.

- Name each state uniquely to reflect what is happening for the duration of the state or what the state is waiting for.
- Do not use composite states unless the state behaviour is genuinely complex.

- Do not show too much complexity on a single state machine. If there are more than seven states consider using substates. Even with a small number of states, a state machine may be too complex if there are a large number of transitions between them. Arguably the state machine in Fig. 11.20 would be better represented on three diagrams, one for the high-level state machine with the detail of the Active state hidden and one diagram for each of the two submachines, Running and Monitoring.
- Use guard conditions carefully to ensure that the state machine describes possible behaviour unambiguously.

State machines are not flowcharts. Activity diagrams (see Chapters 6 and 10) are used to model procedural behaviour. Typical symptoms of state machines that are too much like flowcharts include the following.

- Most transitions are fired by state completion.
- Many messages are sent to 'self', reflecting code reuse rather than actions triggered by events.
- States do not capture state dependent behaviour associated with the class.

Of course, a model that was intended to be a state machine but turns out to be an activity diagram describing procedural flow may be a valuable model; it just is not a state machine.

11.9 Summary

The specification of the dynamic and behavioural aspects of an application is an important aspect of both analysis and design. They are described in part by interaction diagrams but these focus only on a use case or an operation. In order to capture fully the behavioural constraints for each class it is necessary to model the impact of events on that class and to model the resulting state changes with their attendant limitations on behaviour. It is only necessary to prepare state machines for classes that have state dependent variations in behaviour. UML's state machine notation permits the construction of detailed models that may include the nesting of states and the use of concurrent states to capture complex behaviour.

State machines must be checked for consistency with their associated class and interaction diagrams and this may highlight the need to make modifications to these other models.

The notations provided by UML are very detailed and should be used with some care. There is no advantage in producing a state machine that utilizes every UML feature unless this is really necessary for the application that is being modelled. Ideally, state models should be kept as simple as possible but should have sufficient detail to make them unambiguous and informative. The use of multiple nested states does not aid clarity unless the behaviour being described is itself complex.

Review Questions

- 11.1** Define event, state and transition.

- 11.2** What is the effect of a guard condition?
- 11.3** Why should all the guard conditions from a state be mutually exclusive?
- 11.4** What is a composite state?
- 11.5** What does it mean to say that an object can be in concurrent states?
- 11.6** How do nested states differ from concurrent states?
- 11.7** When are entry and exit pseudostates used?
- 11.8** What is the difference between shallow history and deep history pseudostates?
- 11.9** What UML modelling elements have their behaviour described by a state machine?
- 11.10** What is a trigger?
- 11.11** What are the indications that a state machine has not been drawn to model state changes?
- 11.12** Describe the difference between behavioural and protocol state machines.
- 11.13** Against which other UML diagrams should a state machine be cross-checked?
- 11.14** What cross-checks should be carried out?

Case Study Work, Exercises and Projects

- 11.A** Using the interaction sequence diagrams that you prepared for Exercises 9.A–9.C, list events that affect a `ProductionLine` object and identify appropriate states for this class.
- 11.B** Prepare a state machine for the class `ProductionLine`.
- 11.C** List any changes that may have to be made to the class diagram for the `FoodCo` case study in the light of preparing this state machine.

Further Reading

State machines have been used widely to model complex control behaviour. Various non-object-oriented approaches have used state machines very effectively. In particular, the texts by Ward and Mellor (1985, 1986) and Hatley and Pirbhai (1987) provide detailed descriptions of their application in real-time applications. From an object-oriented perspective both Rumbaugh et al. (1991) and Booch (1994) provide useful descriptions of the Harel (1987) notation used in UML. Object-oriented real-time development approaches are well discussed by Douglass (1999) and also by Selic et al. (1994). The latter text is based on the ROOM (Real-time Object-Oriented Modeling) approach. Cook and Daniels (1994) give an interesting alternative perspective on the modelling of events and states. Useful advice on preparing state machines can also be found in the IBM-Rational Unified Process (IBM, 2005). For a more recent view of state machines from Harel, see Harel and Politi (1998), which presents the STATEMATE approach.

Agate Ltd Case Study—Further Analysis

Agate Ltd

A4.1 Introduction

In this chapter we show how the Analysis Model presented in Chapter A3 has been refined in a further iteration. The refinement has been carried out with two particular aims in mind.

First we aim to improve our understanding of the domain and thereby increase the general usefulness of the model in a wider context. This essentially means identifying opportunities for reuse through the elaboration of generalization, composition and aggregation structures in the class model, as described in Chapter 8.

Second, we aim to improve the level of detail of the model and also the accuracy with which it reflects user requirements. This is addressed partly through appropriate allocation of behaviour to classes, derived from the analysis of class interaction using sequence diagrams and state machines. We also seek to specify the behavioural aspects of the model in more detail through the specification of operations. The related techniques are described in Chapters 9, 10 and 11.

As a result of these activities, the analysis class model is revised to reflect our greater understanding of the domain and of the requirements.

The following sections include:

- samples of the sequence diagrams and state machines that help us to understand the behavioural aspects of the model
- specifications for some operations that capture this behaviour and communicate it to the designers
- a revised analysis class diagram that shows the effects of further analysis on the static structure of the model.

Together, the class diagram and operation specifications comprise an analysis class model.

A4.2 Sequence Diagrams

The first sequence diagram, shown in Fig. A4.1, is for the use case Add a new advert to a campaign. The second sequence diagram, shown in Fig. A4.2, is for the use case Check campaign budget. Both these sequence diagrams are discussed in some detail in Chapter 9; note that for simplicity we show here the version of Add a new advert to a campaign that does not include boundary and control classes.

Sequence diagrams help the requirements analyst to identify at a detailed level the operations that are necessary to implement the functionality of a use case. It is worth mentioning that, although at this point we are still primarily engaged in analysis—in other words, an attempt to understand the demands that this information system will fulfil—there is already a significant element of design in our models. There is no one correct sequence diagram for a given use case. Instead, there are a variety of possible sequence diagrams, each of which is relatively more or less satisfactory in terms of how well it meets the needs of the use case. The sequence diagrams illustrated here are the product of experimentation, judgement and several iterations of modelling carried out by analysts and users together.

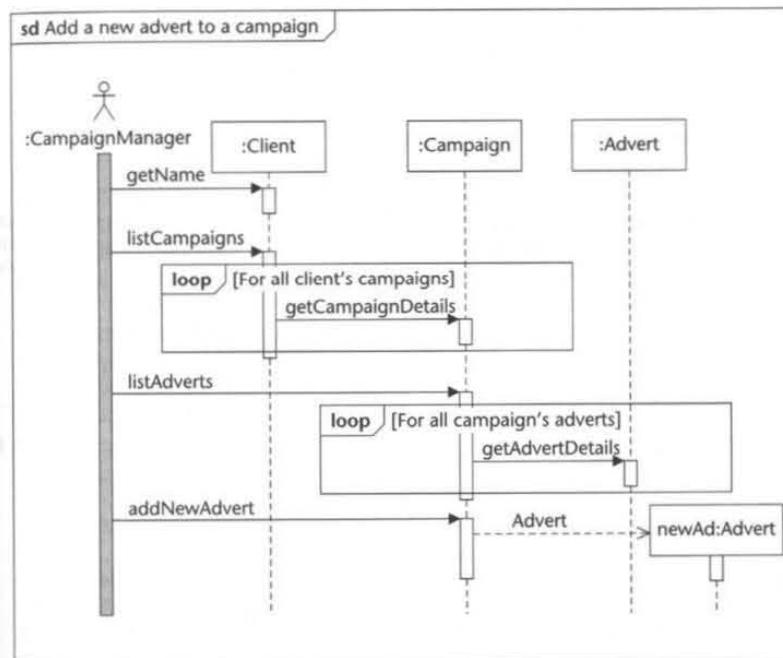


Figure A4.1 Sequence diagram for Add a new advert to a campaign.

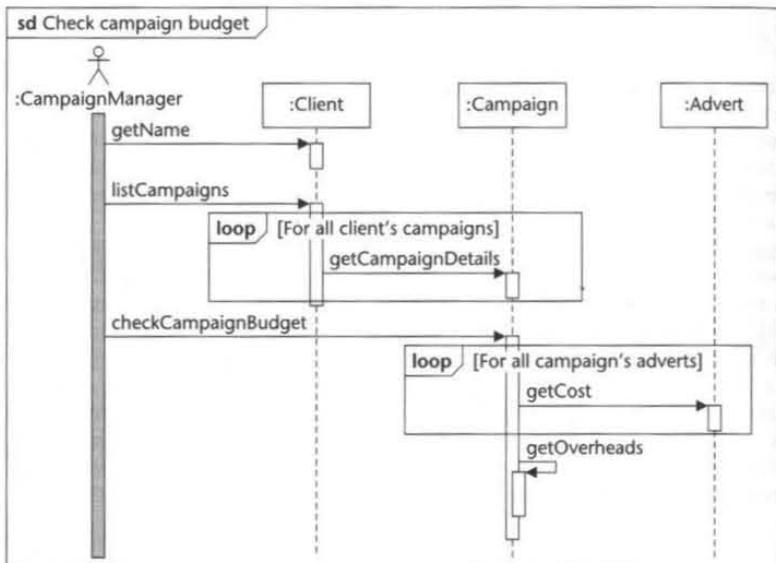


Figure A4.2 Sequence diagram for Check campaign budget.

A4.3 State Machines

In this section we present the final state machine for Campaign (Fig. A4.3), which has already been discussed at some length in Chapter 11, and an initial state machine for StaffMember (Fig. A4.4), which is presented here for the first time. These represent the behaviour of objects of significant classes in the Campaign Management and Staff Management analysis packages, respectively.

In conjunction with sequence diagrams, state machines help to identify the operations that are required and to allocate those operations to appropriate classes. All operations shown on sequence diagrams and state machines are added to the relevant class definitions. Each operation must also in due course be specified, and it is to this that we turn in the next section.

A4.4 Operation Specifications

The operation specifications given below define all operations identified for the sequence diagram Check campaign budget, which is shown above in Fig. A4.2.

Note that in all cases the logic of the operation is very simple; for some it consists of little more than returning the value of an attribute. Each operation, and, indeed, each object, has responsibility for only a small part of the processing required to realize the use case.

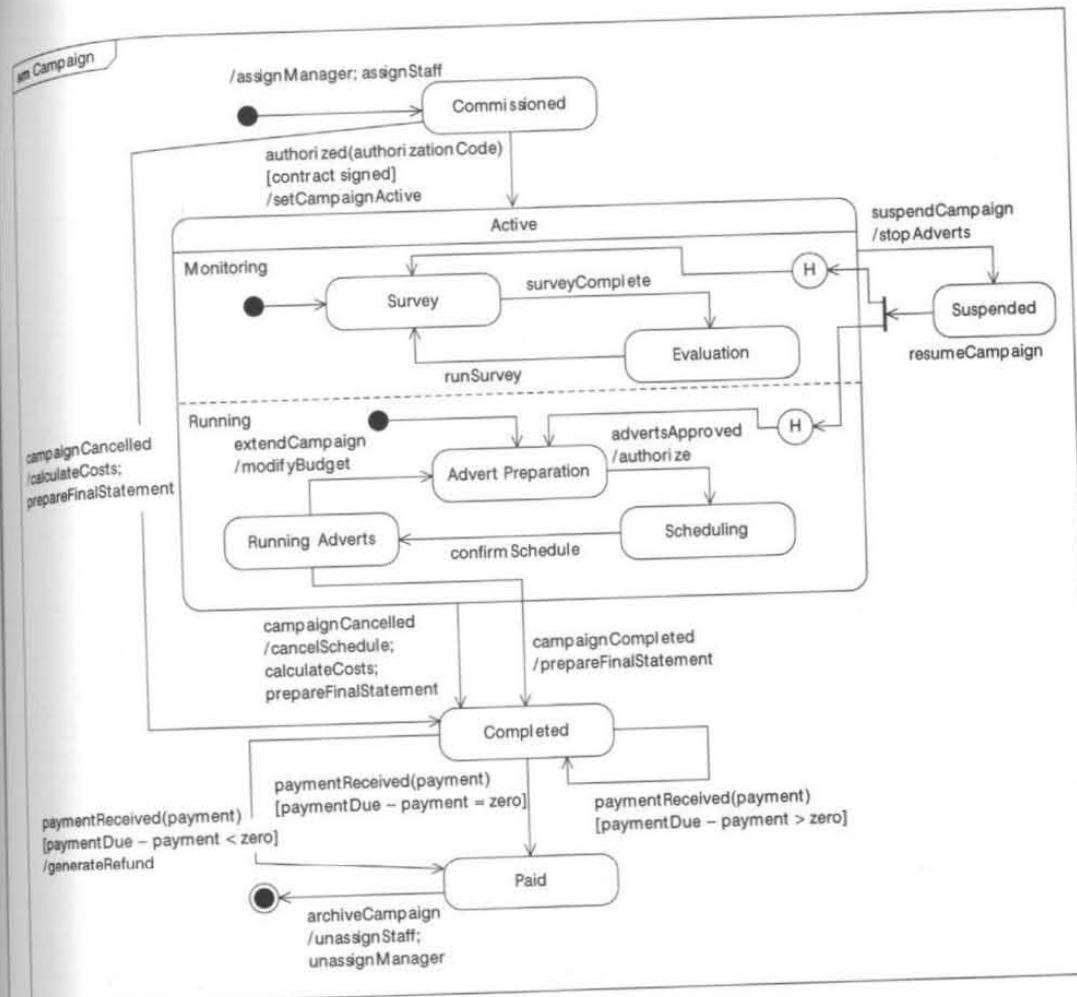


Figure A4.3 State machine for Campaign.

By reading the operation specifications in conjunction with the sequence diagram, it is easy to see how the Client, Campaign and Advert objects collaborate to realize this use case.

Note that this view of collaborating objects is simplified to some extent, in that it does not include control and boundary objects and their operations. However, operations in these objects are no more complex than those shown below, since their primary role is simply to call and co-ordinate operations on the entity objects.

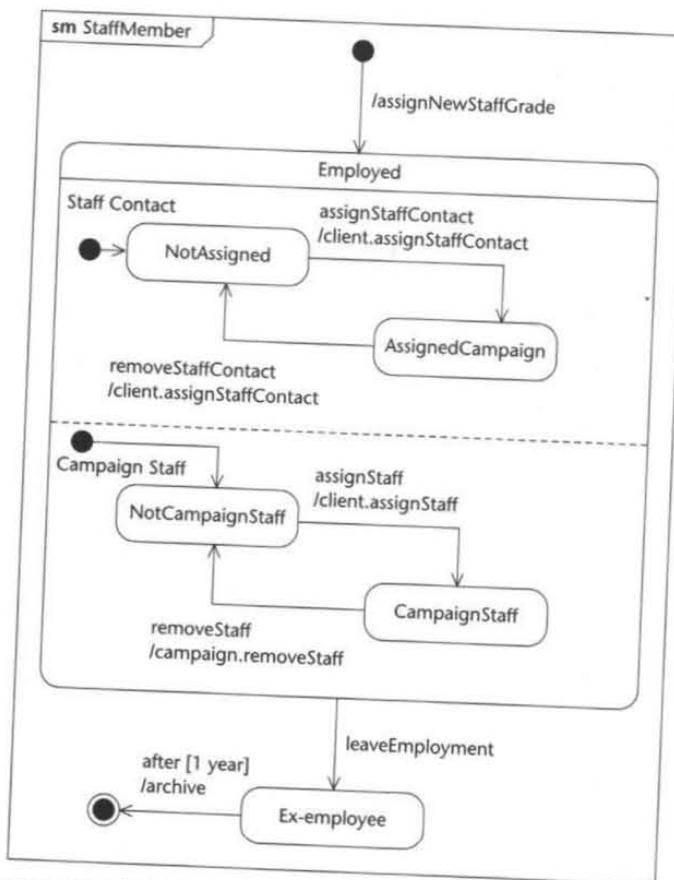


Figure A4.4 Initial state machine for StaffMember.

Context: Campaign

Operation specification: checkCampaignBudget()

Operation intent: return campaign budget and actual costs.

Operation signature: Campaign::checkCampaignBudget()

Logic description (pre- and post-conditions):
budgetCostDifference:Money

pre: self->exists()

post: result = self.originalBudget-self.estimatedCost **and**
self.estimatedCost =
self.adverts.estimatedCost->sum()

Other operations called: Advert.getCost(), self.getOverheads()
Events transmitted to other objects: none
Attributes set: none

Response to exceptions: none defined

Non-functional requirements: none defined

Operation specification: getCampaignDetails()

Operation intent: return the title and budget of a campaign.

Operation signature: Campaign::getCampaignDetails()

title:String, campaignBudget:Money

Logic description (pre- and post-conditions):

pre: self->exists()

post: result = self.title, self.estimatedCost

Other operations called: none

Events transmitted to other objects: none

Attributes set: none

Response to exceptions: none defined

Non-functional requirements: none defined

Operation specification: getOverheads()

Operation intent: calculate the total overhead cost for a campaign.

Operation signature: Campaign::getOverheads()

campaignOverheads:Money

Logic description (pre- and post-conditions):

pre: self->exists()

post: result = self.campaignOverheads

Other operations called: none

Events transmitted to other objects: none

Attributes set: none

Response to exceptions: none defined

Non-functional requirements: none defined

Context: Client

Operation specification: getName()

Operation intent: return the client name.

Operation signature: Client::getName() name:String

Logic description (pre- and post-conditions):

pre: self->exists

post: result = self.name

Other operations called: none

Events transmitted to other objects: none

Attributes set: none

Response to exceptions: none defined

Non-functional requirements: none defined

Operation specification: listCampaigns()

Operation intent: return a list of campaigns for a client.

Operation signature: Client::listCampaigns() titles:String[]

Logic description (pre- and post-conditions):

pre: self->exists

post: result = self.campaign->collect(campaign.title)

Other operations called: Campaign.getCampaignDetails

Events transmitted to other objects: none

Attributes set: none

Response to exceptions: none defined

Non-functional requirements: none defined

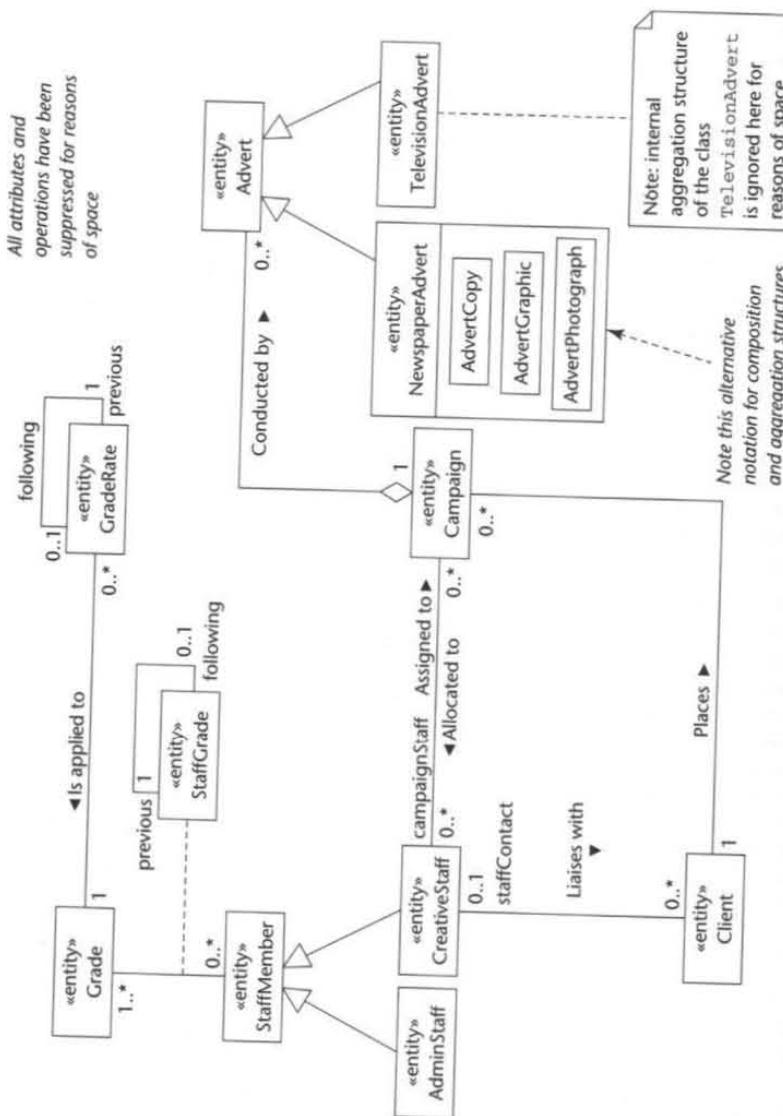


Figure A4.5 Revised analysis class diagram with generalization and aggregation structures.

Context: Advert

Operation specification: `getCost()`

Operation intent: return the actual cost for an advert.

Operation signature: `Advert::getCost()`

`actualAdvertCost:Money`

Logic description (pre- and post-conditions):

pre: self->exists()

post: result = self.actualAdvertCost

Other operations called: none

Events transmitted to other objects: none

Attributes set: none

Response to exceptions: none defined

Non-functional requirements: none defined

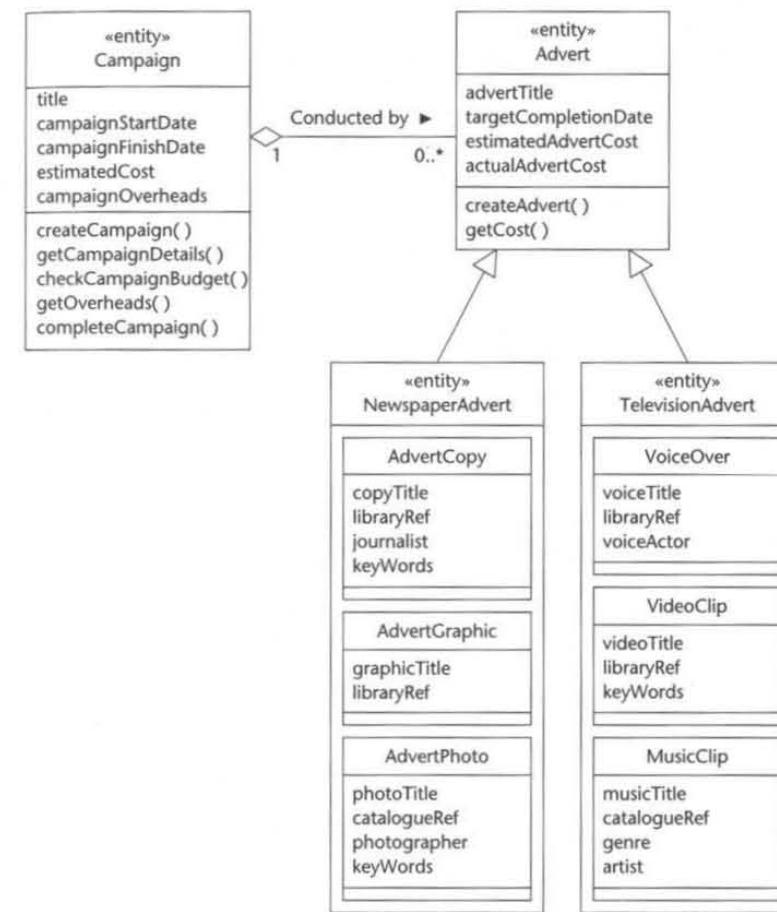


Figure A4.6 Generalization and aggregation structure for Advert.

A4.5 Further Refinement of the Class Diagram

Figure A4.5 shows the revised analysis class diagram, after inheritance and aggregation structures have been added. For reasons of space, all attributes and operations have been suppressed from this view.

Figure A4.6 shows an excerpt from the analysis class diagram, detailing the generalization and aggregation structure for *Advert* with attributes and operations visible. This partial diagram reflects a further iteration of investigation and requirements modelling, which revealed that there is a requirement to keep track of the various elements used to create an advertisement. This is because photographs, music clips and so on can often be used for more than one advertisement in a campaign, and it has been a problem to identify and retrieve these elements when they are needed.

A4.6 Further Activities of Requirements Analysis

Figure A4.7 shows an activity diagram that illustrates the relationship between the products of the analysis model before and after this iteration of analysis. Some details are worth highlighting.

- The analysis class model now includes some detailed class definition. In particular, all operations should be specified at least in outline.

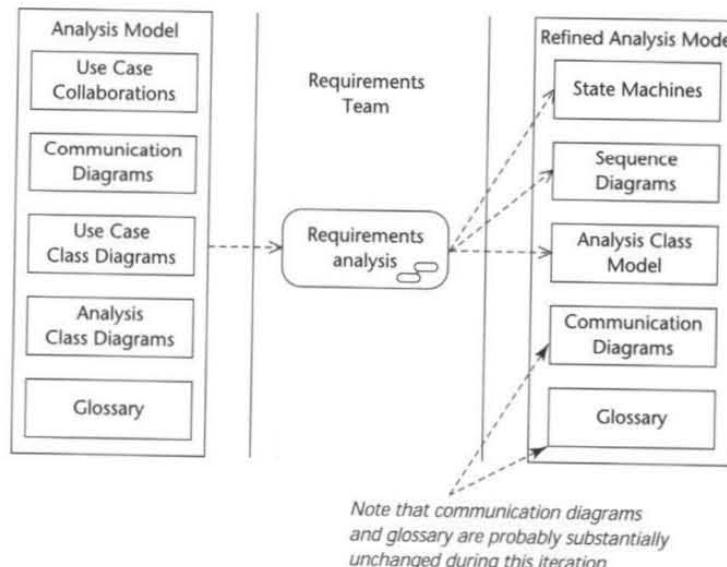
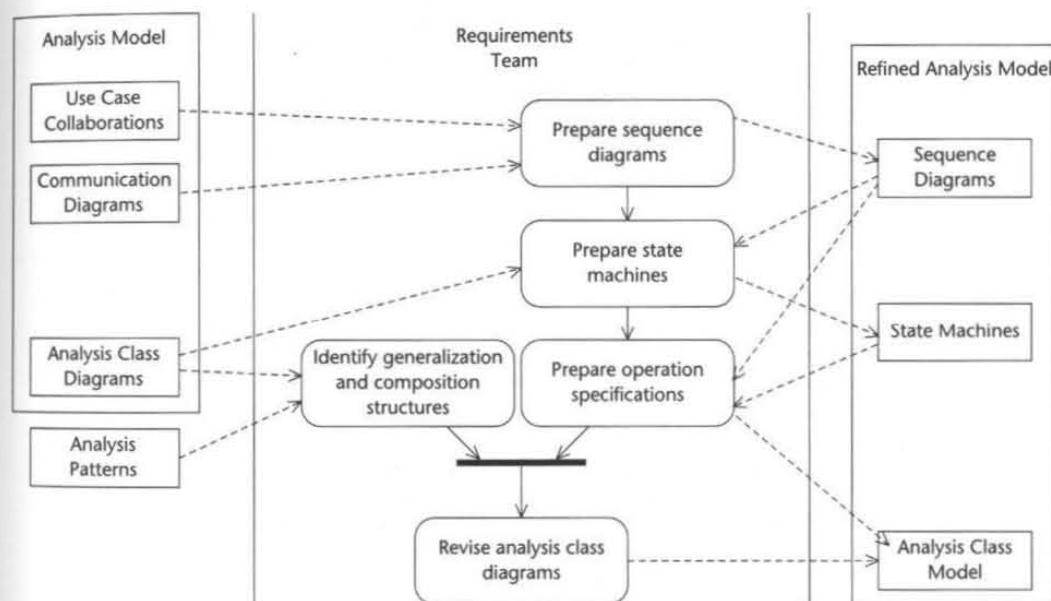


Figure A4.7 High-level activity diagram showing how elements of the analysis model are created or updated during this iteration of analysis.

- Some parts of the analysis model may be substantially unchanged during this iteration, for example the communication diagrams and the glossary. Although this is not necessarily the case, we have shown these as unaffected in Fig. A4.7.
- As a result of the operation specification activity, many attributes may also have been specified in more detail. Some, particularly those that are required to provide parameters to operations in other classes, will certainly now be typed. We have not shown this yet, since the typing of attributes is essentially a design activity. But in practice, some design decisions are made in parallel with the more detailed analysis that we describe in this chapter.

Figure A4.8 shows a more detailed view of the activities that are carried out and the products directly used or affected during this iteration. In this diagram, we have tried to suggest a sensible outline sequence for carrying out the various activities. However, it should be noted that this is no more than a guide, and is certainly not meant to be prescriptive. An iterative approach should always be followed that is sensitive to the needs of the project, to the skill of the developers and to the often haphazard manner in which understanding grows during the modelling and analysis of requirements.



For clarity, we have detailed only those activities and products that are most directly involved in this iteration. Note also that the flow of activities is indicative and is not intended to be prescriptive.

Figure A4.8 The activities that are carried out and the products directly used or affected during this iteration of analysis.

12

CHAPTER

System Architecture

Learning Objectives

In this chapter you will learn

- what is meant by architecture in information systems development
- the factors that influence the architecture of a system
- the range of architectural styles that can be used as the basis of an architecture.

12.1 Introduction

In Chapter 5 we described the Unified Software Development Process (USDP) (Jacobson et al., 1999) as 'architecture-centric' and at several points in previous chapters we have mentioned the idea of producing architectural models of the proposed information system. In Chapter 8 we introduced the idea of patterns and the use of architectural patterns such as the Model–View–Controller architecture that is embodied in the use of Boundary, Control and Entity classes that we have used in the class models that we have produced. However, up to this point we have not defined what we mean by architecture in the context of information systems; nor have we explained how to design the architecture of a system.

Systems architecture is a broad topic and one that is the subject of many books. In this chapter we aim to give an introduction to systems architecture and to explain why it is important. Every system has an architecture of some sort. If the designers and developers do not take the time or have the skills to produce explicit architectural models of the system, the system will still have an architecture. However, that architecture will be implicit and will be influenced by factors such as the choice of programming language, database and platform, and the skills and experience of the development team. Any such implicit architecture is likely to result in a system that does not meet the non-functional requirements and is difficult to maintain or enhance. Producing an explicit architecture means that

the architect has to consider the non-functional requirements, the context of the system and how it and its components may be used and further developed in the future.

In the rest of this chapter, we explain what is meant by systems architecture, what are the factors that influence the development of an architecture and the kind of issues that are addressed by an architecture.

12.2 What Do We Mean by Architecture?

The use of the term architecture in the development of information systems obviously derives from the practice of architecture in the built environment. The Royal Institute of British Architects (RIBA) describes 'What Architects Do' as follows.

Architects are trained to take your brief and can see the big picture—they look beyond your immediate requirements to design flexible buildings that will adapt with the changing needs of your business.

Architects solve problems creatively—when they are involved at the earliest planning stage, they gain more opportunities to understand your business, develop creative solutions, and propose ways to reduce costs.

If we replaced the word 'buildings' with 'information systems' many systems architects and software architects would happily sign up to this definition of what they do. There are certain key features in these two sentences that apply as much to systems architecture as to the architecture of buildings.

- Systems architects act on behalf of the client. Part of their role is to understand the client's business and how that business can best be supported by an information system. However, the client may make conflicting demands on the new information system, and part of the systems architect's role is to resolve those conflicts.
- Systems architecture addresses the big picture. The architecture of an information system is a high-level view of the system: it is modelled in terms of the major components and the way they are interconnected; it does not normally address the detailed design of the system, though it may set standards to be applied.
- If flexibility is important, then systems architects will produce an architecture that is intended to deliver this quality. In the current climate of rapid change in the business environment, flexibility is often cited as a reason for adopting certain types of systems architecture. However, there are other qualities of information systems that may be more important for a particular client, in which case those qualities will be addressed by the architecture.
- Systems architects are concerned with solving problems. In information systems development, problems manifest themselves in terms of risks to the success of the project. The reason that the Unified Process is architecture-centric is that by concentrating on the architecture and making architectural decisions early in the project lifecycle, the risks can be reduced or mitigated.
- Reducing costs is not a primary objective of systems architects. However, proposing unnecessarily expensive solutions never wins anyone any friends, and

producing an explicit architecture for a new system means that the specific needs of that system are addressed and unnecessary features eliminated. It also means that risks are tackled early in the project lifecycle and that the chance is minimized of discovering late in the project that the new system will not meet some requirement, with the need for costly design changes or reworking.

Of these, probably the most important is that architecture is about the big picture. Analysis is inevitably about detail: the business analyst needs to understand and document every requirement in a clear and unambiguous way; the systems analyst must consider use cases and other requirements and translate them into a complete model of the classes necessary to support those use cases, their attributes and responsibilities or operations and a first-cut view of how instances of those classes will interact. Design is about translating every aspect of the analysis model into a design model that will effectively implement the requirements: the designer must consider the type of every attribute and design each operation to take the necessary parameters, return the right value and be efficient in its working. Architecture, on the other hand, looks at the large-scale features of the system and how those features work together as a whole: the architect groups classes together into packages, models the system as a set of interacting components and considers what platforms to deploy those components on in order to deliver the required qualities of the system.

There are a number of different views of architecture in the development of information systems. Our focus here is on systems architecture and software architecture. In Section 12.4 we discuss enterprise architecture and technical architecture and their relationship with systems and software architectures.

In their book on large-scale software architecture, Garland and Anthony (2003) use the definition of architecture from the Institute of Electrical and Electronics Engineers (IEEE) standard IEEE 1471–2000 (IEEE, 2000). This provides the following definitions of key terms.

- **System** is a set of components that accomplishes a specific function or set of functions.
- **Architecture** is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution¹.
- **Architectural Description** is a set of products that document the architecture.
- **Architectural View** is a representation of a particular system or part of a system from a particular perspective.
- **Architectural Viewpoint** is a template that describes how to create and use an architectural view. A viewpoint includes a name, stakeholders, concerns addressed by the viewpoint, and the modelling and analytic conventions.

Given this definition of architecture, then **Software Architecture** is the organization of a system in terms of its software components, including subsystems and the relationships and interactions among them, and the principles that guide the design of that software system.

¹ From IEEE Standard 1471–2000, Copyright 2000 IEEE.

The IEEE definition is important because it stresses the fact that the same system can be shown in different views that emphasize different aspects of that system. Bass et al., (2003) point out that architecture is often defined as something like ‘the overall structure of the system’, but criticize this because it implies that a system has only a single structure. They suggest asking anyone who takes this position exactly which structure of the system the architecture represents.

Soni et al. (1995) identify four different aspects of software architecture, which are shown in Fig. 12.1.

In terms of object-oriented development, the conceptual architecture is concerned with the structure of the static class model and the connections between the components of the model. The module architecture describes the way the system is divided into subsystems or modules and how they communicate by exporting and importing data. The code architecture defines how the program code is organized into files and directories and grouped into libraries. The execution architecture focuses on the dynamic aspects of the system and the communication between components as tasks and operations execute.

The Rational Unified Process uses five views of the system, known as the ‘4 + 1 views’ (Kruchten, 2004). The four views are the *logical view*, the *implementation view*, the *process view* and the *deployment view*. The one view that ties them all together is the *use case view*. These five views are explained in Fig. 12.2.

These five views conform to the IEEE 1471 definition of what constitutes a view. They provide a description of the system from a particular perspective. The static structural relationships between classes and packages in the logical view present a different aspect of the system from the dynamic relationships between runtime processes in the process view. A single diagram or model cannot easily combine both these perspectives.

Different views are like different maps of a country. It is possible to find maps that show the physical topography—mountains, hills, rivers and lakes; maps that show the human geography—towns, cities and the road and rail networks; maps that show the land use—farming, woodland, industry and human settlements; and maps that show schematically the volume of transport flow between major conurbations. However, trying to combine all these views of the country in a single map would make it confusing and difficult to understand.

Maps conform to particular conventions for how they represent the geography of a country. For example, the physical topography is shown using contour lines, colour or shading, or some combination of these three, to represent the height of features and the location of water. Clearly, models that represent different views of

Type of architecture	Examples of elements	Examples of relationships
Conceptual	Components	Connectors
Module	Subsystems, modules	Exports, imports
Code	Files, directories, libraries	Includes, contains
Execution	Tasks, threads, object interactions	Uses, calls

Figure 12.1 Four aspects of software architecture according to Soni et al. (adapted from Weir and Daniels, 1998)

View	Explanation
Use case view	The important use cases in the system and scenarios that describe architecturally significant behaviour
Logical view	Important design classes and interfaces in a package structure, with composite structure diagrams
Implementation view	Architectural decisions made for the implementation in terms of subsystems and components and relationships among them
Process view	A description of the processes (operating system processes and threads) and inter-process communications using stereotyped classes
Deployment view	Physical nodes for the likely deployment platform, components deployed on the nodes and the communication channels between them, using deployment diagrams

Figure 12.2 The 4 + 1 views.

a system must adopt some conventions for the different features that are shown in the model. The use of conventions makes it possible for the systems architect to communicate with stakeholders about the system and to provide guidance to designers and developers. A set of conventions for drawing architectural models is known as an *architecture description language* (ADL). Bass et al. (2003) present their own ADL, which consists of elements of four types, representing features in systems architectures, software architectures, process models and reference models. However, we would recommend using UML as an ADL. UML 2.0 has specific features that have been added and adapted in order to make it more suitable for modelling architectures as well as producing analysis and design models. The UML 2.0 Request for Proposals (OMG, 2000), which solicited proposals for the changes that should be made to the specification, had as one of its specific objectives the following.

Enable the modeling of structural patterns, such as component-based development and the specification of run-time architectures.

This has resulted in the introduction of composite structure diagrams and changes to the component diagram notation.

12.3 Why Produce Architectural Models?

A software architect uses architectural models based on different views in order to reason about the proposed system and the way it will operate from different perspectives. In particular, this makes it possible to assess how well the system will deliver the non-functional requirements. Bass et al. (2003) do not like the term non-functional requirements. They argue that what they term *quality attributes* of a system, such as performance, security or fault tolerance, are intimately bound up with the behaviour of the system and the way that it responds to inputs. They believe that defining a set of non-functional requirements that are somehow separate from the functional behaviour of the system is dangerous, as it implies that the functionality of the system can be addressed first and then the non-functional

requirements can be tacked onto the system towards the end of the development process. We have used the term non-functional requirements because it is widely understood and because it focuses attention during requirements gathering on all those aspects of how well the system will deliver the functionality. However, we do not believe that this is a licence to ignore such requirements until the end of the development process.

Development processes based on the Unified Process are architecture-centric. This means that getting the architecture right is a priority, and this in turn means that from the start of the project the architects are trying to address the non-functional requirements of the system, because the architecture provides the framework for delivering these quality attributes of the system. Getting the architecture right early on is also about reducing the risks in the project. If one of the requirements of a new system is that it should be able to handle very large peak processing volumes (for example, in an online order processing system), then it is important to prove as soon as possible in the project that the architecture supports the achievement of these peak loads. If the early work addresses only the ability to process orders but does not ensure that the design can be scaled up to handle the peak loads, then there is always the risk that the fact that the system cannot handle the loads will not be discovered until late in the project, and that this will result in delays while the software is redesigned to cope with the peak volumes.

Using architectural models, the architect can assess the ability of the system architecture to deliver quality attributes such as high performance. The way that the different views in the 4 + 1 view of the system can contribute to assessing performance is shown in Fig. 12.3.

View	Contribution to assessing performance
Use case view	The use cases that require high performance can be identified and the scenarios used to walkthrough how the other views will affect the performance requirement.
Logical view	The logical view of classes will show whether techniques such as creating lightweight objects or value objects have been used to reduce the overheads associated with passing values around.
Implementation view	The more components or subsystems involved, the more likely there are to be communication overheads, so the implementation view should show a small number of components used in the process.
Process view	The process view can be used to assess how many running processes will exist, and whether there will be multiple instances of the same process so that the work can be shared out by a special process that handles load-balancing. The kind of interprocess communication that is used will affect how efficiently data can be passed between processes.
Deployment view	The deployment view will show where different components are deployed, and whether data has to travel from machine to machine, or whether all the processes needed to deliver a high-performance use case are located on the same machine.

Figure 12.3 The contribution of the 4 + 1 views to assessing performance.

It is important to realize that some of the features shown in Fig. 12.3 to increase performance will not contribute to the achievement of other quality attributes. For example, adding lightweight versions of classes will mean that for every business class there are two versions, and any change to the attributes of the business class means an associated change to the attributes of the lightweight version; this makes the code more complex to maintain. Similarly, reducing the number of components involved in a process may mean that functionality that does not naturally belong together is grouped into the same component or subsystem, and this reduces the flexibility of the system.

12.4 Influences on System Architecture

The systems architect developing the architecture for a new system does not operate in isolation. In any organization there will be existing systems that will constrain and influence the architecture. Many large organizations are now developing or have developed an *enterprise architecture*, which provides a framework for all system development. An enterprise architecture links the design of the business to the information systems that are needed to support that business. Either as part of an enterprise architecture or as a separate framework, many organizations have technology standards or a *technical reference architecture* that lays down a set of technologies, often including specific products that are acceptable, and defines how they should be used.

In the following subsections, we explain each of these influences in turn and the effect that they have on the architecture. In Section 12.5 we explain the range of *architectural styles* that are typically applied within the organization's information systems and that the architect can choose to adopt in developing new systems.

12.4.1 Existing systems

In many cases, the architecture of a new system will be designed to conform to the existing systems in the organization. This applies to the technical aspects such as choice of operating system, database and programming language, and to the way in which the components of the new system will be chosen, designed and interconnected. An organization that has adopted Java 2 Enterprise Edition (J2EE) or Microsoft .NET for its systems will expect new systems to be developed to fit in with this framework. Frameworks such as J2EE and .NET are well documented in books and web resources, but any business that adopts them is also likely to maintain a set of technology standards or a technical reference architecture that explains how to use the framework in the particular company.

Where there are existing systems, any new system may be able to take advantage of reuse of components in those systems. This is particularly the case when the new system and the old share the same architecture. In Chapter 8 we introduced the idea of reusable components, and we develop it further in Chapter 20. Organizations that plan for software reuse will typically use some kind of searchable repository in which they store reusable assets. The OMG, the body that manages the UML standard, also maintains the standard for the *Reusable Asset Specification* (RAS), which provides a set of guidelines about the structure, content and description of reusable software assets. Products such as LogicLibrary's Logidex

and Select Component Manager from Select Business Solutions provide tools to help manage collections of components.

Heritage systems

Sometimes the existing systems may not provide a pattern for development of new systems. The technologies that were used to develop them may be out of date and difficult to support. The systems may still be doing a good job of supporting the business, but a decision has been made to adopt new technologies. The term *heritage system* is sometimes used in preference to legacy system to describe a system that uses out-of-date technology but is still delivering a clear benefit to the business or is key to its operations. If a heritage system is not being replaced, the new system may need to access data from it via some kind of interface. *Enterprise Application Integration* (EAI) tools are software tools that connect to systems in order to integrate them with one another. If the heritage system uses a well-known technology, there is likely to be an *adapter* available that will connect it to the EAI tool and enable the EAI tool to extract data from the old system and make it available to the new or pass data into it in order to use its functionality.

Services

A technique for connecting to heritage systems that is growing in popularity is to wrap them in a layer of software that exposes the required functionality as *services*. *Web services* are the most recent technique applied to this problem, but the idea of a *Service-Oriented Architecture* (SOA) has been around for longer than the Web.

The wrapper acts as a service proxy, so that it looks the same as other services to the client systems that invoke operations on the service, as shown in Fig. 12.4.

There may be many kinds of interface to legacy systems. Sometimes they have been written to provide an Application Programming Interface (API), in which case it may be possible to use this, although there may be a limited choice of programming languages to use. Often they do not have an API, but may provide some other kind of interface: they listen on a TCP/IP socket for connections, or they check for files placed in a certain directory and treat the file as input. Sometimes the only way to access a legacy system is for the wrapper to pretend to be a terminal (*terminal emulation*) and connect to it and send text and terminal codes as though a user were typing the data in, and then to read the data that comes back and extract what is required from the mix of prompts, actual data values and control sequences. This is known as *screen-scraping*.

Reverse-engineering and model-driven architecture

Model-Driven Architecture (MDA) is one of the reasons for some of the changes that were made to UML to produce Version 2.0. The idea of MDA is to separate the business and application logic of a system from the underlying platform-technology. This abstract view of what the system must do is known as a platform-



Figure 12.4 Wrapping a legacy system as a service.

independent model (PIM). The PIM is then combined with a definition of the platform architecture in order to produce a platform-specific model (PSM) that can be built and executed on a particular platform. In order to produce a PIM, it is necessary to be able to specify actions that must be carried out within a system in a precise and verifiable way. Using this approach, it should be possible to build a platform-independent specification of a system and then, using different standard mappings, to transform it into a platform-specific model. The PSM is then further transformed to implementation code using automated tools that are already available for building software from models. A single PIM could thus be implemented in different ways: J2EE, .NET, CORBA.

The OMG also has an initiative to promote MDA. UML is central to the MDA initiative. Although earlier versions of UML provided ways of modelling structures (class diagrams), interaction (sequence diagrams) and lifecycles (state machine diagrams), the specification of actions in activity diagrams was not up to the task of precisely defining how classes should carry out operations. The developers of UML 2.0 have added a precise *action semantics* to the language. Combined with the notation of activity diagrams (which have also been defined more precisely in UML 2.0), this is intended to make UML the language of choice for producing PIMs.

As well as creating applications by transforming PIMs, the OMG also promotes the idea of reverse-engineering existing applications into PIMs. The idea is that if the business and application logic of a legacy system can be separated from the implementation details, and represented in an abstract specification language (UML with action semantics), then that PIM can then be used to re-implement the functionality of the system on a different, more modern platform. Products such as ArcStyler from InteractiveObjects not only provide a way of producing implementations from PIMs, but also of reverse-engineering existing application code into a PIM.

12.4.2 Enterprise architectures

In large, complex organizations, particularly those that operate in many countries and have different divisions of the business that address different markets, there is a risk that system development will be unco-ordinated. Indeed, there is a risk that nobody will have an overall understanding of the business, let alone the systems that support it. When a project for a new system is proposed, it is difficult to analyse the effect of that new system. Questions that might be asked include the following.

- How does the system overlap with other systems in the organization?
- How will the system need to interface with other systems?
- Will the system help the organization to achieve its goals?
- Is the cost of the system justified?

This last question about the cost of the system is an important factor for any organization, but is particularly significant when the system is being paid for from the public purse. In the United States, the *Clinger-Cohen Act* (CCA) of 1996 (formerly known as the IT Management Reform Act) was enacted to require public

bodies to manage their information systems investment in a way that protected the interests of the taxpayers. The US Congress had been concerned about the effectiveness of investment in IT systems after a number of spectacular failures; the CCA was designed to prevent further failures.

The resulting pressure on public bodies, and particularly federal government departments, to conform to the requirements of the CCA resulted in the development of a number of enterprise architecture frameworks in the USA. The best known are the Federal Enterprise Architecture Framework (FEAF) and the Treasury Enterprise Architecture Framework (TEAF). There are no equivalent regulations that apply across the European Union, but a number of individual countries have developed frameworks for enterprise architecture, for example the German Standards and Architectures for eGovernment Applications (SAGA, 2003) and the United Kingdom's e-Government Interoperability Framework (e-GIF, 2004).

The Sarbanes-Oxley Act (SOX) was enacted in the USA following the collapse of Enron and a number of other large private-sector company failures. It imposes standards for financial recording, reporting and audit on US companies and overseas subsidiaries. Businesses have to be able to show that they are compliant with the requirements of SOX.

These legislative changes in the USA have sparked a worldwide concern with corporate governance. By governance we mean the decision-making processes, responsibilities and structures within an organization. In a world where information technology is pervasive in the financial accounting, control and management of both public and private bodies, the concept of governance extends into the way in which information systems are defined, developed, procured and managed.

Enterprise architectures provide a way of modelling the enterprise and aspects of the way it conducts business and of driving these concepts down into the practical issues of how the information systems are intended to support the business. Outside the world of US federal government, there is really only one significant approach to developing an enterprise architecture, and this is the Zachman framework (Zachman, 1987), developed originally by John Zachman, and extended in collaboration with John Sowa (Sowa and Zachman, 1992).

The Zachman framework seeks to build explicit models of the enterprise from two views. The first asks the questions What? How? Where? Who? When? and Why? The second looks at the system at different levels, from the most conceptual, business view down to the view of the actual implemented system. The two dimensions are usually viewed as a matrix and the values that fill the thirty-six cells in the matrix are the actual models of aspects of the enterprise at different levels and from different perspectives.

The task of the *enterprise architect* is to build up a total picture of the enterprise using these categories. This total picture of the enterprise and its systems supports the process of ensuring that any IT investment is aligned to the goals of the business. Clearly this is a daunting task for a large organization, and one of the criticisms of the Zachman framework is that it is a heavyweight approach to enterprise architecture.

In an organization that has adopted any kind of enterprise architecture framework, that framework should be the starting point for identifying constraints on the architecture of new systems.

12.4.3 Technical reference architectures

Whereas enterprise architectures address the entire organization and its systems, *technical reference architectures* focus on the technology that is used within the enterprise, the standards for the technologies to apply and guidance on how to apply that technology. This may be in terms of a standards document, or a list of approved technologies or architectural models that show how different technologies should be applied in a typical system.

For organizations that do not have the time or resources to develop their own framework for technology standards, The Open Group produced The Open Group Architecture Framework (TOGAF) in 1995 (The Open Group, 2002). The current version is 8.1.

TOGAF consists of three main parts.

- The Architecture Development Method describes an approach for developing enterprise IT architectures.
- The Enterprise Continuum shows the continuum of architectures:
 - Foundation Architectures, which consist of abstract building blocks that support all kinds of technical architectures
 - Common Systems Architectures, which provide models for typical domains based on the Foundation Architectures
 - Industry Architectures, which show how Common Systems Architectures are typically implemented in different types of industry
 - Organization Architectures, which address the specific architectural needs of particular organizations.
- The Resources section provides a range of useful information and examples of architectural patterns, principles and other guidance.

The Open Group also maintains an online Standards Information Base (SIB) that lists hundreds of IT standards categorized according to the building blocks in the Foundation Architectures model.

12.5 Architectural Styles

Architects designing buildings do not start from scratch every time they are given a new commission. They design buildings that are similar to others that they or other architects have built previously, and they learn what works and what does not. Systems architects are very similar: they design systems that conform to the prevailing standards, and fashions in systems architecture come and go, like flying buttresses on Gothic churches or lifts on the outside of buildings.

In systems architecture, the term *architectural styles* is used to apply to these ways of designing systems that conform to the prevailing fashion. Often these fashions are the result of changes in technology: until the advent of the PC, it would not have been possible to implement client-server system architectures using PCs connected to mini-computers. Architectural styles also apply to software architecture. Bass et al. (2003) describe five main types: independent components, data flow, data centred, virtual machine, and call and return, each with subtypes. Each style has characteristics that make it more or less suitable for certain types of application. We will consider some of the major alternatives. It is worth noting that

software architectures have been documented in the patterns form by Buschmann et al. (1996) and Schmidt et al. (2000) amongst others.

12.5.1 Subsystems

A subsystem typically groups together elements of the system that share some common properties. An object-oriented subsystem encapsulates a coherent set of responsibilities in order to ensure that it has integrity and can be maintained. For example, the elements of one subsystem might all deal with the human-computer interface, the elements of another might all deal with data management and the elements of a third may all focus on a particular functional requirement.

The subdivision of an information system into subsystems has the following advantages.

- It produces smaller units of development.
- It helps to maximize reuse at the component level.
- It helps the developers to cope with complexity.
- It improves maintainability.
- It aids portability.

Each subsystem should have a clearly specified boundary and fully defined interfaces with other subsystems. A specification for the interface of a subsystem defines the precise nature of the subsystem's interaction with the rest of the system but does not describe its internal structure (this is a high-level use of contracts, which are described in Chapter 10). A subsystem can be designed and constructed independently of other subsystems, simplifying the development process. Subsystems may correspond to increments of development that can be delivered individually as part of an incremental lifecycle (if the developers are using the spiral lifecycle model or an iterative and incremental approach such as the Unified Process).

Dividing a system into subsystems is an effective strategy for handling complexity. Sometimes it is only feasible to model a large complex system piece by piece, with the subdivision forced on the developers by the nature of the application. Splitting a system into subsystems can also aid reuse, as each subsystem may correspond to a component that is suitable for reuse in other applications. A judicious choice of subsystems during design can reduce the impact on the overall system of a change to its requirements. For example, consider an information system that contains a presentation subsystem that deals with the human-computer interface (HCI). A change to the data display format need not affect other subsystems. Of course there may still be some changes to the requirements that affect more than one subsystem. The aim is to localize the consequences of change, so that a change in one subsystem does not trigger changes in other subsystems (sometimes referred to as the ripple effect). Moving an application from one implementation platform to another can be much easier if the software architecture is appropriate. An example of this would be the conversion of a Windows application so that it could run in a Unix environment. This would require changes to the software that implements the human-computer interface. If this is dealt with by specialized subsystems then the overall software change is localized to these subsystems. As a result, the system as a whole is easier to port to a different operating environment.

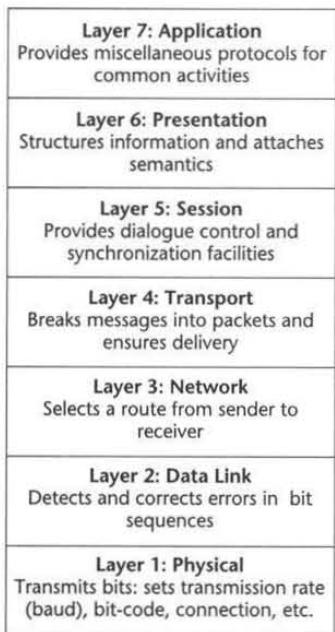


Figure 12.7 OSI Seven Layer Model (adapted from Buschmann et al., 1996).

- the further subdivision of complex layers
- performance reductions due to a closed layered architecture.

The OSI model has seven layers only because it covers every aspect of the communication between two applications, ranging from application-oriented processes to drivers and protocols that directly control network hardware devices. Many layered architectures are much simpler than this. Figure 12.8 shows a simple example of a three layer architecture.

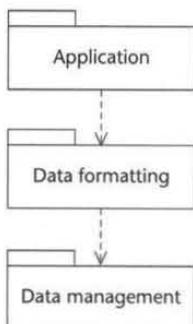


Figure 12.8 Simple layered architecture.

The lowest layer of the architecture in Fig. 12.8 consists of data management library classes. The layer immediately above this, the data formatting layer, uses services that are provided by the data management library classes in order to get data from a database management system. This data is formatted before it is passed upwards to the application layer. Supposing this system were to be modified to allow it to use a different database management system, the layered architecture limits major changes to the data management library class layer with some possible changes to the data formatting layer.

The following steps are adapted from Buschmann et al. (1996) and provide an outline process for the development of a layered architecture for an application. Note that this does not suggest that the specification of a system's architecture is a rule-based procedure. The steps offer guidelines on the issues that need to be addressed during the development of a layered architecture.

1. Define the criteria by which the application will be grouped into layers. A commonly used criterion is level of abstraction from the hardware. The lowest layer provides primitive services for direct access to the hardware while the layers above provide more complex services that are based upon these primitives. Higher layers in the architecture carry out tasks that are more complex and correspond to concepts that occur in the application domain.
2. Determine the number of layers. Too many layers will introduce unnecessary overheads while too few will result in a poor structure.
3. Name the layers and assign functionality to them. The top layer should be concerned with the main system functions as perceived by the user. The layers below should provide services and infrastructure that enable the delivery of the functional requirements.
4. Specify the services for each layer. In general it is better in the lower layers to have a small number of low-level services that are used by a larger number of services in higher layers.
5. Refine the layering by iterating through steps 1 to 4.
6. Specify interfaces for each layer.
7. Specify the structure of each layer. This may involve partitioning within the layer.
8. Specify the communication between adjacent layers (this assumes that a closed layer architecture is intended).
9. Reduce the coupling between adjacent layers. This effectively means that each layer should be strongly encapsulated. Where a client–server communication protocol will be used, each layer should have knowledge only of the layer immediately below it.

One of the simplest application architectures has only two layers—the application layer and a database layer. Tight coupling between the user interface and the data representation would make it more difficult to modify either independently, so a middle layer is often introduced in order to separate the conceptual structure of the problem domain. This gives the architecture shown in Fig. 12.9, which is commonly used for business-oriented information systems.

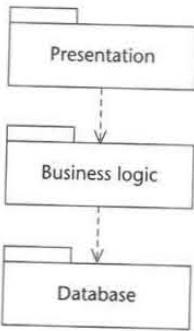


Figure 12.9 Three layer architecture.

A common four layer architecture separates the business logic layer into application logic and domain layers, and this is illustrated in Fig. 12.10. The approach that has been adopted during the analysis activity of use case realization results in the identification of boundary, control and entity classes. It is easy to see that it is possible to map the boundary classes onto a presentation layer, the control classes onto an application logic layer and the entity classes on a domain layer. Thus from an early stage in the development of an information system some element of layering is being introduced into the software architecture. However, it is important to appreciate that as we move through design, the allocation of responsibility amongst these types of class may be adjusted to accommodate non-functional requirements.

Separation of the application logic layer from the domain layer may be further justified because several applications share (or are likely to share) one domain layer, or because the complexity of the business objects forces a separation into two layers. It can also be used when the objects are physically distributed (see

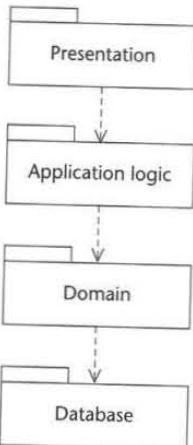


Figure 12.10 Four layer architecture.

Chapter 19). However, it must be emphasized that there is no perfect solution to this kind of design problem. There are only solutions that have different characteristics (perhaps different levels of efficiency or maintainability). A good design solution is one that balances competing requirements effectively.

Layered architectures are used quite widely. J2EE (Sun Java Centre, 2005) adopts a multi-tiered³ approach and an associated patterns catalogue has been developed. The architecture has five layers (client, presentation, business, integration and resource tiers) and the patterns catalogue addresses the presentation, business and integration tiers.

Partitioned subsystems

As suggested earlier, some layers within a layered architecture may have to be decomposed because of their intrinsic complexity. Figure 12.11 shows a four layer architecture for part of Agate's campaign management system that also has some partitioning in the upper layers.

In this example the application layer corresponds to the analysis class model for a single application, and is partitioned into a series of subsystems. These subsystems are loosely coupled and each should deliver a single service or coherent group of services. The Campaign Database layer provides access to a database that contains all the details of the campaigns, their adverts and the campaign teams. The Campaign Domain layer uses the lower layer to retrieve and store data in the database and provides common domain functionality for the layers above. For example, the Advert subsystem might support individual advert costing while the Campaign Costs subsystem uses some of the same common domain functionality when costing a complete campaign. Each application subsystem has its own presentation layer to cater for the differing interface needs of different user roles⁴.

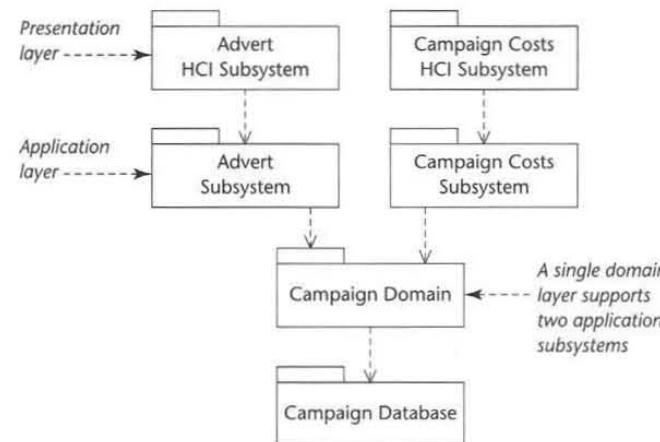


Figure 12.11 Four layer architecture applied to part of the Agate campaign management system.

³ These use the term tier as broadly equivalent to layer.

⁴ This example is for illustrative purposes only. Our analysis class model for Agate is too small to justify this kind of partitioning in practice.

A system may be split into subsystems during analysis because of the system's size and complexity. However, the analysis subsystems should be reviewed during design for coherence and compatibility with the overall system architecture.

The subsystems that result from partitioning should have clearly defined boundaries and well specified interfaces, thus providing high levels of encapsulation so that the implementation of an individual subsystem may be varied without causing dependent changes in the other subsystems. The process of identifying subsystems within a particular layer can be detailed in much the same way as for subsystem layers.

12.5.3 Model–View–Controller

Many interactive systems use the Model–View–Controller (MVC) architecture. This structure was first used with Smalltalk but has since become widely used with many other object-oriented development environments. The MVC architecture separates an application into three major types of component: models that comprise the main functionality, views that present the user interface and controllers that manage the updates to views. This structure is capable of supporting user requirements that are presented through differing interface styles, and it aids maintainability and portability.

It is common for the view of an information system that is required for each user to differ according to their role. This means that the data and functionality available to any user should be tailored to his or her needs. The needs of different types of user can also change at varying rates. For both these reasons it makes sense to give each user access to only the relevant part of the functionality of the system as a whole. For example, in the Agate case study many users need access to information about campaigns, but their perspectives vary. The campaign manager needs to know about the current progress of a campaign. She is concerned with the current state of each advertisement and how this impacts on the campaign as a whole—is it prepared and ready to run, or is it still in the preparation stage? If an advert is behind schedule, does this affect other aspects of the campaign? The graphic designer also needs access to adverts but he is likely to need access to the contents of the advert (its components and any notes that have been attached to it) as well as some scheduling information. A director may wish to know about the state of all live campaigns and their projected income over the next six months. This gives at least three different perspectives on campaigns and adverts, each of which might use different styles of display. The director may require charts and graphs that summarize the current position at quite a high level. The campaign manager may require lower level summaries that are both textual and graphical in form. The graphic designer may require detailed textual displays of notes with a capability to display graphical images of an advert's content. Ideally, if any information about a campaign or an advert is updated in one view then the changes should also be immediately reflected in all other views. Figure 12.12 shows a possible architecture, but some problems remain.

The design of such varied and flexible user interfaces that still incorporate the same core functionality is likely to be expensive because elements of functionality may have been duplicated for different interfaces. This makes the software more complex and thus also more error prone. There is an impact on maintainability too, since any change to core functionality will necessitate changes to each interface subsystem.

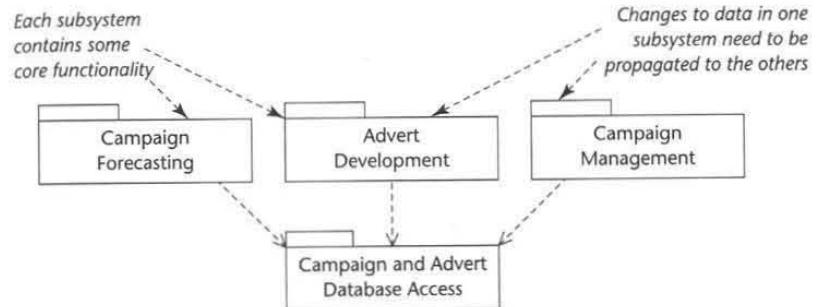


Figure 12.12 Multiple interfaces for the same core functionality.

We repeat below some of the difficulties that need to be resolved for this type of application.

- The same information should be capable of presentation in different formats in different windows.
- Changes made within one view should be reflected immediately in the other views.
- Changes in the user interface should be easy to make.
- Core functionality should be independent of the interface to enable multiple interface styles to co-exist.

While the four layer architecture in Fig. 12.11 resolves some of these problems it does not handle the need to ensure that all view components are kept up to date. The MVC architecture solves this through its separation of core functionality (model) from the interface and through its incorporation of a mechanism for propagating updates to other views. The interface itself is split into two elements: the output presentation (view) and the input controller (controller).

Figure 12.13 shows the basic structure of the MVC architecture.

The responsibilities of the components of an MVC architecture are listed below.

- **Model.** The model provides the central functionality of the application and is aware of each of its dependent view and controller components.
- **View.** Each view corresponds to a particular style and format of presentation of information to the user. The view retrieves data from the model and updates its presentations when data has been changed in one of the other views. The view creates its associated controller.
- **Controller.** The controller accepts user input in the form of events that trigger the execution of operations within the model. These may cause changes to the information and in turn trigger updates in all the views ensuring that they are all up to date.
- **Propagation Mechanism.** This enables the model to inform each view that the model data has changed and as a result the view must update itself. It is also often called the dependency mechanism.

Figure 12.14 represents the capabilities offered by the different MVC components as they might be applied to part of the campaign management system at Agate.

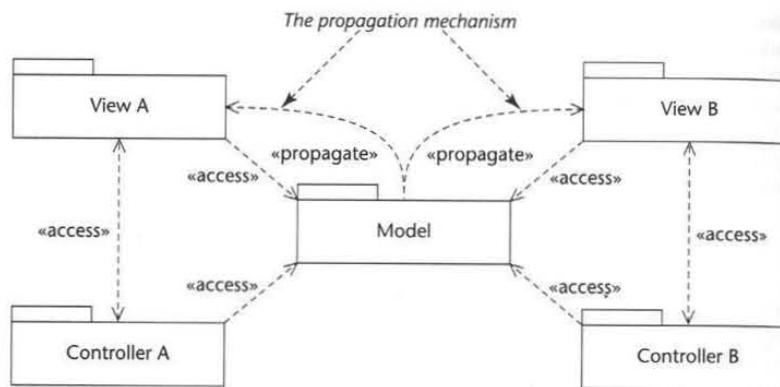


Figure 12.13 General structure of Model–View–Controller (adapted from Hopkins and Horan, 1995).

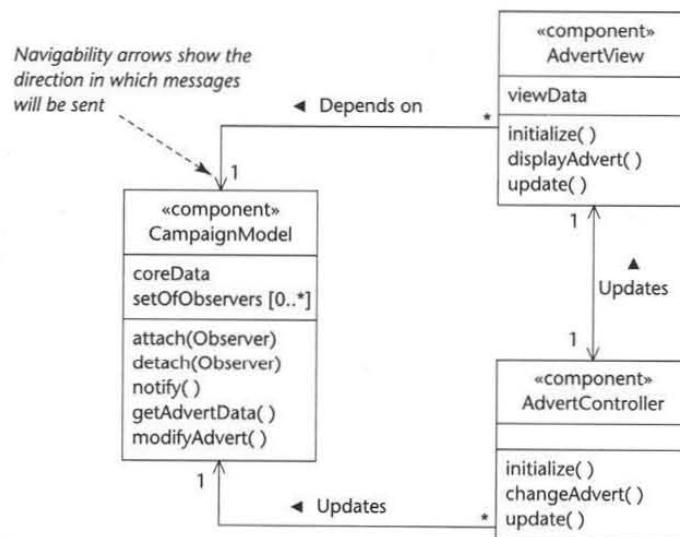


Figure 12.14 Responsibilities of MVC components, as applied to Agate.

The operation `update()` in the `AdvertView` and `AdvertController` components triggers these components to request data from the `CampaignModel` component⁵. This model component has no knowledge of the way that each view and controller component will use its services. It need only know that all view and controller components must be informed whenever there is a change of state (a modification either of object attributes or of their links).

⁵ In this example the `CampaignModel` will hold details of campaigns and their adverts.

The `attach()` and `detach()` services in the `CampaignModel` component enable views and controllers to be added to the `setOfObservers`. This contains a list of all components that must be informed of any change to the model core data. In practice there would be separate views, each with its own controller, to support the requirements of the campaign manager and the director.

The interaction sequence diagram in Fig. 12.15 illustrates the communication that is involved in the operation of an MVC architecture. (The choice of message type—synchronous or asynchronous—shown in this diagram is only one of the possibilities that could be appropriate, the features of the implementation environment would influence the actual design decision.) An `AdvertController` component receives the interface event `changeAdvert`. In response to this event the controller invokes the `modifyAdvert` operation in the `CampaignModel` object. The execution of this operation causes a change to the model.

For example, the target completion date for an advertisement is altered. This change of state must now be propagated to all controllers and views that are currently registered with the model as active. To do this the `modifyAdvert` operation invokes the `notify` operation in the model, which sends an `update` message to the view. The view responds to the `update` message by executing the `displayAdvert` operation which requests the appropriate data from the model via the `getAdvertData` operation. The model also sends an `update` message to the `AdvertController`, which then requests the data it needs from the model.

One of the most important aspects of the MVC architecture is that each model knows only which views and controllers are registered with it, but not what they do. The `notify` operation causes an update message to all the views and

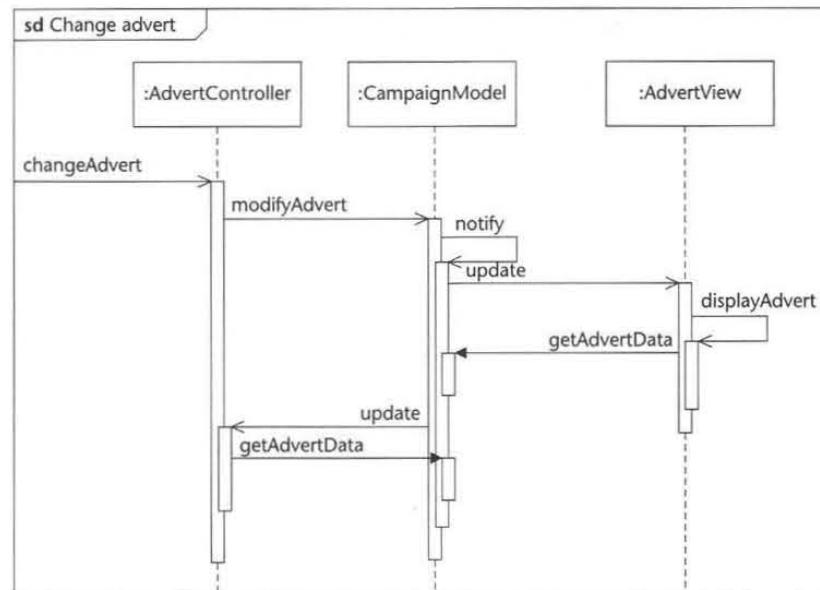


Figure 12.15 MVC component interaction.

controllers (for clarity, only one view and one controller are shown in the diagram, but interaction with the others would be similar). The update message from the model is in effect saying to the views and controllers 'I have been updated and you must now ensure that your data is consistent'. Thus the model, which should be the most stable part of the application, is unaffected by changes in the presentation requirements of any view or controller. The change propagation mechanism can be structured so that further views and controllers can be added without causing a change to the model. Each of these may support different interface requirements but require the same model functionality. However, since views and controllers need to know how to access the model in order to get the information they require, some changes in the model will inevitably still cause changes in other components.

Other kinds of communication may take place between the MVC components during the operation of the application. The controller may receive events from the interface that require a change in the way that some data is presented to the user but do not cause a change of state. The controller's response to such an event would be to send an appropriate message to the view. There would be no need for any communication with the model.

12.5.4 Architectures for distributed systems

Distributed information systems have become more common as communications technology has improved and have also become more reliable. An information system may be distributed over computers at the same location or at different locations. Since Agate has offices around the world, it may need information systems that use data that is distributed among different locations. If Agate grows, it may also open new offices and require new features from its information systems. An architecture that is suitable for distributed information systems needs also to be flexible so that it can cope with change. A distributed information system may be supported by software products such as distributed database management systems or object request brokers or may adopt a service-oriented architecture (these are discussed in Chapter 19).

A general *broker* architecture for distributed systems is described by Buschmann et al. (1996). A simplified version of the broker architecture is shown in Fig. 12.16.

A broker component increases the flexibility of the system by decoupling the client and server components. Each client sends its requests to the broker rather

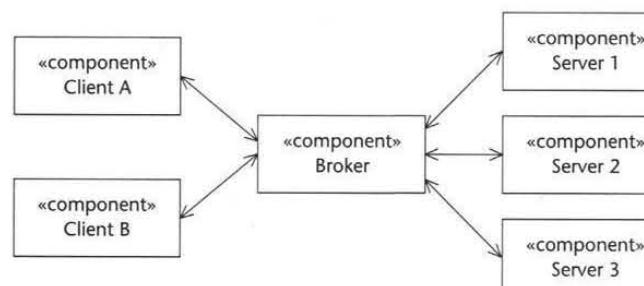


Figure 12.16 Schematic of simplified broker architecture.

than communicating directly with the server component. The broker then forwards the service request to an appropriate server. A broker may offer the services of many servers and part of its task is to identify the relevant server to which a service request should be forwarded. The advantage offered by a broker architecture is that a client need not know where the service is located, and it may therefore be deployed on either a local or a remote computer. Only the broker needs to know the location of the servers that it handles.

Figure 12.17 shows a sequence diagram for client–server communication using the broker architecture. The diagram is drawn with asynchronous message types but the actual implementation may involve both synchronous and asynchronous message types. In this example the server subsystem is on a local computer. In addition to the broker itself, two additional *proxy* components have been introduced to insulate the client and server from direct access with the broker. On the client side a *ClientSideProxy* receives the initial request from the client and packs the data in a format suitable for transmission. The request is then forwarded to the *Broker* which finds an appropriate server and invokes the required service via the *ServerSideProxy*.

The *ServerSideProxy* then unpacks the data and issues the service request sending the service message to the *Server* object. The *service* operation then executes and on completion responds to the *ServerSideProxy*. The response is then sent to the *Broker* which forwards it to the originating *ClientSideProxy*.

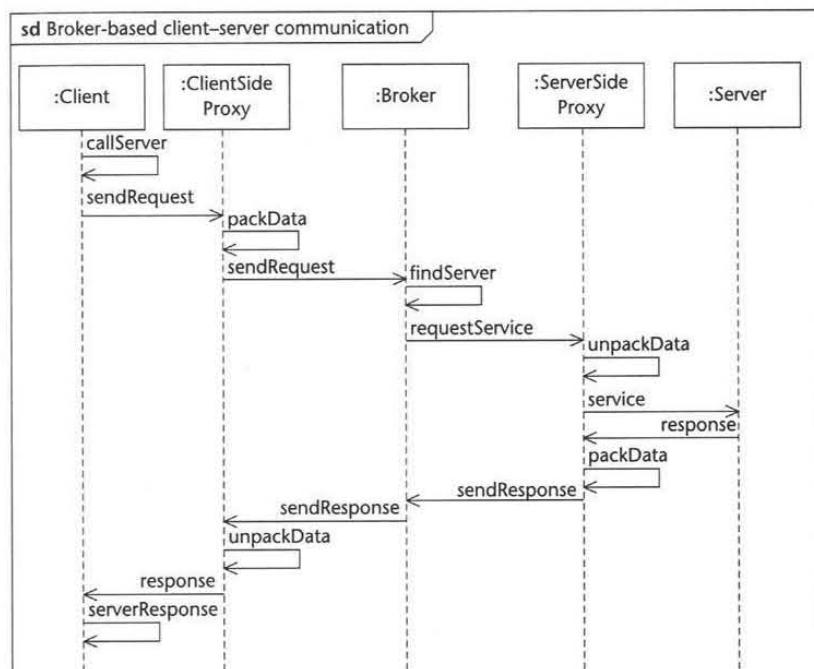


Figure 12.17 Broker architecture for local server (adapted from Buschmann et al., 1996).

Note that these are both new messages and not returns. The reason for this is that a broker does not wait for each response before handling another request. Once its `sendRequest` activation has been completed, the broker will in all probability deal with many other requests and thus requires a new message from the `ServerSideProxy` object to cause it to enter a new activation. Unlike the broker, the `ClientSideProxy` has remained active; this then unpacks the message and the response becomes available to the Client as control returns.

Figure 12.18 shows how the participants in this interaction can be allocated to different processes, with the client and its proxy running in one process thread, the broker in another and the server and its proxy in a third.

Figure 12.19 shows a schematic broker architecture that uses *bridge* components to communicate between two remote processors. Each bridge converts service requests into a network specific protocol so that the message can be transmitted. Figure 12.20 shows a possible allocation of these components to processes.

12.5.5 Organization structures for architecture and development

Dividing a system into subsystems has benefits for project management. Each subsystem can be allocated to a single development team, which can operate independently of other teams, provided that they adhere to the interface requirements for their subsystem. Where a subsystem must be split between two development teams, there is a heavy communications overhead that is incurred in ensuring that the different parts of the subsystem are constructed to consistent standards. In such cases the structure of either the organization or of the software tends to change so that they become more closely aligned with each other; this helps to minimize the communications overhead and is sometimes known as Conway's Law⁶ (Coplien,

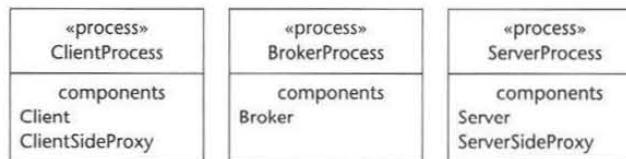


Figure 12.18 Process allocation of components in Figure 12.17.

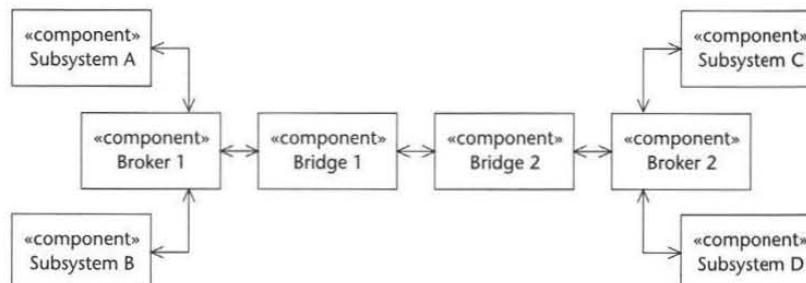


Figure 12.19 Schematic of broker architecture using bridge components.

⁶ This is an example of an organizational pattern.

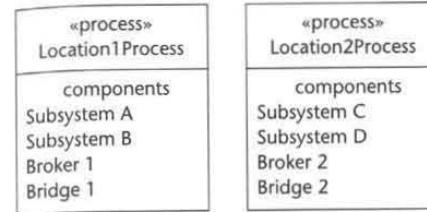


Figure 12.20 Process allocation of components in Figure 12.19.

1995). If a subsystem that is being developed by more than one team is cohesive, and the way it is split between teams has no apparent functional basis, then the teams may coalesce in practice and operate as one. Teams that are working on the same subsystem are sometimes inhibited from merging, perhaps because they are located on different continents. The subsystem should then be treated as if it were two separate subsystems. An interface between these two new subsystems can be defined and the teams can then operate autonomously. Where the allocation of one subsystem to two teams is such that one team deals with one set of requirements and the other deals with a different set of requirements, the subsystem can also be treated as if it were actually two subsystems, with a defined interface between them.

12.6 Concurrency

In most systems there are many objects that do not need to operate concurrently but some may need to be active simultaneously. Object-oriented modelling captures any inherent concurrency in the application principally through the development of interaction diagrams and state machines. The examination of use cases also helps with the identification of concurrency. There are several ways of using these models to identify circumstances where concurrent processing may be necessary. First, a use case may indicate a requirement that the system should be able to respond simultaneously to different events, each of which triggers a different thread of control. Second, if a state machine reveals that a class has complex nested states which themselves have concurrent substates, then the design must be able to handle this concurrency. The state machine for the class `Campaign` has nested concurrent states within the `Active` state (see Fig. 11.20) and there may be the possibility of concurrent activity. In this particular example, the concurrent activity that occurs in the real world need not necessarily be represented as concurrent processing in the computerized information system.

In cases where an object is required to exhibit concurrent behaviour it is sometimes necessary to split the object into separate objects in order to avoid the need for concurrent activity within any one object. Concurrent processing may also be indicated if interaction diagrams reveal that a single thread of control requires that operations in two different objects should execute simultaneously, perhaps because of asynchronous invocation. This essentially means that one thread of control is split into two or more active threads. An example of this is shown in Fig. 12.21.

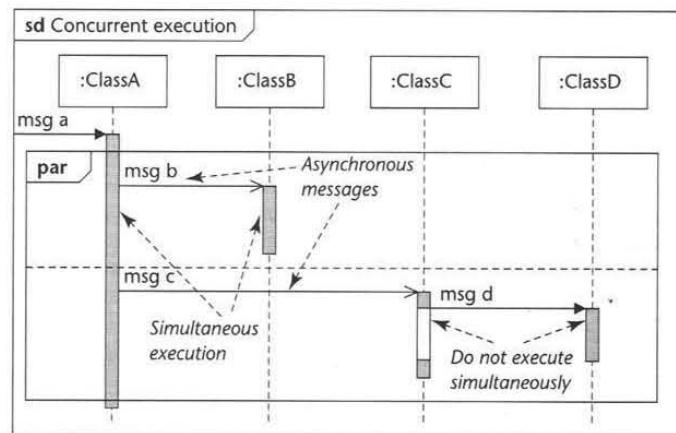


Figure 12.21 Concurrent activity in an interaction diagram.

Different objects that are not active at the same time can be implemented on the same logical processor (and thus also on the same physical processor—this distinction is explained below). Objects that must operate concurrently must be implemented on different logical processors (though perhaps still on the same physical processor).

The distinction between logical and physical concurrency is as follows. There are a number of ways of simulating the existence of multiple processors using only a single physical processor. For example, some operating systems (Unix and Windows XP) allow more than one task to appear to execute at the same time, and are thus called multi-tasking operating systems. In fact, only one task really takes place at any one time, but the operating system shares the processor between different tasks so quickly that the tasks appear to execute simultaneously. Where there are no tight time constraints a multi-tasking operating system can provide a satisfactory implementation of concurrency. But it is important to ensure that the hardware configuration of the computer can cope with the demands of multi-tasking.

When there are tight time constraints a scheduler subsystem can be introduced that ensures that each thread of control operates within the constraints on its response time. Figure 12.22 illustrates a possible relationship between a scheduler and the other parts of a system. Events that are detected by the I/O (input/output) subsystems generate interrupts in the scheduler. The scheduler then invokes the appropriate thread of control. Further interrupts may invoke other threads of control and the scheduler allocates a share of physical processor time to each thread.

Another way of implementing concurrency is to use a multi-threaded programming language (such as Java). These permit the direct implementation of concurrency within a single processor task. Finally, a multi-processor environment allows each concurrent task to be implemented on a separate processor.

Most concurrent activity in a business information system can be supported by a multi-user environment. These are designed to allow many users to perform tasks

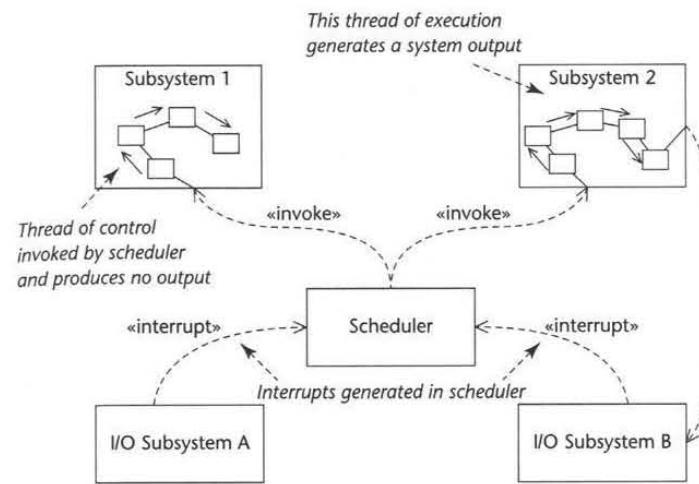


Figure 12.22 Scheduler handling concurrency.

simultaneously. Multi-user concurrent access to data is normally handled by a separate database management system (DBMS)—these are introduced briefly in Section 13.7 and are discussed in more detail in Chapter 17.

12.7 Processor Allocation

In the case of a simple, single-user system it is almost always appropriate for the complete system to operate on a single computer. The software for a multi-user information system (all or part of it) may be installed on many computers that use a shared database server. More complex applications sometimes require the use of more than one type of computer, where each provides a specialized kind of processing capability for a specific subsystem. An information system may also be partitioned over several processors, either because subsystems must operate concurrently or because some parts of the application need to operate in different locations (in other words, it is a distributed system). Information systems that use the Internet or company intranets for their communications are now widespread. Such distributed information systems operate on diverse computers and operating systems.

The allocation of a system to multiple processors on different platforms involves the following steps.

- The application should be divided into subsystems.
- Processing requirements for each subsystem should be estimated.
- Access criteria and location requirements should be determined.
- Concurrency requirements for the subsystems should be identified.
- Each subsystem should be allocated to an operating platform—either general purpose (PC or workstation) or specialized (embedded micro-controller or specialist server).

- Communication requirements between subsystems should be determined.
- The communications infrastructure should be specified.

The estimation of processing requirements requires careful consideration of such factors as event response times, the data throughput that is needed, the nature of the I/O that is required and any special algorithmic requirements. Access and location factors include the difficulties that may arise when a computer will be installed in a harsh operating environment such as a factory shop floor.

12.8 Agate Software Architecture

In the case study chapters, A2, A3 and A4, we have developed the models for the Agate system. The initial package architecture was shown in Fig. A2.8. However, this does not reflect a proper layering or partitioning of the software architecture. We may begin with a four layer architecture that separates responsibility for the user interface, the application logic, the domain classes and the database. A simple view of this is shown in Fig. 12.23.

However, we know that Agate requires the system to be capable of distribution. One option would be to adopt a thin-client architecture. In this approach, all four of the layers shown in Fig. 12.23 would be located on one or more servers, and the user interface would be generated as HTML and displayed in a browser. However, this would not give us the kind of interactivity that we have been modelling in our prototype user interfaces, which rely on a client program running on the users' PCs. So we need to decide where to split the system between the client side and the server side. The Agate Control package could be split into a client-side package that co-ordinates the user interface, playing the role of Controller, and a server-side package that orchestrates the business logic of the use cases and interacts with the domain classes. If we adopt this approach, we will break the closed layering of

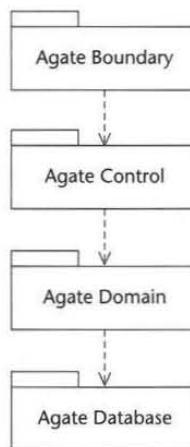


Figure 12.23 Four layer architecture for Agate.

the architecture of Fig. 12.23. Both the client-side and the server-side classes will need to understand the structure of the entity objects in the domain package (Advert, Campaign, Client etc.). If we develop in Java, the jar file containing these classes will need to be located on the client as well as the server, even if their operations are not invoked on the client. One way to reduce this dependency is to use lightweight versions of the entity classes in the Agate Domain package. These are classes that have the attributes of the entity classes, but do not have any operations apart from constructors and those operations necessary to get and set attribute values. This is an established pattern in J2EE systems, and is shown in Fig. 12.24.

Note that the dependency between the Agate Client Control package and the Agate Value Objects package is no more than that, a dependency. It does

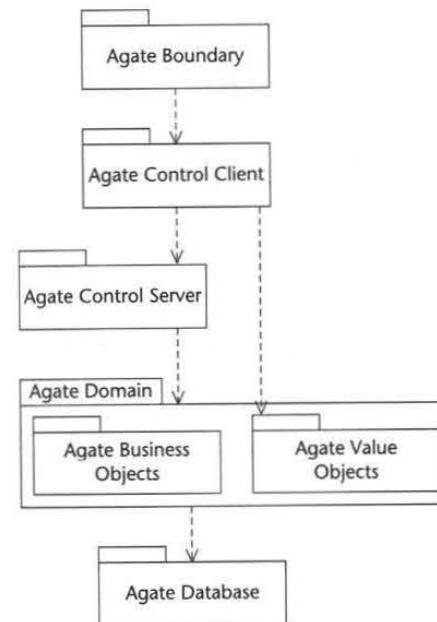


Figure 12.24 Package architecture for Agate.

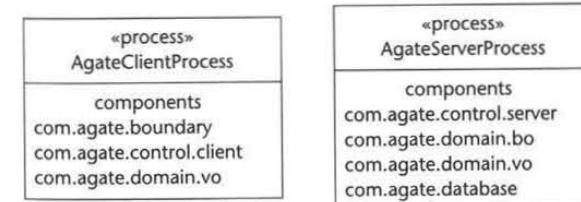


Figure 12.25 Process allocation for Agate.

not imply that there is some kind of communication across the network between the two. In fact if we implement the packages as Java packages, and deploy them, the value object package (`com.agate.domain.vo`) will exist in both the client process and the server process. This is shown in Figure 12.25.

We shall revisit this architecture in the case study chapter A5, once we have considered in more detail the design of classes, the user interface and the database.

12.9 Summary

Systems and software architecture have aspects in common with the architecture of buildings, and architectural models are typically produced using different views, which address different aspects of the architecture. In information systems, ‘architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution’ (IEEE, 2000). Many architects now use UML in order to produce architectural models of systems.

One of the key concerns of architects is to ensure that the architecture of the system will enable it to meet the quality attributes (non-functional requirements) that are expected of it. The models allow them to reason about how well the proposed structures and relationships will support demands on performance, reliability, reusability and other quality attributes.

The architecture of new systems is often constrained by existing systems, because they define either explicitly or implicitly the way in which systems are built within the organization, or because the new systems will have to inter-operate with the old. There is a growing interest in wrapping up existing systems as services to support a service-oriented architecture, or in extracting the business logic from heritage systems using reverse-engineering to produce platform-independent models, and then deriving new implementations in more modern technologies from the models. The Model-Driven Architecture movement places UML at the centre of its work and many of the features that have been improved or added in UML 2.0 are there to support MDA.

Large organizations may mandate approaches to architecture development based on enterprise architecture or technical reference architectures that lay down models of the business and how it operates (in the former case) or of standard technologies to be applied (in the latter case). Experienced architects also draw on architectural styles, which act as architectural patterns and provide well understood ways of constructing the high-level architecture of new systems.

Review Questions

- 12.1** Give a definition of architecture in an information systems context.
- 12.2** What is the difference between an architectural view and an architectural viewpoint?
- 12.3** What are the 4 + 1 views of architecture in the Unified Process?
- 12.4** What are the benefits of adopting an architecture-centric approach?

12.5 How do existing systems influence the architecture of new systems in the same organization?

12.6 Explain the difference between a PIM and a PSM.

12.7 What is meant by enterprise architecture?

12.8 What are the advantages of dividing a system into a collection of subsystems?

12.9 What is the difference between client-server and peer-to-peer communication between subsystems?

12.10 Why is an open layered architecture more difficult to maintain?

12.11 What are the disadvantages of the closed layered architecture?

12.12 What advantages would there be if the `Advert HCI` subsystem in Fig. 12.11 were designed to have direct access to the `Campaign Database` layer?

12.13 What are the main differences between the MVC architecture and the layered and partitioned architecture?

12.14 In what sense does a broker decouple two subsystems that need to communicate with each other? How does this work?

12.15 How do architectural divisions of systems help with project management?

12.16 Why is it sometimes necessary to design information systems that have explicitly concurrent behaviour?

12.17 How should you go about allocating system tasks to processors?

Case Study Work, Exercises and Projects

12.A Compare Soni’s four aspects with the UP 4 + 1 views. What do they have in common and how do they differ?

12.B Consider a system that you use regularly. What, if anything, can you tell about the architecture of the system from the user’s perspective?

12.C Develop a series of steps for the identification of partitioned subsystems within a layer in a layered architecture. Use the process for the identification of layers described in Section 12.5.2 as a starting point. Highlight any significant differences that you feel exist between the two processes.

12.D Investigate a framework for enterprise architecture. What support is there for it in modelling tools?

12.E Suggest a suitable layered architecture with any necessary partitioning for the FoodCo case study by following the procedures defined above.

Further Reading

Bass et al. (2003) is an updated version of their 1998 book, in which they have adopted UML as their architecture description language. The book mixes case studies from different kinds of projects with theory and practical guidance.

Garland and Anthony (2003) provide an excellent and practical approach to developing software architectures using UML.

Buschmann et al. (1996, 2000) provide further details of the architectures discussed in this chapter and describe other interesting alternatives.

Details of the OMG's MDA initiative can be found at www.omg.org/mda/.

The Zachman Institute for Framework Advancement (www.zifa.com) provides information about the Zachman framework.

Systems Design

Learning Objectives

In this chapter you will learn

- the difference between analysis and design
- the difference between logical and physical design
- the difference between system and detailed design
- the characteristics of a good design
- the need to make trade-offs in design.

13.1 | Introduction

Two questions that may be asked by those who are new to systems analysis and design are 'What is the difference between analysis and design?' and 'Why treat analysis and design as separate activities?' In the development of information systems, as in the development of many kinds of system, the process of analysis is distinguished from the process of design (Section 13.2). Analysis is often said to be about the 'What?' of a system, and design is described as being about the 'How?'. Design can start before or after the decision has been made about the hardware and software to be used in implementing the system. Implementation-independent or logical design is distinguished from implementation-dependent or physical design (Section 13.3). Design also takes place at several levels: system design or architectural design, which addresses architectural aspects that affect the overall system, and detailed design, which addresses the design of classes and the detailed working of the system (Chapter 14). We have already discussed the key role of architecture in design and now explore further aspects of systems design (Section 13.4).

In producing a design for a system, a designer will be working within a framework of general quality criteria and will also be trying to achieve measurable objectives for the design that are specific to the particular system (Sections 13.5

and 13.6). Data management issues are important for most information systems development projects (Section 13.7) and the design activity should be conducted within an agreed set of design and development standards (Section 13.8). Some of the design objectives may conflict with one another, and constraints on the design may result in the need for trade-offs to be made (Section 13.9). Whether the system being designed is replacing an existing computer system or is a new system implementation issues should be considered (Section 13.10). Planning for design is discussed in Chapter 22, which is available from the book's website.

13.2 How is Design Different from Analysis?

Design has been described by Rumbaugh (1997) as stating 'how the system will be constructed without actually building it'. The models that are produced by design activities show how the various parts of the system will work together; the models produced by analysis activities show what is in the system and how those parts are related to one another.

13.2.1 Moving to design

The word *analysis* comes from a Greek word meaning to break down into component parts. When we analyse an organization and its need for a new system, the analysis activity is characterized as asking *what* happens in the current system and *what* is required in the new system. It is a process of seeking to understand the organization, investigating its requirements and modelling them. The result of this analysis activity is a specification of what the proposed system will do based on the requirements.

Design is about producing a solution that meets the requirements that have been analysed. The design activity is concerned with specifying *how* the new system will meet the requirements. There may be many possible design solutions, but the intention is to produce the best possible solution in the circumstances. Those circumstances may reflect constraints such as limits on how much can be spent on the new system or the need for the new system to work with an existing system. Jacobson et al. (1992) regard design as part of the construction process (together with implementation). The systems designer has his or her attention focused on the implementation of the new system, while the systems analyst is focused on the way the business is organized and a possible better organization; the foci of these two activities are very different.

A simple example of this can be seen in the Agate case study. Analysis identifies the fact that each Campaign has a `title` attribute, and this fact is documented in the class model. Design determines how this will be entered into the system, displayed on screen and stored in some kind of database together with all the other attributes of Campaign and other classes.

Design can be seen either as a stage in the systems development lifecycle or as an activity that takes place within the development of a system. In projects that follow the waterfall lifecycle model (Fig. 3.2), the Analysis stage will be complete before the Design stage begins. However, in projects that follow an iterative lifecycle, design is not such a clear-cut stage, but is rather an activity that will be carried out on the evolving model of the system. Rumbaugh (1997) distinguishes

between the idea of design as a stage in the waterfall lifecycle and design as a process that different parts of the model of the system will go through at different times.

In the Unified Process (Jacobson et al., 1999), design is organized as a workflow—a series of activities with inputs and outputs—that is independent of the project phase. In the Rational Unified Process (Kruchten, 2004), analysis and design are combined into a single workflow—the analysis activities produce an overview model, if it is required, but the emphasis is on design—and the workflow is similarly independent of the project phase. We have adopted a similar approach to the Unified Process in the process outlined in Chapter 5. A project consists of major phases (Inception, Elaboration, Construction and Transition); each phase requires one or more iterations and, within the iterations, the amount of effort dedicated to the activities in each workflow gradually increases and then declines as the project progresses. The difference between this kind of approach and the traditional waterfall model is that in the traditional approach Analysis, Design, Construction and other stages in the waterfall are both activities and stages: during the Analysis stage, for example, all the analysis activity is meant to take place. Real projects are not like this: during the early part of the project, which may be called Analysis, some design activity may take place; during the later part of the project, which may be called Design, some analysis activity may take place. Process models such as the Unified Process recognise this and give the phases different names to decouple them from the activities. As long as less and less analysis and more and more design take place as the project develops, the project is making progress. Despite this, many projects still treat analysis and design as separate stages rather than activities that gradually elaborate the model as the project progresses.

13.2.2 Design in the iterative lifecycle

There are advantages to be gained from using an iterative lifecycle such as the Unified Software Development Process. These are concerned with:

- risk mitigation
- change management
- team learning
- improved quality.

Risk mitigation. An iterative process enables the identification of potential risks and problems earlier in the life of a project. The early emphasis on architecture and the fact that construction, test and deployment activities are begun early on make it possible to identify technological problems and take action to reduce them. Integration of subsystems is begun earlier and is less likely to throw up unpleasant surprises at the last minute.

Change management. Users' requirements do change during the course of a project, often because of the time that projects take, and often because until they see some results they may not be sure what they want. This last point is sometimes referred to as IKIWIWI—'I'll Know It When I See It'. In a waterfall lifecycle changing requirements are a problem; in an iterative lifecycle there is an expectation that some requirements activities will still be going on late in the project and it is easier to cope with changes. It is also possible to revise decisions about

technology during the project, as the hardware and software available to do the job will almost certainly change during the project.

Team learning. Members of the team, including those concerned with testing and deploying are involved in the project from the start, and it is easier for them to learn about and understand the requirements and the solution from early on. They are not suddenly presented with a new and unfamiliar system. It is also possible to identify training needs and provide the training while people are still working on an aspect of the system.

Improved quality. Testing of deliverables begins early and continues throughout the project. This helps to prevent the situation where all testing is done in a final 'big bang' and there is little time to resolve the bugs that are found.

One of the arguments put forward for the use of object-oriented approaches is that the same model (the class diagram or object model) is used right through the life of the project. Analysis identifies classes, those classes are refined in design, and the eventual programs will be written in terms of classes. While this so-called *seamlessness* of object-oriented methods may seem like an argument for weakening the distinction between analysis and design, when we move into design different information is added to the class diagram, and other different diagrams are used to support the class diagram. Rumbaugh (1997) distinguishes between analysis and design in terms of the amount of detail that is included in the model. On a continuum, the analysis stage provides an abstract model of 'what to do' while the design stage documents 'exactly how to do it'. As the project moves from one end of this continuum to the other, additional detail is added to the model until a clear specification of 'how to do it' is provided. This additional detail is added in the form of diagrams such as interaction diagrams, state machine diagrams and deployment diagrams that supplement the information in the class diagram. The class diagram is also enhanced during design by the addition of detail about attributes and operations and additional classes to handle the implementation of the user interface, communication between layers and data storage.

13.3 Logical and Physical Design

At some point in the life of a systems development project a decision must be made about the hardware and software that are to be used to develop and deliver the system—the hardware and software platform. In some projects this is known right from the start. Many companies have an existing investment in hardware and software, and any new project must use existing system software (such as programming languages and database management systems) and will be expected to run on the same hardware. This is more often the case in large companies with mainframe computers. In such companies the choice of configuration has been limited in the past to the use of terminals connected to the mainframe. However, client-server architectures (see Chapter 12) and open system standards, which allow for different hardware and software to operate together, have meant that even for such companies, the choice of platform is more open. For many new projects the choice of platform is relatively unconstrained and so at some point in the life of the project a decision must be made about the platform to be used.

Some aspects of the design of systems are dependent on the choice of platform. These will affect the system architecture, the design of objects and the interfaces with various components of the system. Examples include the following.

- The decision to create a distributed system with elements of the system running on different machines will require the use of some *middleware*, such as is provided by CORBA, to allow objects to communicate with one another across the network. This will affect the design of objects.
- The decision to write programs in Java and to use a relational database that supports ODBC (Object Data Base Connectivity) will require the use of JDBC (Java Data Base Connectivity) and the creation of classes to map between the objects and the relational database.
- The choice of Java as a software development language will mean that the developer has the choice of using the standard Java AWT (Abstract Windowing Toolkit), the Java Swing classes, the Standard Widget Toolkit (SWT) or proprietary interface classes for designing the interface.
- Java does not support multiple inheritance; other object-oriented languages such as C++ do. If the system being developed appears to require multiple inheritance then in Java this will have to be implemented using Java's interface mechanism.
- If the system has to interface with special hardware, for example bar-code scanners, then it may be necessary to design the interface so that it can be written in C as a *native method* and encapsulated in a Java class, as Java cannot directly access low-level features of hardware.

Java has been used here as an example. The same kinds of issues will arise whatever platform is chosen.

It is also the case that there are many design decisions that can be made without knowledge of the hardware and software platform.

- The interaction between objects to provide the functionality of particular use cases can be designed using interaction diagrams or communication diagrams.
- The layout of data entry screens can be designed in terms of the fields that will be required to provide the data for the objects that are to be created or updated, and the order in which they will appear on the screen can be determined. However, the exact nature of a textbox and whether it is a Borland C++ TEdit, a Delphi Edit, a Java JTextField, a C# TextBox, a Visual Basic TextBox or something else can be left until later.
- The nature of commands and data to be sent to and received from special hardware or other systems can be determined without needing to design the exact format of messages.

Because of this, design is sometimes divided into two stages. The first is *implementation-independent* or *logical* design and the second is *implementation-dependent* or *physical* design. Logical design is concerned with those aspects of the system that can be designed without knowledge of the implementation platform; physical design deals with those aspects of the system that are dependent on the implementation platform that will be used.

Having an implementation-independent design may be useful if you expect a system to have to be re-implemented with little change to the overall design but on

a different platform, for example a Windows program that is to be ported to MacOS and Linux. Or a program that must run on different types of handheld using Windows CE, EPOC and PalmOS.

In many projects, design begins after hardware and software decisions have been made. However, if this is not the case, then the project manager must ensure that the plan of work for the project takes account of this and that logical design activities are tackled first. In an iterative project lifecycle, logical design may take place in the early design iterations or, if the system is partitioned into subsystems, the logical design of each subsystem will take place before its physical design.

13.4 System Design

Design of systems takes place at two levels: system design and detailed design. System design is concerned with the overall architecture of the system and the setting of standards, for example for the design of the human-computer interface; detailed design is concerned with designing individual components to fit this architecture and to conform to the standards. In an object-oriented system, the detailed design is mainly concerned with the design of objects and classes. Detailed design is considered in Chapter 14.

During system design the designers make decisions that will affect the system as a whole. The most important aspect of this is the overall architecture of the system (Chapter 12). Many modern systems use a client-server architecture in which the work of the system is divided between the clients (typically PCs on the users' desks) and a server (usually a Unix or Windows NT machine that provides services to a number of users). This raises questions about how processes and objects will be distributed on different machines, and it is the role of the system designer or system architect to decide on this. The design will have to be broken down into subsystems and these subsystems may be allocated to different processors. This introduces a requirement for communication between processors, and the systems designer will need to determine the mechanisms used to provide for this communication. Distributing systems over multiple processors also makes it possible for different subsystems to be active simultaneously or concurrently. This concurrency needs to be designed into the system explicitly rather than left to chance.

Many organizations have existing standards for their systems. These may involve interface design issues such as screen layouts, report layouts or how online help is provided. Decisions about the standards to be applied across the whole system are part of system design, whereas the design of individual screens and documents (to comply with these standards) is part of detailed design.

When a new system is introduced into an organization, it will have an impact on people and their existing working practices. Job design is often included in system design and addresses concerns about how people's work will change, how their interest and motivation can be maintained and what training they will require in order to carry out their new jobs. How people use particular use cases will be included in the detailed design of the human-computer interface.

13.5 Qualities and Objectives of Analysis and Design

There are a number of criteria for a good design and these are discussed in Section 13.5.2 and the following section. Perhaps the most obvious measure of design quality is whether the finished application is of high quality. This assumes that the analysis that preceded the design work was itself of high quality. However, this is a rather vague and circular way of assessing quality: in this section we look briefly at aspects of quality in analysis and in more detail at quality in design. Design quality cannot be measured only against general criteria, and in Section 13.6 we consider more specific measures that can be applied to particular projects.

13.5.1 What makes for good analysis?

The cost of fixing faults in a system increases as the system progresses through the systems development lifecycle. If an error occurs in the analysis of a system, it is cheaper to fix it during an early phase than it is later, when that error may have propagated through numerous aspects of the design and implementation. It is most expensive to fix it after the system has been deployed and the error may be reflected in many different parts of the system. The quality of the design is, therefore, dependent to a large extent on the quality of the analysis.

Some methodologies have explicit quality criteria that can be applied to the products of every stage of the lifecycle, but these quality criteria typically check syntactic aspects of the products, that is, whether the notation is correct in diagrams, rather than semantic aspects, that is, whether the diagrams correctly represent the organization's requirements. To provide a sound foundation for design, analysis should meet the following four criteria:

- correct scope
- completeness
- correct content
- consistency.

These are described in more detail below.

Correct scope. The scope of a system determines what is included in that system and what is excluded. It is important first that the required scope of the system is clearly understood, documented and agreed with the clients and second, that everything that is in the analysis models *does* fall within the scope of the system. In the case of the Agate system, it is not a requirement to replace the existing accounting system that is used to invoice clients. It is, however, a requirement that the new system should interface with the accounting system to provide for the transfer of data relating to financial aspects of advertising campaigns. The scope of the system therefore excludes use cases for accounting but should include use cases both to handle the entry of data that will be transferred to the accounting system and to handle the transfer itself. Coad et al. (1997) include a *not this time* component with their other four components (problem domain, human interface, data management and system interaction). The *not this time* component is used to document classes and business services that emerge during the analysis but are not part of the requirements this time. This is a useful way of forcing consideration of the scope of the system.

Completeness. Just as there is a requirement that everything that is in the analysis models is within the scope of the system, so everything that is within the scope of the system should be documented in the analysis models. Everything that is known about the system from the requirements capture should be documented and included in appropriate diagrams. Often the completeness of the analysis is dependent on the skills and experience of the analyst. Knowing what questions to ask in order to elicit requirements comes with time and experience. However, analysis patterns and strategies, as proposed by Coad et al. (1997) and Fowler (1997), can help the less experienced analyst to identify likely issues. (The use of patterns, which draw on past experience, can be a good way of ensuring that the analysis is effective.)

Non-functional requirements should be documented even though they may not affect the analysis models directly. Rumbaugh (1997) suggests that some of the requirements found during analysis are not analysis requirements but design requirements. These should be documented, but the development team may only have to consider them once the design phase has begun. An example in the Agate system is the requirement that the system should be usable in different offices around the world and should handle multiple currencies. This would be noted during analysis; during design, it will mean that the system must be designed to support localization (adaptation to local needs) and to display different national currency symbols (perhaps using the Unicode standard).

Correct content. The analysis documentation should be correct and accurate in what it describes. This applies not only to textual information and diagrams but also to quantitative features of the non-functional requirements. Examples include correct descriptions of attributes and any operations that are known at this stage, correct representation of associations between classes, particularly the multiplicity of associations, and accurate information about volumes of data. Accuracy should not be confused with precision. FoodCo owns 1,500 acres of land (to the nearest 100 acres). To state that the company owns 1,700 is inaccurate. To state that it owns 1,523 is more precise. To state that it owns 1,253 is still inaccurate, although the precision gives a spurious impression of accuracy.

Consistency. Where the analysis documentation includes different models that refer to the same things (use cases, classes, attributes or operations), the same name should be used consistently for the same thing. Errors of consistency can result in errors being made by designers, for example, creating two attributes with different names that are used in different parts of the system but should be the same attribute. If the designers spot the inconsistency, they may try to resolve it themselves, but may get it wrong because the information they have about the system is all dependent on what they have received in the specification of requirements from the analysts.

Errors of scope or completeness will typically be reflected in the finished product not doing what the users require; the product will either include features that are not required or lack features that are. Errors of correctness and consistency will typically be reflected in the finished product performing incorrectly. Errors of completeness and consistency will most often result in difficulties for the designers; in the face of incomplete or inconsistent specifications, they will have to try to decide what is required or refer back to the analysts.

One general way of ensuring that the analysis models reflect the requirements is to use walkthroughs. Walkthroughs are described by Yourdon (1985) (and also in

an appendix to Yourdon, 1989). They provide a structured review with other developers. They may be used at various points in the systems development life-cycle. For example, walkthroughs may also be used during program design and development

13.5.2 What makes for good design?

The quality of the design will clearly be reflected in the quality of the finished system that is delivered to the clients. Moreover, in the same way as the quality of analysis affects the work of designers, the quality of the design has an impact on the work of the programmers who will write the program code in order to implement the system based on the design. Some of the criteria given below for a good design will bring benefits to the developers, while some will provide benefits for the eventual users of the system.

13.5.3 Objectives and constraints

The designers of a system seek to achieve many objectives that have been identified as the characteristics of a good design since the early days of information systems development. Yourdon and Constantine (1979) cite efficiency, flexibility, generality, maintainability and reliability; DeMarco (1979) proposes efficiency, maintainability and buildability; and Page-Jones (1988) suggests that a good design is efficient, flexible, maintainable, manageable, satisfying and productive. These latter two points highlight issues concerned with human-computer interaction and remind us of the need for the design to produce a usable system. Other characteristics of a good design are that it should be functional, portable, secure and economical; in the context of object-oriented systems, reusability is a priority objective.

Functional. When we use a computer system, we expect it to perform correctly and completely those functions that it is claimed to perform; when an information system is developed for an organization, the staff of that organization will expect it to meet their documented requirements fully and according to specification. So, for example, the staff of Agate will expect their system to provide them with the functionality required to document advertising campaigns, record notes about campaigns and store information about the advertisements to be used in those campaigns. If it does not perform these functions, it is not fully functional. Referring back to Rumbaugh's definition of design as 'how the system will be constructed without actually building it', a functional design should show how every element of the required system will work.

Efficient. It is not enough that a system performs the required functionality; it should also do so efficiently, in terms both of time and resources. Those resources can include disk storage, processor time and network capacity. This is why design is not just about producing any solution, but about producing the best solution. This objective may apply to Agate's requirement to store textual notes about ideas for campaigns and advertisements. A sample two lines of text containing twenty words takes up about 100 bytes in text format, but stored in the format of a well-known word-processor takes up 13,800 bytes (this is without the use of any special fonts or styles). A poor design might use object linking and embedding (OLE) to handle the word-processing of the notes but would pay a severe penalty in increased storage requirements!

Economical. Linked to efficiency is the idea that a design should be economical. This applies not only to the fixed costs of the hardware and software that will be required to run it, but also to the running costs of the system. The cost of memory and disk storage is very low compared to twenty years ago, and most small businesses using Microsoft Windows probably now require more disk space for their programs than they do for their data. However, the growth of multimedia systems for business purposes may make it once more important to calculate the storage requirements of a system carefully. For large enterprises, it is still important to plan carefully for storage requirements and their anticipated growth.

Reliable. The system must be reliable in two ways: first, it should not be prone to either hardware or software failure; second, it should reliably maintain the integrity of the data in the system. Hardware reliability can be paid for: some manufacturers provide systems with redundant components that run in parallel or that step in when an equivalent component fails; RAID (redundant arrays of inexpensive disks) technology can provide users with disk storage that is capable of recovering from failure of one drive in an array. The designers must design software reliability into the system. In physical design, detailed knowledge of the development environment is likely to help ensure reliability.

Reliability depends to some extent on the ability of the system to be tested thoroughly. A well analysed and designed system will specify the valid and invalid combinations of data that can be handled. It will also show clearly the structure of the system and which elements of the system are dependent on others so that testing can work up through classes, groups of classes, subsystems and eventually the whole system.

Secure. Systems should be designed to be secure against malicious attack by outsiders and against unauthorized use by insiders. System design should include considerations of how people are authorized to use the system and policies on passwords. It should also cover protection of the system from outsiders, including firewalls in either hardware or software to protect the system from access via public networks, such as the Internet. In European countries that are members of the European Union, there are data protection laws that are designed to protect the interests of individuals about whom data is held in information systems, such as the Data Protection Acts (1984 and 1998) and the Computer Misuse Act (1990) in the UK. Where such legislation exists, the designer should ensure that the design of the system will comply with its requirements.

Flexible. Some authors treat flexibility as the ability of the system to adapt to changing business requirements as time passes. Yourdon and Constantine (1979) call this feature *modifiability*. By flexibility they mean the ability to configure the system to handle different circumstances based on control values that are not compiled into the system but are available for the user to set at run-time. In the Agate system, this could be reflected in the choice of ODBC as the means to access the database. This provides a standard mechanism for accessing databases, and changing the ODBC driver used would allow either the system to access a local or a remote version of the database or for the system to be migrated to a different database engine at a later date. Another possibility would be to ensure that all the prompts and error messages used by the system are held in an external data file that can be loaded up when the program runs or in response to a menu option. This allows for the creation of multiple files of messages and would enable users to

set the language that they wish to use. The use of Unicode, which provides a character set that includes ideographic characters (for example, Chinese, Japanese and Korean) as well as all the world's alphabets, would enable a system to be developed for Agate that could be localized for each location in which it is used.

General. Generality describes the extent to which a system is general-purpose. It is more applicable to utility programs than to large information systems. However, it includes the issue of *portability*, which applies to the Agate system that is to be developed in Java so that it can run on different hardware (PCs and Apple Macs). The system may also exhibit generality from the point of view of the developers, who may wish to use the same system for other clients in the advertising industry. Reuse is discussed below.

Buildable. From the perspective of the programmer who has to write the program code to build the system, it is important that the design is clear and not unnecessarily complex. In particular, the physical design should relate closely to the features that are available in the development language. Not all object-oriented languages offer the same features, for example in the visibility of attributes and operations (public, private, protected, friend etc.), in the ability to handle multiple inheritance or in the availability of utility classes such as collections or linked lists in the base language. Designs that rely on features such as these will force the programmer to work around them if a different language is being used from the one that the designer had in mind.

Manageable. A good design should allow the project manager to estimate the amount of work involved in implementing the various subsystems. It should also provide for subsystems that are relatively self-contained and can be marked off as completed and passed on for testing without fear that changes to other parts of the system still in development will have unforeseen consequences on them.

Maintainable. Maintenance is cited as taking up as much as 60% of the data-processing budget of organizations. Maintenance activities include fixing bugs, modifying reports and screen layouts, enhancing programs to deal with new business requirements, migrating systems to new hardware and fixing the new bugs that are introduced by all of the above. A well-designed and documented system is easier to maintain than one that is poorly designed and documented. If maintenance is easy then it is less costly. It is particularly important that there is a close match between the developed program code and the design. This makes it easier for the maintenance programmer to understand the intention of the designer and to ensure that it is not subverted by the introduction of new code.

Usable. Usability covers a range of aspects including the idea, mentioned above, that a system should be both satisfying and productive. It may seem odd to suggest that people should enjoy using their computer systems and find it a satisfying experience. However, if you think about the times that you have used a computer system and have found it a source of dissatisfaction, then you can perhaps imagine a satisfying system as one with an absence of dissatisfying features. Many of the features that contribute to user satisfaction are characteristic of good human-computer interface (HCI) design. For example, the concept of *affordance* (meaning that objects on the interface suggest their function) can reduce the number of errors made by users. Reducing error rates and ensuring that if users do make an error it is clear both where they went wrong and how to recover from the

error can contribute to the satisfaction of users. Productivity can be enhanced by ensuring that the tasks that users wish to carry out using the system are straightforward to carry out and do not introduce an overhead of keystrokes or mouse-clicks to achieve. If usability requirements have been captured (see Section 6.2.2), then the design should take these into account. Usability is considered in more detail in Chapter 16.

Reusable. Reusability is the Holy Grail of object-oriented development. Many of the features of object-oriented systems are geared to improve the possibility of reuse. Reuse affects the designer in three ways. First, he or she will consider how economies can be made by designing reuse into the system through the use of inheritance; second, he or she will look for opportunities to use design patterns, which provide templates for the design of reusable elements; and, third, he or she will seek to reuse existing classes or components either directly or by subclassing them. Design patterns are described in detail in Chapter 15. Existing classes could be classes that have been developed for other projects, classes in class libraries that are associated with the development language (such as the Java AWT) or classes that are bought in from outside vendors. To date, object-oriented development has not achieved the levels of reuse that were expected. In order to reuse a software class, a designer must be aware of the existence of the class, and be able to determine both that its interface matches the interface for the class that he or she requires and that the methods of the class match those required. It is arguable that in order to determine whether an available class matches requirements, the required class must already have been designed. The economies from reuse thus appear during the construction of the software and require a change to a culture of project management that supports reuse; this means that project managers must be able to recognize the effort that is saved by **not** writing and testing lines of code (because a class is being reused). The development of strategies to parcel up classes as components and the provision of component management software are discussed in Chapter 20.

There is clearly some overlap between the categories that have been listed here. Aspects of maintainability overlap with flexibility, generality with reuse, efficiency with economy. What is often the case, however, is that some design objectives will conflict with one another. This happens more often at the level of specific objectives rather than general ones such as those described above. However, it should be possible to see that functionality, reliability and security could all conflict with economy. Many of the conflicts result from constraints that are imposed on the system by the users' non-functional requirements. The management of the resulting design trade-offs is discussed later.

13.5.4 Coupling and cohesion

Yourdon and Constantine (1979) defined a series of criteria for structured design that could be used in breaking systems and programs down into modules to ensure that they are easy to develop and maintain. These criteria concern two issues: *cohesion* and *coupling*. Criteria to maximize desirable types of cohesion have as their aim the production of modules—sections of program code in whatever language is used—that carry out a clearly defined process or a group of processes that are functionally related to one another. This means that all the elements of the module contribute to the performance of a single function. Poor cohesion is found

when processes are grouped together in modules for other reasons. Examples of poor types of cohesion include:

- processes that are grouped together for no obvious reason (*coincidental cohesion*)
- logically similar processes that are handled together, such as inputs (*logical cohesion*)
- processes that happen at the same time—for example when the system initializes—(*temporal cohesion*).
- the outputs of one process being used as inputs by the next (*sequential cohesion*).

By aiming to produce modules that are functionally cohesive, the designer should produce modules that are straightforward to develop, easy to maintain and have the maximum potential to be reused in different parts of the system. This will be assisted if coupling between modules is also reduced to the minimum.

Criteria to minimize the coupling between modules have as their aim the production of modules that are independent of one another and that can be amended without resulting in knock-on effects to other parts of the system. Good coupling is achieved if a module can perform its function using only the data that is passed to it by another module and using the minimum necessary amount of data.

The concepts of coupling and cohesion can be applied to object-oriented systems development. Coad and Yourdon (1991) suggested several ways in which coupling and cohesion can be applied within an object-oriented approach. Larman (2005) also considers the application of these criteria. The criteria can be used within object-orientation during both analysis and design as described below (adapted from Coad and Yourdon, 1991).

Coupling describes the degree of interconnectedness between design components and is reflected by the number of links an object has and by the degree of interaction the object has with other objects. Cohesion is a measure of the degree to which an element contributes to a single purpose. The concepts of coupling and cohesion are not mutually exclusive but actually support each other.

Interaction Coupling is a measure of the number of message types an object sends to other objects and the number of parameters passed with these message types. Interaction coupling should be kept to a minimum to reduce the possibility of changes rippling through the interfaces and to make reuse easier. When an object is reused in another application it will still need to send these messages (unless the object is modified before it is reused) and hence needs objects in the new application that provide these services. This affects the reuse process as it requires groups of classes to be reused rather than individual classes. (In Chapter 8 we introduced the idea of the *component* as the unit of reuse and discuss it further in Chapter 20. Components are groups of objects that together provide a clearly defined service.)

Inheritance Coupling describes the degree to which a subclass actually needs the features it inherits from its base class. For example, in Fig. 13.1 the inheritance hierarchy exhibits low inheritance coupling and is poorly designed. The subclass *LandVehicle* needs neither the attributes *maximumAltitude* and *takeOffSpeed* nor the operations *checkAltitude()* and *takeOff()*. They have been inherited unnecessarily. In this example it would appear that the base class, *Vehicle*, would perhaps be better named *FlyingVehicle* and the inheritance relationship is somewhat suspect. A land vehicle is not a kind of flying vehicle (not

normally anyway). However, many systems developers view designs with a small degree of unnecessary inheritance as being acceptable if the hierarchy is providing valuable reuse and is meaningful. It can be argued that if attributes and operations are inherited unnecessarily it is merely a matter of not using these features in the subclass. However, a subclass with unnecessary attributes or operations is more complex than it needs to be and objects of the subclass may take more memory than they actually need. The real problems may come when the system needs maintenance. The system's maintainer may not realize that some of the inherited attributes and operations are unused and may modify the system incorrectly as a result. Alternatively the system's maintainer may use these unneeded features to provide a fix for a new user requirement, making the system even more difficult to maintain in the future. For these reasons, unnecessary inheritance should be kept as low as possible.

Operation Cohesion measures the degree to which an operation focuses on a single functional requirement. Good design produces highly cohesive operations, each of which deals with a single functional requirement. For example in Fig. 13.2 the operation `calculateRoomSpace()` is highly cohesive.

Class Cohesion reflects the degree to which a class is focused on a single requirement. The class `Lecturer` in Fig. 13.2 exhibits low levels of cohesion as it has three attributes (`roomNumber`, `roomLength` and `roomWidth`) and one operation (`calculateRoomSpace()`) that would be more appropriate in a class `Room`. The class `Lecturer` should only have attributes that describe a `Lecturer` object (e.g. `lecturerName` and `lecturerAddress`) and operations that use them.

Specialization Cohesion addresses the semantic cohesion of inheritance hierarchies. For example in Fig. 13.3 all the attributes and operations of the `Address` base class are used by the derived¹ classes: the hierarchy has high inheritance coupling. However, it is true neither that a person is a kind of address nor that a company is a kind of address. The example is only using inheritance as a syntactic structure for sharing attributes and operations. This structure has low

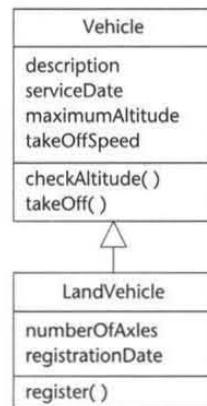


Figure 13.1 Inheritance coupling.

¹ A derived class is another term for subclass; the superclass is known as the base class.

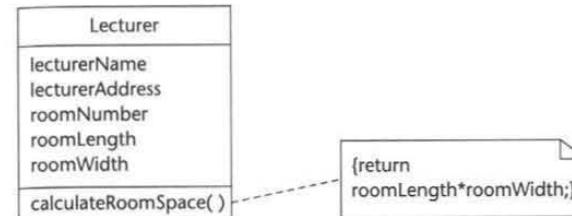


Figure 13.2 Good operation cohesion but poor class cohesion.

specialization cohesion and shows poor design. It does not reflect meaningful inheritance in the problem domain. A better design is shown in Fig. 13.4, in which a common class `Address` is being used by both the `Person` and `Company` classes. All the design criteria explained above may be applied at the same time to good effect.

13.5.5 Liskov Substitution Principle

The *Liskov Substitution Principle (LSP)* is another design criterion that is applicable to inheritance hierarchies. Essentially the principle states that, in object interactions, it should be possible to treat a derived object as if it were a base object. If the principle is not applied, then it may be possible to violate the integrity of the derived object. In Fig. 13.5 objects of the class `MortgageAccount` cannot be treated as if they are objects of the class `ChequeAccount` because `MortgageAccount` objects do not have a debit operation whereas `ChequeAccount` objects do. The `debit` operation is declared private in `MortgageAccount` and hence cannot be used by any other object. Figure 13.5 shows an alternative structure that satisfies LSP. Interestingly, this inheritance hierarchy has maximal inheritance coupling, and enforcing the LSP normally produces structures with high inheritance coupling.

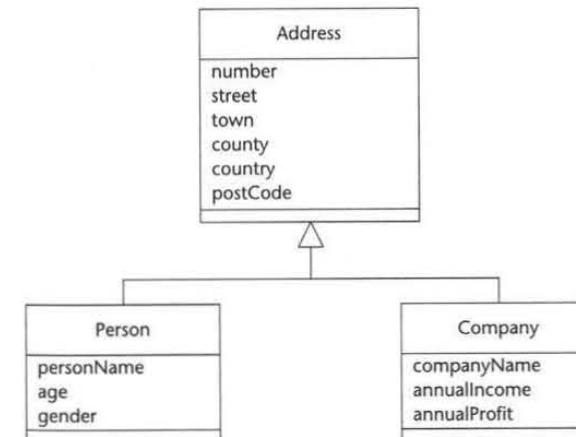


Figure 13.3 Poor specialization cohesion.

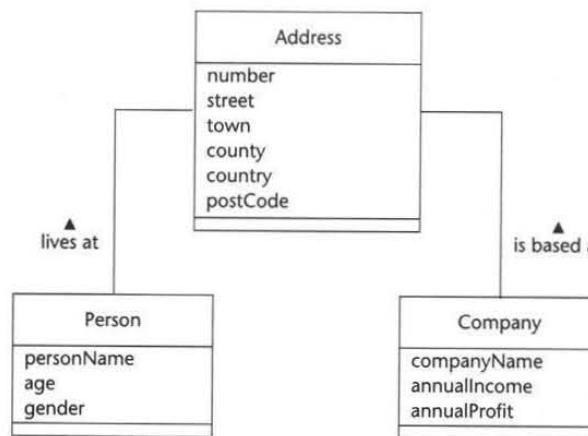


Figure 13.4 Improved structure using Address class.

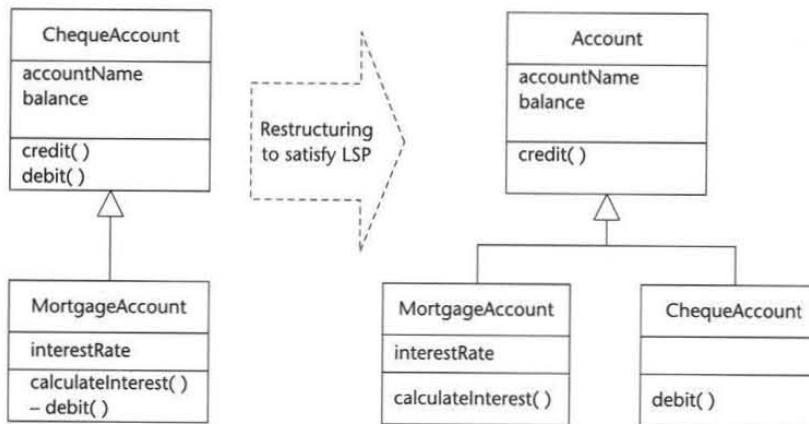


Figure 13.5 Application of the Liskov Substitution Principle.

13.6 Measurable Objectives in Design

In the previous section, we discussed some of the general objectives of the designers in a systems development project. Some objectives are specific to a particular project, and it is important to be able to assess whether these objectives have been achieved. One way of doing this is to ensure that these objectives are expressed in measurable terms so that they can be tested by simulation during the design phase, in prototypes that are built for this purpose or in the final system.

Measurable objectives often represent the requirements that we referred to as non-functional requirements in Chapter 6. They also reflect the fact that

information systems are not built for their own sake, but are developed to meet the business needs of some organization. The system should contribute to the strategic aims of the business, and so should help to achieve aims such as:

- provide better response to customers
- increase market share.

However, such aims are vague and difficult to assess. If they are expressed in measurable terms, then it is possible to evaluate whether they can be achieved by the design or whether they have been achieved by the finished system. Ideally, they should be phrased in a way that shows how these objectives are attributable to the system. If a company expects to increase its market share as a result of introducing a new computer system but does not achieve this, it should be possible to tell whether this is a failure of the new system or the outcome of other factors outside the control of the system developers, such as economic recession. The system may contribute to business objectives such as those above by providing better information or more efficient procedures, but for the objectives to be measurable they need to be phrased as operational objectives that can be quantified, such as:

- to reduce invoice errors by one-third within a year
- to process 50% more orders during peak periods.

These set clear targets for the designers and a way of checking whether these objectives can be achieved (within the constraints on the system) and whether they have been achieved once the system is up and running.

13.7 Data Management Issues

Suitable data management approaches for an information system can vary from simple file storage and retrieval to sophisticated database management systems of various types. In some applications where data has to be accessed very rapidly, the data may be kept in main memory while the system executes. However, most data management is concerned with storing data, often large volumes, so that it may be accessed at a later stage either by the same system or by another. A file-based approach to data management is suitable for simple data management, but the data storage and retrieval must be coded explicitly within the application. If the data is to be accessed by multiple users simultaneously, the management of the data storage and retrieval directly in the application becomes complex. However, a requirement for fast access to data may justify the use of a file-based approach and its resulting code complexity.

Database management systems (DBMS) provide various facilities that are useful in many applications. A DBMS typically offers support for:

- different views of the data by different users
- control of multi-user access
- distribution of the data over different platforms
- security
- enforcement of integrity constraints

- access to data by various applications
- data recovery
- portability across platforms
- data access via query languages
- query optimization.

These capabilities make a DBMS the obvious choice for many applications. However, a DBMS does have a significant performance overhead and the standard data access mechanisms may be inappropriate for specialized systems. Once a decision has been made to use a DBMS, the most appropriate type must be selected. A relational DBMS is likely to be appropriate if there are large volumes of data with varying (perhaps ad hoc) access requirements. An object-oriented DBMS is more likely to be suitable if specific transactions require fast access or if there is a need to store complex data structures and there is not a need to support a wide range of transaction types. A third type of DBMS is emerging—the object-relational DBMS—that is similar to an object-oriented DBMS in its support for complex data structures, but that also provides effective querying facilities. In some systems there may be different data management requirements for different subsystems and it may be best then to use a mix of DBMS types. These issues and the detailed design consequences of the choice of DBMS are explored further in Chapter 18.

The DBMS used by the application will typically be accessed from programs using specialized class libraries to provide the database access functionality. It is rarely the case today that a developer must construct low-level primitives in order to access a DBMS since suitable commercial class libraries are widely available.

13.8 Development Standards

All information systems development projects should operate with clearly defined guidelines within which all members of the development team work. Many organizations will have corporate style guides that govern the production of software development artefacts, including the delivered system. In some organizations these corporate guides may be adapted for particular development projects. Modelling standards have been discussed in Chapter 5. From the design perspective it is important to specify guidelines for the development of I/O subsystems and their interfaces and standards for the development of code.

13.8.1 HCI guidelines

Standards for the human-computer interface are an important aspect of the design activity, since it is with the interface that users actually interact. Some characteristics of good dialogues and the subject of style guides for HCI are discussed in Chapter 16.

13.8.2 Input/output device guidelines

Where an application interacts with mechanical or electronic devices such as temperature and pressure sensors or actuators that control heaters or motors, it is equally important to develop guidelines. The objective is to use a standard form of

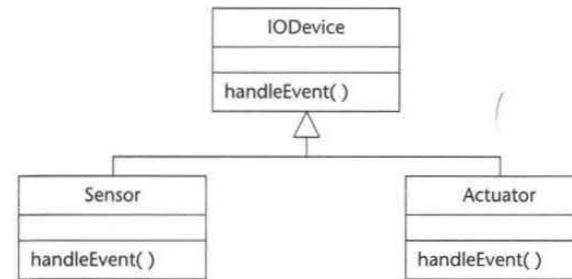


Figure 13.6 I/O device hierarchy.

interface with the devices so that hardware may be changed or updated without occasioning any changes to the core system functionality. Sensor and controller devices usually have fully specified communications protocols and standardization is probably best achieved by encapsulating all direct access with each device in a single I/O device object. An I/O device class can be subclassed so that for each particular device involved with the application there is a class that deals with its communications protocol. Since all the I/O device classes would then be subclasses of one inheritance hierarchy, they can easily be constructed to provide consistent interfaces to the rest of the application. One way of achieving this is by the use of polymorphism, as shown in Fig. 13.6. In this diagram, each subclass overrides the abstract operation `handleEvent()`, so that each can respond to a `handleEvent()` message in its own appropriate way, but other objects in the system do not necessarily need to know which particular type of object will receive the message.

13.8.3 Construction guidelines

Construction guidelines may not appear relevant at this stage in a systems development project. However, they are pertinent to system design because there is a growing tendency for developers to use CASE environments that have code generation capabilities. It is also the case that when a rapid development approach or an iterative approach is followed, the demarcation between design and construction activities can become blurred.

Construction guidelines will normally include advice on the naming of classes, of operations and of attributes, and where this is the case these guidelines are also applicable during the analysis activity. Wherever possible, consistent naming conventions should be enforced throughout the project since this makes it easier to trace an analysis class directly through to its implementation. Other guidelines for construction might relate to the use of particular software features (for example, using only standard language constructs in order to aid portability) and the layout of the code. These issues are addressed in more detail in Chapter 19.

13.9 Prioritizing Design Trade-offs

Design frequently involves choosing the most appropriate compromise. The designer is often faced with design objectives or constraints that are mutually

incompatible and he or she must then decide which objective is the more important.

Design constraints arise from the context of the project as well as from the users' requirements. The clients' budget for the project, the timescale within which they expect the system to be delivered, the skills of staff working on the project, the need to integrate the new system with existing hardware or systems and standards set as part of the overall systems design process can all constrain what can be achieved. Resolving conflicts between requirements and constraints results in the need for compromises or trade-offs in design. A couple of examples should illustrate how these can occur.

- If the users of Agate's new system require the ability to change fonts in the notes that they write about campaigns and adverts, then they will want to be able to edit notes with the same kind of functionality that would be found in a word-processor. As pointed out earlier in Section 13.5.3 when we discussed efficiency, this will seriously impact the storage requirements for notes. It will also have an effect on network traffic, as larger volumes of data will need to be transferred across the network when users browse through the notes. The designers will have to consider the impact of this requirement. It may be that the users will have to accept reduced functionality or the management of Agate will have to recognize that their system will have higher costs for storage than first envisaged. Compromise solutions may involve transferring only the text of a note (without the overhead of all the formatting information) when users are browsing a note and transferring the full file only when it needs to be viewed or edited. However, this will increase the processing load on the server. Another compromise solution might be to use a different file format such as RTF (rich text format) rather than the word-processor format. For the short text file discussed above this reduces the byte count to 1,770 while retaining formatting information.
- Agate would like the system to be configurable so that prompts, help and error messages are displayed in the language of the user. This means that each prompt and error message must be read into the programs from data files or the database. While this is good software design practice and makes the system more flexible, it will increase the workload of the designers when they design elements of the interface. Without this requirement, it is enough for each designer to specify that messages such as 'Campaign' or 'Not on file' appear on screen; there is a minimal need for liaison between designers. With this requirement, the designers will need to draw up a list of prompts, labels and error messages that can be referred to by number or by some other key so that the same message is used consistently wherever it is applicable. This means that the programmers will not hard code messages into the system, but will refer to them as elements in an array of messages, for example. While this increases the flexibility and to some extent the maintainability of the system, it is likely to increase the cost of the design phase.

It is important that these design decisions are clearly documented and the reasoning behind compromises and trade-offs is recorded. Keeping the client informed of these decisions is always important to ensure that the decisions are consistent with the client's needs.

The requirements model may indicate the relative priorities of the different objectives or constraints but, if it does not, then it is useful to prepare general

guidelines. These guidelines must be agreed with clients since they determine the nature of the system and what functionality will be delivered. Guidelines for design trade-offs ensure consistency between the decisions that are made at different stages of development. They also ensure consistency between different subsystems. However, no guidelines can legislate for every case. Design experience and further discussions with the client will remain necessary to resolve those situations that cannot be anticipated—at least some of these occur on almost every project.

13.10 | Design for Implementation

The introduction of an information system within an organization frequently requires some form of data conversion. System design must consider how best to deal with this and other system initialization issues. Special programs may be designed and written that convert data from one format to another and procedures may have to be specified for the manual entry of data that configures the final system. A new system may need a batch input capability so that existing data can be loaded, even where the day-to-day use of the system is intended to be fully interactive with online update of all data.

The design and construction of the data conversion or initialization subsystems may represent a significant part of development and can be complex. Logistical considerations such as the overall volumes of data and the time available for conversion and initialization will have a significant impact upon the approach that is taken. The data of an enterprise represents one of its most valuable resources and every care must be taken not to corrupt or lose it.

13.11 | Summary

While analysis looks to the business in order to establish requirements, design looks to the technology that will be used to implement those requirements. An effective design will meet general objectives that will make the system both easier to build and maintain and more usable and functional for the end-users. The design of a system should also meet specific objectives relating to the business needs of the users, and these specific objectives should be phrased in quantifiable, operational terms that allow them to be tested. This process of design takes place in the context of constraints that are imposed by the users, their budget and existing systems, the available technology and the skills and knowledge of the design and development team.

System design focuses on determining a suitable architectural structure (discussed in Chapter 12) for the system and defines the context within which the remaining design activity is performed. We have explored other key aspects of design. Selecting an appropriate data management strategy is essential for most information systems development projects. Specifying development standards enforces consistency and helps to improve quality. Design objectives may conflict and these conflicts must be resolved to the client's satisfaction. Finally, the process of systems implementation and installation must be considered to ensure continuity of service for the client where this is appropriate.

In Chapter 14 we describe detailed design of the classes in the required system. Chapter 15 explains how patterns can be used to assist the design process. Chapters 16 and 17 look specifically at the design of the human-computer interface and Chapter 18 discusses the design of data storage.

Review Questions

- 13.1** What are the advantages of separating the analysis and design phases of a project?
- 13.2** What are the advantages of an iterative lifecycle?
- 13.3** Users at Agate require a report of unpaid campaigns. Which of the following aspects of the report represents analysis, logical design and physical design?
The size of the paper and the position of each field in the report.
The fact that the user wants a report of completed campaigns that have not yet been paid for by the client.
The selection of the business objects and their attributes used by the report.
- 13.4** Which of the following sentences describing an element of the FoodCo system represents analysis, logical design and physical design?
The reason for stopping a run will be selected from one of the values displayed in a listbox (Java Choice) in the Record Line Stop dialogue window.
When a production line stops during a run, the reason for stopping will be recorded.
The reason for stopping a run will be entered into the system by selecting from a list of valid reasons.
- 13.5** What is meant by *seamlessness* in object-oriented systems development?
- 13.6** What are the differences between system design and detailed design?
- 13.7** Explain the difference between cohesion and coupling.
- 13.8** How does the use of the Liskov Substitution Principle improve the quality of the design?
- 13.9** List four quality criteria for good analysis.
- 13.10** List twelve quality criteria for good design.
- 13.11** Reread the description of the FoodCo case study in Case Study B1. Identify any constraints that you think might be imposed on the design of the new system.
- 13.12** Based on the same information try to identify possible measurable objectives for the new FoodCo system.
- 13.13** Agate wants the new system to provide access to the same data from every office around the world. Maintaining a network that is constantly connected between all the offices is considered too expensive, while using a network that dials up remote offices as required would provide response times that are too slow. What kind of compromise solution can you come up with to this problem?
- 13.14** What facilities are typically offered by a DBMS?

Case Study Work, Exercises and Projects

- 13.A** FoodCo requires a data entry screen for entering details of staff holidays. Without knowing what software or hardware is going to be used to develop this data entry screen, list as many features of the design as you can that are not dependent on the implementation platform.
- 13.B** Design applies to a wide range of artefacts, for example cars, buildings, books and packaging. Choose an artefact that you use and try to identify what makes for a good design in this context. Are there aspects that do not apply to systems design? Are there aspects of systems design that should perhaps apply to the design of artefacts that you use?
- 13.C** Find out what laws (if any) exist in your country to protect computer systems against malicious attack from hackers. What implications does the law have for the design of systems?
- 13.D** One aspect of system design is concerned with the setting of standards across the system. Choose a system that you use regularly as part of your work or study and try to identify standards that are applied to the design of the interface. (This could be a software package you use, the library system you use to access your library catalogue or even Windows.)
- 13.E** In Section 13.5.2 we pointed out that some criteria for good quality in design will bring benefits to the designers, while others will bring benefits to the eventual users of the system. Try to decide which of the characteristics discussed in Section 13.5.3 bring benefits to the designers as well as the end users.
- 13.F** Identify a suitable data management approach for the FoodCo case study and, if appropriate, specify the type of DBMS to be used.

Further Reading

For those with an interest in the historical development of systems design, the classics of structured design are Jackson (1975) and Yourdon and Constantine (1979). DeMarco (1979) deals with structured analysis. Two more recent books that are likely to be more easily available are Yourdon (1989) and Page-Jones (1988). Sommerville (2004) and Pressman (2004) both provide detailed discussions of design issues.

If you are interested in an approach to the analysis and design of requirements that is completely different from object-oriented approaches, SSADM (Structured Systems Analysis and Design Method) makes a very clear distinction between requirements analysis, logical design and physical design. A separate stage in SSADM is used to carry out the choice of development environment (Technical System Options). Skidmore et al. (1994), Goodland and Slater (1995) or any other book on SSADM explains the way in which these stages are handled. For more information on SWT which is used in Eclipse, see www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html.

Detailed Design

Learning Objectives

In this chapter you will learn

- how to design classes
- how to design associations
- the impact of integrity constraints on design
- how to design operations.

14.1 Introduction

Detailed design or object design is concerned with the design of the classes, objects and their interactions. It is completed within the overall architecture defined during system design and according to agreed design guidelines and protocols. Detailed design is particularly concerned with the specification of the attribute types, how operations function and how objects are linked to other objects. There are many sources of information that guide the detailed design process. During the design process the analysis models undergo some degree of transformation. There is a commonly accepted view that changes made to analysis artefacts to produce the design model should be kept to a minimum, as the analysis model is a coherent and consistent description of the requirements (Ward and Mellor, 1985; Goldsmith, 1993). However, as the analysis model is transformed into a design model that can guide software construction, further aspects of the system need to be designed and modelled (Section 14.2). Detailed design produces a complete specification of the classes and uses the UML notation described in Section 14.3 for attributes and operation signatures. An important aspect of every class is what attributes (if any) and operations are generally accessible, and this is considered in Section 14.4.

Much design activity focuses on adding detail to the analysis specification and makes no change to the structures identified during analysis. However, there may

be a need to modify the analysis structures for various reasons. The associations modelled in analysis have to be designed to minimize coupling. A series of common designs for associations are discussed in Section 14.5. Analysis will have identified integrity constraints that have to be adhered to by the information system if it is to maintain its consistency and integrity. Various strategies for the design of integrity constraints are examined in Section 14.6. The functionality of the information system is defined in the operation specifications. These have now to be translated into suitable designs (Section 14.7). If the information system is going to utilize relational database technology, it may be appropriate to normalize the data structures and this is considered in detail in Chapter 18. Concurrent activities may be placed in separate subsystems and may require an analysis class to be split into two design classes. As new classes are added to the design model to support data management and the operation of the interface, analysis classes have to be modified to build associations and communication pathways to these new subsystems. As we shall see in Chapter 15, the use of design patterns offers structures and modes of interaction that support non-functional requirements.

14.2 What Do We Add in O-O Detailed Design?

Traditionally, detailed design has been about designing inputs, outputs, processes and file or database structures; these same aspects of the system also have to be designed in an object-oriented system, but they will be organized in terms of classes. During the analysis phase of a project, concepts in the business will have been identified and elaborated in terms of classes, and use cases will have been identified and described. The classes that have been included in the class diagram will reflect the business requirements but they will only include a very simplistic view of the classes to handle the interface with the user, the interface with other systems, the storage of data and the overall co-ordination of the other classes into programs. These classes will be added in design with greater or lesser degrees of detail depending on the hardware and software platform that is being used for the new system.

Different authors describe these additional aspects of the system in different ways. Coad and Yourdon (1991) suggest that in addition to the problem domain or business classes developing a detailed design involves three further elements (they used the word component but not in the sense of component-based development). These are:

- human interface
- data management
- task management.

Coad et al. (1997) propose a slightly different set of additional aspects to be considered in design:

- human interface
- data management
- system interaction.

What should be apparent, whatever the terminology, is that it is necessary to decide on the architecture that will be used and that this will involve the designers in designing classes to handle these aspects of the system. The specification of these aspects of the system, in addition to the design of the analysis classes themselves, is detailed design.

14.2.1 Aspects requiring attention in O-O detailed design

Certain aspects of the detailed design require special attention in the development of object-oriented systems. These include reuse and assignment of responsibilities to classes.

One of the arguments for the use of object-oriented languages is that they promote reuse through encapsulation of functionality and data together in classes and through the use of inheritance. This is not just a programming issue, but one that also affects analysis and design. There is a growing recognition of the need to reuse analysis results in object-oriented systems development. Design reuse already takes place at two levels: first through the use of design patterns, which are discussed in detail in Chapter 15; and second by recognizing during design that business classes that have been identified during analysis may be provided by reusing classes that have already been designed within the organization, or even bought in from outside vendors.

The assignment of responsibilities to classes is an issue that is related to reuse. Larman (2005) highlights this activity as a key task in design. In an object-oriented system, it is important to assign responsibility for operations to the right classes and there is often a choice. In the FoodCo system, there will be a need to produce invoices for customers that include the calculation of Value Added Tax (VAT). (Value Added Tax is a tax used throughout Europe that is applied at each stage of the supply chain and not just as a purchase tax paid by the final end-user or consumer.) The calculation of VAT could be carried out by one of a number of classes in the model (Fig. 14.1):

- **Invoice**—which organizes the total information for the whole sale
- **InvoiceLine**—which contains the detail of each item sold and to which the tax applies
- **Product**—to which different VAT rates may apply
- **TaxRate**—which carries the details of the percentage that applies for each valid rate.

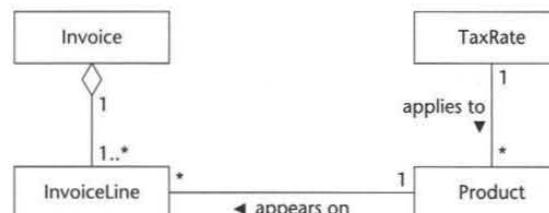


Figure 14.1 Partial class diagram for FoodCo.

If the designer makes the wrong decision, the resulting class will be less reusable and may constrain the design of other classes. If the responsibility for tax calculation is allocated to **Invoice** or **InvoiceLine**, then this has implications for **CreditNote** and **CreditNoteLine**, which may also need to calculate tax. If it is assigned to **Product**, then it cannot be reused in the Agate project where VAT applies to services as well as products. Clearly it needs to be assigned to **TaxRate** in order to maximize the reuse that can be made of the classes in this design.

14.3 Class Specification

14.3.1 Attributes and operation signatures

Attributes

During analysis we have not considered in detail the data types of the attributes, although on occasions it may be useful to record data type information in analysis. For example, an attribute **temperature** may be a floating-point data type if it holds the temperature in Centigrade or it may be an enumerated data type if it holds one of the values, 'hot' or 'cold'. The attribute has a different meaning and would be manipulated differently for each of these data types and it is important to determine during analysis which meaning is appropriate.

Common primitive data types include Boolean (true or false), Character (any alphanumeric or special character), Integer (whole numbers) and Floating-Point (decimal numbers)¹. In most object-oriented languages more complex data types, such as Money, String, Date, or Name, can be constructed from the primitive data types or may be available in standard libraries. An attribute's data type is declared in UML using the following syntax:

```

name ':' type-expression '=' initial-value
      '{property-string}'
  
```

The name is the attribute name, the type-expression is its data type, the initial-value is the value the attribute is set to when the object is first created and the property-string describes a property of the attribute, such as constant or fixed. The characters in single quotes are literals. The attribute name is the only feature of its declaration that is compulsory.

Figure 14.2 shows the class **BankAccount** with attribute data types declared. The attribute **balance** in a **BankAccount** class might be declared with an initial value of zero using the syntax:

```

balance:Money = 0
  
```

The attribute **accountName** might be declared with the property string indicating that it must have a value and may not be null using the syntax:

```

accountName:String {not null}
  
```

Attribute declarations can also specify the multiplicity of the attribute in much the same as the multiplicity of an association may be specified. For example, an

¹ A list of Java primitive data types can be found in Deitel and Deitel (2004).

BankAccount
nextAccountNumber: Integer
accountNumber: Integer
accountName: String (not null)
balance: Money = 0
/availableBalance: Money
overdraftLimit: Money
open(accountName: String): Boolean
close(): Boolean
credit(amount: Money): Boolean
debit(amount: Money): Boolean
viewBalance(): Money
getBalance(): Money
setBalance(newBalance: Money)
getAccountName(): String
setAccountName(newName: String)

Figure 14.2 BankAccount class.

Employee class might include an attribute to hold a list of qualifications that would be declared using the syntax:

```
qualification:String[0..10]
```

This declaration states that the attribute qualification may hold from zero to 10 qualifications.

Operations

Each operation also has to be specified in terms of the parameters that it passes and returns. The syntax used for an operation is:

```
operation name '('parameter-list ')' ':'  
           return-type-expression
```

An operation's *signature* is determined by the operation's name, the number and type of its parameters and the type of the return value if any. The BankAccount class might have a credit() operation that passes the amount being credited to the receiving object and has a Boolean return value. The operation would be defined using the syntax:

```
credit(amount:Money):Boolean
```

A credit() message sent to a BankAccount object could have the format:

```
creditOK = accObject.credit(500.00)
```

where creditOK holds the Boolean return value that is available to the sending object when the credit() operation has completed executing. This Boolean value may be tested to determine whether the credit() operation performed successfully. In an O-O language like Java this would typically be done as

```
try{  
    creditOK = accObject.credit(500.00);  
} catch (UpdateException) { //some error handling; }
```

Notice that this uses Java syntax for handling exceptions. An exception is a way of handling errors in a programming language.

The UML is a modelling language and does not determine what operations should be shown in a class diagram. It provides the notation to use and suggestions about presentation, but it does not tell the analyst or designer what to include and what not to include.

Some authors of books on object-oriented analysis and design give specific guidelines about operations. Coad and Yourdon (1991) talk about services: an object provides services to other objects. They talk about services as being like responsibilities at the system level, but then go on to specify services as what we call the operations of classes. Yourdon (1994) talks about some services being implicit services. These are services to create instances of objects, to modify attributes of instances, to select instances based on some kind of key or identifier, and to delete instances. (These are also known as *primary operations* and include constructor, destructor, get and set operations.) They say that these usually need not be shown on diagrams, as they clutter up the diagrams and make them difficult to read. (However, all operations have to be specified somewhere, and it is important to recognize that a class may have several different constructors for instances of that class.) They also point out that sometimes it is important to be able to see these services. However, this is an issue about the functionality offered by modelling tools rather than methodologies. Ideally, it should be possible to switch off the display of any operation that the analyst or designer does not wish to have displayed in the operations compartment in a class.

An alternative approach would be to follow the way that some CORBA tools work. The IDL2JAVA tool will generate two Java operations for every attribute: one to set the values of the attribute and one to get the value of the attribute. The following example shows the operations generated to set and get the value of the attribute smallUnit in a Currency class:

```
short smallUnit;  
...  
public void smallUnit(short smallUnit) {  
    // implement attribute writer...  
    smallUnit = smallUnit;  
}  
public short smallUnit() {  
    // implement attribute reader...  
    return smallUnit;  
}
```

Note that the naming convention of the operations differentiates between get (reader) and set (writer) operations by the number of parameters and the return type.

The framework for this fragment of code (together with the rest of the Java implementation of this CORBA interface) was generated automatically from the following IDL interface definition:

```
interface Money {  
    attribute long largeUnit;  
    attribute short smallUnit;
```

```

        attribute string format;
        Money Money( in long large, in short small);
        string toString();
    };

```

So, if you have a modelling tool that can generate set and get operations for every attribute, you do not need to include them in your class diagram.

We have shown primary operations on some class diagrams for pedagogic reasons to emphasize their presence and hence have taken a pragmatic approach. In most parts of the book, we have shown those operations that were useful to show. We have usually included the operations that are not primary operations (where primary operations are the create, destroy, get and set operations). Typically, these are the operations that Coad and Yourdon describe as algorithmically complex. We have included some primary operations, usually because they are referred to in the text or some related diagram.

One commonly held approach is normally not to show primary operations on analysis class diagrams as it can be assumed that such functionality is available. During analysis, issues such as the visibility of operations or the precise data types of attributes may not have been finally decided. However, when a design class diagram is completed it may be important to indicate that certain primary operations have public or protected visibility and, as such, these may justifiably be shown on the diagram. Those that are private may be omitted as they do not constitute part of the class's public interface.

Exceptionally, primary operations may usefully be included on analysis class diagrams either if they reflect particular functionality that has to be publicly visible or if it is important to indicate, for example, that more than one constructor is required. A class may need more than one constructor if objects could be instantiated in one of several initial states that require different input parameters. Each constructor would have a different signature.

There are clearly alternative approaches and it is important that appropriate documentation standards are clearly defined at the outset of a project so that the absence of primary operations on a class diagram is not misinterpreted.

14.3.2 Object visibility

The concept of encapsulation was discussed in Chapter 4 and is one of the fundamental principles of object-orientation. During analysis various assumptions have been made regarding the encapsulation boundary for an object and the way that objects interact with each other. For example, it is assumed that the attributes (or more precisely the values of the attributes) of an object cannot be accessed directly by other objects but only via 'get' and 'set' operations (primary operations) that are assumed to be available for each attribute. Moving to design involves making decisions regarding which operations (and possibly attributes) are publicly accessible. In other words we must define the encapsulation boundary.

Figure 14.2 shows the class `BankAccount` with the types of the attributes specified and the operation parameters defined. The class has the attribute `balance`, which, we might assume during analysis, can be accessed directly by the simple primary operations `getBalance()` and `setBalance()`. However, the balance

should be updated through the operations `credit()` and `debit()` that contain special processing to check whether these transactions should be permitted and to ensure that the transactions are logged in an audit trail². In these circumstances, it is important that changes to the value of the `balance` attribute can only occur through the `debit()` and `credit()` operations. The operation `setBalance()` should not be publicly available for use by other classes. Note also that the attribute `availableBalance` is a derived attribute indicated in UML by the symbol '/'. A derived attribute is one whose value can be calculated or determined from the value of other attributes.

Meyer (1997) introduces the term 'secret' to describe those features that are not available in the public interface. Programming languages designate the non-public parts of a class, which may include attributes and operations, in various ways. The four commonly accepted terms³ used to describe *visibility* are listed in Fig. 14.3. Visibility may also be shown as a property string, for example:

```
balance : Money {visibility = private}
```

To enforce encapsulation the attributes of a class are normally designated private (Fig. 14.4). The operation `setBalance()` is also designated private to ensure that objects from other classes cannot access it directly and make changes that are not recorded in the audit trail. Private operations can, of course, be invoked from operations in the same class. Commonly, complex operations are simplified by factoring out procedures into private operations.

In Fig. 14.4 the operation `getBalance()` is assigned protected visibility so that subclasses of `BankAccount` can examine the value of the `balance` attribute. For example, the `debit()` operation might be redefined polymorphically in a `JuniorBankAccount` subclass. The redefined operation would use `getBalance()` to access the balance and check that a debit would not result in a negative balance.

Visibility symbol	Visibility	Meaning
+	Public	The feature (an operation or an attribute) is directly accessible by an instance of any class
-	Private	The feature may only be used by an instance of the class that includes it
#	Protected	The feature may be used either by instances of the class that includes it or of a subclass or descendant of that class
-	Package	The feature is directly accessible only by instances of a class in the same package

Figure 14.3 Visibility.

² In general, an audit trail records the details of transactions so that they can be checked to ensure that, for example, fraud has not occurred.

³ The precise meaning of the non-public categories of visibility depends on the programming language being used. When a designer is determining the visibility of parts of a model, he or she must be aware of the visibility (or scoping) offered by the implementation environment.

BankAccount
- nextAccountNumber: Integer
- accountNumber: Integer
- accountName: String {not null}
- balance: Money = 0
- /availableBalance: Money
- overdraftLimit: Money
+ open(accountName: String): Boolean
+ close(): Boolean
+ credit(amount: Money): Boolean
+ debit(amount: Money): Boolean
+ viewBalance(): Money
getBalance(): Money
- setBalance(newBalance: Money)
getAccountName(): String
setAccountName(newName: String)

Figure 14.4 BankAccount class with visibility specified.

The attribute `nextAccountNumber` in Fig. 14.4 is an example of a class-scope attribute (indicated by underlining). A class-scope attribute occurs only once and is attached to the class, not to any individual object. In this example `nextAccountNumber` holds the account number for the next new `BankAccount` object created. When a new `BankAccount` is created `nextAccountNumber` is incremented by one. The attribute `accountNumber` is an example of an instance-scope attribute (hence no underlining). Each `BankAccount` object has an instance-scope `accountNumber` attribute, which holds its unique account number.

14.4 Interfaces

On occasions a class (or some other component) may present more than one external interface to other classes or the same interface may be required from more than one class. An interface in UML is a group of externally visible (i.e. public) operations. The interface contains no internal structure, it has no attributes, no associations and the implementation of the operations is not defined. Formally, an interface is equivalent to an abstract class that has no attributes, no associations and only abstract operations. Figure 14.5 illustrates two alternative notations for an interface. The simpler of the two UML interface notations is a circle. This is attached by a solid line to the classes that support the interface. For example, in Fig. 14.5 the `Advert` class supports two interfaces, `Manageable` and `Viewable`, that is, it provides all of the operations specified by the interface (and maybe more). The circle notation does not include a list of the operations provided by the interface type, though they should be listed in the repository.

Some modelling tools show a list of the operations under the circle, though this can usually be suppressed. The dashed arrow from the `CreativeStaff` class to the `Manageable` interface circle icon indicates that it uses or needs, at most, the operations provided by the interface.

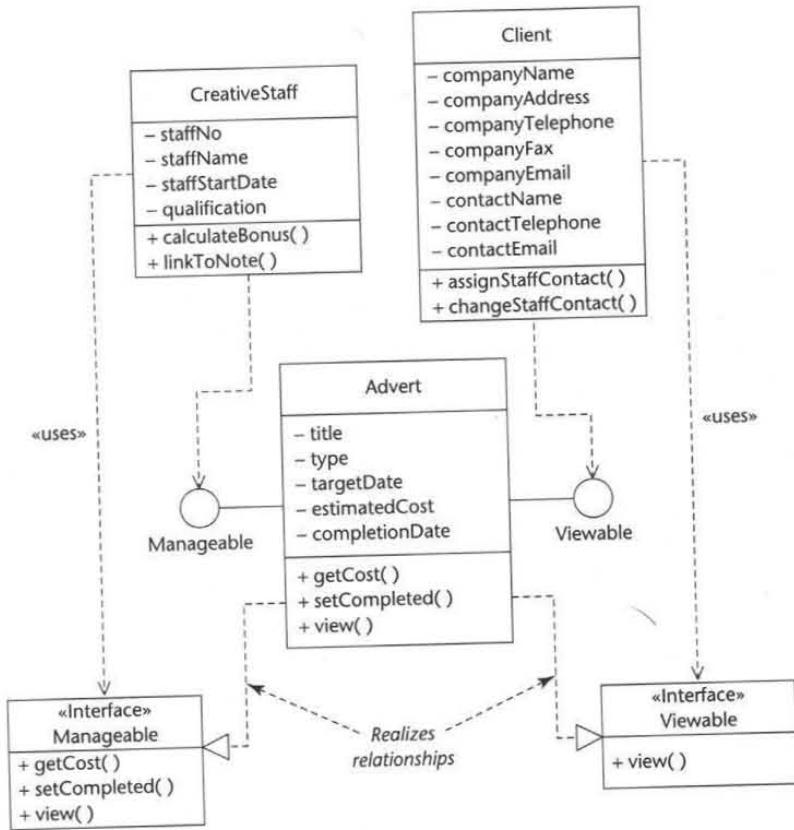


Figure 14.5 Interfaces for the `Advert` class.

The alternative notation⁴ uses a stereotyped class icon. As an interface only specifies the operations and has no internal structure, the attributes compartment is omitted. This notation lists the operations on the diagram. The *realize* relationship, represented by the dashed line with a triangular arrowhead, indicates that the client class (e.g. `Advert`) supports at least the operations listed in the interface (e.g. `Manageable` or `Viewable`). Again the dashed arrow from `CreativeStaff` means that the class needs or uses no more than the operations listed in the interface. The notation used for the *realize* relationship (the triangular arrowhead) is deliberately reminiscent of the notation for inheritance, as in a sense `Advert` inherits the `Manageable` interface. This concept can be implemented using the interface programming language construct in Java.

⁴ Normally only one of these notations would be used in a diagram.

14.5 Designing Associations

An association between two classes indicates the possibility that links will exist between instances of the classes. The links provide the connections necessary for message passing to occur. When deciding how to implement an association it is important to analyse the message passing between the objects tied by the link.

14.5.1 One-to-one associations

In Fig. 14.6 objects of the class *Owner* need to send messages to objects of the class *Car* but not vice versa. This particular association may be implemented by placing an attribute to hold the object identifier (some authors prefer to use the term object reference) for the *Car* class in the *Owner* class. Thus *Owner* objects have the *Car* object identifier and hence can send messages to the linked *Car* object. As a *Car* object does not have the object identifier for the *Owner* object, it cannot send messages to the *Owner* object. The *owns* association is an example of a one-way association: the arrowhead on the association line shows the direction along which it may be navigated.

So before an association can be designed it is important to decide in which direction or directions messages may be sent. (If messages are not sent in either direction along an association, then the need for its existence should be questioned.) Essentially we are determining the navigability of the association.

In general an association between two classes A and B should be considered with the questions:

1. Do objects of class A have to send messages to objects of class B?
2. Does an A object have to provide some other object with B object identifiers?

If either of these questions is answered ‘yes’, then A objects need B object identifiers. However, if A objects get the required B object identifiers as parameters in incoming messages, A objects need not remember the B object identifiers. Essentially, if an object needs to send a message to a destination object, it must have the destination object’s identifier either passed as a parameter in an incoming message just when it is required, or the destination object’s identifier must be

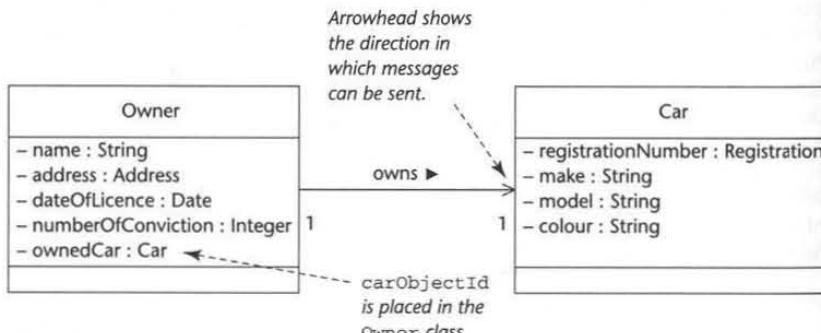


Figure 14.6 One-way one-to-one association

stored in the sending object. An association that has to support message passing in both directions is a two-way association. A two-way association is indicated with arrowheads at both ends⁵. As discussed earlier, it is important to minimize the coupling between objects. Minimizing the number of two-way associations keeps the coupling between objects as low as possible.

14.5.2 One-to-many associations

In Fig. 14.7, objects of the class *Campaign* need to send messages to objects of the class *Advert* but not vice versa. If the association between the classes was one-to-one, the association could be implemented by placing an attribute to hold the object identifier for the *Advert* class in the *Campaign* class. However, the association is in fact one-to-many and many *Advert* object identifiers need to be tied to a single *Campaign* object. The object identifiers could be held as a simple one-dimensional array in the *Campaign* object, but program code would have to be written to manipulate the array. Another way of handling the group of *Advert* object identifiers, which is more amenable to reuse, is to place them in a separate

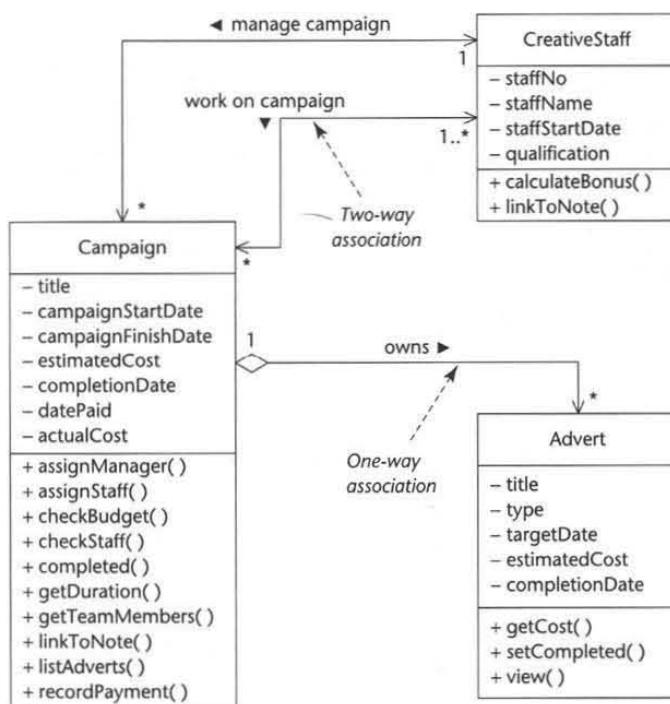


Figure 14.7 Fragment of class diagram for the Agate case study.

⁵ In UML a two-way association may be represented by drawing the association without the navigability arrowheads. However, an association without arrowheads may also represent an uncommitted association, that is, an association for which navigability is not yet decided.

object, a collection object that has operations to manage the object identifiers and that behaves rather like an index of adverts for the Campaign object. This is shown in the class diagram fragment in Fig. 14.8. There will be many instances of the collection class, as each Campaign object has its own collection of Advert object identifiers. Notice that the AdvertCollection class has operations that are specifically concerned with the management of the collection. The `findFirst()` operation returns the first object identifier in the list and the `getNext()` gets the next object identifier in the list.

When a Campaign object wants to send a message to each of its Advert objects the Campaign object first sends a `findFirst()` message to the class to get the first object identifier. The Campaign object can now send a message to its first Advert object. The Campaign then uses `getNext()` to get the next object identifier from the collection class and sends the message to the next Advert object. The Campaign object can then iterate through the collection of object identifiers and send the message to each of the Advert objects in turn⁶.

Figure 14.9 shows the sequence diagram for the interaction that would enable the Campaign object to prepare a list of its adverts with their titles. The Campaign object holds the object identifier of the collection class so that it can send messages

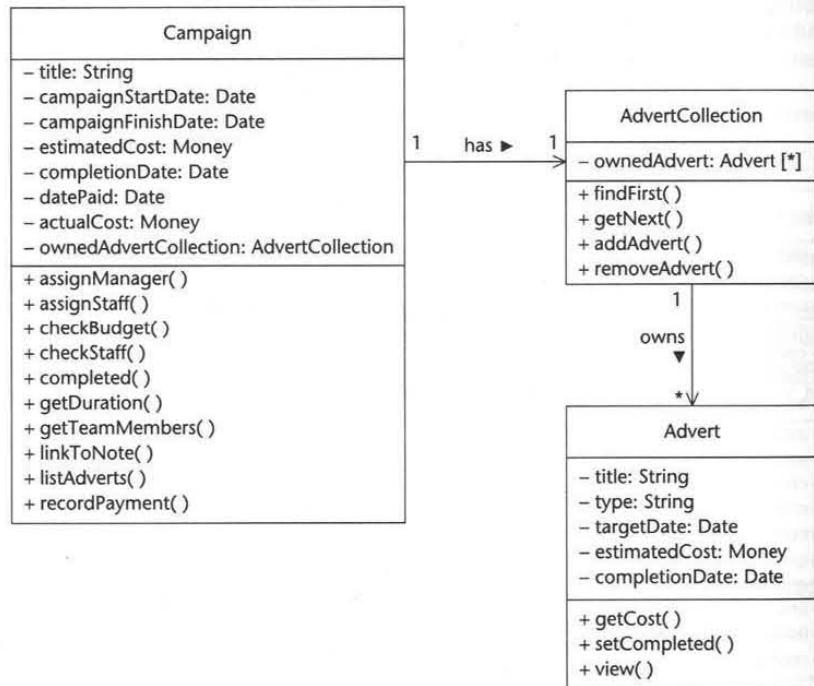


Figure 14.8 One-to-many association using a collection class.

⁶ O-O languages and class libraries provide various features to handle collection classes. For example, in Java an Iterator could be applied to a collection class instance to provide the ability to iterate through the elements of the collection.

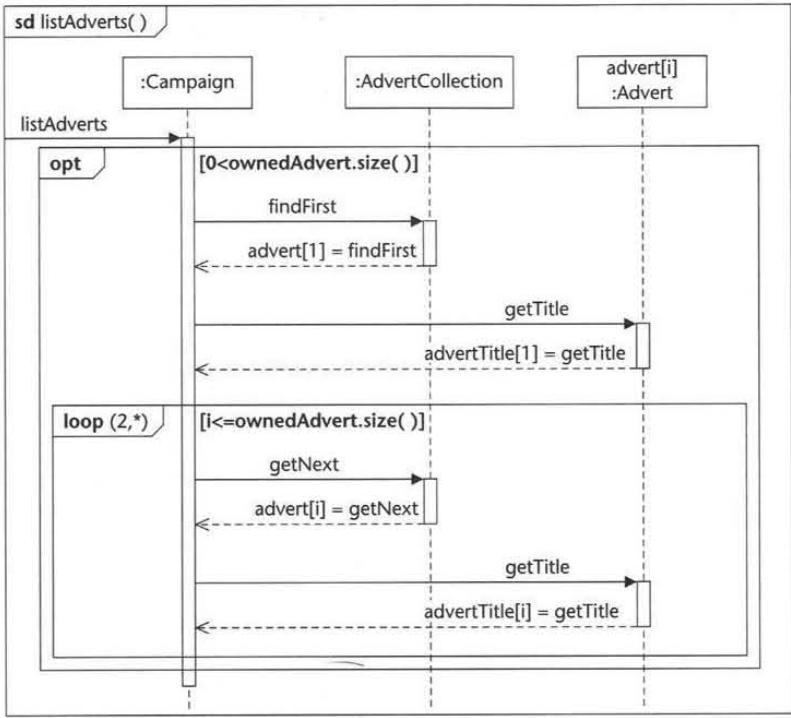


Figure 14.9 Sequence diagram for `listAdverts()`.

to it. As an Advert object does not have the object identifier for the Campaign object to which it belongs, it cannot send messages to the Campaign object. The interaction constraints in `listAdverts()` use the term `ownedAdvert.size()`, where `ownedAdvert` is the name of the attribute (in `AdvertCollection`) that holds the collection of object identifiers and `size()` is an OCL function that gives the number of elements in a collection. The `opt` interaction operator specifies that the interaction fragment will only execute if the constraint is satisfied—in this example, if there is at least one advert associated with the Campaign.

14.5.3 Many-to-many associations

The design of the many-to-many association work on campaign between CreativeStaff and Campaign (see Fig. 14.7) follows the principles described above. Assuming this is a two-way association, each Campaign object will need a collection of CreativeStaff object identifiers and each CreativeStaff object will need a collection of Campaign object identifiers. The designed association with the collection classes is shown in Fig. 14.10. Both the CreativeStaff and Campaign classes contain an attribute to hold the object identifiers of their respective collection classes.

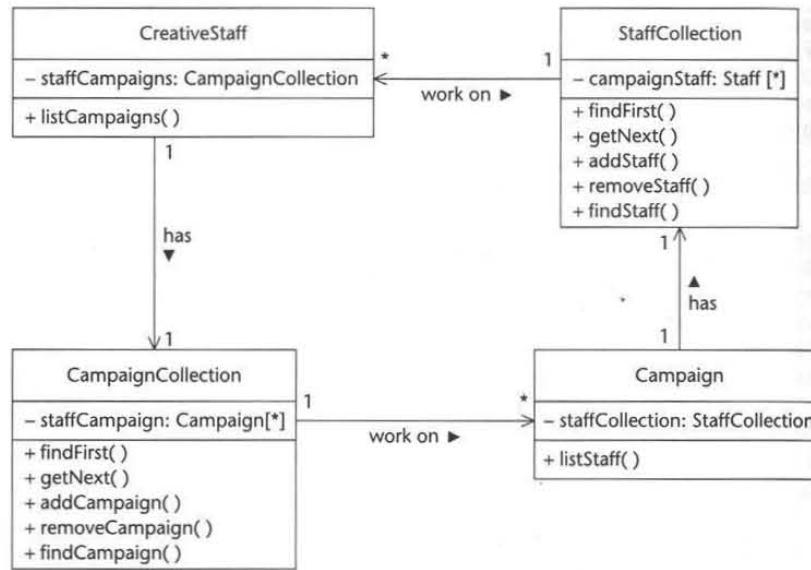


Figure 14.10 Two-way many-to-many association.

Collection classes can be designed to provide additional support for object navigation. For example, if there is a requirement to find out if an employee works on a campaign with a particular title, a message may be sent from the Creative Staff object to each Campaign object the employee works on to get its title until either a match is found or the end of the collection is reached. Two separate messages are required to access each Campaign object. So, if an employee works on four campaigns, a maximum of eight messages must be sent to find the campaign. In general, if the employee works on N campaigns a maximum of $2N$ messages must be sent.

An alternative search approach uses a `findCampaign()` operation in the collection class. This operation may be designed either to access an index in the collection object itself or to take responsibility to cycle through the Campaign objects searching for a title match. In the first case only the `findCampaign()` message is needed to find the campaign as the collection class indexes the campaigns by title. In the second case the collection object may have to send a maximum of N messages (one for each campaign on which the employee works). So in either case the inclusion of the `findCampaign()` operation in the collection class reduces the message passing required.

The management of object identifiers using collection classes may appear to increase appreciably the development effort required. In fact, the opposite is generally the case, as object-oriented languages normally provide collection classes of various types with standard collection management operations. The standard collection classes may offer various forms of indexing. They may also be subclassed to add additional application functionality. For example, Java provides, among others, standard List, Stack, Hashtable and Dictionary collection classes that may be subclassed to add application-specific behaviour (Deitel and Deitel, 2004).

14.5.4 Keeping classes to a minimum

The association between Campaign and AdvertCollection is one-to-one (commonly the case with collection classes), and this suggests that one implementation strategy is to place the collection class inside the Campaign object. This approach generally produces more complex classes and limits extensibility. However, in this case it is likely that collection class behaviour will largely be provided by a library class or by a feature of the development language being used, so the issue of increased complexity may not be so significant. The problem of any reduction in extensibility is also less significant in this case: as only a Campaign object would want to know which Advert objects are tied to it, and any request to access a Campaign's Adverts would be directed to the Campaign first. So, on balance, placing the collection class inside the Campaign class is a sensible design decision, and using library collection classes where possible maximizes reuse and reduces development effort. Clearly, if another class, apart from Campaign, needs to use the list independently of the Campaign class then it is more appropriate to keep the collection class separate.

14.6 Integrity Constraints

Systems analysis will have identified a series of integrity constraints that have to be enforced to ensure that the application holds data that is mutually consistent and manipulates it correctly. These integrity constraints come in various forms:

- *referential integrity*, which ensures that an object identifier in an object is actually referring to an object that exists
- *dependency constraints*, which ensure that attribute dependencies, where one attribute may be calculated from other attributes, are maintained consistently
- *domain integrity*, which ensures that attributes only hold permissible values.

14.6.1 Referential integrity

The concept of referential integrity as applied to a relational database management system (see Chapter 18) is discussed by Howe (2001). Essentially the same principles apply when considering references between objects. In Fig. 14.7 the association manage campaign between CreativeStaff and Campaign is two-way, and an object identifier called `campaignManagerId`, which refers to the particular CreativeStaff object that represents the campaign manager, is needed in Campaign. (CreativeStaff needs a collection of Campaign object identifiers to manage its end of the association.) To maintain referential integrity the system must ensure that the attribute `campaignManagerId` either is null (not referencing any object) or contains the object identifier of a CreativeStaff object that exists. In this particular case the association states that a Campaign must have a CreativeStaff instance as its manager, and it is not correct to have a Campaign with a null `campaignManagerId` attribute. In order to enforce this constraint, the constructor for Campaign needs as one of its parameters the object identifier of the CreativeStaff object that represents the campaign manager, so that the `campaignManagerId` attribute can be instantiated with a valid object identifier.

Problems in maintaining the referential integrity of a Campaign may occur during its lifetime. For instance, the campaign manager, Nita Rosen, may leave the company to move to another job and Nita's CreativeStaff object will then be deleted⁷. Referential integrity is maintained by ensuring that the deletion of a CreativeStaff object that is a campaign manager always involves allocating a new campaign manager. The task of invoking the operation `assignManager()` is included in the `CreativeStaff` destructor, and it will request the object identifier of the new campaign manager. Similarly, any attempt to remove the current campaign manager from a Campaign must always involve allocating the replacement.

The multiplicity of exactly one represents a strong integrity constraint for the system. In the example just discussed, it seems to be appropriate that a campaign should always have a manager, even when it has just been created. However, great care should be taken when assigning a multiplicity of exactly one (or in general a minimum of one) to an association, as the consequences in the implemented system can be quite dramatic. Let us imagine that the campaign manager, Nita Rosen, does leave Agate but that there is no replacement campaign manager available. The strict application of the integrity constraint implied by the `manageCampaign` association means that integrity can only be enforced if all the campaigns that Nita managed are deleted. Of course, because each Advert must be linked to a Campaign, all the Advert objects for each of the Campaigns must also be deleted in order to maintain referential integrity. This is an example of a cascade delete: deleting one object results in the deletion of many objects as referential integrity is applied. In the case of Agate, deleting the information about Nita's campaigns and their adverts would be disastrous. There are two solutions: either the constraint on the association is weakened by changing the cardinality to zero or one, or when Nita leaves a dummy `CreativeStaff` object is created and allocated as campaign manager to Nita's campaigns. Although the second solution is a fix, it has the advantage of providing an obvious place-holder, highlighting the problem of unmanaged campaigns but maintaining the integrity constraint. Of course, the minimum of one multiplicity was assigned to the association to reflect company policy that a campaign must always have a manager, but this may not be a viable solution from a business perspective.

14.6.2 Dependency constraints

Attributes are dependent upon each other in various ways. These dependencies may have been identified during analysis and must now be dealt with during design. A common form of dependency occurs when the value of one attribute may be calculated from other attributes. For instance, a requirement to display the total advertising cost may be satisfied either by storing the value in the attribute `totalAdvertCost` in the `Campaign` class or by calculating the value every time it is required. The attribute `totalAdvertCost` is a derived attribute and its value is calculated by summing the individual advert costs. Placing the derived attribute in the class reduces the processing required to display the total advertising cost as it does not require calculation. On the other hand, whenever the cost of an advert changes, or an advert is either added to or removed from the campaign, then the

⁷ This is a somewhat simplistic view. When an employee leaves the company the `CreativeStaff` object would be set to the state `Ex-employee`. In this case, although the object still exists, it is not appropriate for it to be referenced as a campaign manager.

attribute `totalAdvertCost` has to be adjusted so that it remains consistent with the attributes upon which it depends. An example of the UML symbol ('/') used to indicate that a modelling element (attribute or association) is derived is shown in Fig. 14.4.

In order to maintain the consistency between the attributes, any operation that changes the value of an Advert's cost must trigger an appropriate change in the value of `totalAdvertCost` by sending the message `adjustCost()` to the `Campaign` object. The operation `adjustCost()` is an example of a *synchronizing operation*. The operations that have to maintain the consistency are `setAdvertCost()` and the `Advert` destructor. When a new advert is created, the constructor would use `setAdvertCost()` to set the advert cost. This would invoke `adjustCost()` and hence ensure that the `totalAdvertCost` is adjusted. So any change to an Advert's cost takes more processing if the derived attribute `totalAdvertCost` is used. Thus one part of the system executes more quickly while another part executes more slowly. Generally, it is easier to construct systems without derived attributes, as this obviates the need for complex synchronizing operations. Derived attributes should only be introduced if performance constraints cannot be satisfied without them. If performance is an issue, then one of the skills needed in design is how to optimize the critical parts of the system without making the other parts of the system inoperable.

Another form of dependency occurs where the value of one attribute is constrained by the values of other attributes. For example, let us assume that the sum of the total advertising cost, the staff costs, the management costs and the ancillary costs must not exceed the campaign's authorized budget. Any changes to these values must check that the authorized budget is not exceeded by the sum of the costs. If a change to any of these dependent values would cause this constraint to be broken, then some action should be taken. There are two possibilities. Either the system prohibits any change that violates the constraint and an exception is raised, or it permits the change and an exception is raised. It is most likely that the violation of the constraint would occur as a result of an attempt interactively to change one of the constrained values and the exception raised would be a warning message to the user. If it is considered permissible for the constraint to be broken, then all access to or reporting about these values should produce a warning message to the user.

Dependency constraints can also exist between or among associations. One of the simplest cases is shown in Fig. 14.11 where the `chairs` association is a subset of the `is a member of` association.

This constraint is stating that the chair of a committee must be a member of the committee, and it can be enforced by placing a check in the `assignChair()` operation in `Committee` to confirm that the `Employee` object identifier passed as a parameter is already in the collection class of committee members. More complex constraints, which require several associations, may also exist. Derived associations may also be introduced to improve performance if absolutely necessary and, as in the case of derived attributes, synchronizing operations are needed to ensure that the derived links are consistent with the links on which they depend.

14.6.3 Domain integrity

Domain integrity is concerned with ensuring that the values an attribute takes are

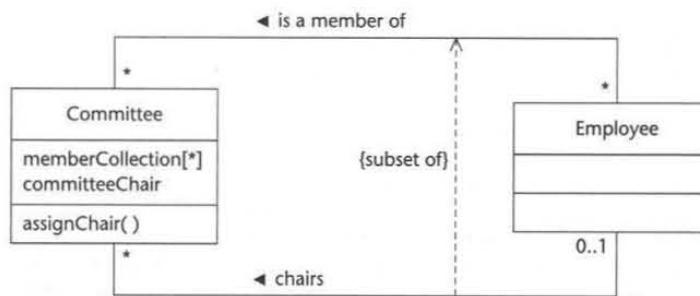


Figure 14.11 Constraints between associations.

from the appropriate underlying domain. For instance, the attributes from the Cost domain might reasonably be non-negative decimal values with two decimal places. These constraints may be viewed as an extended form of those implied by data types. The necessary integrity checking code is normally placed in the 'set' operations or in any interactive interface that permits the entry of values.

14.7 Designing Operations

The design of operations involves determining the best algorithm to perform the required function. In the simplest case, primary operations require little design apart from the inclusion of code to enforce integrity checks. For more complex operations, algorithm design can be an involved process. Various factors constrain algorithm design:

- the cost of implementation
- performance constraints
- requirements for accuracy
- the capabilities of the implementation platform.

Generally it is best to choose the simplest algorithm that satisfies these constraints, as this makes the operation easier to implement and easier to maintain. Rumbaugh et al. (1991) suggest that the following factors should be considered when choosing among alternative algorithm designs.

- *Computational complexity.* This is concerned with the performance characteristics of the algorithm as it operates on increasing numbers of input values. For example, the bubble sort algorithm has an execution time that is proportional to $N \times N$ where N is the number of items being sorted.
- *Ease of implementation and understandability.* It is generally better to sacrifice some performance to simplify implementation.
- *Flexibility.* Most software systems are subject to change and an algorithm should be designed with this in mind.

- *Fine-tuning the object model.* Some adjustment to the object model may simplify the algorithm and should be considered.

Designing the main operations in a class is likely to highlight the need for lower-level private operations to decompose complex operations. This process is much the same as traditional program design. Techniques such as step-wise refinement (Budgen, 1994) or structure charts (Yourdon and Constantine, 1979) may well be used to good effect. UML offers activity diagrams (see Chapters 5 and 10) as a technique both to document and to design operations. In circumstances where high levels of formality are required in operation design, formal specification techniques such as Z or VDM may be used⁸.

Responsibilities identified during analysis may map onto one or more operations. The new operations that are identified need to be assigned to classes. In general, if an operation operates on some attribute value then it should be placed in the same class as the attribute. On occasions, a particular operation may modify attributes in more than one class and could sensibly be placed in one of several classes. In choosing where to locate the operation, one view is that minimizing the amount of object interaction should be a major criterion, while another significant criterion is simplicity. However, in some cases it is not a clear-cut decision.

During analysis use case realization, control classes are introduced to control the execution of use cases. Typically these control classes may be the best place for operations that are particular to the use case or that have no obvious owning entity class. Some designers may choose to allocate control class responsibility to boundary or entity classes during design to achieve performance or other implementation requirements. However, this results in boundary or entity classes that have less well focused functionality (lower class cohesion) and can make maintenance more difficult. This yet again reflects the trade-offs that have to be made during design.

14.8 Summary

Object design is concerned with the detailed design of the system and is conducted within the architectural framework and design guidelines specified during system design. The detailed design process involves determining the data types of the attributes and defining the operation signatures. Interfaces may be specified. Associations have to be designed to support the message passing requirements of the operations. This involves determining how best to place object references in the classes. The application of integrity constraints is included in the design of operations. Operations have to be designed to enforce these integrity constraints. If derived attributes are included in any of the classes then synchronizing operations are required to maintain their consistency. The detailed design process is guided by a series of criteria that incorporate the fundamental principles of coupling and cohesion. Normalization is another important design technique and this is discussed in Chapter 18.

⁸ Z and VDM are formal languages that can be used to specify a system using mathematical entities such as sets, relations and sequences.

Review Questions

- 14.1** What aspects of the system are added to the class diagram(s) in object-oriented detailed design?
- 14.2** What levels of visibility may be assigned to an attribute or an operation?
- 14.3** Why should attributes be private?
- 14.4** What are primary operations?
- 14.5** What is a class-scope attribute?
- 14.6** What does the term interface mean in UML?
- 14.7** How can collection classes be used when designing associations?
- 14.8** Under what circumstances should a collection of object references be included in a class?
- 14.9** How can referential integrity be enforced in an object-oriented system?
- 14.10** Under what circumstances should derived attributes be used?
- 14.11** Under what circumstances should derived associations be used?

Case Study Work, Exercises and Projects

- 14.A** Specify the attribute types and the operation signatures for the class ProductionLine in the FoodCo case study.
- 14.B** For each association in which ProductionLine participates allocate object identifiers to design the association.
- 14.C** Show how referential integrity can be enforced for the associations designed in Exercise 14.B.
- 14.D** For an object-oriented programming language of your choice investigate the language features available to support the use of collection classes.

Further Reading

Rumbaugh et al. (1991) and Booch (1994) provide good advice on object design. Budgen (1994) offers descriptions of various design techniques. Meyer (1997) contains a comprehensive discussion of object-oriented software design and provides many interesting insights. Sommerville (2004) and Pressman (2004) both provide detailed discussions of design issues. There is an excellent treatment of normalization in Howe (2001) and detailed discussion of integrity constraints from a database perspective in Date (2003).

Design Patterns

Learning Objectives

In this chapter you will learn

- what types of patterns have been identified in software development
- how to apply design patterns during software development
- the benefits and difficulties that may arise when using patterns.

15.1 Introduction

Successful software development relies on the knowledge and expertise of the developer, among other factors. These are built up and refined during the developer's working life. A systems analyst or software engineer applies potential solutions to development problems, monitors their success or failure and produces more effective solutions on the next occasion. It is in the nature of software development that the same problems tend to recur, though in different contexts. Individual developers may expend a great deal of development time and effort on solving these recurring problems from first principles each time they occur, and the solution that each individual produces may not be the most appropriate that could be achieved. This can result in information systems that are inflexible, difficult to maintain or inefficient or that possess some other undesirable features. The perpetuation of this cycle of reinventing the wheel has been encouraged partly because there were no effective mechanisms for communicating successful solutions to recurring problems.

Patterns provide a means for capturing knowledge about problems and successful solutions in software development. Experience that has been gained in the past can be reused in similar situations, thus reducing the effort required to produce systems that are more resilient, more effective and more flexible.

Patterns have been introduced in Section 8.4 where we considered analysis patterns in particular, and in Chapter 12 we discussed several architectural

patterns. In Section 15.2 we consider further detailed aspects of patterns in general. Patterns are documented in various formats and these are considered in Section 15.3. Although patterns have been applied to many aspects of software development we focus our attention on some better known design patterns (Section 15.4) to illustrate their application. Guidelines for using patterns are discussed in Section 15.5. Patterns have many benefits but there are also potential disadvantages associated with their use. These advantages and disadvantages are discussed in the context of the examples used in Section 15.6.

15.2 Software Development Patterns

15.2.1 Frameworks

Patterns have the potential to be an important aspect of the reuse strategy within an organization. *Frameworks* also offer opportunities for reuse. There can be confusion between patterns and frameworks but there are important differences. Frameworks are partially completed software systems that may be targeted at a specified type of application, for example sales order processing. An application system tailored to a particular organization may be developed from the framework by completing the unfinished elements and adding application- and organization-specific elements. This may involve the specialization of classes, the implementation of operations and perhaps the addition of new classes. Essentially the framework is a reusable mini-architecture that provides structure and behaviour common to all applications of this type.

The major differences between patterns and frameworks can be summarized as follows.

- Patterns are more abstract and general than frameworks. A pattern is a description of the way that a type of problem can be solved, but the pattern is not itself a solution.
- Unlike a framework, a pattern cannot be directly implemented in a particular software environment. A successful implementation is only an example of a design pattern.
- Patterns are more primitive than frameworks. A framework can employ several patterns but a pattern cannot incorporate a framework.

15.2.2 Pattern catalogues and languages

Patterns are grouped into catalogues and languages. A *pattern catalogue* is a group of patterns that are related to some extent and may be used together or independently of each other. The patterns in a *pattern language* are more closely related, and work together to solve problems in a specific domain. For example, Cunningham (1995) documented the 'Check Pattern Language of Information Integrity', which consists of eleven patterns that address issues of data validation. All were developed from his experience of developing interactive financial systems in Smalltalk.

One of these patterns, Echo, describes how data input should be echoed back to the user after it has been modified and validated by the information system (since

Cunningham uses the Model–View–Controller structure he talks about this in terms of changes made by the model). Typically, users enter small batches of values and then look at the screen to check that they have been correctly entered. The sequence in which a user can enter data into fields may not be fixed and so validation feedback should be given one field at a time. For example, a user enters a value as 5.236. This might be echoed back by the system as 5.24 (correctly rounded to two decimal places). The user receives direct visual feedback that the value has been accepted and how it has been modified.

15.2.3 Software development principles and patterns

Patterns are intended to embody good design practice and hence are based upon sound software development principles, many of which have been identified since the early days of software development and applied within development approaches that are not object-orientated. Buschmann et al. (1996) suggest that the following are the key principles that underlie patterns:

- abstraction
- encapsulation
- information hiding
- modularization
- separation of concerns
- coupling and cohesion
- sufficiency, completeness and primitiveness
- separation of policy and implementation
- separation of interface and implementation
- single point of reference
- divide and conquer (this means breaking a complex problem into smaller, more manageable ones).

These principles should already be familiar to the reader as they have been discussed in earlier chapters.

15.2.4 Patterns and non-functional requirements

Patterns address the issues that are raised by non-functional requirements (Chapters 6 and 12). Buschmann et al. (1996) identify these as the important non-functional properties of a software architecture:

- changeability
- interoperability
- efficiency
- reliability
- testability
- reusability.

These properties may be required of a complete system or a part of a system. For example, a particular set of functional requirements may be seen as volatile and

subject to change. It is important to develop a structure for these requirements that can cope with change. Another requirement may be that a particular aspect of an application must be highly reliable. Again, this requirement must be met by the design.

15.3 Documenting Patterns—Pattern Templates

15.3.1 Template contents

Patterns may be documented using one of several alternative templates. The *pattern template* determines the style and structure of the pattern description, and these vary in the emphasis they place on different aspects of patterns. The differences between pattern templates may mirror variations in the problem domain but there is no consensus as to the most appropriate template even within a particular problem domain. Nonetheless it is generally agreed that a pattern description should include the following elements (at least implicitly).

Name. A pattern should be given a meaningful name that reflects the knowledge embodied by the pattern. This may be a single word or a short phrase. These names become the vocabulary for discussing conceptual constructs in the domain of expertise. For instance, the names of three of the Gamma design patterns¹, Bridge, Mediator and Flyweight, give a strong indication of how they are intended to work.

Problem. This is a description of the problem that the pattern addresses (the intent of the pattern). It should identify and describe the objectives to be achieved, within a specified context and constraining forces. For example, one problem might be concerned with producing a flexible design, another with the validation of data. The problem can frequently be written as a question, for example 'How can a class be constructed that has only one instance and can be accessed globally within the application?'. This question expresses the problem addressed by the Singleton pattern (discussed in Section 15.4.2).

Context. The context of the pattern represents the circumstances or preconditions under which it can occur. The context should provide sufficient detail to allow the applicability of the pattern to be determined.

Forces. The forces embodied in a pattern are the constraints or issues that must be addressed by the solution. These forces may interact with and conflict with each other, and possibly also with the objectives described in the problem. They reflect the intricacies of the pattern.

Solution. The solution is a description of the static and dynamic relationships among the components of the pattern. The structure, the participants and their collaborations are all described. A solution should resolve all the forces in the given context. A solution that does not resolve all the forces fails.

15.3.2 Other aspects of templates

A pattern template may be more extensive than the elements described above. Some other features that have figured in pattern templates are:

- an example of the use of a pattern that serves as a guide to its application

- the context that results from the use of the pattern
- the rationale that justifies the chosen solution
- related patterns
- known uses of the pattern that validate it (some authors suggest that until the problem and its solution have been used successfully at least three times—the *rule of three*—they should not be considered as a pattern)
- a list of aliases for the pattern ('also known as' or AKA)
- sample program code and implementation details (commonly used languages include C++, Java and Smalltalk).

Gamma et al. (1995) use a template that differs from that described above. Although very detailed, this does not explicitly identify the forces. Cunningham's (1995) 'Check Pattern Language of Information Integrity' is described in the Portland Form². Variations in template style and structure make it difficult to compare patterns that are documented in different templates, and this limits their reusability since it is more difficult to use a pattern that is documented in an unfamiliar template.

15.4 Design Patterns

15.4.1 Types of design pattern

In the catalogue of 23 design patterns presented by Gamma, Helm, Johnson and Vlissides (1995) patterns are classified according to their scope and purpose. The four authors are known as the Gang of Four and the patterns they describe are sometimes referred to as the GOF patterns or the Gamma patterns. The GOF patterns are categorized as *creational*, *structural* or *behavioural* to reflect their different purposes. An example of each of these categories is described in the following sections. The scope of a pattern may be primarily at either the class level or at the object level. Patterns that are principally concerned with objects describe relationships that may change at run-time and hence are more dynamic. Patterns that relate primarily to classes tend to be static and identify relationships between classes and their subclasses that are defined at compile-time. The GOF patterns are generally concerned with increasing the ease with which an application can be changed, through reducing the coupling among its elements and maximizing their cohesion. The patterns are based on principles of good design, which include maximizing encapsulation and the substitution of composition for inheritance wherever possible. Using composition as a design tactic produces composite objects whose component parts can be changed, perhaps dynamically under program control, hence resulting in a highly flexible system. Nonetheless, patterns will frequently use both inheritance and composition to achieve the desired result.

Changeability involves several different aspects (Buschmann et al., 1996): maintainability, extensibility, restructuring and portability. Definitions of these terms vary but we use the following.

- *Maintainability* is concerned with the ease with which errors in the information system can be corrected.

¹ Also known as GOF patterns, see Section 15.4.1.

² Named after Portland, Oregon where it originated, this is basically free-format text.

- *Extensibility* addresses the inclusion of new features and the replacement of existing components with new improved versions. It also involves the removal of unwanted features.
- *Restructuring* focuses on the reorganization of software components and their relationships to provide increased flexibility.
- *Portability* deals with modifying the system so that it may execute in different operating environments, such as different operating systems or different hardware.

15.4.2 Creational patterns

A creational design pattern is concerned with the construction of object instances. In general, creational patterns separate the operation of an application from how its objects are created. This decoupling of object creation from the operation of the application gives the designer considerable flexibility in configuring all aspects of object creation. This configuration may be dynamic (at run-time) or static (at compile-time). For example, when dynamic configuration is appropriate, an object-oriented system may use composition to make a complex object by aggregating simpler component objects. Depending upon circumstances different components may be used to construct the composite object and, irrespective of its components, the composite object will fulfil the same purpose in the application. A simple analogy illustrates this. A systems development department in an organization will vary in its composition from time to time. When permanent staff are on holiday contract staff may be employed to perform their roles. This enables the department to offer the same service to the organization at all times.

Creating composite objects is not simply a matter of creating a single entity but also involves creating all the component objects. The separation of the creation of a composite object from its use within the application provides design flexibility. By changing the method of construction of a composite object, alternative implementations may be introduced without affecting the current use.

Singleton pattern

As an example we consider the creational pattern, Singleton, which can be used to ensure that only one instance of a class is created. In order to understand the use of the pattern we need to consider the circumstances under which a single instance may be required.

The Agate campaign management system needs to hold information regarding the company. For example, its name, its head office address and the company registration details need to be stored so that they can be displayed in all application interfaces and printed on reports. This information should be held in only one place within the application but will be used by many different objects. One design approach would be to create a global data area that can be accessed by all objects, but this violates the principle of encapsulation. Any change to the structure of the elements of global data would require a change to all objects that access them. The creation of a Company class overcomes this problem by encapsulating the company attributes (Fig. 15.1). These are then accessible to other objects through the operations of the Company object. But there is still a problem with this proposal. An object that wants to use the Company object needs to know the Company

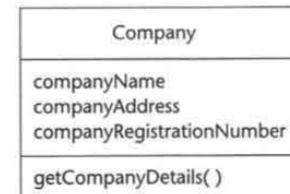


Figure 15.1 Company class for the Agate case study.

object's identifier so that it can send messages to it. This suggests that the Company object identifier should be globally accessible—but again this is undesirable since it violates encapsulation.

Some object-oriented programming languages (including Java and C++) provide a mechanism that enables certain types of operations to be accessed without reference to a specified object instance. These are called *class* (or *static*) *operations*. This mechanism offers a solution to the problem of providing global access without the need to globally define the object identifier. For example, a static operation `getCompanyInstance()` can be defined in such a way that it will provide any client object with the identifier for the Company instance. This operation can be invoked by referencing the class name as:

```
Company.getCompanyInstance()
```

When a client object needs to access the Company object it can send this message to the Company class and receive the object identifier in reply. The client object can now send a `getCompanyDetails()` message to the Company object.

There is one further aspect to this design problem. It is important that there should be only one instance of this object. To ensure system integrity the application should be constructed so that it is impossible to create more than one. This aspect of the problem can be solved by giving the Company class sole responsibility for creating a Company object. This is achieved by making the class constructor private so that it is not accessible by another object. The next issue that needs to be addressed is the choice of an event that causes the creation of the company object. Perhaps the simplest approach is to create the Company object at the moment when it is first needed. When the Company class first receives the message `getCompanyInstance()` this can invoke the Company class constructor. Once the Company object has been created, the object identifier is stored in the class (or static) attribute `companyInstance` so that it can be passed to any future client objects. When the Company object is first created its attribute values may, for example, be read in from a database or a file.

So far we have produced a design for the Company class (Fig. 15.2) that provides a single global point of access via the class operation `getCompany Instance()` and that also ensures that only one instance is created.

A simple version of the logic for the `getCompanyInstance()` operation is:

```
If (companyInstance == null)
{
    companyInstance = new Company()
} return companyInstance
```

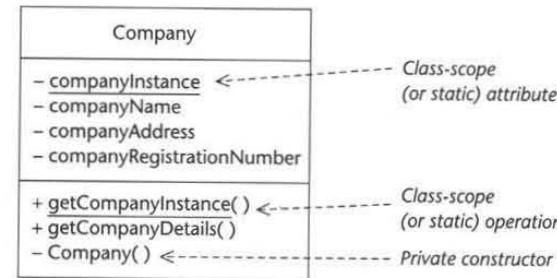


Figure 15.2 Company class with class-scope operation and attribute.

The sequence diagram in Fig. 15.3 shows the interaction Get company name for display. In this it can be seen that the :RequestingObject receives the object identifier (held in companyInstance) for the Company object in the same way whether the object has just been created or is already in existence.

The design may need to accommodate further requirements. Since Agate operates as a separate company in each country (each owned by the same multinational), variations in company law from country to country may necessitate different company registration details to be recorded for each country. This suggests a requirement for different types of Company class each with its own variation of the registration details. The creation of a separate subclass for each

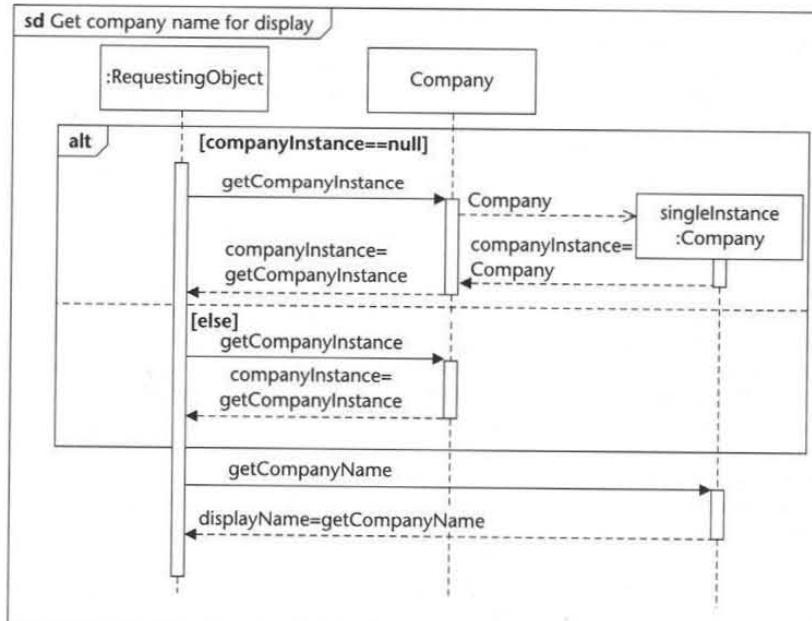


Figure 15.3 Sequence diagram for the interaction Get company name for display.

style of company registration details provides a solution to this aspect of the problem (Fig. 15.4). When the `getCompanyInstance()` operation is first called, the appropriate subclass is instantiated. If the `Company` object has not yet been instantiated, its constructor operation can access details of the country (say, held in a `CurrentCountry` object) to determine which subclass should be instantiated. This could use the Factory Method pattern (Gamma et al., 1995) to create the appropriate subclass.

This part of the design for the campaign management system now has:

- a class `Company`, which is only instantiated once
- an instance of this class, which is globally accessible
- different subclasses of `Company`, which are instantiated as needed, depending on run-time circumstances.

This is an application of the Singleton pattern. The pattern is described below in more general language using the key elements of the pattern template that we discussed in Section 15.3.

Name. Singleton.

Problem. How can a class be constructed that should have only one instance and that can be accessed globally within the application?

Context. In some applications it is important that a class have exactly one instance. A sales order processing application may be dealing with sales for one company. It is necessary to have a `Company` object that holds details of the company's name, address, taxation reference number and so on. Clearly there should be only one such object. Alternative forms of a singleton object may be required depending upon different initial circumstances.

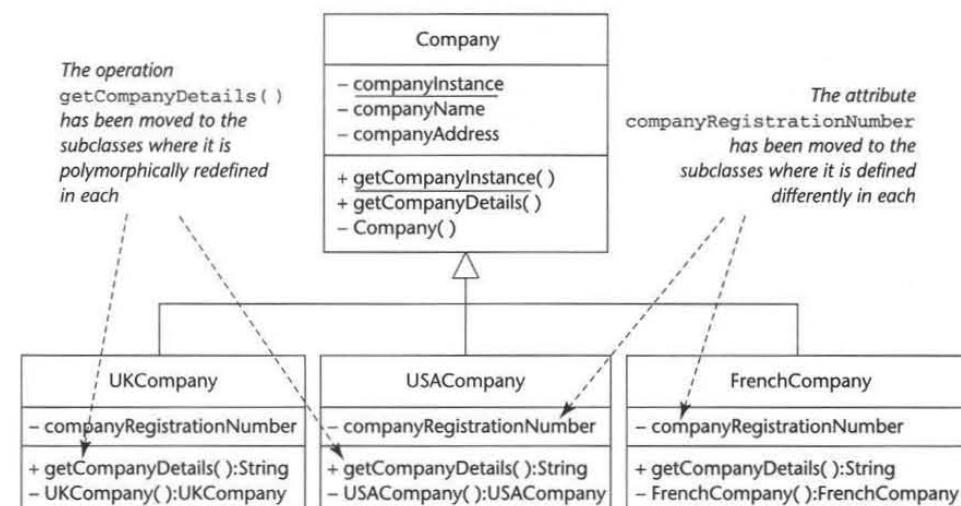


Figure 15.4 Company class with subclasses.

Forces. One approach to making an object globally accessible is to make it a global variable, but in general this is not a good design solution as it violates encapsulation. Another approach is not to create an object instance at all but to use class operations and attributes (called ‘static’ in C++ and Java). However, this limits the extensibility of the model since polymorphic redefinition of class operations is not possible in all development environments (for example C++).

Solution. Create a class with a class operation `getInstance()`, which, when the class is first accessed, creates the relevant object instance and returns the object identity to the requesting object. On subsequent accesses of the `getInstance()` operation no additional instance is created but the object identity of the existing object is returned. A class diagram fragment for the singleton pattern is shown in Fig. 15.5.

The singleton pattern offers several advantages but also has some disadvantages.

- + It provides controlled access to the sole object instance as the Singleton class encapsulates the instance.
- + The namespace is not unnecessarily extended with global variables.
- + The Singleton class may be subclassed. At system start-up user-selected options may determine which of the subclasses is instantiated when the Singleton class is first accessed.
- + A variation of this pattern can be used to create a specified number of instances if required.
- Using the pattern introduces some additional message passing. To access the singleton instance the class scope operation, `getInstance()`, has to be accessed first rather than accessing the instance directly.
- The pattern limits the flexibility of the application. If requirements change, and many instances of the singleton class are needed, then accommodating this new requirement necessitates significant modification to the system.
- The singleton pattern is quite well known and developers are tempted to use it in circumstances that are inappropriate. Patterns must be used with care.

In UML terms a pattern describes a collaboration and the interaction of the participating elements that provide the required functionality. A pattern may be documented as a template collaboration, as shown in Fig. 15.6. In essence a template collaboration is a parameterized collaboration that may be implemented (that is, instantiated) with different classes or objects. The notation describes the

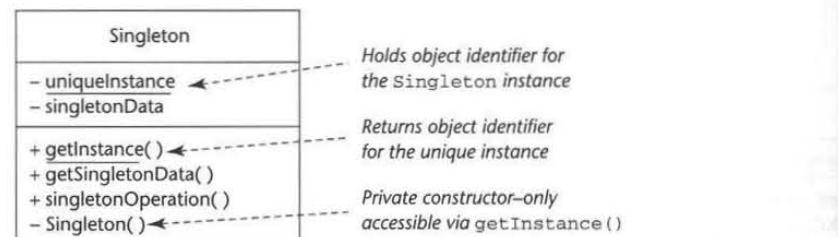


Figure 15.5 Creational patterns—Singleton.

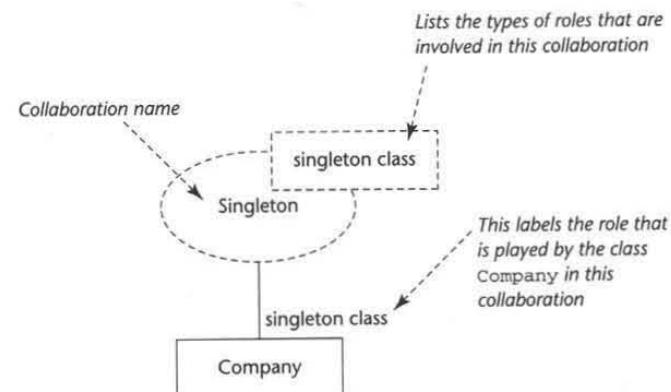


Figure 15.6 Template collaboration for Singleton Pattern implemented with the class Company.

pattern in terms of the roles that the participants occupy. In the case of the Singleton pattern there is only one role, `singleton class`, and this is occupied by the `Company` class as in Fig. 15.6.

15.4.3 Structural patterns

Structural patterns address issues concerned with the way in which classes and objects are organized. Structural patterns offer effective ways of using object-oriented constructs such as inheritance, aggregation and composition to satisfy particular requirements. For instance, there may be a requirement for a particular aspect of the application to be extensible. In order to achieve this, the application should be designed with constructs that minimize the side-effects of future change. Alternatively, it may be necessary to provide the same interface for a series of objects of different classes.

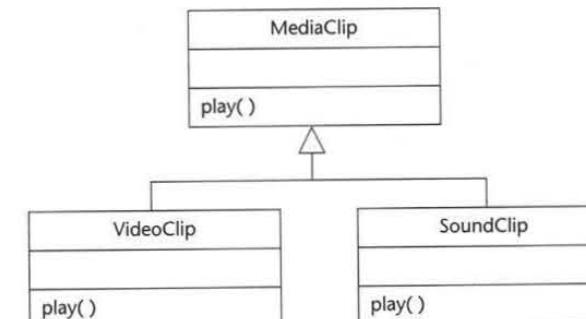


Figure 15.7 MediaClip inheritance hierarchy.

Composite pattern

It may be appropriate to apply the Composite structural pattern in a design for the Agate case study. In the following example we assume that further work is required to design a multimedia application that can store and play components of an advert.

Here an advert is made up of sound clips and video clips, each of which may be played individually or as part of an advert. The classes `SoundClip` and `VideoClip` have attributes and operations in common and it is appropriate that these classes are subclassed from `MediaClip` (Fig. 15.7). However, not all advert clips are primitive (that is, made up of only a single `MediaClip`). Some consist of one or more sequences of clips, such that each sequence is in turn an aggregation of `SoundClip` and `VideoClip` objects (Fig. 15.8).

These two orthogonal³ hierarchies can be integrated by treating `AdSequence` both as a subclass of `MediaClip` and as an aggregation of `MediaClip` objects (see Fig. 15.9).

All the subclasses in Fig. 15.9 include the polymorphically redefined operation `play()`. For the subclasses `VideoClip` and `SoundClip` this operation actually plays the object. But for an `AdSequence` object, an invocation of the `play()` operation results in it sending a `play()` message to each of its components in turn. This structure is a straightforward application of the Composite pattern where a `MediaClip` object and an `AdSequence` object provide a common interface in part.

Figure 15.10 shows the template collaboration diagram for the Composite pattern with its roles and types and Fig. 15.11 shows the allocation of particular classes to the roles. Figure 15.12 contains another type of UML diagram, a composite structure diagram. The top compartment of the diagram references an interaction diagram that describes the interaction and the lower compartment

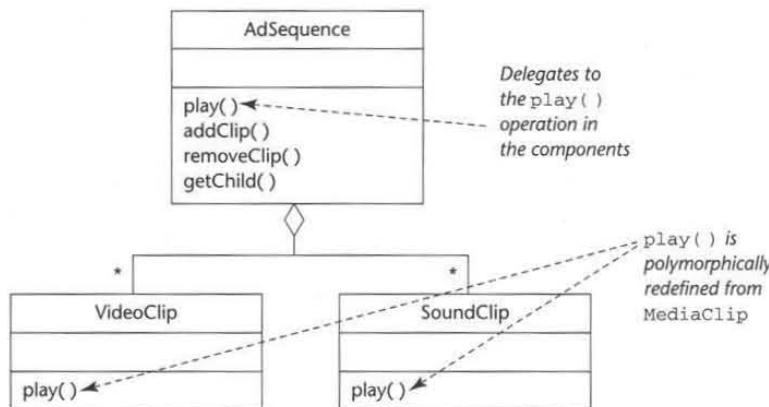


Figure 15.8 AdSequence aggregation hierarchy.

³ The term literally means 'at right angles to each other'. It is more loosely used here to describe hierarchies that cannot be directly mapped onto each other.

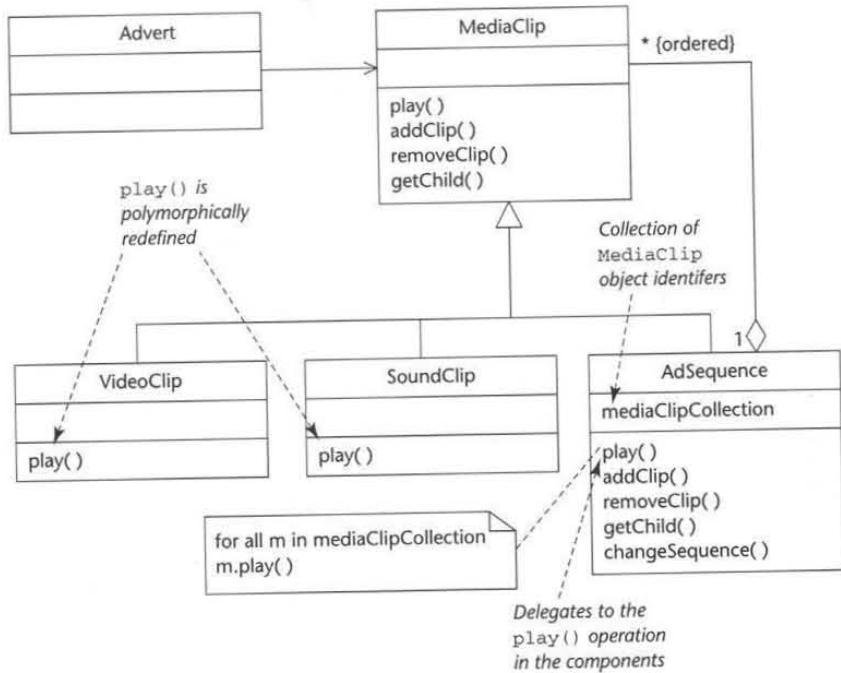


Figure 15.9 Integrating the two hierarchies for MediaClip.

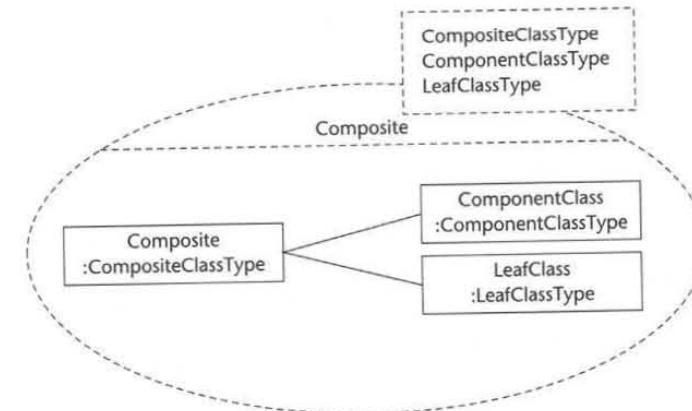


Figure 15.10 Template collaboration for Composite Pattern.

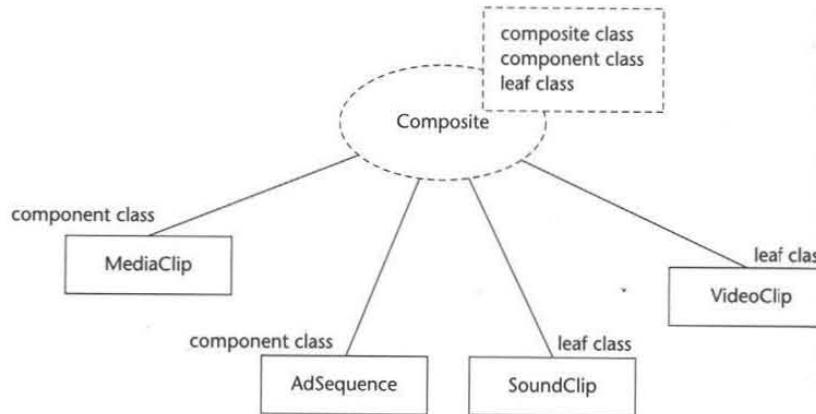


Figure 15.11 Template collaboration for Composite Pattern with implementation classes shown.

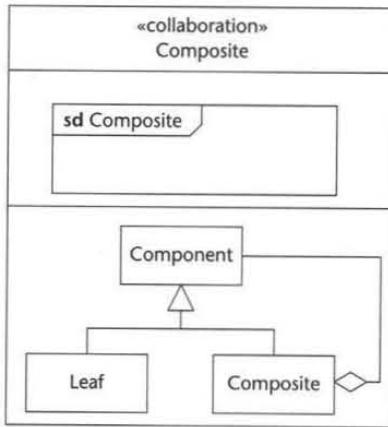


Figure 15.12 Composite structure diagram for the Composite Pattern.

shows the key relationships between the participating elements. A sequence diagram describing the interaction for the operation `play()` in an `AdSequence` object is shown in Fig. 15.13. The interaction constraint uses the OCL `size()` operation to get the size of the collection `mediaClipCollection`. The Composite pattern is described more generally below.

Name. Composite.

Problem. There is a requirement to represent whole-part hierarchies so that both whole and part objects offer the same interface to client objects.

Context. In an application both composite and component objects exist and are required to offer the same behaviour. Client objects should be able to treat composite or component objects in the same way. A commonly used example for the

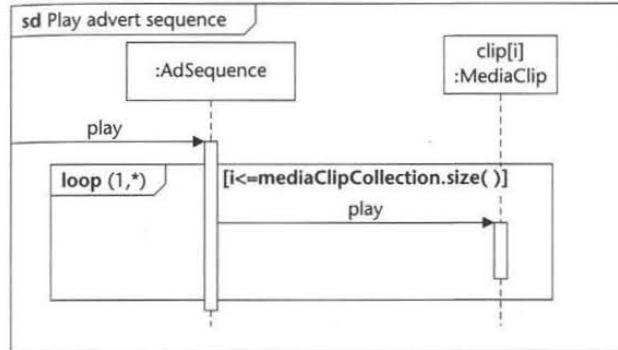


Figure 15.13 Sequence diagram for the interaction Play advert sequence.

composite pattern is a graphical drawing package. Using this software package a user can create (from the perspective of the software package) atomic objects like circle or square and can also group a series of atomic objects or composite objects together to make a new composite object. It should be possible to move or copy this composite object in exactly the same way as it is possible to move or copy an individual square or a circle. See Figs 8.5 and 8.6 which illustrate a straightforward composition without using the Composite pattern.

Forces. The requirement that the objects, whether composite or component, offer the same interface suggests that they belong to the same inheritance hierarchy. This enables operations to be inherited and to be polymorphically redefined with the same signature. The need to represent whole-part hierarchies indicates the need for an aggregation structure.

Solution. The solution resolves the issues by combining inheritance and aggregation hierarchies. Both subclasses, `Leaf` and `Composite`, have a polymorphically redefined operation `anOperation()`. In `Composite` this redefined operation invokes the relevant operation from its components using a simple loop construct (Fig. 15.14). The `Composite` subclass also has additional operations to manage the aggregation hierarchy so that components may be added or removed.

Further requirements may need to be considered for the Agate example. Perhaps `VideoClip` and `SoundClip` objects must be played in a particular sequence. This can be handled if the aggregate `AdSequence` maintains an ordered list of its components. This is shown in Fig. 15.9 by the `{ordered}` property on the aggregation association. Each component object can be given a sequence number, and two components that have the same sequence number are played simultaneously. The operation `changeSequence()` allows a component `MediaClip` object to be moved up or down within the sequence of clips in the advertisement (Fig. 15.9).

15.4.4 Behavioural patterns

Behavioural patterns address the problems that arise when responsibilities are assigned to classes and in designing algorithms. Behavioural patterns not only suggest particular static relationships between objects and classes but also describe how the objects communicate. Behavioural patterns may use inheritance structures

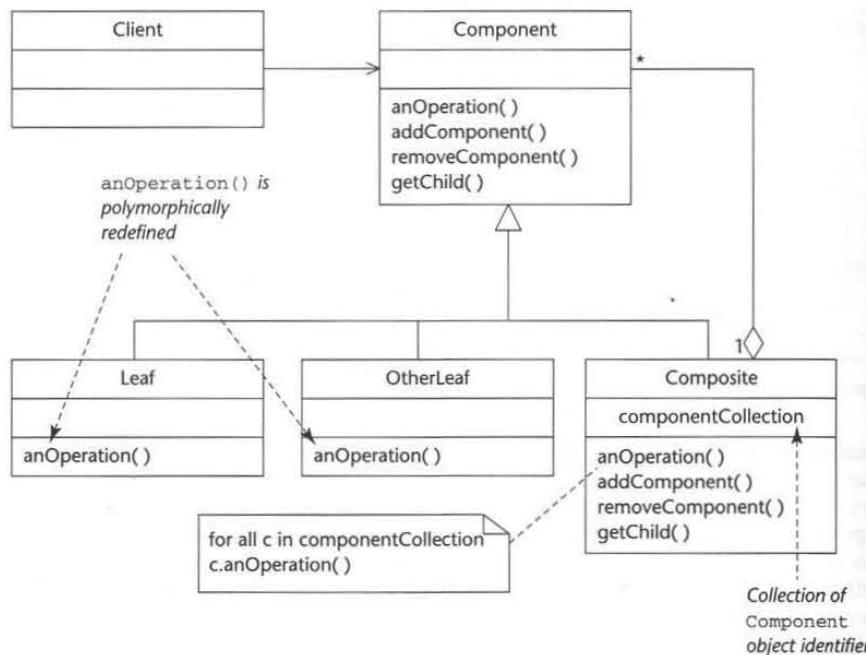


Figure 15.14 Structural patterns—Composite.

to spread behaviour across the subclasses or they may use aggregation and composition to build complex behaviour from simpler components. The State pattern, which is considered below, uses both of these techniques.

State pattern

Let us examine the Agate case study to determine whether it has features that may justify the application of the state pattern. First, are there any objects with significant state-dependent behaviour? Campaign objects have behaviour that varies according to state; a Campaign object may be in one of four main states, as shown in Fig. 11.20 (for simplicity we ignore the substates of the Active state). Clearly a Campaign object's state changes dynamically as the campaign progresses, thus necessitating changes in the behaviour of the object.

For example, when the FoodCo campaign is planned a Campaign object is created in the Commissioned state. It remains in this state until a campaign budget has been agreed and only then does it become possible to run advertisements, although some preparatory work may be done for the campaign in the meantime. Once a Campaign object enters the Active state, all advert preparation and any other work that is done is subject to an agreed billing schedule. Several operations, for example addAdvert() and calcCosts(), will behave differently depending upon the state of the Campaign object. It would be possible to construct a working version of the software using the design for the Campaign

class that is shown in Fig. 15.15. However, this would be a complex class that is further complicated by state-dependent operations such as calcCosts(), which would need to be specified with a series of case or if-then-else statements to test the state of the object. It would be simpler to subdivide the operations that have state-dependent behaviour, which in this case would result in four separate calcCosts() operations, one for each state. The inclusion of calcCostsCommissioned(), calcCostsActive() and so on within Campaign would simplify the operation calcCosts(), but the class as a whole would become even more complex.

Another possibility is to create additional classes, one for each state, so that each holds a state-specific version of the operations, and this is how the State pattern works. A class diagram fragment illustrating this application of the State pattern is shown in Fig. 15.16. Since the subclasses of CampaignState have no attributes specific to a particular Campaign object, it is possible to have only one instance of each in the system. Thus there will be a maximum of four CampaignState objects, one for each state, and the additional overhead of manipulating the objects is unlikely to be significant (Fig. 15.16). These are examples of *pure state classes* as they contain no attributes. They could be implemented using a variation of the Singleton pattern that ensures one and only one instance of each subclass will exist

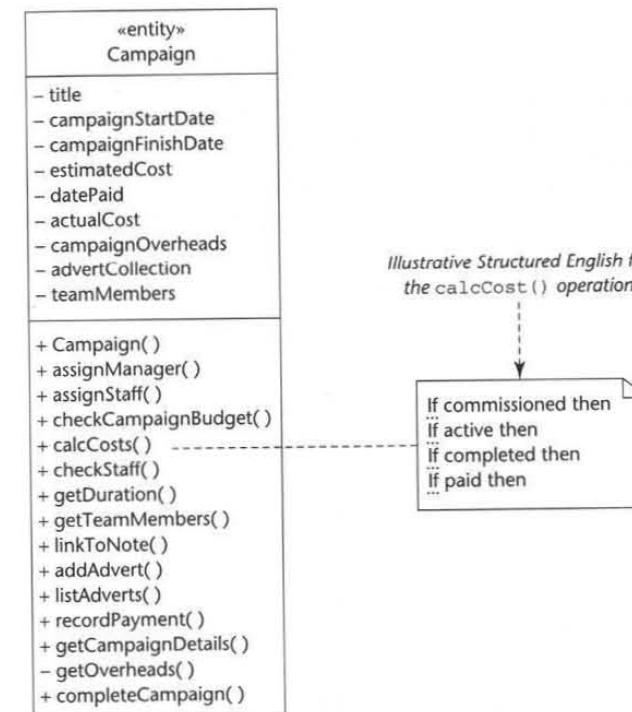


Figure 15.15 The Campaign class.

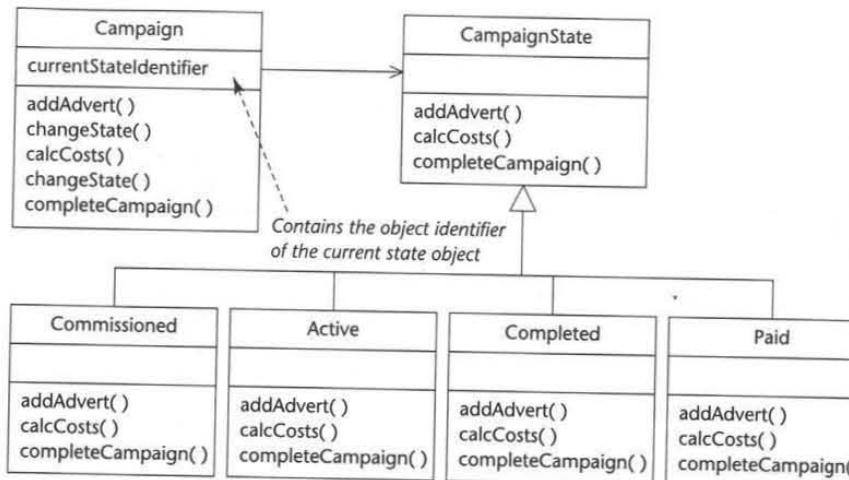


Figure 15.16 State pattern for Agate showing a simplified version of Campaign class.

(this differs from its more usual application of ensuring that there is at most one object for a whole hierarchy).

Figure 15.17 shows some sample Campaign objects linked to their state objects. The object `c:Campaign` and `e:Campaign` both have links to the `:Active` state object. This is a pure state object and has no attributes for any particular campaign and can be used by more than one Campaign object at a time. Any state dependent messages that are received by `c:Campaign`, for example, `calcCosts()`, will essentially be passed on to the object whose identifier is held in the `currentState` attribute. In this case, for `c:Campaign` a `calcCosts()` message will be sent to the `:Active` object. If `d:Campaign` receives a `calcCosts()` message, this will result in a `calcCosts()` message being sent to `:Completed`. The state object that receives the `calcCosts()` message is determined by the value of `currentState` without the need for any conditional statements.

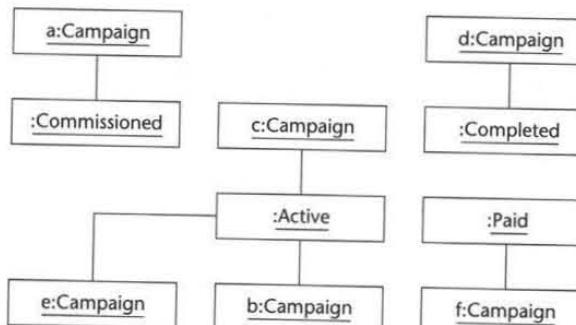


Figure 15.17 Some State pattern objects for Agate.

Figure 15.18 shows the template collaboration for the State Pattern and the classes allocated to the roles. The State pattern is described more generally below.

Name. State.

Problem. An object exhibits different behaviour when its internal state changes making the object appear to change class at run-time.

Context. In some applications an object may have complex behaviour that is dependent upon its state. In other words, the response to a particular message varies according to the object's state. One example is the `calcCosts()` operation in the Campaign class.

Forces. The object has complex behaviour, which should be factored into less complex elements. One or more operations have behaviour that varies according to the state of the object. Typically the operation would have large, multi-part conditional statements depending on the state. One approach is to have separate public operations for each state but client objects would need to know the state of the object so that they could invoke the appropriate operation. For example, four operations `calcCostsCommissioned()`, `calcCostsActive()`, `calcCostsCompleted()` and `calcCostsPaid()` would be required for the Campaign object. The client object would need to know the state of the Campaign object in order to invoke the relevant `calcCosts()` operation. This would result in undesirably tight coupling between the client object and the Campaign object. An alternative approach is to have a single public `calcCosts()` operation that invokes the relevant private operation (`calcCostsCommissioned()` would be private). However, the inclusion of a separate private operation for each state may result in a large complex object that is difficult to construct, test and maintain.

Solution. The state pattern separates the state-dependent behaviour from the original object and allocates this behaviour to a series of other objects, one for each state. These state objects then have sole responsibility for that state's behaviour. The original object, shown as Context in Fig. 15.20 delegates responsibility to the appropriate state object. The original object becomes an aggregate of its states,

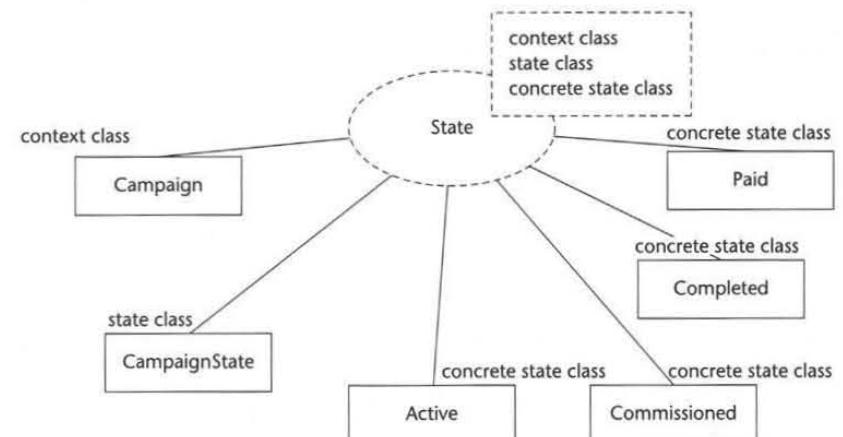


Figure 15.18 Template collaboration for State Pattern.

only one of which is active at one time. The state objects form an inheritance hierarchy.

The responsibility for transitions from one state to another may either be taken by the Context object or it may be shared among the State subclasses. If the rules for state changes are volatile and subject to change, it may be better for the current State object to be responsible for the next transition. In this way the current state object always knows all the states into which the object may move next. However, this has the disadvantage of producing dependencies between state subclasses.

For example, in Fig. 15.19 the :Campaign lifeline (the Context object) receives the completeCampaign() message. This results in a complete Campaign() message being sent to the current state object, in this case represented by the :Active lifeline. The completeCampaign operation in the Active class then sends the changeState(nextState) message back to the :Campaign and places the object identifier for the next state object (:Completed) in the currentState attribute. When the :Campaign object next receives a state-dependent message, it will be passed on to the :Completed object. In Fig. 15.19 it

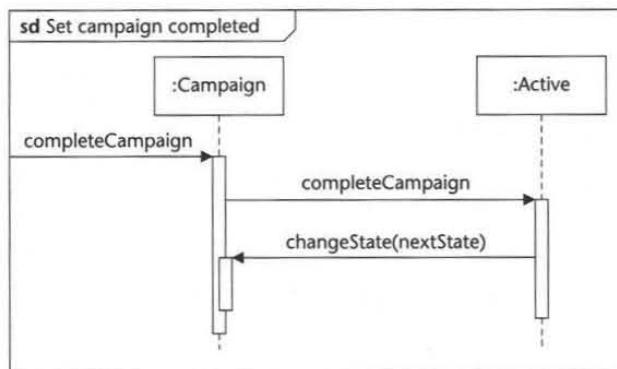


Figure 15.19 Sequence diagram for the interaction Set campaign completed.

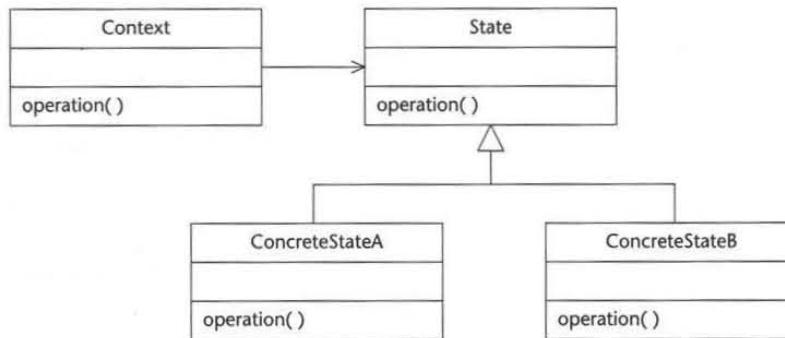


Figure 15.20 Behavioural patterns—State.

is assumed that the Active state object already exists. As it is a pure state object, it could be created using the Singleton pattern the first time it is accessed.

If the Campaign class was taking responsibility for managing the state, there would be a nextState() message from the :Campaign object to itself when it received the completeCampaign() message.

If it is not appropriate to use pure state classes, then a new state object may have to be created whenever a Context object changes state and the old state object may have to be deleted. This increases processing load for various transactions and may affect performance.

Use of the State pattern has both advantages and disadvantages, the latter particularly in terms of its possible side effects on system performance.

- + State behaviour is localized and the behaviour for different states is separated. This eases any enhancement of the state behaviour, in particular the addition of extra states.
- + State transitions are made explicit. The state object that is currently active indicates the current state of the Context object.
- + Where a state object has no attributes relevant to a specific Context object it may be shared among the Context objects. This State object is a Singleton!
- If the State objects cannot be shared among the Context objects, each Context object will have to have its own State object thus increasing the number of objects and the storage requirements for the system.
- State objects may have to be created and deleted as the Context object changes state, thus introducing a processing overhead.
- Use of the State pattern introduces at least one extra message, the message from the Context class to the State class, thus adding a further processing overhead.

15.5 How to Use Design Patterns

The use of a pattern requires careful analysis of the problem that is to be addressed and the context in which it occurs. Before contemplating the application of patterns within a software development environment it is important to ensure that all members of the team receive appropriate training.

When a developer identifies a part of the application that may be subject to high coupling, a large, complex class or any other undesirable feature, there may be a pattern that addresses the difficulty. The following issues should be considered before employing a pattern to resolve the problem.

- Is there a pattern that addresses a similar problem?
- Does the pattern trigger an alternative solution that may be more acceptable?
- Is there a simpler solution? Patterns should not be used just for the sake of it.
- Is the context of the pattern consistent with that of the problem?
- Are the consequences of using the pattern acceptable?
- Are constraints imposed by the software environment that would conflict with the use of the pattern?

Gamma et al. (1995) suggest a seven-part procedure that should be followed after an appropriate pattern has been selected in order to apply it successfully:

1. Read the pattern to get a complete overview.
2. Study the structure of the pattern, its participants (e.g. classes) and its collaborations in detail.
3. Examine sample code to review an implementation of the pattern.
4. Choose names for the classes that relate to the application context.
5. Define the classes.
6. Give the operations application-specific names.
7. Implement operations that perform the necessary responsibilities and collaboration.

A pattern should not be viewed as a prescriptive solution but rather as guidance on how to find a suitable solution. It is quite likely (in fact almost certainly the case) that a pattern will be used differently in each particular set of circumstances. At a simple level the classes involved will have attributes and operations that are determined by application requirements. Often a pattern is modified to accommodate contextual differences. For example, the inclusion of the `changeSequence()` operation shown in Fig. 15.9 represents a variation on the Composite pattern to allow the sequence of `MediaClip` objects in `AdSequence` to be changed. As an alternative to using a variation of a pattern, the pattern may suggest some other solution to the developer that produces a better design than using the pattern itself.

If patterns are going to be used by a developer, then relevant pattern catalogues and languages should be made available and easily accessible. Many patterns are documented in hypertext on the Internet or on company intranets. It is important to consider the way a pattern is documented so that it is easy for the developer to determine its applicability. We suggest that the minimum information needed is that described earlier in Section 15.3. Modelling tool support for patterns is developing and is provided by some vendors. Some tools now allow the user to select classes and apply a pattern to them, or to import a pattern template set of classes into a model and rename the classes to fit the application domain.

Software developers may wish to capture their experience in a pattern format and build their own pattern catalogue or language. The process of identifying patterns is known as *pattern mining* and requires careful validation and management to ensure that the patterns that are captured suggest good solutions to recurring problems. Again, the pattern elements described earlier provide a checklist against which any candidate pattern can be compared.

If a pattern satisfies these criteria, then its quality should be assured via a walkthrough. The most commonly used form of walkthrough for a pattern is known as a pattern writer's workshop. This involves a small group of pattern authors who constructively comment upon each other's patterns. The focus of a workshop helps the participants describe useful patterns effectively.

15.6 Benefits and Dangers of Using Patterns

One of the most sought-after benefits of object-orientation is reuse. Reuse at the object and class levels has proved more elusive than was expected in the early days

of object-orientation. Patterns provide a mechanism for the reuse of generic solutions for object-oriented and other approaches. They embody a strong reuse culture. Within the design context, patterns suggest reusable elements of design and, most significantly, reusable elements of demonstrably successful designs. This reuse permits the transfer of expertise to less experienced developers so that a pattern can be applied again and again.

Another benefit gained from patterns is that they offer a vocabulary for discussing the problem domain (whether it be analysis, design or some other aspect of information systems development) at a higher level of abstraction than the class and object, making it easier to consider micro-architectural issues and systems architecture as well. Pattern catalogues and pattern languages offer a rich source of experience that can be explored and provide patterns that can be used together to generate effective systems.

Some people believe that the use of patterns can limit creativity. Since a pattern provides a standard solution, the developer may be tempted not to spend time on considering alternatives. The use of patterns in an uncontrolled manner may lead to over-design. Developers may be tempted to use many patterns irrespective of their benefits, thus rendering the software system more difficult to develop, maintain and enhance. When a pattern is used in an inappropriate context, the side effects may be disastrous. For example, the use of the State pattern may significantly increase the number of objects in the application with a consequent reduction in performance. If the concrete state classes are not pure state classes (i.e. they hold attributes), then it will not be possible to share state classes, as is shown in Fig. 15.17. Each context class may need its own state class and the total number of objects could be doubled, resulting in an additional processing load for object creation and deletion as states change.

The introduction of any new approach to software development has costs for the organization. Developers need to spend time understanding the relevant pattern catalogues, they need to be provided with easy access to the relevant catalogues and they need to be trained in the use of patterns. Another aspect of the introduction of patterns is the necessary cultural change. Patterns can only be used effectively in the context of an organizational culture of reuse. Ironically, the introduction of a patterns approach may arouse less opposition to the encouragement of a reuse culture than an attempt to introduce procedures for code reuse. Developers need to think about how to apply a pattern to their current context, and thus there are greater opportunities for individual creativity.

These issues emphasize that the use of patterns in software development requires care and planning. In this respect patterns are no different from any other form of problem solving: they must be used with intelligence and an awareness of any side effects. It is also important to appreciate that patterns address only some of the issues that occur during systems development. In no way should patterns be viewed as a 'silver bullet' that conquers all problems in systems development.

15.7 Summary

This chapter has considered how patterns can be used in software development. Patterns have been identified in many different application domains and are applicable at many different stages of the software development process. The most

significant aspect of the growth of interest in patterns is the increased awareness of design issues that follows as a consequence. Patterns represent a significant change in the reuse culture in software development. Reuse need no longer be focused solely on elements of code, whether these are individual classes or complex frameworks, but can also realistically include the reuse of analysis or design ideas as described by patterns.

Related patterns are grouped together in catalogues. A pattern language is a group of patterns focused on a particular aspect of a problem domain so that when used together they provide solutions to the problems that arise.

Review Questions

- 15.1** What is the difference between a pattern and a framework and how is each used?
- 15.2** What are the main aspects of changeability?
- 15.3** Why is the class constructor private in the Singleton pattern?
- 15.4** What are the advantages of using the Singleton pattern?
- 15.5** What are the disadvantages of using the Singleton pattern?
- 15.6** What implementation problems may occur when using the State pattern?
- 15.7** What are the differences between a pattern language and a pattern catalogue?
- 15.8** List two general dangers and two general benefits of the use of patterns.
- 15.9** What seven steps are suggested by Gamma et al. for the effective use of patterns?
- 15.10** What are the advantages of using the Composite pattern?

Case Study Work, Exercises and Projects

- 15.A** Read the design patterns Bridge and Façade in Gamma et al. (1995) and rewrite their description using the structure of the template given in Section 15.3.
- 15.B** In the FoodCo case study the `ProductionLine` class might be a candidate for design using the State pattern. Show how a variation of the State pattern could handle this requirement. What benefits and disadvantages are there in applying this solution?
- 15.C** Where and how could the Singleton pattern be used in the FoodCo case study? Prepare a design class specification for a suitable Singleton class.

Further Reading

Gamma et al. (1995) and Buschmann et al. (1996) are two excellent texts that give important advice concerning software construction and should be on the essential reading list of any software developer. Schmidt et al. (2000) provide a further set of architectural patterns. Even where the patterns they discuss are not directly relevant, there is much to learn from their approach to solving design problems.

Both Fowler (1997) and Coad et al. (1997) are useful sources of information from an analysis perspective.

The 'Pattern Languages of Program Design' (known as the PLOP) books (Coplien and Schmidt, 1995; Vlissides et al., 1996; Martin et al., 1998; Harrison et al., 2000) catalogue a wide range of patterns for all aspects of software development.

The patterns home page has links to a wealth of patterns and is well worth browsing. This can be found at <http://hillside.NET/patterns/patterns.html>.

Further useful patterns are available in the Portland Pattern Repository at <http://c2.com/cgi-bin/wiki?PortlandPatternRepository>. Organizational patterns can be found at <http://i44.pc48.info.uni-karlsruhe.de/cgi-bin/OrgPatterns>. IBM Patterns for e-Business website. <http://www-128.ibm.com/developerworks/patterns/>.

A series of Java related patterns (some architectural and some design) are available at <http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns/>. Two other more recent texts are *Core J2EE Patterns* (Alur et al., 2003) and Metsker's (2002) *Design Patterns Java Workbook*.

Some argue that the architectural approach adopted by Shlaer and Mellor (1988, 1992) is pattern-based; they certainly have interesting ideas.

Human–Computer Interaction

Learning Objectives

In this chapter you will learn

- the importance of good user interface design
- what is meant by metaphors in human–computer interaction
- about different approaches to human–computer interaction
- how to apply the techniques of scenario-based design
- how standards and the law affect interface design.

16.1 | Introduction

Designing the user interface can be critical in the development of an information system. The interface is what the users see. To them it *is* the system. Their attitude towards the entire system can be coloured by their experience of the user interface. Effective design of the interaction between people and the information systems they use is a discipline in its own right—human–computer interaction (HCI)—that combines the techniques of psychology and ergonomics with those of computer science. Before we move on to the design of the classes that make up the user interface in Chapter 17, we will address some of the HCI issues that influence the design of the user interface.

In Chapter 13 we raised the issue of designing the inputs and outputs of a system. This chapter is about the human factors aspects of designing the inputs and outputs. The inputs and outputs can be in the conventional form of data entry and enquiry screens and printed reports, or they can take the form of speech recognition, scanners, touch screens and virtual reality environments. We shall be concentrating on the conventional inputs and outputs used in information systems, although we recognize that the growth in the use of multimedia systems means that even quite conventional business information systems may incorporate multiple media as inputs and outputs.

There are two metaphors that are widely used to represent the user interface: first, the idea that the user is conducting a dialogue with the system, and, second, the idea that the user is directly manipulating objects on screen (Section 16.2). Much HCI work in the past has concentrated on producing guidelines for dialogue design, and we include a section on the characteristics of a good dialogue.

In the second part of the chapter, we consider an informal approach to the HCI design of a system and describe some of the activities that may be carried out. We then introduce three different approaches to HCI that reflect different beliefs about the best way to design the user interface of a system (Section 16.3). Examples of some of the techniques used in these approaches are given.

Finally, we make reference to the international standards for the ergonomics of workstation design and, for European readers, to the legal obligations for HCI design that are imposed by the European Union's directive on health and safety requirements for work with display screens (Section 16.4). With the growth of the Internet and the use of World Wide Web (WWW) technologies for intranets and access to information systems within organizations, accessibility of user interfaces for people with disabilities has become an important HCI issue. We highlight the legal requirements for accessibility and some of the initiatives that set guidelines to improve the accessibility of browser-based user interfaces.

16.2 | The User Interface

16.2.1 What is the user interface?

Users of an information system need to interact with it in some way. Whether they are users of FoodCo's telesales system entering orders made over the telephone by customers, or members of the public using a touch screen system to find tourist information, they will need to carry out the following tasks:

- read and interpret information that instructs them how to use the system
- issue commands to the system to indicate what they want to do
- enter words and numbers into the system to provide it with data to work with
- read and interpret the results that are produced by the system, either on screen or as a printed report
- respond to and correct errors.

It is important to note that these are mostly secondary tasks: they are concerned with using the system, not with the users' primary objectives. In the examples above, the primary tasks are to take a customer order and to find tourist information. If the system has been designed well, the secondary, system-related tasks will be easy to carry out; if it has not been designed well, the secondary tasks will intrude into the process and will make it more difficult for the users to achieve their primary tasks.

16.2.2 The dialogue metaphor

In the design of many computer systems, interaction between the user and the system takes the form of a *dialogue*. The idea that the user is carrying on a dialogue

with the system is a *metaphor*. (A metaphor is a term that is used figuratively to describe something but is not applied literally.) There is no real dialogue in the sense of a conversation between people going on between the user and the computer¹, but as in dialogues between people, messages are passed from one participant to the other. Figure 16.1 shows the human-computer dialogue in schematic form.

Figure 16.2 describes what is meant by each of the types of message that can be found in this dialogue.

Figure 16.3 shows a sample screen layout from FoodCo's existing system, which runs on a mini-computer with displays on dumb terminals. Although this shows only one screen, you can describe it in terms of the dialogue between the user and the system.

- The user may enter a command by selecting an option from a menu (not shown).
- The system responds with this data entry screen and automatically displays the Order Date and next Order No.
- The user enters a Customer Code.
- The system responds with the name and brief address of the customer as a confirmation that the correct number has been entered.

And so on ...

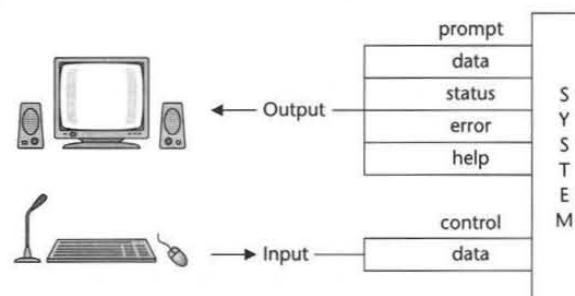


Figure 16.1 Schematic form of human-computer dialogue.

Output	prompt	Request for user input
	data	Data from application following user request
	status	Acknowledgement that something has happened
	error	Processing cannot continue
	help	Additional information to user
Input	control	User directs which way dialogue will proceed
	data	Data supplied by user

Figure 16.2 Types of messages in human-computer dialogue.

¹ Although speech recognition and text to speech systems now make voice input and output possible, even if it is not yet a true dialogue.

CUSTORD1	Customer Order Entry		25/08/2005		
Order Date	25/08/2005	Order No.	37291		
Customer Code	CE102_ Central Stores, Lytham St Annes				
Customer Order Ref.	R20716_				
Prod. Code	Product Description	Unit Price	Line Price		
01 12-75_	Sandwich spread 24x250g	3	18.00 54.00		
02 09-103_	Brown sauce 30x500g	10	24.60 246.00		
03 _____		_____	_____		
04 _____		_____	_____		
05 _____		_____	_____		
06 _____		_____	_____		
07 _____		_____	_____		
08 _____		_____	_____		
		Total	300.00		
		Tax	52.50		
		Order			
		Total	352.50		
F1-Help	F2-Save	F3-Cancel	F4-New	F5-Cust.	F6-Prod.
F10-Exit				Cust.	Lookup
					Lookup

Figure 16.3 FoodCo customer order entry screen layout with sample data.

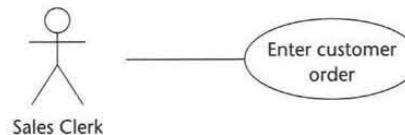
The dialogue may not take exactly the same form each time that a user enters data into this screen. Sometimes the user may not know the Customer Code and may have to use some kind of index look-up facility, perhaps entering the first few characters of the customer name in order to view a display of customers that start with those characters. Sometimes an order may consist of one line, usually it will

Output	prompt	Request for user input and labels for automatically generated data, shown in bold, for example Customer Code
	data	Automatic display of Order Date and next Order No., automatic calculation of totals and tax (shown in italics to distinguish it from input data)
	status	Screen heading; could include display to confirm that a new order has been saved
	error	Messages to warn of incorrect data entered, for example if a Customer Code is entered that does not exist or if a negative Quantity is entered
	help	Additional information to user in response to the user pressing F1; may be general about the order entry screen or context-sensitive—specific to a particular type of data entry
Input	control	Use of function keys to control dialogue
	data	Numbers, codes and quantities typed in by user

Figure 16.4 Examples of types of messages in human-computer dialogue.

consist of more and, if it consists of more than eight, it will be necessary to clear those that have been entered from the screen and display space for a further eight lines to be entered. It also illustrates elements of the interface that support some of the message types listed in Fig. 16.2. These are described in Fig. 16.4.

In the requirements model of the new system for FoodCo, there will be a use case for Enter customer order, as in Fig. 16.5. This will be supported by a use case description, which may be quite brief early in the project. As the project progresses through further iterations, the use case description will be filled out in more detail. Not all the use cases will be for interactive dialogues: some will be for enquiries and some will be for printed reports. Figure 16.6 shows some of the use cases that the FoodCo sales clerks use. For each of these use cases there may be a sequence diagram to show the interaction between the collaborating objects. However, these sequence diagrams will not yet show the details of the interaction between the user and the system at the interface. This will be covered in Section 17.5.



Enter customer order

Enter Customer Code or look up customer in index. For each line of the order, enter Product No. and Quantity. Calculate Line Price, Tax and totals automatically.

Figure 16.5 Use case diagram and description for Enter Customer Order.

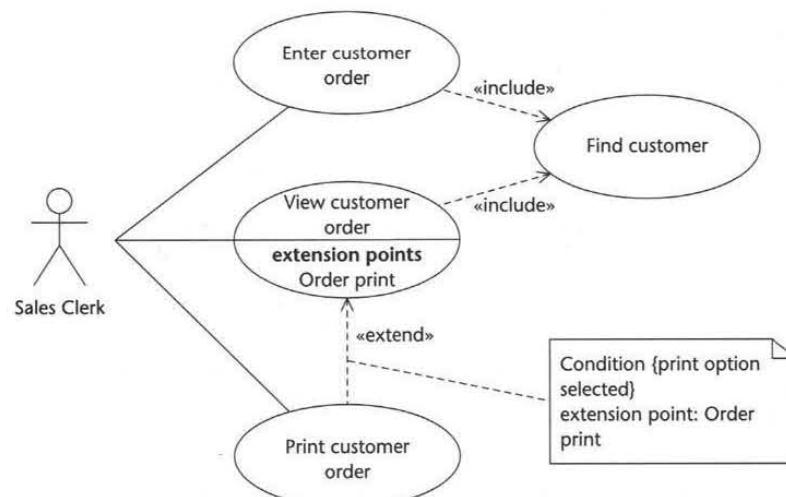


Figure 16.6 Use case diagram showing use cases used by sales clerks at FoodCo.

16.2.3 The direct manipulation metaphor

The other metaphor for the design of the user interface, which has become more widespread in the last few years, is the *direct manipulation* metaphor. Most people are now familiar with this through the use of GUIs. When you use a software package with this kind of interface you are given the impression that you are manipulating objects that are used on the screen through the use of the mouse. This metaphor is reflected in the concrete nature of the terms used. You can:

- drag and drop an icon
- shrink or expand a window
- push a button
- pull down a menu.

(Though many of the metaphors break down if examined too closely. Why would you put an open window on your desktop?) Such interfaces are *event-driven*. Graphical objects are displayed on the screen and the window management part of the operating system responds to events. Most such events are the results of the user's actions. The user can click on a button, type a character, press a function key, click on a menu item or hold down a mouse button and move the mouse. The design of user interfaces to support this kind of interaction is more complicated than for text-based interfaces using the dialogue metaphor. Figure 16.7 shows the interface of a Java program to implement the use case Check campaign budget for the Agate case study. In this use case, the user first selects the name of a client from a list box labelled Client. Having selected the client, a list of all active campaigns for that client is placed in the list box labelled Campaign. At this point, no campaign is selected, and the user can click on the arrow at the end of the list box to view the list and select a campaign. When a campaign has been selected, the user can click on the button labelled Check. The program then totals up the cost of adverts in that campaign, subtracts it from the budget and displays the balance as

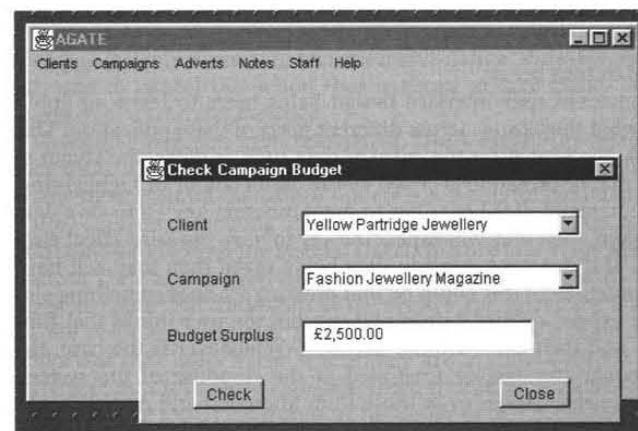


Figure 16.7 Interface for the use case Check Campaign Budget developed in Java.

a money value (negative if the campaign is over budget). In this interface design, there is no point in the user selecting a campaign until a client has been selected or clicking the Check button until a client and a campaign have been selected. The designer may choose to disable the Campaign list box until the client has been selected, and disable the button until both client and campaign have been selected. Having checked one campaign, the user may choose a different client, in which case the contents of the **Campaign** list box have to be changed and the button disabled again until a different campaign has been selected. In Section 17.8 we shall use state machine diagrams to model the state of elements of a user interface like this in order to ensure that we have correctly specified the behaviour of the interface.

Windows like the one in the example above are usually called *dialogue boxes* in GUI environments. In terms of the metaphors that we have discussed, they combine elements of a dialogue with the user with direct manipulation of buttons and lists.

16.2.4 Characteristics of good dialogues

Many authors of books and reports on HCI have produced sets of guidelines to help designers to produce good designs for the user interface. Some such guidelines are specific to certain types of interface. Shneiderman and Plaisant (2004) propose five high-level objectives for data entry dialogue design that date back to the original edition in 1986 when most interfaces were text-based. Other authors such as Gaines and Shaw (1983), also writing at a time when text-based interfaces were predominant, have proposed as many as seventeen.

Regardless of whether a system is being developed for a text-based environment or for a GUI environment, there are a number of important general characteristics of good dialogue design. These include:

- consistency
- appropriate user support
- adequate feedback from the system
- minimal user input.

These are considered in turn below.

Consistency. A consistent user interface design helps users to learn an application and to apply what they know across different parts of that application. This applies to commands, the format for the entry of data such as dates, the layout of screens and the way that information is coded by the use of colour or highlighting. As an example of a command, if the user has to press function key F2 to save data in one part of the system, then they will expect the key to have the same effect elsewhere in the system. If it does not result in data being saved then they will have made an error. The outcome of this could be that pressing F2 does something else that the user did not expect, or that it does nothing, but the user thinks that they have saved the data and then exits without saving it. Whichever is the case, the user is likely to become annoyed or frustrated at the response of the system. Guidelines in corporate style guides or in those from Microsoft and Apple help to prevent this kind of user frustration. Style guides are discussed in Section 16.2.5.

Appropriate user support. When the user does not know what action to take or has made an error, it is important that the system provides appropriate support at the interface. This support can be informative and prevent errors by providing help messages, or it can assist the user in diagnosing what has gone wrong and in recovering from their error. Help messages should be context-sensitive. This means that the help system should be able to detect where the user has got to in a dialogue and provide relevant information. In a GUI environment, this means being able to detect which component of the interface is active (or has the *focus*) and providing help that is appropriate to that part of the interface. The help provided may be general, explaining the overall function of a particular screen or window, or it may be specific, explaining the purpose of a particular field or graphical component and listing the options available to the user. It may be necessary to provide a link between different levels of help so that the user can move between them to find the information they require. The hypertext style of help in Microsoft Windows provides this facility. Help information may be displayed in separate screens or windows, it may be displayed simultaneously in a status line as the user moves through the dialogue or it may be displayed using *tooltips* that appear as the user positions the cursor over an item. Many web page designers provide help about elements of their pages by displaying messages in the status line at the bottom of the browser window or by displaying a tooltip-style message in a box as the cursor moves over an item on the page.

Error messages serve a different purpose and require careful design to ensure that they inform rather than irritate the user. An error message that tells the user that he or she has just deleted an essential file and then expects the user to click on a button marked OK when it is anything but OK is likely to annoy the user. Error messages should explain what has gone wrong and they should also clearly explain what the user can or should do to recover the situation. This information should be in language that the user can understand. This may mean using jargon from the user's business that they will recognize and understand rather than using computer jargon. Figure 16.8 shows three different error message boxes for the same situation. Only one is really of any help to the user.

Warning messages can prevent the user from making serious errors by providing a warning or caution message before the system carries out a command from the user that is likely to result in an irreversible action. Warning messages should allow the user to cancel the action that is about to take place. Figure 16.9 shows an example of a warning message.

Adequate feedback from the system. Users expect the system to respond when they make some action. If they press a key during data entry, they expect to see the character appear on the screen (unless it is a control command or a function key); if they click on something with the mouse, they expect that item to be highlighted and some action from the system. Users who are uncertain whether the system has noticed their action keep on pressing keys or clicking with the mouse, with the possible result that these further keypresses and clicks are taken by the system to be the response to a later part of the dialogue, with unpredictable results. It is important that users know where they are in a dialogue or direct manipulation interface: in a text-based interface there should be a visible cursor in the current active field; in a GUI environment the active object in the interface should be highlighted. The **Yes** button in Fig. 16.9 is highlighted in this way, and this means that it will respond to the user pressing `<Return>` on the keyboard.

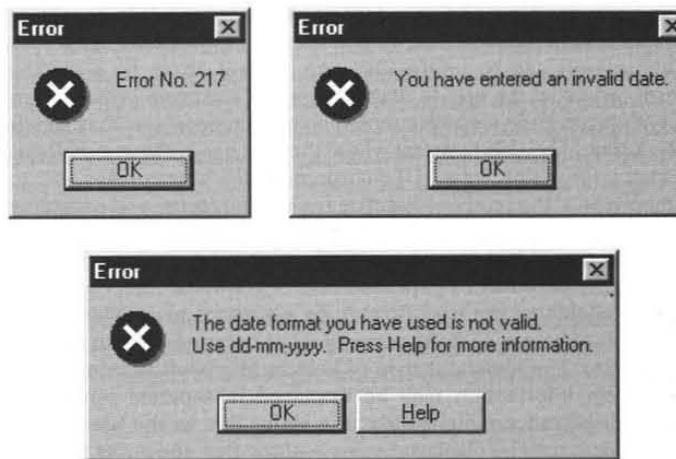


Figure 16.8 Example error messages for the same error.

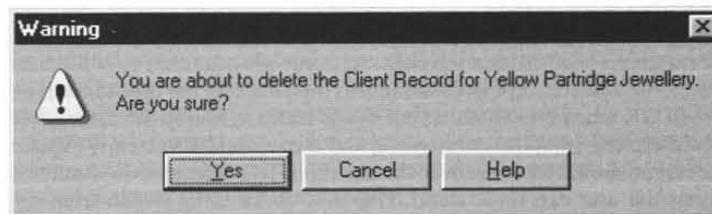


Figure 16.9 Example warning message.

The system's response time should be appropriate to the type of user action: responses to data entry keypresses should be instantaneous, while responses to commands in menus or by means of buttons may take longer. If a system response is going to take some time, the system should respond first with some kind of feedback indicating that it is busy. This can be changing the cursor to a different form, or it can be displaying a progress monitor that shows what proportion of the task has been completed. If possible, the user should be given the option of cancelling the command. The purpose of this kind of feedback is to reduce the user's uncertainty about whether the system has received the input and is doing something about it or is waiting for the next input.

Minimal user input. Users resent making what they see as unnecessary keypresses and mouse clicks. Reducing unnecessary input also reduces the risk of errors and speeds data entry. The interface should be designed to minimize the amount of input from the user. The user can be helped in this way by:

- using codes and abbreviations
- selecting from a list rather than having to enter a value

- editing incorrect values or commands rather than having to type them in again
- not having to enter or re-enter information that can be derived automatically
- using default values.

Some of these have a basis in the psychological aspects of the discipline of HCI. For example, being able to select values from a list rather than having to enter them from memory allows the user to work by recognizing information rather than having to recall it.

It is also possible to reduce the amount of input as users become more familiar with a system by providing shortcuts or accelerators, key combinations that the user can use instead of selecting a command from a menu. However, these require the user to remember the key combinations and are less useful for new users, who will find menus easier to use.

16.2.5 Style guides

In Section 16.2.4 consistency of the interface has been highlighted as one of the characteristics of good dialogue design. Some organizations provide standard guidelines for the design of user interfaces. One way in which standardization of user interface design has come about is through the domination of the PC market by Microsoft. Microsoft produces a book of guidelines *The Windows Interface Guidelines for Software Design* (Microsoft, 1997) that lays down the standards to which developers must adhere if they are to be granted Windows certification. Similar guidelines are available from Apple for the Apple Macintosh operating system—*Macintosh Human Interface Guidelines* (Apple, 1996). The effect of such guidelines is apparent in the similarity of many applications from different sources that make use of toolbars, status bars and dialogue boxes with buttons and other graphical components placed in similar positions. The benefit of this similarity for users is that different applications will look similar and behave in similar ways. This means that users can transfer their knowledge of existing applications to a new one and make inferences about the way that the new one will respond to certain types of user interaction.

Guidelines for user interface design are usually referred to as *style guides*, and large organizations with many different information systems produce their own style guides for the design of systems to ensure that all their applications, whether they are produced in-house or by outside software companies, conform to a standard set of rules that will enable users quickly to become familiar with a new application. Figure 16.3 reflects the use of an existing style guide in FoodCo. The layout of the screen with standard heading information at the top, the use of bold text to highlight prompts and labels, the position of the information about function keys and the use of specific function keys for particular commands are all standards within FoodCo for the design of text-based screens. This is important, as it means that a user can be confident that pressing function key F2 in any data entry screen will save the data on the screen.

The use of style guides and the characteristics of a good dialogue relate to dialogue and interface design in general. In the next part of this chapter, we consider how to ensure that the user interface is appropriate to the specific application for which it is being designed.

16.3 Approaches to User Interface Design

16.3.1 Informal and formal approaches

There are many different ways of designing and implementing the elements of the user interface that support the interaction with users. The choices that the designer makes will be influenced by a number of factors. These include:

- the nature of the task that the user is carrying out
- the type of user
- the amount of training that the user will have undertaken
- the frequency of use
- the hardware and software architecture of the system.

These factors may be very different for different systems. They are listed in Figure 16.10 for the FoodCo telesales system and a WAP (Wireless Access Protocol) tourist information system. Systems that are used by members of the public are very different from information systems used by staff. The WWW and WAP have made information systems available to people who are unlikely to receive training in using these systems, and who may have no experience of information systems in business settings.

This way of comparing the two systems and identifying factors that affect their design is very informal. More formal and methodical approaches to the analysis of

	Telesales System	WAP Tourist Information System
The nature of the task that the user is carrying out	Routine task; closed solution; limited options	Open-ended task; may be looking for information that is not available
The type of user	Clerical user of the system; no discretion about use (must use it to do their job)	Could be anyone; discretion about use of system; novice in relation to this system
The amount of training that the user will have undertaken	Training provided as part of job	No training provided
The frequency of use	Very frequent; taking an order every few minutes	Very occasional; may never use it again
The hardware and software architecture of the system	Mini-computer, dumb terminals with text screens, keyboard data entry. All software runs on the mini-computer. Structured programs with subroutines for data access and screen-painting	Mobile telephone screen with keypad and scroll buttons to move through menus. WAP browser runs on mobile telephone, WAP gateway connects to server, which generates WML for WAP browsers and HTML for other browsers using XML and stylesheets

Figure 16.10 User interface design factors for two systems.

usability requirements have been developed by researchers in the discipline of HCI. These approaches can be categorized under three headings:

- structured approaches
- ethnographic approaches
- scenario-based approaches.

These approaches are very different from one another. However, they all carry out three main steps in HCI design:

- requirements gathering
- design of the interface
- interface evaluation.

Each of these approaches has similar objectives in each of these main steps. Typical objectives are shown in Fig. 16.11. However, they differ in the ways that they set out to achieve these objectives. This is described below.

Structured approaches

Structured approaches to user interface design have been developed in response to the growth in the use of structured approaches to systems analysis and design during the 1980s. Structured analysis and design methodologies have a number of characteristics. They are based on a model of the systems development lifecycle, which is broken down into stages, each of which is further broken down, for example into steps that are broken down into tasks. Specific analysis and design techniques are used, and the methodology specifies which techniques should be used in which step. Each step is described in terms of its inputs (from earlier steps), the techniques applied and the deliverables that are produced as outputs (diagrams and documentation). These approaches are more structured than the simple waterfall model of the lifecycle, as they provide for activities being carried out in

Step	Objectives
Requirements gathering	Determine characteristics of the user population: types of user, frequency of use, discretion about use, experience of the task, level of training, experience of computer systems
Design of the interface	Determine characteristics of the task: complexity of task, breakdown of task, context/environment of task
Interface evaluation	Determine constraints and objectives: choice of hardware and software, desired throughput, acceptable error rate
	Allocate elements of task to user or system; determine communication requirements between users and system
	Design elements of the interface to support the communication between users and system in the light of characteristics of the users, characteristics of the task and constraints on design
	Develop prototypes of interface designs
	Test prototypes with users to determine if objectives are met

Figure 16.11 Steps in HCI design and objectives in each step.

parallel where possible rather than being dependent on the completion of the previous step or stage. Typically such structured approaches use data flow diagrams to model processes in the system and take a view of the system that involves decomposing it in a top-down way. Structure charts or structure diagrams are used to design the programs that will implement the system.

Proponents of structured approaches argue that they have a number of benefits.

- They make management of projects easier. The breakdown of the project into stages and steps makes planning and estimating easier, and thus assists management control of the project.
- They provide for standards in diagrams and documentation that improve understanding between the project staff in different roles (analyst, designer and programmer).
- They improve the quality of delivered systems. Because the specification of the system is comprehensive, it is more likely to lead to a system that functions correctly.

Advocates of structured approaches to HCI believe that similar benefits can be brought to HCI by adopting structured approaches. These approaches assume that a structured approach to analysis and design of a system is being used and that a structured approach to the HCI design can take place at the same time and be integrated to some extent into the project lifecycle. Two examples of such approaches are discussed briefly below.

- STUDIO (Structured User-interface Design for Interface Optimization) developed with KPMG Management Consulting in the UK (Browne, 1994)
- The RESPECT User Requirements Framework developed for the European Union Telematics Applications Programme by a consortium of Usability Support Centres (Maguire, 1997)

Structured approaches make use of diagrams to show the structure of tasks and the allocation of tasks between users and the system. They also make extensive use of checklists in order to categorize the users, the tasks and the task environments. Evaluation is typically carried out by assessing the performance of the users against measurable usability criteria. STUDIO is used here as an example of a structured approach.

STUDIO is divided into Stages, and each Stage is broken down into Steps. The activities undertaken in each of the Stages are shown in Figure 16.12. STUDIO uses a number of techniques such as:

- task hierarchy diagrams
- knowledge representation grammars
- task allocation charts
- state machines.

It is not possible to provide examples of all of these here. Statecharts in STUDIO are similar to UML state machines and based on the work of Harel (1988). Examples of UML state machines applied to user interface design are included in Section 17.7. A sample task hierarchy diagram for Take an Order is shown in Fig. 16.13. This diagram applies to the order entry screen of Fig. 16.3. The diagram

Stage	Summary of activities
Project Proposal and Planning	Decide whether user interface design expenditure can be justified. Produce quality plan
User Requirements Analysis	Similar to systems analysis, with focus on gathering information relating to user interface design rather than general functionality
Task Synthesis	Synthesize results of requirements analysis to produce initial user interface design. Produce user support documentation
Usability Engineering	Prototyping combined with impact analysis to provide an approach to iterative development that is easy to manage
User Interface Development	Handover of the user interface specification to developers to ensure that usability requirements are understood

Figure 16.12 Summary of activities in each Stage of STUDIO.

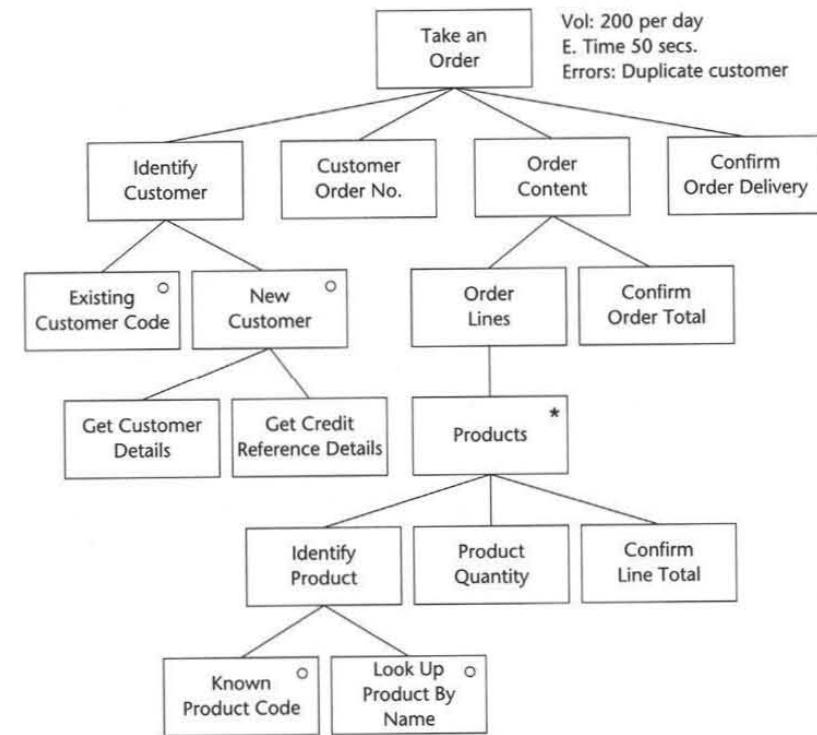


Figure 16.13 Task hierarchy diagram for Take an Order.

is read from top to bottom and left to right. In it, the boxes with a small circle in the top right-hand corner are selections, only one of which will take place each time an order is taken; the box with an asterisk in the top right-hand corner is an iteration, which will take place usually more than once.

Structured approaches may involve evaluation of the user interface designs in a laboratory situation. This reflects the need to have operational measures of usability that can be tested and used to assess the effectiveness of the design. These operational measures are derived from objectives that were gathered during the requirements analysis phase of the project described in Chapter 6. Examples include measures of how quickly users learn to use the system, error rates and time taken to complete tasks.

Serco Usability Services, which emerged out of the National Physical Laboratory in the UK, one of the partners in the consortium that produced the RESPECT methodology, has a usability laboratory in which interface designs are tested with users under laboratory conditions. One criticism of this approach is that people do not use systems under laboratory conditions, they use them in busy offices, noisy factories or crowded public spaces. Usability tests conducted in laboratories lack *ecological validity*. This means that they do not reflect the real conditions in the environment in which the systems will be used. As an alternative to laboratory-based usability studies, ethnographic approaches to the establishment of usability requirements, and to the testing of those requirements, have been proposed.

Ethnographic approaches

The term ethnography is applied to a range of techniques used in sociology and anthropology and reflects a particular philosophy about how scientific enquiry should be carried out in these disciplines. Researchers who employ an ethnographic method seek to involve themselves in whatever situation they are studying. They believe that only by doing this and becoming part of that situation can they truly understand and interpret what is happening. Ethnographic methods belong to a wide range of *qualitative* research methods. (Qualitative means 'concerned with the quality of something' and is usually taken to be the opposite of *quantitative*, which means 'concerned about the quantity of something'. Quantitative methods typically rely on statistics to draw conclusions.) Hammersley and Atkinson (1995:1) provide a definition of ethnography:

In its most characteristic form it involves the ethnographer participating, overtly or covertly, in people's daily lives for an extended period of time, watching what happens, listening to what is said, asking questions—in fact, collecting whatever data are available to throw light on the issues that are the focus of the research.

In HCI this means that the professional charged with carrying out the user interface design spends time with the users immersed in their everyday working life. Only by spending time in this way can the real requirements of the users be understood and documented. Ethnographic methods also emphasize that different users interpret their experience of using systems subjectively, and it is this subjective interpretation that the HCI professional must understand rather than assuming that the system can be assessed objectively.

Some HCI methods are criticised for failing to capture information about the context in which people are using systems, by focusing on the user and their tasks. Ethnographic approaches attempt to answer this criticism by the professional

experiencing the daily working life of the people who will be the users of the system so that it can be better designed to meet their needs. Some structured approaches have also attempted to respond to the criticism about failing to take context into account, typically by adding some kind of contextual analysis questionnaire to the battery of checklists that they use.

There is no one ethnographic approach that covers the three steps in user interface development: requirements gathering, design of the interface and interface evaluation. There are a number of approaches that can be classed as ethnographic, although their originators may use particular terms to describe their approaches.

Contextual enquiry is an approach developed by John Whiteside and others at Digital Equipment Corporation (Whiteside et al., 1988). It is used to carry out evaluation of the usability of a system in the users' normal working environment. The aim of contextual enquiry is to get as close to the users as possible and to encourage them to develop their interpretation of the system.

Participative or co-operative design and evaluation involve users actively in the design and evaluation of the user interface of systems (Greenbaum and Kyng, 1991). The social and political issues in the workplace that affect the use of systems are part of the material that is captured by these approaches.

Ethnographic approaches use a range of techniques to capture data: interviews, discussions, prototyping sessions and videos of users at work or using new systems. These data are analysed from different perspectives to gain insights into the behaviour of the users. Video is also used in other approaches, particularly in laboratory-based usability studies. Analysis of video can be particularly time-consuming.

Scenario-based approaches

Scenario-based design has been developed by John Carroll and others (Carroll, 1995). It is less formal than the structured approaches but more clearly defined than most ethnographic approaches. Scenarios are step-by-step descriptions of a user's actions that can be used as a tool in requirements gathering, interface design and evaluation. Use cases are similar to scenarios, and Carroll's book includes chapters by Ivar Jacobson and Rebecca Wirfs-Brock, who have developed use cases and responsibility-based approaches to modelling interaction in object-oriented analysis and design. Of the three approaches discussed here, scenario-based design fits best with use case modelling.

Scenarios can be textual narrative describing a user's actions or they can be in the form of storyboards (a series of pictures that depict those actions), video mock-ups or even prototypes. Figure 16.14 shows a scenario that describes the actions of Peter Bywater from Agate when he demonstrates how he creates notes following the interview in Chapter 6.

Scenarios can be used like this in requirements gathering to document the actions that a user carries out in their current system. They can also be used to document ideas about how the user would see themselves using the new system. This is called envisioning the design. Alternative scenarios describing different approaches to the design can be compared by the designers and the users. Figure 16.15 shows a scenario describing how a staff member at Agate might use the new system to create a new note about an advert.

For evaluation of the system, more detailed scenarios are prepared so that the

Pete starts up the word-processor.
 He types in a title for the note and changes its style to *Title*.
 He types in two paragraphs describing his idea for an advertisement for the Yellow Partridge campaign to be used in fashion magazines in Europe during the summer of 2005.
 He types his initials and the date and time.
 He uses the short-cut keys to save the file.
 The save-as dialogue box appears and, using the mouse, he changes to the *Summer 2005 Campaign* folder in the *Yellow Partridge* folder on the server.
 He scrolls to the bottom of the list of files already in the folder and reads the title of the last note to be added, *Note 17*, he calls the new note *Note 18* and clicks on Save.
 He exits from the word-processor.

Figure 16.14 Scenario describing Pete Bywater of Agate adding a new note.

actual system can be compared against the expectations that the designer has of how the user will interact with it. Carroll (1995) claims that scenarios can be used in more than just these three ways. He lists the following roles for scenarios:

- requirements analysis
- user–designer communication
- design rationale
- envisionment
- software design
- implementation
- documentation and training
- evaluation

The user selects Add a Note from the menu. A new window appears.
 From the list box at the top of the window she selects the name of the client.
 A list of campaigns appears in the list box below, and she selects a particular campaign.
 A list of adverts appears in the next list box, and she selects a specific advert.
 She types a few paragraphs into a text box to describe her idea for the advert. She fills the space on screen and a vertical scrollbar appears and the text in the text box scrolls up.
 She enters her initials into a text box, and the system checks that she is allocated to work on that campaign.
 The date and time are displayed by the system, and the Save button is enabled.
 She clicks on the Save button and the word Saved appears in the status bar.
 The text box, the text field for initials and the date and time are cleared.

Figure 16.15 Scenario describing how a user might add a note in the new system.

- abstraction
- team building.

Two of these are worth further comment: user–designer communication and design rationale.

In Chapter 6 we pointed out that the diagrams used by systems analysts and designers are used to communicate ideas, among other things. Information systems professionals need to communicate with the end-users of the systems that they are developing. Scenarios provide a means of communication that can be used by professionals and end-users to communicate about the design of the users' interaction with the system. They are simple enough that users can produce them without the need for the kind of training that they would need to understand class diagrams, for example. Scenarios can be used with use cases. The use cases can provide a description of the typical interaction; scenarios can be used to document different versions of the use case, for example, to document what happens when a user is adding a new note but is not authorized to work on the project they try to add it to. Use cases are concerned with the functionality offered by the system, while scenarios focus on the interaction between the user and the system.

Scenarios can be supported by additional documentation to justify design decisions that have been taken. Carroll (1995) calls these design justifications *claims*. The designer can document the reasoning behind alternative designs, explaining the advantages and disadvantages of each. Figure 16.16 shows some claims for the scenario in Fig. 16.15. These usability claims from design can be checked during evaluation of the software or of prototypes.

Scenario-based design can result in large volumes of textual information that must be organized and managed so that it is easily accessible. There is a document management task to be undertaken that requires a rigorous approach to control different versions of scenarios and to cross-reference them to claims and feedback from users. Developers run the risk of delaying implementation while they work through and document alternative scenarios for different parts of the system. Rosson and Carroll (1995) present one way to try to prevent this happening. They use a computer-based tool to develop and document their scenarios and to develop working models of the scenarios in Smalltalk as they go along. This allows them to document software implementation decisions at the same time, and they propose

The Save button is disabled until the user has selected a client and a campaign, entered some text and entered his or her initials. This prevents the user attempting to save the note before all data has been entered and getting an error message.

The initials of the user could be entered automatically from their network login, but observation shows that the creative staff often work together as a group and different people will come up with ideas that they record as notes. It would be inconvenient for them to be logging in and out of the system each time a different person wants to enter a new note. For this reason, they are required to enter their initials.

The initials, date, time and text fields are cleared after a note is saved, but the client, campaign and advert list boxes are left untouched so that the user can enter another note for the same advert or campaign without having to reselect these items.

Figure 16.16 Claims for the design scenario in Fig. 16.15.

that there are benefits to recording design decisions and software implementation decisions together in this way.

These three types of approach have been presented as though they were very separate. However, there are elements that they have in common. Some structured approaches have attempted to take on board the criticisms that they fail to address the context in which people work and use computer systems. Ethnographic methods may use the same data gathering techniques as other approaches, and may be used to provide information that can be used as the basis for drawing up scenarios. What they all share is a concern to enhance the usability of information systems and a recognition that usability issues must be integral to the design of computerized information systems.

16.3.2 Achieving usability

People often talk about how *user-friendly* a piece of software is, but it is often very difficult to tell what it is they mean by this. As a concept it is very vague. Usability may seem like a similar concept, but the HCI community has developed definitions of usability that can be used to test a piece of software. Shackel (1990) produced definitions of four criteria that were originally developed in the 1980s:

- *Learnability*—how much time and effort is needed to achieve a particular level of performance
- *Throughput*—the speed with which experienced users can accomplish tasks and the number of errors made
- *Flexibility*—the ability of the system to handle changes to the tasks that users carry out and the environment in which they operate
- *Attitude*—how positive an attitude is produced in users of the system.

In Chapter 6 we mentioned the International Standards Organization (ISO) definition of usability as ‘the degree to which specific users can achieve specific goals within a particular environment; effectively, efficiently, comfortably and in an acceptable manner’. These criteria can be used in conjunction with the users’ acceptance criteria documented during requirements gathering to assess how easy a software product is to use. Some of these can be quantified, for example, we can count the number of errors made by staff at FoodCo using the new system and compare that with the number of errors made with the old system and the objectives that they have set for the new system.

Sometimes conflicts will exist between different criteria and between usability criteria and other design objectives, and the designers will have to make compromises or trade-offs between different objectives. In particular, increasing flexibility is likely to conflict with the objective of developing the system at a reasonable cost.

16.4 Standards and Legal Requirements

In Section 16.2.5 we discussed style guides, which set standards for the design of user interfaces. Style guides like these determine the use of standard layouts, colour and function keys and the overall appearance of the system. The International Standards Organization (ISO) has produced standards that have a broader impact

on the use of computer systems. ISO 9241 is an international standard for the ergonomic requirements for work with Visual Display Terminals, including both hardware and software. The standard covers physical aspects of the user’s workstation (including positioning of equipment and furniture), the design of the computer equipment and the design of the software systems. ISO 14915-2002 is a further standard, entitled ‘Software ergonomics for multimedia user interfaces’, which gives recommendations for, and guidance on, the design, selection and combination of interactive user interfaces that integrate and synchronize different media. These standards are intended to ensure the quality of systems and to prevent local standards becoming barriers to free trade.

In the European Union (EU), this has been taken one step further and the EU Council issued a directive on 29 May 1990 that has the force of law for member states. In the UK, for example, this directive has been implemented in the Health and Safety (Display Screen Equipment) Regulations 1992. Under these regulations all workstations must now comply with certain minimum requirements, and employers have a duty in law to ensure the health and safety of employees using display screen equipment.

The regulations provide a number of definitions:

- display screen equipment—any alphanumeric or graphic display screen
- user—an employee who habitually uses display screen equipment as a significant part of his or her normal work (see table of criteria in Health and Safety Executive, 2003)
- operator—self-employed person as above
- workstation—display screen equipment, software providing the interface, keyboard, optional accessories, disk drive, telephone, modem, printer, document holder, work chair, work desk, work surface or other peripheral item and the immediate work environment around the display screen equipment.

The definition of display screen equipment excludes certain types of equipment, such as equipment in the cab of a vehicle, cash registers and some portable equipment.

As well as covering the physical equipment that is used by the user, the regulations cover environmental factors such as position of equipment, lighting, noise, heat and humidity in the workplace. Employers are required to:

- analyse workstations to assess and reduce risks
- take action to reduce risks identified
- ensure workstations meet the requirements of the regulations by the necessary dates
- plan the work activities of users to provide breaks
- provide eyesight tests for users
- provide corrective appliances for eyes if required
- provide training relevant to health and safety issues and workstations
- provide information to employees about health and safety risks and measures taken to reduce them.

The analysis of workstations in order to reduce risks includes analysis of the software, and the guidelines published to assist employers to meet their responsibilities state the following requirements.

Employers must take into account the following principles in designing, choosing, commissioning and modifying software and in designing tasks for people that require them to use display screen equipment.

- The software that is used must be suitable for the task.
- The software must be easy to use and able to be adapted to the level of knowledge or experience of the operator or user.
- The employer is not allowed to use any kind of quantitative or qualitative checking facility without the knowledge of the operators or users.
- Systems must give feedback to operators or users about the performance of the systems that they are using.
- Systems must display information for users both in a format and at a pace that are adapted to the operators or users.
- The principles of software ergonomics must be applied, particularly to the way that people process data.

Clearly the effect of this is to require employers, and so also software developers, to demonstrate that they are applying good HCI practice in the way that they design software.

Many countries in the world have regulations in place to promote good practice in workstation use. The USA is a significant exception to this. The Occupational Safety and Health Administration (OSHA) has proposed rules designed to prevent musculo-skeletal disorders caused by poor work design, bad posture and repetitive activities and covering workstation design and layout. These were rejected by Congress, backed by industry lobbyists. However, this is the exception rather than the rule in developed countries and there is plenty of online material on the subject, including the OSHA's own website, <http://www.osha.gov>, which provides advice on good workstation design.

One area where the United States does have some strong legislation is in the area of access to information technology by people with disabilities. The Americans with Disabilities Act (1990) ensures equal opportunity for persons with disabilities in employment, State and local government services, public accommodations, commercial facilities and transportation. Because so many services are now delivered, booked or accessed electronically, and so much work involves the use of IT, this has had an impact on the design of information systems and US businesses have improved the accessibility of their systems.

In the UK, the Disability Discrimination Act (1995) has, since 1999, required that service providers take reasonable steps to change practices that make it unreasonably difficult for disabled people to make use of their services. This requirement is defined in the Code of Practice (Disability Rights Commission, 2002), which specifically mentions websites and gives the example of an airline online booking system as the kind of system that is covered by the Act.

Access to information systems by people with disabilities has improved in many countries, driven by the expansion of the Internet and by legislation. The World Wide Web Consortium runs the Web Accessibility Initiative to develop guidelines and techniques for improving the accessibility of material on the Internet.

16.5 Summary

System designers must take account of the requirements of the people who will use their software if they are to reduce errors and maximize the satisfaction of the users with the system. The user interface can be viewed as part of a dialogue between the user and the system and there are a number of characteristics of good dialogue and design that can be used to ensure that the user is supported by the interface and assisted in carrying out their primary task.

It is possible to apply an informal approach to determining characteristics of the users, the task and the situation that will affect the interface design, or to apply a more formal approach using either structured, ethnographic or scenario-based techniques or some combination of these. The main aim of this is to produce software that can be demonstrated to meet the usability requirements of the people who will use it. This may be done in order to ensure compliance with international standards or it may be to meet legal requirements in some countries.

Review Questions

- 16.1** Think of a computerized information system that you use regularly. This could be a library system, an automated teller machine (ATM) that you use to get cash, a database that you use in your work or any other system that you are familiar with. Write down which elements of the interface support the five tasks listed at the start of Section 16.2.1.
- 16.2** For each of the elements of the interface that you have listed in Question 16.1, write down your ideas about how they could be improved.
- 16.3** What is the difference between the dialogue and direct manipulation metaphors?
- 16.4** Make a list of direct manipulation metaphors that are used in a GUI that you are familiar with. Are there any metaphors that do not work as you might expect?
- 16.5** What are the four characteristics of good dialogues described in Section 16.2.4?
- 16.6** Figure 16.9 shows the *Yes* button in a dialogue highlighted. What do you think is the risk associated with making this the active button by default?
- 16.7** For the system that you wrote about in Question 16.1, note down information relevant to the design factors in Fig. 16.10.
- 16.8** List as many differences as you can think of between structured, ethnographic and scenario-based approaches.
- 16.9** Make your own list of what you think the advantages and disadvantages could be of structured, ethnographic and scenario-based approaches.

Case Study Work, Exercises and Projects

- 16.A** Using a user interface that you are familiar with as an example, try to identify features that you think might be part of the style guidelines for that user interface. This could be a GUI, a website or even a mobile phone.

16.B Using the four criteria for good dialogues discussed in Section 16.2.4, evaluate an application that you use regularly. Identify the ways in which it meets these criteria and the ways in which it does not meet the criteria. Suggest ways in which it could be improved.

16.C Write a scenario to describe what is done when Rik Sharma of FoodCo starts to plan staff allocation, based on the interview transcript in Exercise 6.B (Chapter 6). (Make sure that you concentrate on what he does and not on what is done by other staff at other times.)

16.D For the system that you wrote about in Question 16.1, identify measurable objectives that could be used to measure how usable that system is. (You may like to start by thinking about how long it takes you to use it and how many errors you make.)

16.E Find out whether there are any legal requirements on software designers to comply with legislation that covers ergonomics or HCI in your country. Write a short report to summarize these requirements as though you were an analyst reporting to your manager on this legislation.

16.F Look at the World Wide Web Consortium's website on access for users with disabilities at <http://www.w3.org/WAI/>. Identify some of the practical recommendations. Choose a website (perhaps one you have developed yourself) and evaluate it against these criteria. Write a short report summarizing what improvements need to be made.

Further Reading

Many computer science and information systems courses now include HCI as a subject. If you have not come across HCI before and want to find out more, there are a number of suitable textbooks, such as Booth (1989), Preece et al. (1994) and Dix et al. (1998). A classic text in this area is Shneiderman and Plaisant (2004), which has been updated to cover graphical user interfaces in more detail since its first publication.

Style guidelines are available from some of the largest companies in the industry. Links are available to the online versions of these in the book's website.

For a structured method for user interface design Dermot Browne's book (Browne, 1994) provides a step-by-step approach to user requirements analysis, task analysis, usability and interface design. Browne also uses Harel state machines to model the behaviour of the interface in a more thorough way than many authors. Horrocks (1999) provides another view of how to use state machines in interface design. Carroll (1995) provides a good coverage of scenario-based methods and is very practical in approach.

In the UK, Her Majesty's Stationery Office (HMSO) publishes a booklet that explains the requirements of the Display Screen Regulations. This book's website includes links to other resources on health and safety and workstation ergonomics. Australian and Canadian government organizations provide a good starting point for investigating standards and legislation. The US OSHA also has a good website, <http://www.osha.gov>, with guidelines and checklists for ergonomic workstation design.

One of the best overview sites on access to information systems for people with disabilities is the World Wide Web Consortium's Web Accessibility Initiative (<http://www.w3.org/WAI/>), which has links to information on legislation in a number of countries.

Designing Boundary Classes

Learning Objectives

In this chapter you will learn

- what we mean by the presentation layer
- how prototyping can be applied to user interface design
- how to add boundary classes to the class model
- how to model boundary classes in sequence diagrams
- how design patterns can be applied to the design of the user interface
- how to model the control of the user interface using state machine diagrams.

17.1 Introduction

In Chapter 12 we introduced the three-tier system architecture. The presentation layer in this architecture contains the boundary classes that handle the interface with the user—usually windows and reports—or with other systems. We shall be concentrating here on interaction with the human user. We can use the techniques and diagrams of UML to model the user interface by adding detail to the boundary classes in the class model. These classes handle the user interface and allow us to design business (or entity) classes that do not contain details of how they will be presented. This enables the reuse of the business classes.

The reasons for adopting a layered architecture are revisited in Section 17.2. Different authors use different terms for the user interface or presentation layer, and these terms are briefly introduced.

Practical example window layouts are used to show how the interaction between the user and the system is designed. There is rarely one right solution to the problem of designing the user interface for a particular application, and prototyping can be used to try out different interface designs (Section 17.3). The UML notation for packages and for package dependency can be used to show how class diagrams can reference classes from reusable class libraries (Section 17.4) and how

boundary classes can be placed in separate packages. The window layouts can be modelled as classes if required, but this is not always necessary unless the behaviour of the interface or of graphical objects in the user interface is the subject of the application being developed.

The sequence diagrams developed in Chapter 9 can be extended to include the detail of interaction with the boundary classes (Section 17.5), and the model of the boundary classes is developed iteratively as we increase our understanding of the interaction (Section 17.6). This can include the development of an inheritance hierarchy of boundary classes in order to model common features of dialogues and reports that occur in many use cases.

- Patterns can be used to provide generic models for the way that the interaction will work. Many systems written in Smalltalk use the Model–View–Controller (MVC) architecture, which separates the model (domain or business classes) from classes that handle the interaction between user and system. In Section 17.7 we examine the MVC architecture in terms of patterns.

The dynamic behaviour of the user interface is modelled with state machine diagrams (Section 17.8). UML state machines were used in Chapter 11 to model the response of objects to events that take place during their lifetimes and to show how they change state as time passes. The same notation can be used to show the state of the user interface and how it responds to events such as mouse clicks on buttons or the entry of text into data entry screens. A state machine for the control class that manages the user interface for one use case is developed and alternative versions are presented.

17.2 The Architecture of the Presentation Layer

In Chapter 7 the idea of boundary classes was introduced, and in Chapters 12 and 13 a layered model of the system was presented. The three-tier architecture is a common way to separate out user interface classes from the business and application logic classes and from mechanisms for data storage. There are a number of reasons for doing this, and these are shown in Fig. 17.1.

This is not to say that classes should contain no means of displaying their contents to the outside world. It is common practice to include in each class a `print()` (or a `toString()`) method that can be used to test the classes before the presentation layer has been developed. Such methods typically take an output stream (a file or a terminal window) as a parameter and produce a string representation of their attributes on that stream. This enables the programmer to check the results of operations carried out by classes without needing the full system in place. (See Chapter 19 for more on testing.)

The three-tier architecture was discussed in Section 12.5.2. Different approaches to object-oriented development use different names for the layers of the three-tier architecture. The Unified Process uses the terms boundary, control and entity classes for the three types of classes and these are the terms that we have used. Developers using Smalltalk to implement systems have for many years adopted a similar approach using the Model–View–Controller (MVC) approach that was described in Chapter 12. In the (MVC) approach a system is divided into three

Logical design	The project team may be producing analysis and design models that are independent of the hardware and software environment in which they are to be implemented. For this reason, the entity classes, which provide the functionality of the application, will not include details of how they will be displayed.
Interface independence	Even if display methods could be added to classes in the application, it would not make sense to do so. Object instances of any one class will be used in many different use cases: sometimes their attributes will be displayed on screen, sometimes printed by a printer. There will not necessarily be any standard layout of the attributes that can be built into the class definition, so presentation of the attributes is usually handled by another class.
Reuse	One of the aims is to produce classes that can be reused in different applications. For this to be possible, the classes should not be tied to a particular implementation environment or to a particular way of displaying the attribute values of instances.

Figure 17.1 Reasons for separating business and user interface classes.

components:

- *Model*—the classes that provide the functionality of the system
- *View*—the classes that provide the display to the user
- *Controller*—the classes that handle the input from the user and send messages to the other two components to tell them what operations to carry out.

Whatever approach is chosen in a particular project, all these approaches share the objective of keeping the behaviour of the interface separate from the behaviour of the classes that provide the main functionality of the system. To use the anthropomorphic style of some authors about object-oriented systems, the entity classes ‘know’ nothing about how they will be displayed.

Taking a three-tier architectural approach does not necessarily mean that the different types of classes will end up running on different machines or even that they will be completely separate. It is useful to distinguish between the logical architecture of the system and the physical architecture. The physical architecture may combine layers of the logical architecture on a single physical platform or it may split logical layers across physical systems. If you are designing Java applets, some of the responsibilities for control will be located in the applet class itself, together with the responsibilities of the boundary class, while other control responsibilities may be located in classes on a server together with entity classes. In a distributed system, the entity classes may exist in different databases on different servers and the control classes would pull the data together from these different sources in order to deliver it to the boundary classes.

For the Agate system, we are going to keep the boundary, control and entity classes separate. The boundary classes will run on the users’ machines, while the control classes will be located on servers, and the entity classes will initially be on

local servers but may later be distributed in different offices. (We shall discuss mechanisms for achieving this in Chapters 18 and 19.)

In the next four sections, we shall develop the boundary classes by:

- prototyping the user interface
- designing the classes
- modelling the interaction involved in the interface
- modelling the control of the interface using state machines.

17.3 Prototyping the User Interface

Prototyping was discussed in Chapter 3 as an approach to the development life-cycle and in Chapter 6 as a way of helping to establish what the requirements for a system are. In Chapter 6 we used it to produce models of the user interface. The UML models we have produced so far have been analysis and design diagrams; they are rather like an architect's drawings—they represent what the finished product will be like but they do not really show how it will look. A prototype is a model that looks, and to some extent behaves, like the finished product, but is lacking in certain features; it is more like an architect's scale model of a new building.

There are different kinds of prototype that can be built. A prototype that only provides a model of the user interface is one example of a *horizontal prototype*. It is horizontal because it deals with one layer of the layered architecture of the system. A *vertical prototype* takes one subsystem of the whole system and develops it through each layer: user interface, business classes, application logic and data storage. A horizontal prototype need not only deal with the user interface; there may be circumstances where it is more important to prototype the middle tier in order to test whether an innovative aspect of the system functionality works correctly.

Another distinction is made between those prototypes that are developed further and eventually, by an iterative process, become part of the finished system, and those prototypes that are simply used to test out design ideas and are then thrown away after they have served their purpose. This second kind of prototype is known as a *throwaway prototype*. A throwaway prototype can be built using any programming language that is suitable for the purpose. Figure 17.2 shows a prototype of the user interface for Agate that was created in a matter of minutes using Microsoft Visual Basic. Although this application will be developed in Java, a visual programming environment such as Visual Basic or Delphi can be used to produce prototypes of the user interface. In fact, the Java prototype examples used in the book took no more time than the Visual Basic examples, as they were also produced in a visual programming environment—Visual Café for Java.

Visual programming environments can be used to develop prototypes of the user interface to applications. These can be shown to the users and used, for example, in conjunction with techniques from scenario-based design (described in Chapter 16) to agree with users how they will interact with the user interface. In this way, prototypes can be used to design the interaction between the system and the user and establish a set of guidelines for how this interaction will take place.

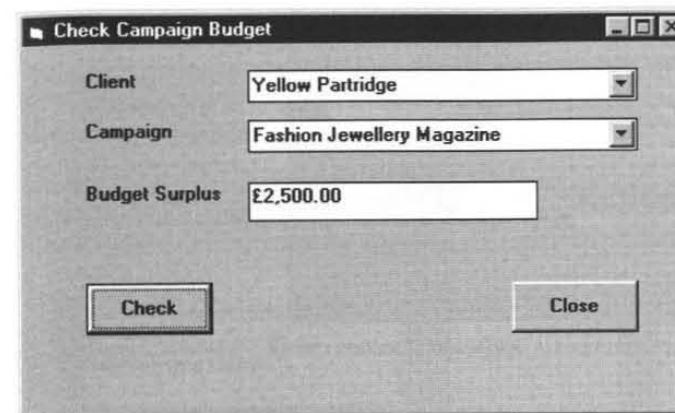


Figure 17.2 Visual Basic prototype of the Check Campaign Budget interface.

Because visual programming environments are so easy to use, developers are often tempted to develop applications from the outside in: starting with the interface and linking the functionality to the visual components that appear on the screen. Without thorough analysis of the business requirements for the system, this can lead to a blurring of the distinction between the presentation layer and the business classes and application logic. Applications developed in this way often have a large amount of program code associated with interface objects such as buttons. This program code should be an operation of a control class or of one or more entity classes. If it is linked to a button it cannot be reused in the same way as if it is carefully encapsulated in a class. Typically, the programmer then needs to reuse the code in another window and copies and pastes the code to another button in the new window; then when a change is made to the code linked to one button it may not be copied to the code linked to the second button and discrepancies creep into the system. This is not to say that it is not possible to develop good applications in visual environments; the important thing is that a thorough analysis should have been carried out first and the business objects and application logic should be kept separate from the visual components.

Prototyping can be used to try out alternative approaches to the same use case. In the example screen layout shown in Fig. 17.2 we have assumed that the users will select first a client and then a campaign from dropdown lists. There are many possible alternatives to this. Three of these are:

- to use a separate look-up window for each class
- to allow the user to enter part of a name (for example, of a client) and for a list of close matches to be returned
- to use a tree structure which shows the instances of clients and campaigns in a tree-like hierarchy.

Prototyping allows us to experiment with these approaches and build models which the users can try out for themselves. Figures 17.3 and 17.4 show screen

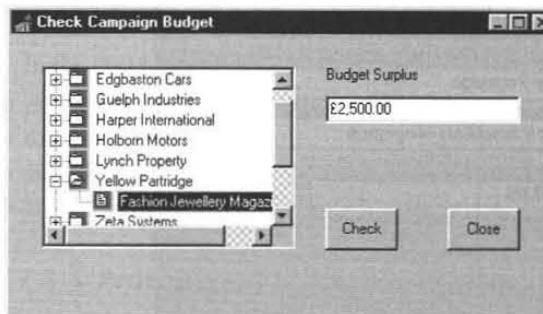


Figure 17.3 Prototype developed in Delphi using TreeView control.

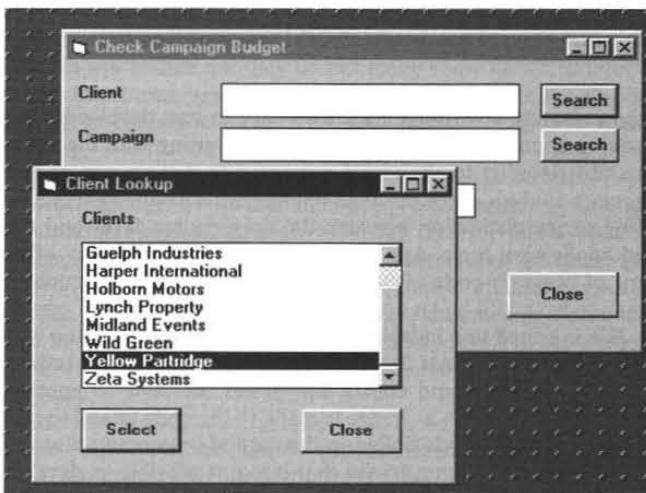


Figure 17.4 Prototype developed in Visual Basic showing separate look-up window.

shots of prototypes based on two of these different ways of handling the look-up process.

The choice of how the look-up in this use case is handled on screen will be determined by the style guidelines that were discussed in Chapter 16. It is important that style guidelines are agreed before development starts; prototyping can be used during design to try out various different interface styles and to get the users' agreement on which style will be followed.

Whichever style is adopted here will be adopted in other use cases in which the user needs to be able to look up clients from a list. The same approach will also be adopted in all use cases in which the user looks up any kind of class. For example, Fig. 17.5 shows the same method as in Fig. 17.4 being used to look up campaigns for a particular client in the use case for Add a concept note.

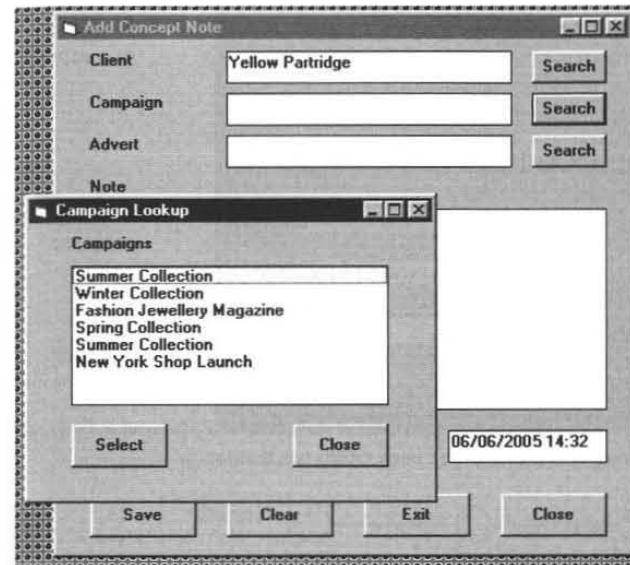


Figure 17.5 Use of the same style of look-up as in Fig. 17.4 in a different use case.

17.4 | Designing Classes

The next step is to design the classes that will provide the user interface. The use case for Check Campaign Budget is used as an example. At the simplest level, there will be an object that provides an interface onto the functionality of this use case. This could be a dialogue window like the one shown in Fig. 17.6 (in the foreground). The analysis collaboration for this use case is shown in Fig. 17.7.

In the simple analysis collaboration in Fig. 17.7 we have not shown the class Client because it does not participate in the main functionality of the use case. In order to calculate what is left in the budget for a campaign we do not need the client object. However, in order to find the right campaign we do need the client: we home in on the right campaign in the context of the particular client. We need to be able to list all the clients and display them in the first dropdown. Once the client has been selected, we then need to list all the campaigns for that client in the second dropdown. In the approach that we have taken in this user interface design, using dropdowns rather than separate dialogue windows, we may want to add further control classes to the collaboration: one to list the clients and the other to list the campaigns. This is shown in Fig. 17.8.

If we had adopted the user interface style of Figs 17.4 and 17.5, then we should have separate user interface classes for each of these, and the collaboration would look like Fig. 17.9.

We will work with the collaboration of Fig. 17.8. We shall treat instances of the boundary class `CheckCampaignBudgetUI` as single objects. We may not want to be concerned about the objects that make it up. In reality, this window may well be

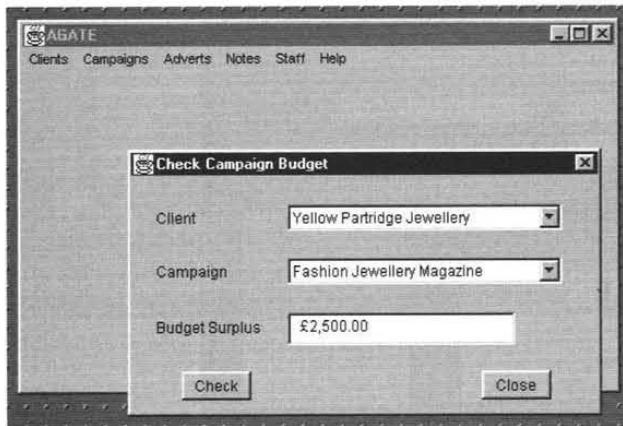


Figure 17.6 Dialogue window for the use case Check Campaign Budget.

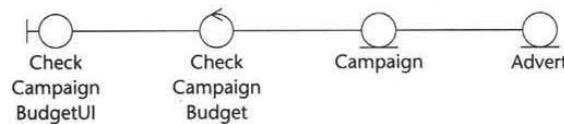


Figure 17.7 Collaboration for the use case Check campaign budget.

an instance of a subclass of a class such as `Dialog` that is available in a library of user interface classes, and it may contain a number of components: buttons, labels, dropdowns and a textbox. This can be shown in a class diagram, as in Fig. 17.10 (Choice is the Java AWT term for a dropdown list). The composition associations represent the fact that the `CheckCampaignBudgetUI` is made up of instances of the other classes. (Alternatively, this can be represented as a class with attributes

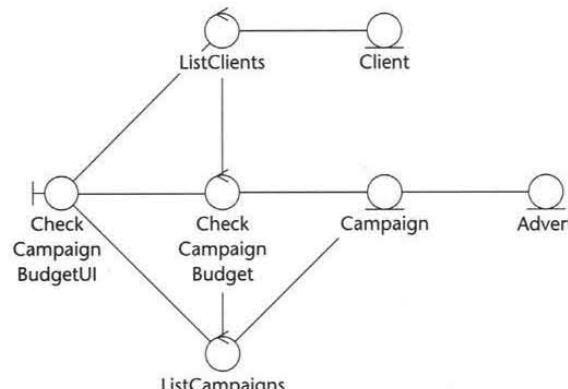


Figure 17.8 Extended collaboration for the use case Check campaign budget.

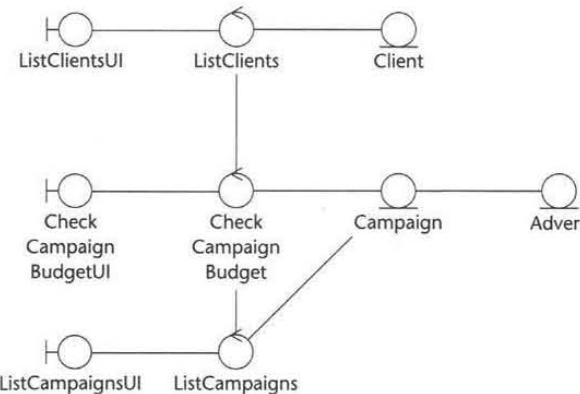


Figure 17.9 Revised collaboration for the use case Check campaign budget.

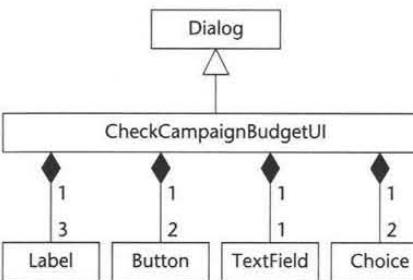


Figure 17.10 Class diagram showing dialogue components.

for each of the components. This is shown in Fig. 17.11 and makes it easier to draw a class diagram for the boundary classes.) The component classes that are used here all come from the Java Abstract Windowing Toolkit (AWT). The `CheckCampaignBudgetUI` class is dependent on the classes in the AWT, and this can be shown using packages in a class diagram, as in Fig. 17.12.

The «import» stereotype on the dependency shows that it will be necessary to import the classes from this AWT package in order to make them available to be used with the classes in the Application Windows package. How this is done will depend on the language that is used for implementation. In Java it is done simply with a line of code:

```
import java.awt.*;
```

In C# this is done with a using statement:

```
using System.WinForms;
```

We have used the «import» stereotype as it is now part of the UML Infrastructure Specification (OMG, 2004c).

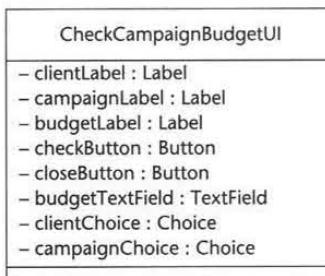


Figure 17.11 Class for dialogue window showing dialogue components as attributes.



Figure 17.12 Class diagram to show dependency between classes in different packages.

Figure 17.13 illustrates how classes from other packages can be shown in the class diagram by adding the pathname of the package, using two colons, as a separator, to the name of the class. The Java AWT is only one example of this. Most object-oriented programming languages or development environments are provided with *class libraries* that contain many of the classes that are needed to build a working system. Microsoft, for example, provides the Windows Forms library, which includes all the classes such as buttons and text fields that are required to build a Windows interface to an application. These classes are grouped together and provided in what UML terms packages. This is an example of the reuse that is claimed as a benefit of object-oriented systems. These user interface classes, whether in Java or C# or another language, have been implemented by other developers and can be reused in many new applications.

If the application being designed is mainly concerned with the behaviour of the objects in the interface itself, for example a drawing package, a CASE Tool or an application with a strong visual element, then it may be advisable to model the user

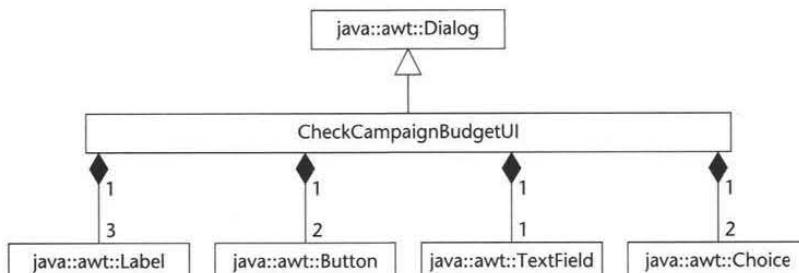


Figure 17.13 Class diagram showing AWT components with their package name.

interface using a class diagram as shown in Fig. 17.13. In most applications in which the user interface will display text and numbers, it is not necessary to produce a model of the classes that make up the interface. It may be useful to show them in the style of Fig. 17.11 in a class diagram in a separate package. Note that there are not normally associations among the classes in the interface package in the way that there are associations between entity classes in the domain model. However, there may be transient links between instances of these classes.

17.5 Designing Interaction with Sequence Diagrams

The sequence diagram for the use case Check campaign budget in Fig. 10.4 did not show the boundary or control classes, but concentrated on the operations of the entity classes. We have shown it again in Fig. 17.14. The communication diagram of Fig. A3.8 showed the boundary and control classes, and we shall now elaborate the interaction in more detail.

We need to add the boundary and control classes of the collaboration in Fig. 17.8 to this sequence diagram. (If we had adopted the design from Fig. 17.9, then we would need to add three boundary classes rather than one.) Rather than trying to draw the entire sequence diagram in one go, we shall build up the interaction step by step. We are assuming here that it is an instance of the control class *CheckCampaignBudget* that is created first and that this creates a new instance of the *CheckCampaignBudgetUI* class to handle the user interface. As soon as it has created the boundary class, the control class needs to have the first dropdown

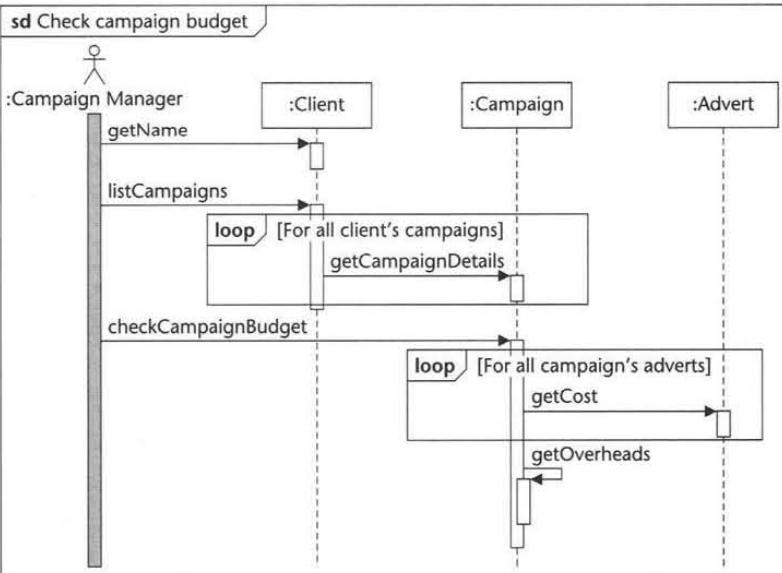


Figure 17.14 Sequence diagram for use case Check campaign budget.

populated with the names of all the clients, so it creates an instance of the control class `ListClients` and requests it to pass back the client names to the boundary class, passing it a reference to the boundary class in the message (`ccbUI`). The instance of `ListClients` sends the message `addClientName(name)` repeatedly to the boundary class until it has finished. It then returns control to the `CheckCampaignBudget` instance and destroys itself. The main control class can now enable the boundary class, allowing the user to select a particular client. This is shown in Fig. 17.15.

We have shown the names of return values and parameters in this diagram to illustrate what is happening. For example, the instance of `CheckCampaignBudget` needs to have a reference to the instance of `CheckCampaignBudgetUI` (`ccbUI`) so that it can pass it to the instance of `ListClients`. This enables `:ListClients` to send the `addClientName` message directly to `:CheckCampaignBudgetUI`. When `:CheckCampaignBudgetUI` is created, it is passed a reference to the control class instance `:CheckCampaignBudget(this)`. In this way, it will be able to send messages back to the main control class when it needs to notify it of events.

Instances of `:CheckCampaignBudgetUI` need to be able to respond to the message `addClientName(name)`. Many other boundary classes will need to allow the user to select a client from a dropdown, for example `AddConceptNoteUI`. We shall want to reuse `ListClients` in all the use cases where a list of clients has to be displayed in a boundary class, but cannot expect `ListClients` to know about all the different boundary classes to which it could send the message `addClientName(name)`. We can use an interface to specify the operations that all these boundary classes must respond to. (See Section 14.4 for an explanation of interfaces.) We could define the interface `ClientLister`, and all the boundary classes that need to display a list of clients will realize it, as `CheckCampaignBudgetUI` does in Fig. 17.16.

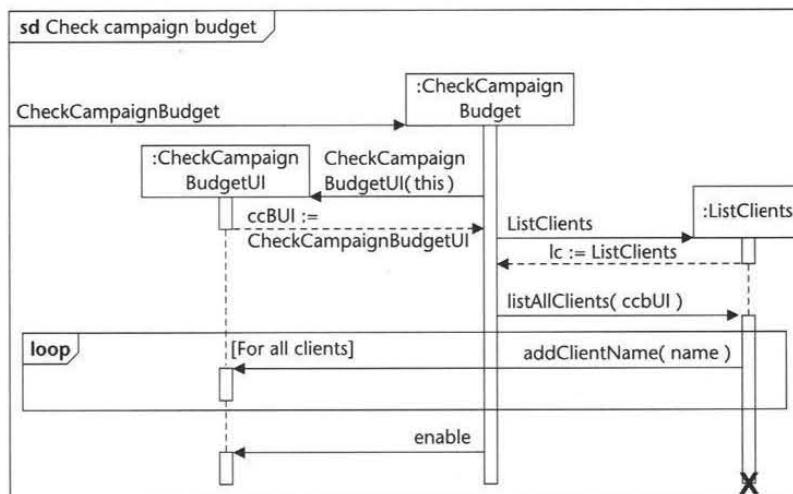


Figure 17.15 First part of detailed interaction for use case Check campaign budget.

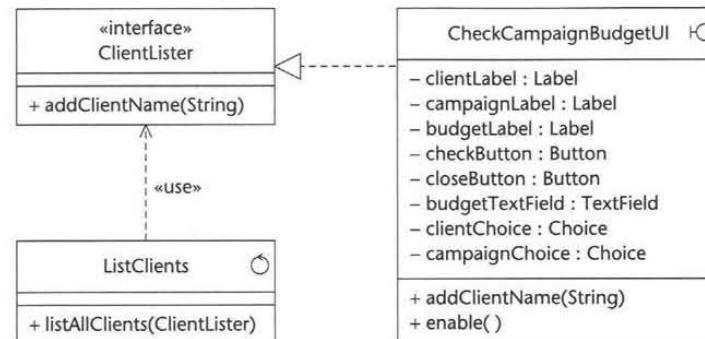


Figure 17.16 ClientLister interface.

Design of other use cases may identify other operations that must be part of this interface, for example `clearAllClientNames()` or `removeClientName(name)`. The operation `addClientName(name)` must be implemented by program code in the class definition for `CheckCampaignBudgetUI`. For example, in Java it may be as in Fig. 17.17 (with much code not shown).

One of the important features of this design is that any object that wants to manipulate the list of clients in the boundary object must do so through operations. We could change the design of the boundary class so that it displayed the list of clients in a scrolling list (a `List` in Java). We would then have to change the Java program that implements `CheckCampaignBudgetUI`, but we would not have to change the classes that use it, because the interface remains the same. (Other object-oriented languages use slightly different ways of achieving the same thing, but the principle is the same.)

Note that in Fig. 17.15 we have not shown what the `ListClients` control class instance does to get the names of all the clients. This level of detail can be hidden in this diagram. UML 2.0 introduces the notation of interaction occurrences to allow fragments of sequence diagrams to be reused in other diagrams. Figure 17.18 shows what happens when an instance of `ListClients` receives a `listAllClients` message. At this stage of the design we have not addressed how

```

import java.awt.*;
public class CheckCampaignBudgetUI extends Frame
    implements ClientLister {
    private Choice clientChoice;
    ...
    public void addClientName(String name) {
        clientChoice.addItem(name);
    }
    ...
}

```

Figure 17.17 Possible Java definition of `addClientName(name)`.

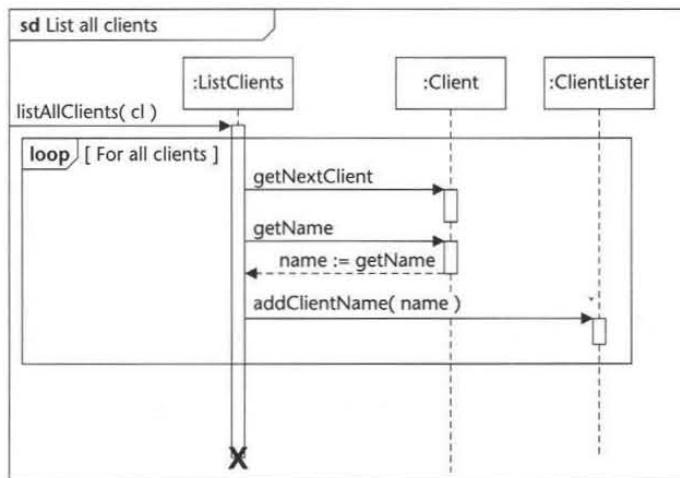


Figure 17.18 `listAllClients()` operation.

clients are to be stored in a database, so this is left vague. Design of the data storage is covered in Chapter 18.

Rosenberg and Scott (1999) use control classes as placeholders for operations that may later be assigned to other classes. This technique could be used in this example, where the `ListClients` control class may become part of some other class that handles the database access for the `Client` class. This approach can help in preventing an object-oriented system from ending up with a lot of classes like `ListClients` that are little more than wrappers for one or two operations, and more like programs than objects. On the other hand, this can break the architectural layering of the system, as the classes in the data storage layer will need to know about classes in the presentation layer.

Returning to the sequence diagram for this collaboration, the next event that will take place is that the user will select a client from the dropdown that has been populated with clients by the processes that we have just described. When a particular client has been selected, then the list of campaigns in the boundary class must be populated with only those campaigns in the database that belong to that client. This is shown in Fig. 17.19.

Part of this sequence diagram is similar to the one in Fig. 17.15. We would apply the same design principles to this, with a `CampaignLister` interface, which must also be implemented by `CheckCampaignBudgetUI`. The interaction is slightly different, as we need to pass the client to `ListCampaigns` so that it knows which client's campaigns to display. We can model the interaction within the boundary class in more detail if we wish.

When the instance of `CheckCampaignBudgetUI` is created it will add the instance of `Choice`, `clientChoice`, to itself and register an interest in events from the user that affect `clientChoice`. (A `Choice` is a dropdown list.) In Java, for example, this means that `CheckCampaignBudgetUI` must implement the `ItemListener` interface. When an event takes place, such as the user selecting a

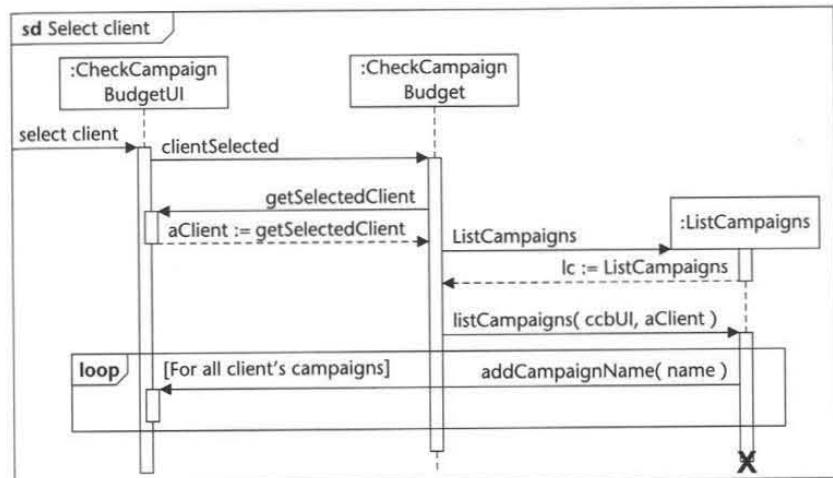


Figure 17.19 Second part of interaction for use case `Check campaign budget`.

client in the dropdown, `:CheckCampaignBudgetUI` will be sent a message `itemStateChanged(evt)`, with the data associated with the event passed in the parameter `evt`. If the source of the event is `clientChoice`, then it should notify the control class by sending it the `clientSelected` message. Figure 17.20 shows this interaction.

At this point in the interaction, the user could either select a campaign from the list of campaigns or could select another client from the client list. In the latter case, then the interaction of Fig. 17.19 could take place again. With the design as it is at present, this would result in the campaigns for the newly selected client being added onto the list of campaigns already in the dropdown. This is clearly incorrect, and this is where the interface needs an operation to clear the list: `clearAllClientNames()` was suggested for the `ClientLister` interface. The equivalent `clearAllCampaignNames()` is shown in Fig. 17.21.

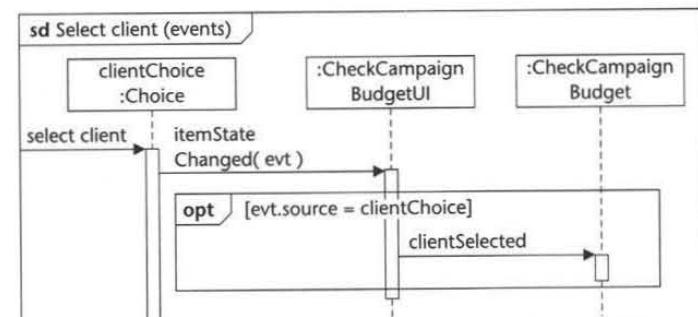


Figure 17.20 Detail of AWT component interaction for use case `Check campaign budget`.

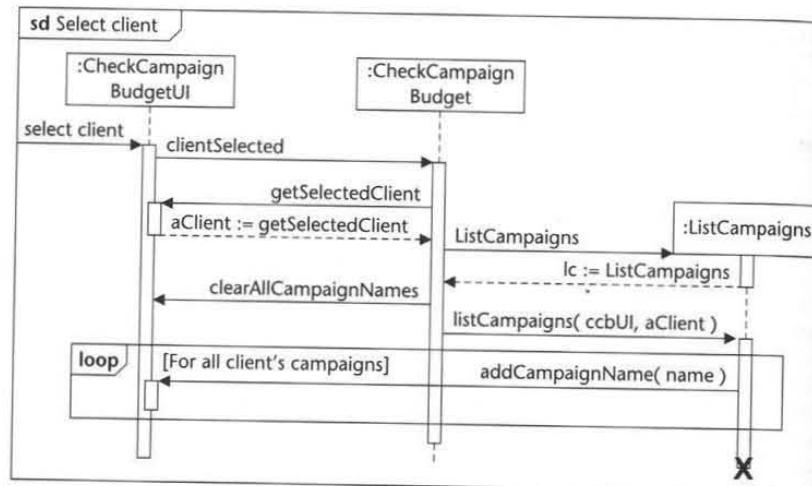


Figure 17.21 Revised second part of interaction for use case Check campaign budget.

If the user does select a campaign, then the button, checkButton, should be enabled, and if the user clicks that, then the budget for the selected campaign should be calculated and displayed in the textfield. This is shown in Fig. 17.22. We have not shown all the parameters and return values in this diagram, as it is very easy to clutter up such a diagram with additional text.

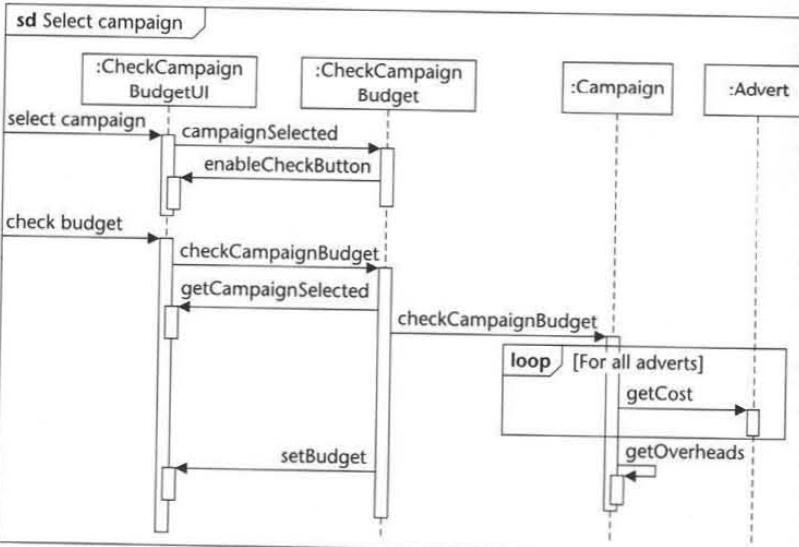


Figure 17.22 Final part of interaction for use case Check campaign budget.

In the same way as we did not show the detail of the interaction within the boundary class on the earlier diagrams, we have left the detail of how :CheckCampaignBudgetUI will get the values from the dropdowns and set the value of the textfield out of this diagram. This provides us with a clean interface, and :CheckCampaignBudget need know nothing of the internal workings of the boundary class. The display of the budget amount could be changed from atextfield to a label, or even digital speech, but the control class only needs to know that the boundary class will respond appropriately to the setBudget message.

There may be many places in the system where the same patterns of interaction as in Figs 17.18 and 17.20 take place. Rather than producing separate interaction sequence diagrams for all of these, we may choose to produce some generic sequence diagrams that show the pattern of interaction that is expected to take place when objects are listed in a dropdown or when an item is selected from a list.

We can further extend the class diagram with the operations that have been identified for our boundary class. Figure 17.23 shows this for the CheckCampaignBudgetUI class. The code that manages the interface is the implementation of an operation of CheckCampaignBudgetUI that is invoked by the control class. It also shows the interfaces that it must realize, both some that are specific to this application and one that is part of the Java AWT event handling model.

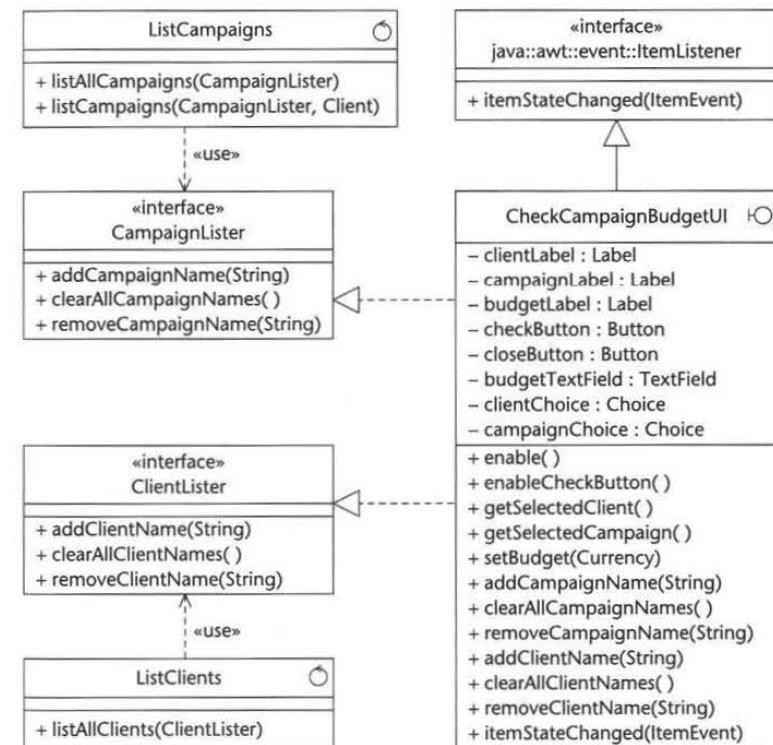


Figure 17.23 Revised class diagram showing CheckCampaignBudgetUI.

Boundary classes include reports as well as screen displays, and reports can be shown in interaction diagrams as well. The simplest form of report is produced by opening an output stream to a device such as a printer. In this case each object that is to be printed out can be sent the message to print itself with a reference to the output stream as a parameter. This is shown in Fig. 17.24 for a simple report of all clients, where a control class co-ordinates the printing of the report. (We have used communication diagrams for the following diagrams to show how they can be used for this kind of design as well.) If the user is required to enter parameters, for example selecting a client in order to print a report of all campaigns for that client, then a dialogue of some sort will also be required.

In the simple solution of Fig. 17.24, the instances of `Client` are responsible for formatting themselves for output to the printed report. An alternative solution is to design a boundary class to handle the formatting of the attributes of one or more instances of one or more classes. This solution is shown in Fig. 17.25.

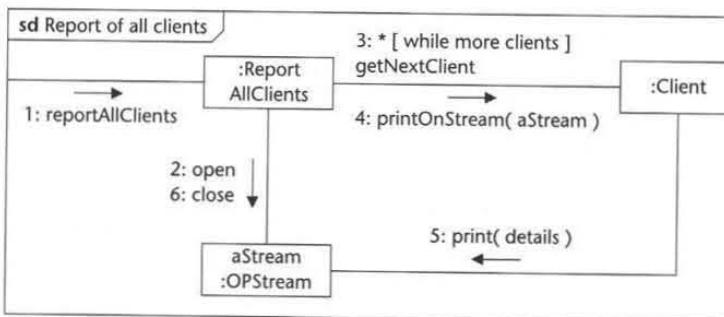


Figure 17.24 Report design in which the `Client` formats its own data.

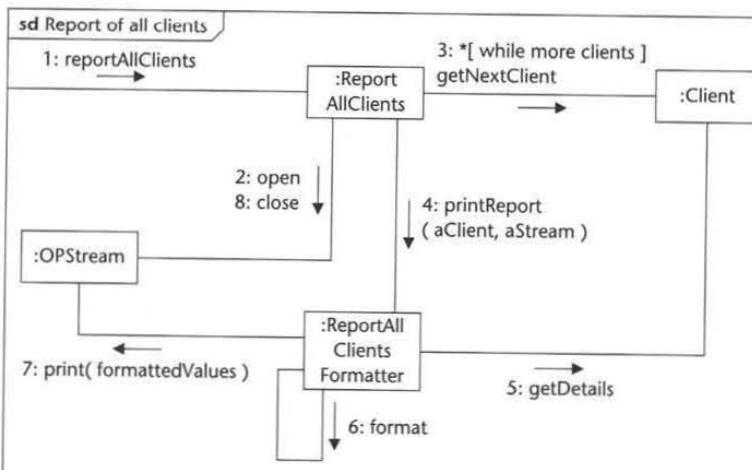


Figure 17.25 Report design in which a boundary class formats the data.

The decision as to which of these approaches is best will depend on the development environment being used.

Control classes to manage the user interface like this are not always necessary. In Java applets, the `Applet` class can handle both the presentation and the control. In some applications, for example real-time applications where the attributes of objects are changing and those changes need to be reflected in different views of the data, it may be better to use a design based on the Smalltalk Model–View–Controller style of developing applications, which is discussed in Section 17.7.

17.6 The Class Diagram Revisited

The boundary classes can be added to the class diagram. They can be shown in a single diagram for the Agate Boundary package (see Fig. 17.12) or in separate diagrams, grouped by type or subsystem. The buttons and other classes that are used to make these interface classes need not be shown in the diagrams. However, all the boundary classes in the application will have a package dependency to the package where the buttons and other classes are held.

There may be some commonality among boundary classes that can be abstracted out into an inheritance hierarchy. For example, all dialogue boxes that are concerned with printed reports may have a standard set of buttons, Print, Cancel and Select Printer, and radio buttons for Portrait and Landscape. These buttons and the associated event-handling mechanisms could be placed in a generic `PrintDialog` superclass, from which all other report dialogue boxes could inherit this functionality. If we had chosen the design for the lists of clients and campaigns that required a separate dialogue box for each, as in Figs 17.4, 17.5 and 17.9, then we would need to add a `LookupDialog` class which would be the parent of all look-up dialogues in the system. This is in turn a subclass of the `Dialog` class from the Java AWT package. The `Dialog` class is also the superclass of `PrintDialog`, the parent of all dialogues used to run reports. Figure 17.26 shows the beginnings of a possible inheritance hierarchy based on this approach. Note that `LookupDialog` and `PrintDialog` are both abstract classes: there will never be instances of either.

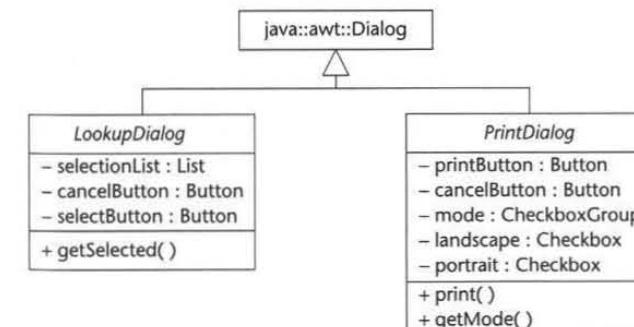


Figure 17.26 Beginnings of an inheritance hierarchy for possible `Dialog` classes.

17.7 User Interface Design Patterns

We have suggested that in an application of this sort generic patterns of interaction can be documented for the design of the user interface and the boundary classes. It is also possible to use standard design patterns as we discussed in Chapter 15.

The Model–View–Controller (MVC) architecture is the classic object-oriented pattern for designing the user interface. It has been mentioned more than once in this chapter, although we have not used it here, and was described in Chapter 12. The Model class is the business object, the View is its presentation to the user and the Controller defines how the user interface responds to the user's actions.

Java uses MVC in the user interface by means of the `EventListener` subinterfaces and `EventObject` subclasses. There are a number of subclasses of `EventObject` for different types of event, such as `MouseEvent` for mouse events and `ItemEvent` for checkboxes and lists, and different subinterfaces of `EventListener` to handle them (such as `MouseListener`, `ItemListener` and `ActionListener`). We used this mechanism to handle the event when the user selects a client in Fig. 17.20. Because it involves a dropdown list (a `Choice`), we use `ItemEvent` and `ItemListener`.

Any class that implements the `ItemListener` interface must implement the method `itemStateChanged()`; this means that it must include code so that its instances can respond appropriately to the message `itemStateChanged(ItemEvent)` whenever they receive it. Any class that implements `ItemListener` can register its interest in the events that affect an instance of certain user interface components, such as the `Choice` and the `Menu` classes. This is done by sending the message: `AddItemListener(ItemListener)` to an instance of one of these classes. It adds the new instance of `ItemListener` to a list it keeps of all the `ItemListeners` it must notify of any events. Then when the `Menu` is selected or the `Choice` is changed, it sends the message `itemStateChanged()` to each of the objects in its list and passes them an `ItemEvent` object as a parameter.

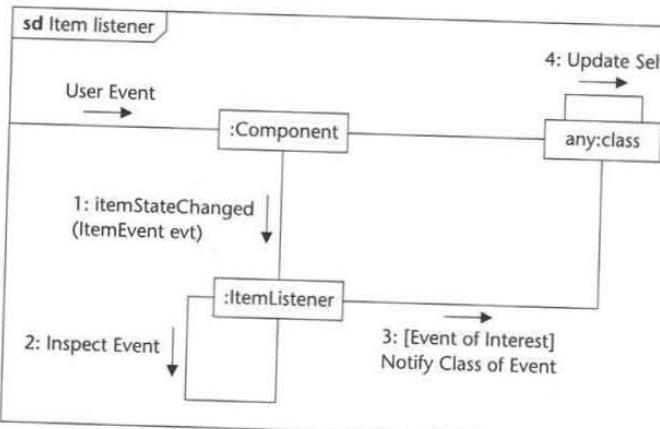


Figure 17.27 Java `ItemListener` response to an external event.

Each of the objects that receives the message can then inspect the `ItemEvent` object and decide whether it is interested in that particular event and whether it needs to take some action in response. The action could be to notify other classes of the event, particularly if the class that is implementing the `ItemListener` interface is a control class. This is shown in Fig. 17.27. The use of the interface mechanism in Java means that the user interface component does not need to know the actual class of the class that implements `ItemListener`; as long as it implements the interface it will know what to do when it is sent the `itemStateChanged()` message.

Gamma et al. (1995) describe the MVC architecture in terms of three patterns: the Observer, the Composite and the Strategy design patterns.

- The Observer pattern provides a mechanism for decoupling business objects from their views. It is based on a publish–subscribe model. There are two main types of objects: Observers, which are the View objects, and Subjects, which are the business objects. Observers can subscribe to a subject and ask to be notified of any changes to the subject. When a change takes place in the subject, it publishes information about the changes to all the observers that have subscribed to it. This is the basis of the Observer–Observable approach adopted in Java 1.1 (and later versions of Java). The Observer pattern is the core of the MVC architecture, but two other patterns also apply.
- The Composite pattern provides a means to structure objects into whole–part hierarchies. Most windowing interfaces use this approach to form composite views made up of smaller components. The class diagram of Fig. 17.10 shows this kind of composite structure. The Composite pattern is a way of structuring views to include nested views. For example, a graphical display may represent the same model in two different views, a graph and a table of numbers that may be nested within another view.
- The Strategy pattern offers a way of defining a family of algorithms, encapsulating each one and making them interchangeable. This allows the strategy to vary independently of client objects that use it. In MVC one controller can be replaced by another. For example, an on-screen calculator could be designed and built initially to respond to mouse clicks on the buttons on the screen. The controller for the calculator detects mouse clicks and notifies the internal application objects and the view objects about these mouse events. Later the calculator could be modified to respond instead to keypresses on the computer numeric keypad, or to both mouse clicks and keypresses. The controller object can be replaced without having any impact on the application objects.

Design patterns provide a way of reusing design experience and best practice. Separating the concerns of the user interface and the control of interaction from the business application objects is a proven technique to produce good designs and maximize the opportunities for reuse of the classes that are designed and developed.

17.8 Modelling the Interface Using State Machines

In Fig. 17.10 we showed the classes that make up the user interface for the use case `Check campaign budget` in the Agate project. That diagram shows the static

structure of the interface, but it provides no information about how it will behave in response to the user. The sequence diagrams developed in Section 17.6 and the prototype in Fig. 17.6 provide additional information, but they tell us nothing about the permitted states of the interface. The sequence diagrams show the sequential view of the user working through the fields on the screen from top to bottom, but it is in the nature of GUI interfaces that the user can click on interface objects out of sequence. So what happens if the user clicks on the check button before a client and a campaign have been selected? The user may choose to check more than one budget. What happens if they select a different client—how does that affect the other fields where data has already been selected? All these issues can be modelled using a state machine diagram. State machines were introduced in Chapter 11, and were used there to model the way that events affect instances of a class over its lifetime. They are used to model shorter timescales in designing the control of real-time systems. They can also be used to model the short-term effects of events in the user interface. Browne (1994) uses state machines in this way to model the user interface as part of the STUDIO methodology. Horrocks (1999) uses state machines in a more rigorous way than Browne in his user interface-control-model (UCM) architecture and relates the use of state machines to coding and testing of the user interface. Browne's approach leads to a bottom-up design of the interface, assembling state machines for components into a complete model of an interface; Horrocks develops his state machines in a top-down way, successively introducing nested substates where they are necessary. We are using Horrocks's approach in what follows.

For the example that follows, we are using the original design for the user interface with dropdowns for Client and Campaign, as in the prototype of Fig. 17.6.

As a design principle in our user interfaces, we want to prevent users from making errors wherever possible rather than having to carry out a lot of validation of data entry in order to pick up errors that have been made. One way of doing this is to constrain what users can do when they are interacting with the interface. For example, in the Check campaign budget user interface it makes no sense to click the check button until both a client and a campaign have been selected. Rather than check whether a client and campaign have been selected every time the button is clicked, we can choose only to enable the button when we know that both have been selected. To do this we need to model the state of the user interface and it is this that we model using state machines. This process involves five tasks.

- Describe the high-level requirements and main user tasks.
- Describe the user interface behaviour.
- Define user interface rules.
- Draw the state machine (and successively refine it).
- Prepare an event-action table.

We have simplified Horrocks's approach here. His book (Horrocks, 1999) provides a full and clear exposition of this approach.

Describe the high-level requirements and main user tasks

The requirement here is that the users must be able to check whether the budget for an advertising campaign has been exceeded or not. This is calculated by

summing the cost of all the adverts in a campaign, adding a percentage for overheads and subtracting the result from the planned budget. A negative value indicates that the budget has been overspent. This information is used by a campaign manager.

Describe the user interface behaviour

There are five elements of the user interface: the client dropdown, the campaign dropdown, the budgettextfield, the check button and the close button. These are shown in Fig. 17.6.

The **client dropdown** displays a list of clients. When a client is selected, their campaigns will be displayed in the campaign dropdown.

The **campaign dropdown** displays a list of campaigns belonging to the client selected in the client dropdown. When a campaign is selected the check button is enabled.

The **budgettextfield** displays the result of the calculation to check the budget.

The **check button** causes the calculation of the budget balance to take place.

The **close button** closes the window and exits the use case.

Define user interface rules

User interface objects with constant behaviour

- The client dropdown has constant behaviour. Whenever a client is selected, a list of campaigns is loaded into the campaign dropdown.
- The budgettextfield is initially empty. It is cleared whenever a new client is selected or a new campaign is selected. It is not editable.
- The close button may be pressed at any time to close the window.

User interface objects with varying behaviour

- The campaign dropdown is initially disabled. No campaign can be selected until a client has been selected. Once it has been loaded with a list of campaigns it is enabled.
- The check button is initially disabled. It is enabled when a campaign is selected. It is disabled whenever a new client is selected.

Entry and exit events

- The window is entered from the main window when the Check Campaign Budget menu item is selected.
- When the close button is clicked, an alert dialogue is displayed. This asks 'Close window? Are you sure?' and displays two buttons labelled 'OK' and 'Cancel'. If 'OK' is clicked the window is exited; if 'Cancel' is clicked then it carries on in the state it was in before the close button was clicked.

Draw the state machine

At the top level, there are three states the application can be in. It can be in the Main Window (and we are assuming that this is modelled in detail elsewhere), in the Check Budget Window or in the Alert Dialogue. Figure 17.28 shows these top-level states.

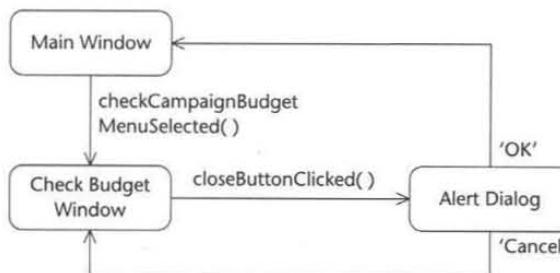


Figure 17.28 Top-level states.

Horrocks uses the convention of names of buttons in single quotes to represent button press events. We have used that notation here for the values that will be returned from the alert dialogue, but have used operation signatures for the other events, as we want to be able to check them against the sequence diagrams.

Within the Check Budget Window state, there are different substates of the user interface that must be modelled. Initially, no client is selected; then the user can select a client. Figure 17.29 shows the resulting two states. Because there will be actions associated with the user selecting a different client, we have shown the clientSelected() event returning to the Client Selected state if it occurs again.

Similarly, once the interface is in the Client Selected state, it can be in either the substate where a campaign has not yet been selected or the substate where a campaign has been selected. These two states are shown in Fig. 17.30.

If the user interface is in the Campaign Selected state, then if the check button is pressed, the result will be displayed in thetextfield, which will initially be blank. This is shown in Fig. 17.31.

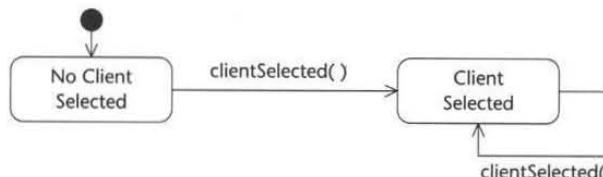


Figure 17.29 Client selection states within the state Check Budget Window.

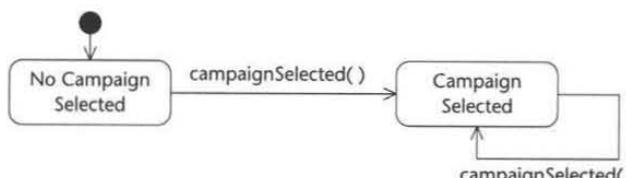


Figure 17.30 Campaign selection states within the state Client Selected.

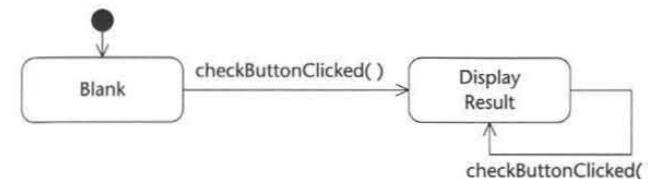


Figure 17.31 Display of result states within the state Campaign Selected.

These various state machines can be combined and nested within the top-level state for the Check Budget Window. This is shown in Fig. 17.32.

Note the use of the *deep history indicator* where the 'Cancel' event returns control from the Alert Dialogue to the Check Budget Window. The H* in a circle shows that when that transition takes place, it will return to the exact same state that it was in before the transition to the Alert Dialogue state, however far down in the nested hierarchy of states it was. This works like a memory. The state

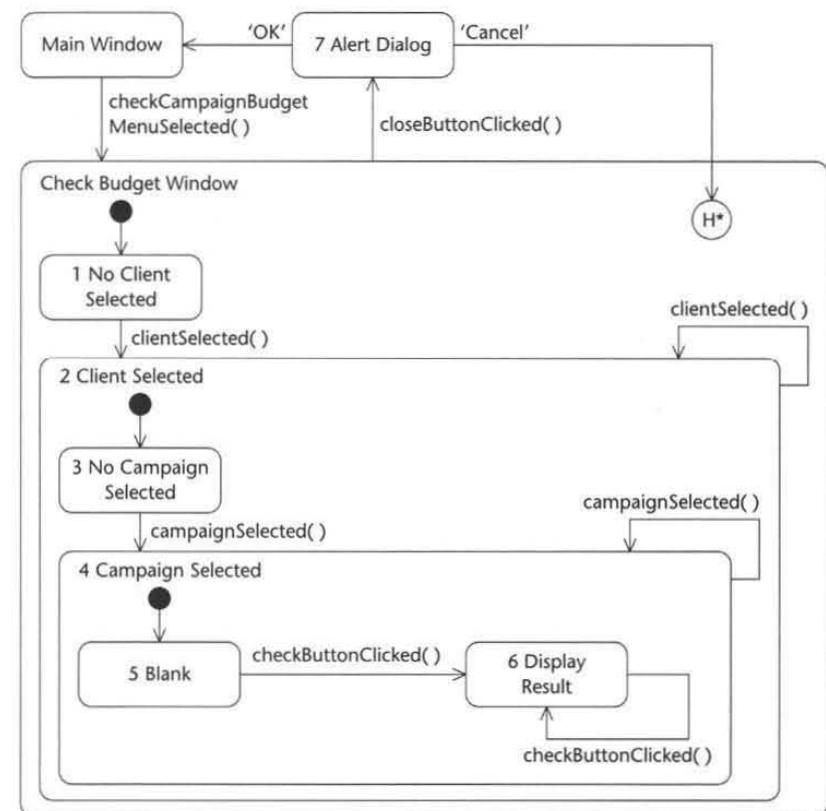


Figure 17.32 Combined state machine with nested states.

of the user interface before the `closeButtonClicked()` event is recorded, and the 'Cancel' event returns it back to that recorded state.

Horrocks's approach has some notational differences from the UML standard. He numbers his states (as in Fig. 17.32), because the numbers of the states can be stored as the values of the state variables that hold the information about the current state of the system. He also explicitly names the state variables in square

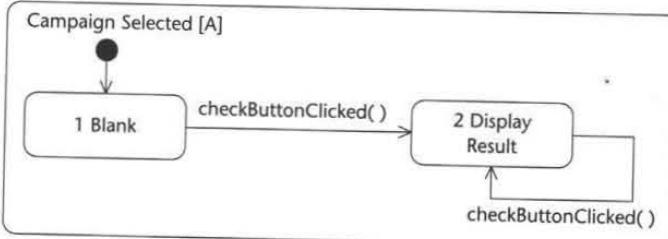


Figure 17.33 State variable and numbered states.

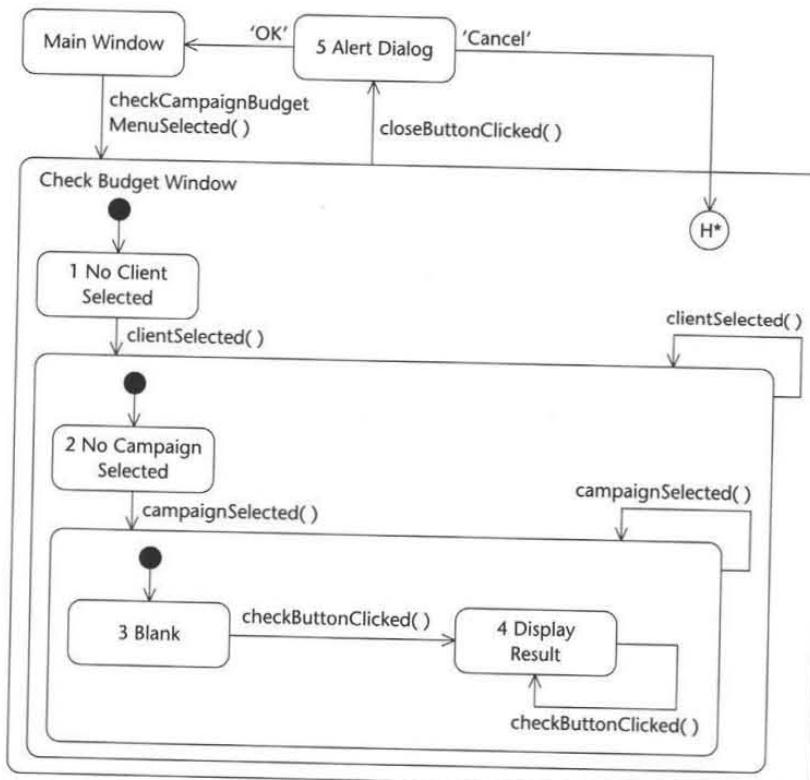


Figure 17.34 Simplified version of state machine in Fig. 17.32.

brackets. Figure 17.33 shows an example of this notation. Using explicit state variables and numbers for states will help in coding the implementation of this design, and the numbered states make the production of the event-action table simpler. Figure 17.32 is slightly more complicated than it needs to be. States 2 and 4 have no real meaning; they can be treated as no more than a grouping of the enclosed substates to keep the number of states and transitions down. Figure 17.34 is a simplification. The simplified state machine has been used to prepare the event-action table in Fig. 17.35.

Prepare an event-action table

UML state machine notation allows you to label transitions and states with actions. On a transition the action can be an action of the object itself or it can involve a message being sent to another object. Within states, *entry* and *exit actions* can be documented, as well as *do actions* that are carried out continuously while the object is in that state and *event actions* that are carried out if a particular event occurs while the object is in that state (as explained in Section 11.3).

The use of these actions on state machine diagrams can make them very cluttered and difficult to read, especially if there are also guard conditions on the transitions as well as actions. UML allows you to put actions both on transitions and on states, although some authors on the subject suggest that you use either actions on transitions or actions on states, but not both.

For complex state machines, rather than displaying the actions in the state machine as in Chapter 11, an alternative is to list the actions in a table. This is an event-action table. From the point of view of the programmer, this table will be easier to use than a state machine labelled with actions. It should also make it

Current State	Event	Action	Next State
-	Check Campaign Budget menu item selected	Display CheckCampaignBudgetUI. Load client dropdown. Disable campaign dropdown. Disable check button. Enable window	1
1	Client selected	Clear campaign dropdown. Load campaign dropdown. Enable campaign dropdown	2
2, 3, 4	Client selected	Clear campaign dropdown. Load campaign dropdown. Clear budget textfield. Disable check button	2
2	Campaign selected	Clear budget textfield. Enable check button	3
3	Check button clicked	Calculate budget. Display result	4
3, 4	Campaign selected	Clear budget textfield	3
4	Check button clicked	Calculate budget. Display result	4
1, 2, 3, 4	Close button clicked	Display alert dialogue	5
5	OK button clicked	Close alert dialogue. Close window	-
5	Cancel button clicked	Close alert dialogue	H*

Figure 17.35 Event-action table for Fig. 17.34.

easier to validate the state machine and to test the code once it has been implemented.

The event-action table lists the following values in columns:

- the current state of the object being modelled
- the event that can take place
- the actions associated with the combination of state and event
- the next state of the object after the event has taken place; if more than one state variable is used, these are shown in separate columns.

Figure 17.35 shows an event-action table for the state machine of Fig. 17.34.

We can now use this information to revisit the sequence diagrams. Indeed, if we know what the names of the messages or operations in the sequence diagrams are, we can use them in the event-action table instead of the natural language descriptions of the actions.

If we examine the first sequence diagram from Fig. 17.15, we can see that we need some additional operations to be shown in the sequence diagram. The sequence diagram shows the boundary class being created, the client dropdown being loaded and the window being enabled, but we have not explicitly disabled the campaign dropdown and the check button. Figure 17.36 shows these additional operations.

We can apply the same approach to the sequence diagram of Fig. 17.18, which shows what happens when the client is selected, and thus the transition from State 1 to State 2. This is shown in Fig. 17.37.

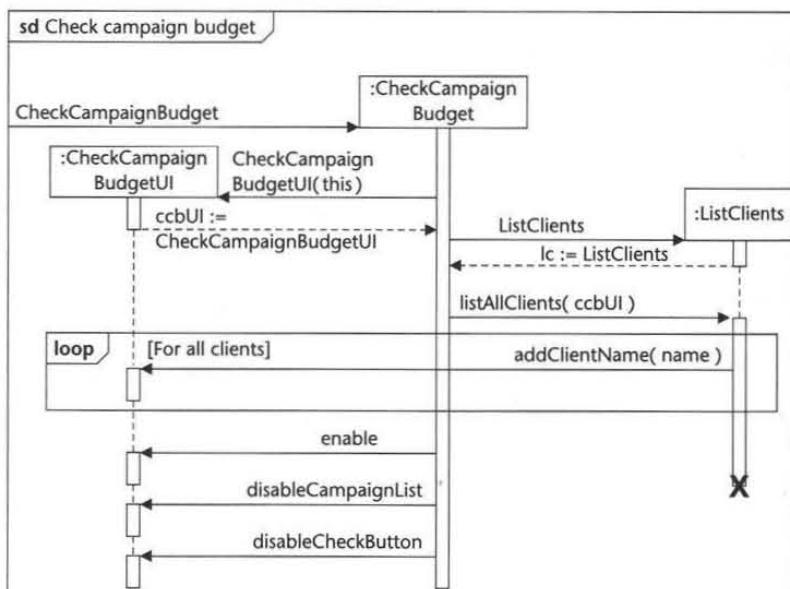


Figure 17.36 Revised sequence diagram for first part of interaction.

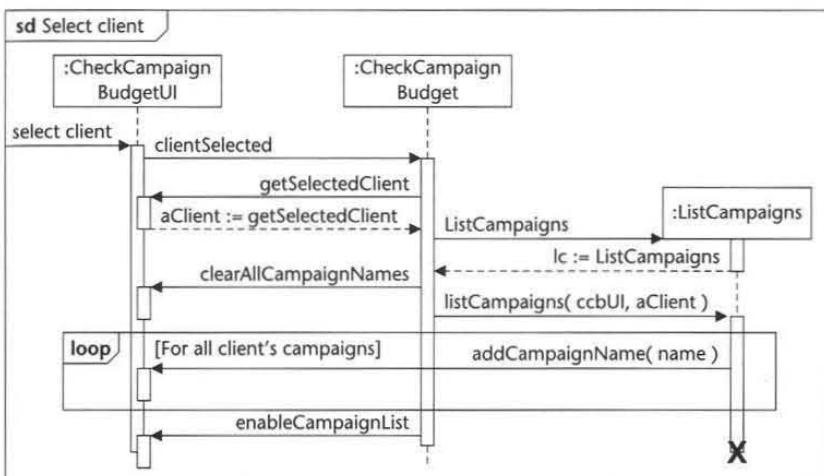


Figure 17.37 Revised sequence diagram for second part of interaction.

If the names of the events and operations have already been decided, because the sequence diagrams have been produced and the operations of classes have been designed, then the event-action table can list them using these names. For example, in the transition from State 1 to State 2 in Fig. 17.35, the event would be called `clientSelected()`, and the actions would be:

```

ListCampaigns();
CheckCampaignBudgetUI.clearAllCampaignNames();
ListCampaigns.listCampaigns();
CheckCampaignBudgetUI.enableCampaignList();
    
```

Working through the design of the interface in this way will lead to the addition of more operations to the class diagram in Fig. 17.22. For example, the operation `disableCheckButton()` needs to be added to `CheckCampaignBudgetUI`, and the operations `enableCampaignList()` and `disableCampaignList()` need to be added to the interface `CampaignLister` and to `CheckCampaignBudgetUI`.

17.9 Summary

Designing the interface objects requires us to model three important aspects of the interface. First, we need to determine the classes of the objects that will participate in the interaction with the user and decide on how we will reuse interface classes that are available in class libraries. The choice of interface objects will depend on the style guidelines that have been adopted for the system. These classes are shown in class and package diagrams. Second, we need to model the interaction with the user in sequence or collaboration diagrams. The way that the interaction is modelled will depend on the architecture that has been chosen for the system. The

three-tier architecture and the Model–View–Controller architecture separate the boundary objects from the entity and control objects using well-defined methods. Third, we need to model the state of the interface to ensure that we understand how the interface will respond to events and what sequences of events are permitted. We use state machine diagrams to do this.

While carrying out these three modelling tasks, we can draw on design patterns to inform the way in which we select the interface classes and design the interaction between them and the business classes. Prototyping can be used to build models of the interface and test how it will work. Users should be involved in this process to ensure that the interface meets their requirements and to validate the analysts' understanding of their requirements for how tasks should be carried out.

Review Questions

- 17.1** Why should the user interface classes be kept separate from the business classes and application logic?
- 17.2** Explain the difference between vertical and horizontal prototyping.
- 17.3** What is meant by a throwaway prototype?
- 17.4** What does the «import» stereotype mean?
- 17.5** What role does each element of the Model–View–Controller architecture play?
- 17.6** What else do we use state machine diagrams for, apart from modelling the state of interface objects?
- 17.7** What are the five steps in preparing a state machine to model a user interface?
- 17.8** What information is held in an Event–Action table?
- 17.9** Convert the communication diagram of Fig. 17.26 into a sequence diagram.
- 17.10** Convert the communication diagram of Fig. 17.27 into a sequence diagram.
- 17.11** What are the differences between the MVC and Java EventListener approaches?
- 17.12** Convert the sequence diagram of Fig. 17.36 into a communication diagram.
- 17.13** Convert the sequence diagram of Fig. 17.37 into a communication diagram.

Case Study Work, Exercises and Projects

- 17.A** Decide how you will handle the interaction between the user and the system for the use case Record problem on line for FoodCo. Draw a prototype user interface design.
- 17.B** Draw a sequence diagram to include the interface objects that are needed for your prototype in Exercise 17.A.
- 17.C** Draw a class diagram to show the classes that are used in the prototype from Exercise 17.A.
- 17.D** Extend your class diagram from Exercise 17.C to show the superclasses of the interface classes.

17.E If you are familiar with a class library such as the Java AWT, Java Swing, Microsoft Windows Forms or Borland Object Windows Library, then try to determine how your interface classes relate to classes in that class library.

17.F Produce a prototype for the use case Record problem on line using a language or visual programming environment with which you are familiar.

17.G Draw a state machine diagram for the interface to the use case Record problem on line to model the behaviour of your prototype developed in Exercise 17.F.

17.H Update your sequence diagram from Exercise 17.B to make sure that it reflects the state machine diagram of Exercise 17.G.

Further Reading

The MVC architecture is explained in a number of books, particularly books that use Smalltalk as a programming language. Gamma et al. (1995) specifically explain it in terms of patterns.

The Java EventListener event model was introduced into Java in Version 1.1. (Version 1.0 used a less efficient model.) There are a number of these EventListener interfaces for different kinds of events. Most introductory Java programming books explain them. The Swing classes can also use the MVC approach.

Few books on object-oriented analysis and design provide much detail on state machines, and those that do often provide simple models of telephone systems as examples rather than user interfaces. Browne (1994) is one of the few authors who seriously applies state machine diagrams to user interface design. Although he does not use the UML notation, his diagrams in Chapter 3 are similar enough to provide a clear idea of how they can be used to model the detail of user interface interaction. Horrocks (1999) applies a more rigorous software engineering approach to the use of state machines to design interfaces. Browne's approach is bottom-up, while Horrocks's is top-down. Both use the state machine notation that was originally developed by Harel (1987). For a more recent view of state machines from Harel, see Harel and Politi (1998), which presents the STATEMATE approach.

Data Management Design

Learning Objectives

In this chapter you will learn

- the different ways of storing persistent objects
- the differences between object and relational databases
- how to design data management objects
- how to extend sequence diagrams to include data management objects
- how frameworks are used to map classes to database tables.

18.1 Introduction

Real information systems require *persistent* data: data that continues to exist even when the system is not active. In this chapter we discuss how persistent data can be modelled using UML. In Section 18.2, we explain what is meant by persistent data. Persistent data can be stored in files or in databases, and files are described in Section 18.3. Using a database management system (DBMS) offers a number of advantages over files. The use of a relational DBMS or an object DBMS will affect the design of the data management. Databases are introduced in Section 18.4.

Many object-oriented systems are constrained by organizations' existing investments in hardware and software and have to use relational database management systems (RDBMS) to store data. *Normalization* can be used to design tables for a relational database. Alternatively there are rules of thumb that can be applied to convert a class diagram to a suitable set of tables. Both approaches are explained in Section 18.5.

Designing for an object DBMS will have a different impact on the design model, and this is discussed in Section 18.6. Depending on the object DBMS, making objects in the class diagram persistent can be straightforward. For many systems, there is a requirement to distribute data in different databases. Various mechanisms, such as CORBA (Common Object Request Broker Architecture), RMI

(Remote Method Invocation) or EJB (Enterprise Java Beans) can be used to separate the persistent objects from the business logic in the application layer and interface objects. CORBA and EJB can provide the infrastructure for creating distributed databases. Section 18.7 briefly explains CORBA as an example of how this is done.

Database objects can be reflected in the UML models that are produced in design (Section 18.8). ODBC (Open Database Connectivity) and the specific Java implementation of this—JDBC (Java Database Connectivity)—can be used to link an object-oriented system to a relational database. Design patterns may be used to show how persistence can be designed into a system using the experience of many previous developers.

18.2 Persistence

18.2.1 The requirement for persistence

For some applications the data that is created or used while the application is running is not required after the application terminates. This applies mainly to simple applications: an example would be the on-screen calculators provided with GUI operating systems. Such data is called *transient data*.

Most applications, however, need to store data between one execution of the program and the next. In some cases, the data that is stored is secondary to the operation of the application. When you use a browser, one of the first things that happens as it loads is that it reads data from files that describe the user's preferences and record the last websites visited. The ability to store user settings and to configure applications in this way is an important factor in their usability, but it is not their primary purpose: a browser still works if the user's preferences and history are not available.

In the case of information systems, storing data is a primary requirement. Businesses and other organizations rely on their information systems to record data about other organizations, people, physical objects and business events and transactions. The data entered into such a system today will be required in the future; operations being carried out on data today rely on data that was stored in the past. Computerized information systems have replaced systems based on paper in files and ledgers, and must provide the same relatively permanent storage that is provided by paper-based systems. In most organizations, it is also important that data can be shared between different users. Data in the memory of a particular computer is not normally accessible to multiple users. It must be written away to some kind of shared data storage system so that other users can retrieve it when they require access to it.

This is what we mean by *persistent data*. It is data that must be stored in a secondary data storage system, not just in computer memory, that must be stored after the program that creates or amends it stops running and that usually must be available to other users. Information systems also use transient data: for example, the results of calculations or lists of objects that are required for a particular purpose such as printing a report, but that are not required permanently and can be destroyed after they have been used.

In an object-oriented system, we are concerned with both *persistent objects* and *transient objects*. Persistent objects are those that must be stored using some kind of storage mechanism, while transient objects will be erased from memory after they have been used.

18.2.2 Overview of storage mechanisms

Ultimately all data in computer systems is stored in files of some sort. In Section 18.3, we explain the different kinds of file organizations and access methods that are available, and the purposes that files are used for. However, most information systems use a database management system of some sort in which to store their data. Database management systems provide a layer of abstraction that hides from the user the fact that the data is stored in files. If the database is a relational database, then the user of the database sees tables containing data. Each table may relate to part of a file, to a single file or to many files, but that is not important to the user of a relational database (who may be a designer or programmer). The way that the database stores tables in files is important to the database administrator, who has to be concerned with where the data is stored, taking backups and so on. If the database is an object database, then the user sees objects and links between them. Again these objects will be stored in files of some sort, but the designer or programmer does not need to know the details.

In an object-oriented system, a database of some sort is the most likely way of providing persistent storage for objects. However, it is possible to store objects in files. Most object-oriented languages provide mechanisms for converting objects into a form that can be written out to a file—*serializing* them—and for reading them back into memory from a file. This is unlikely to provide an efficient mechanism for a business information system. However, files can be used for many other storage purposes in object-oriented systems. They can be used to hold data that is transferred in from other systems, and in Section 19.6 we discuss the conversion of data from a system that is being replaced. Files can also hold configuration information, and in Section 18.3.5 we provide an example of how files can be used to localize the Agate system so that text items such as labels, button captions and menu entries are displayed in the language of the country where the application is being used.

The choice of database management system will have a significant impact on the work that is required of the systems designers. If an object database is used, then there should be little work involved in designing the way that objects are stored. If a relational database is used, then more work is involved. In Section 18.5 we describe two approaches to converting classes to tables and, in Section 18.8, we describe some ways of designing persistence frameworks. In Section 18.8.5 we explain the tools and frameworks that are available to automate the process of mapping classes to tables.

18.2.3 Architecture for persistence

Part of the process of system architecture design is to determine how the requirements for the storage of persistent data will be met by the system. There may be trade-offs to be made between the requirements for a new system and the existing hardware and software that is available. Many organizations also have corporate

standards for the database management systems that they use, and these will influence the architecture of a new system and the design of data storage.

Existing systems may have a different architecture from the one proposed for a new system, but there may be parts of the existing system that can be reused. This is often the case with databases, as organizations often have existing business systems and wish to use the data from those existing systems in new ways. If the new system is to be developed in an object-oriented language, it may be necessary to create a layer in the architecture that wraps the existing data so that it looks like objects even if it is stored in a relational database. This is shown in Fig. 18.1. An example of an existing system like this might use SQL-Server as its database and Visual Basic for the existing client programs. The Visual Basic client programs access the data in the database using ODBC (Open Database Connectivity) and SQL (Structured Query Language). The new system might use Java applications for the presentation layer, and Java components for the business logic layer and for the data access layer. The Java components would provide a wrapper around the rows of data in the relational database, so that they appeared as objects to the business logic layer. The data access layer would probably still use ODBC—or in this case JDBC (Java Database Connectivity)—and SQL to connect to the database, but the objects used to access the data could be reused in other applications. If the data access layer is well designed, it may be possible later to replace the old client-server application by adding to the presentation and business logic layers and reusing the data access layer.

For the architect designing the persistent storage of a system there are a number of questions to be answered.

- Are there parts of the system where storage in files is appropriate?
- Is the system truly an object-oriented system or a simple database system with a GUI user interface? For simple systems, it is possible to write programs in an object-oriented language such as C++ or Java, which provide a front-end to a database. The front-end could connect to the database using ODBC or JDBC and treat the data purely as rows of data in tables, without using any entity objects. We are not taking this approach here, as it is not appropriate for our case studies.
- Will the system use an existing DBMS or is there freedom to choose an appropriate DBMS? If an existing system is to be used, it will constrain the system

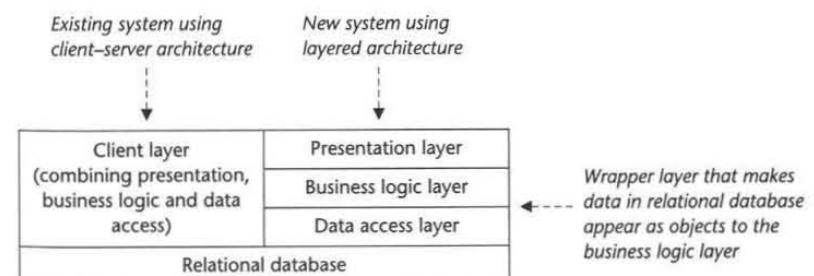


Figure 18.1 Layered architecture for existing and new systems sharing an existing relational database.

design in some ways, but, as we have shown above, it is possible to use a more flexible layered architecture to replace a client–server system while retaining the same DBMS.

- Will the system use a relational DBMS? If a relational DBMS is to be used then classes must be mapped to tables. This can be done using tools that automate this process, or by designing the tables and a suitable mechanism to fetch data from the database and assemble it into objects, and to save the objects back into the database when required.
- Will the system use an object DBMS? If an object DBMS is to be used, then work on designing the persistence mechanisms is likely to be much simpler.
- What is the logical layering of the system? A layered architecture is likely to be more flexible, separating the user interface, the business logic and the access to and storage of data.
- What is the physical layering of the system? More than one logical layer can reside on the same machine, for example the business logic and the data access can be on the same server. In large systems, there may be more than one machine providing the services of a single layer, for example several web-servers handling the presentation layer, which connect to two machines running the business logic, which connect to a single database server. If an application is delivered over the Internet or a company intranet, then much of the presentation layer will reside on the web-server, where Active Server Pages (ASP), Java Server Pages (JSP) or some other related technology will be used to construct the web pages dynamically and deliver them to the client's browser.
- What is the distribution of the system? It is conventional for the presentation layer to be located on many client machines, but, if the entity objects and the business logic are located on multiple machines, then the system must be designed to use mechanisms such as CORBA (Common Object Request Broker Architecture) or EJB (Enterprise Java Beans) to make it possible for clients to locate the objects they need to connect to in order to provide the functionality of the system.
- What protocols will be used to connect between layers of the system, particularly in a distributed architecture? Language- or operating system-specific protocols such as Java's Remote Method Invocation (RMI) or Microsoft's Distributed Component Object Model (DCOM) can be used but restrict the design to implementation on certain platforms. Open standards such as CORBA or SOAP (Simple Object Access Protocol), which uses XML (Extensible Markup Language), make it possible to build component-based systems that are not tied to particular platforms.

Building large, distributed systems is beyond the scope of this book, but in the rest of this chapter we explain some of the mechanisms that can be used to design persistence into an object-oriented system, and we start with the simplest—files.

18.3 File Systems

The simplest means of persistent storage available for computerized information systems uses files. Most users of personal computers are familiar with the idea of

files. Word-processors and spreadsheets store their data in files; browsers download them from websites where they are stored. At the simplest level, a file is a stream of bytes of data stored on some physical medium. In most cases, files are stored magnetically by adjusting the magnetic properties of the surface layer of a disk or tape. Files can also be stored optically on CD-ROMs and other optical storage systems or electronically in special kinds of memory, such as the flash memory used in palmtops and USB memory devices. However, the user is normally shielded from the physical implementation of the file system by the operating system of the computer or by a programming language, which provides high-level functions to create files, store data in them and retrieve that data.

18.3.1 File and record structures

Programming languages, and in some cases operating systems, also impose a structure on files. This structure breaks a file up into individual records, each of which groups together a number of fields representing the data that is to be held in the file. In the same way as each object contains a number of attributes, each one of which holds a particular kind of data about the object, each field in a record holds a particular kind of data about whatever it is that the record describes. For example, each record in a simple address book file would have fields to contain a name, lines of the address, city, postcode and telephone number. Records in files can take different forms, described below.

- Fixed length—Each record is made up of a number of fields, each of which has a fixed length in bytes. If the data in a particular field does not fill it, the field is padded out with special characters (often spaces). Each record is of the same, fixed length and it is possible to skip from the beginning of one record to the beginning of the next by jumping that fixed number of bytes ahead.
- Variable length—Each record is made up of a number of fields, each of which may have a maximum length but has a minimum length of zero bytes. Fields are usually separated or delimited by a special character that would not appear in the data. Records may also be delimited by a special character. The length of each record may also be stored at the start of the record, making it possible to skip to the beginning of the next record by jumping that variable number of bytes ahead.
- Header and detail—Records may be of two types: each transaction recorded consists of a header record, followed by a variable number of detail records. This approach can be used with many business documents, such as orders, invoices and delivery notes that have a variable number of lines on them. Each record will contain a record type field. The number of detail records may be held in the header so that it is possible to tell where the next header record starts.
- Tagged data—The data may have a complex structure, as in object-oriented systems, and it may even be necessary to hold objects of different classes in the same file. Every object and attribute may be tagged with some kind of description that tells a program reading the file what the type of each item is. This approach is used for data in files that use Hypertext Markup Language (HTML) and Extensible Markup Language (XML).

Some systems store information about the structure of the file in a data dictionary and this may be held in a separate file or at the start of the data file itself. This

makes it possible to write programs which can read the data out of any file that uses this format: the program first reads the data dictionary information and configures itself to read the appropriate data structures from the rest of the file.

As well as having alternative ways in which the data can be structured within files, files can have different types of organization, can be accessed in different ways and can serve different purposes in a system.

18.3.2 File organization

There are three ways in which files can be organized: serial, sequential and random.

Serial organization. Each record in the file is written onto the end of the existing records in the file. If a record is to be deleted, the file must be copied from the start up to the deleted record, which is skipped, and then the rest of the file is copied back to the disk.

Sequential organization. In the basic form of sequential organization, each record is written to the file in some pre-determined order, usually based on the value of one of the fields in each record, such as an invoice number. Records must be deleted in the same way as for serial files. Each record must be added to the file in its appropriate place and, if it is necessary to insert a record into a file, the file is copied up to the point where the record is to be inserted, the new record is written to the file and then the rest of the file is copied.

Random organization. The term random is a poor way of describing the organization of random files, as the organization is really anything but random. The records are added to the file by means of precise algorithms that allow records to be written and read directly without having to read through the rest of the file. What this means is that if you choose any record *at random*, it should be possible to access it more or less straightaway without searching through the file. The algorithm usually converts a key field of each record into an address in the file that can be reached directly.

18.3.3 File access

Depending on the file organization chosen, different ways of accessing the data in the files are available to the designer. The main ones are serial, index-sequential and direct.

Serial access. Serial and basic sequential files can only be accessed serially. To find a particular record, it is necessary to read through the file, record by record, until the required record is located.

Index-sequential access. Access to sequential files can be improved by maintaining an index on the field that is used to order the data in the file (the key). Index-sequential files are used where it is necessary to read the file sequentially, record by record, and to be able to go straight to a particular record using its key. The indexing mechanism used for index-sequential files dates back to the time when mainframe operating systems made it possible to allocate the particular disks, cylinders and tracks where a file would be stored.

Records are stored sequentially within blocks (areas of the disk that have a defined size). Enough blocks are allocated to the file for the total anticipated number of records. Records are written into blocks in key order, but the blocks are

not filled up from the start of the file; rather, records are distributed evenly across the blocks, leaving space for new records in each block.

The index on the file can be dense or sparse. In a dense index, there is an entry for every key with a pointer to the first record in the file with that key (there may be more than one). In a sparse index, there is an entry for the last record in each data block. To find a record, a program reads through the index until it finds a key value greater than the value of the key it is searching for. It then jumps to the block pointed to by that index entry and reads through it until it finds the required record.

To support large files, there may be two or more levels of index. For example, there may be a master index and a series of block indexes. The master index holds the value of the key field of the last record in each block index. Each block index holds the value of the key field of the last record of each block in a set of blocks. To find a record by its key, the master index is read until a key value is found that is greater than or equal to the key of the record being sought. This makes it possible to go to the block index for that record. The block index is then read until a key value is found that is greater than or equal to the key of the record being sought. This makes it possible to go to the block in which the record is held. The records in the block are then read sequentially until the desired record is found. A similar approach is taken in order to add records to an index-sequential file. The block in which the record is located is identified using the index, then the records in the block are read into memory and copied to the disk up to the point that the new record is to be inserted, the new record is written into the block, and then the rest of the records in the block are copied back to the block. The index may also need to be updated. Eventually some blocks will fill up, and it will be necessary to write some records into special overflow blocks. The addresses of the overflow blocks will be held in the blocks that have overflowed. Performance tuning of such files involves resizing them so that there are more blocks in the file and no records have to be stored in overflow blocks. Figure 18.2 shows the organization of data and indexes in an index-sequential file.

Index-sequential files have the advantage over sequential files that records can be read and written more quickly, although there is a storage overhead associated with maintaining the indexes. Compared to direct access, which is described next, there is also the overhead of the time taken to access the indexes before the data is reached.

Direct access. Direct access methods rely on the use of algorithms to convert the values of key fields in the records to addresses in the file. (The term random access is sometimes used.) The first and simplest of these is *relative addressing*. This access method requires the use of fixed length records and successive positive integers as keys. If each record is 200 bytes long, then record 1 will start at byte 1 of the file, record 2 at byte 201, record 3 at byte 401 and so on. It is possible to calculate the position of any record in the file by subtracting 1 from its key, multiplying the result by the size of the records and adding 1. Each record can be read directly by reading from that point in the file.

Hashed addressing is the second approach. This can use keys of any form. As with indexed sequential files, a fixed number of blocks is initially allocated to the file. This is usually a prime number of blocks, as this helps to achieve a more even spread of records into blocks. The key is then hashed to determine to which block

Master Index		Block Index (Index block 2)	
Record key	Block index address	Record key	Block address
Feng	1	Finlayson	55
Patel	2	Gomez	56
Zarzycki	3	Hanson	57
		Jacobson	58

		Patel	.84

Data Records in blocks (for simplicity only the keys are shown)			
Block address	Records		
55	Fern	Finch	Finlayson
56	Finn	Firmin	Ford Gomez
57	Gordon	Govan	Hamer Hanson
58	Ho	Ibrahim	Jacobson
...	...		

Figure 18.2 Schematic of indexes and data in an index-sequential file ordered by surname.

a particular record will be allocated. The hashing function is an algorithm that takes an ASCII string and converts it to an integer. There are many approaches. A simple approach is to take the characters in the string and convert them to their ASCII values (for example, 'A' is 65). These ASCII values are summed together. The sum is divided by the number of blocks in the file and the remainder or modulo gives the number of the block in the file into which that record will be placed, starting at block 0. If a block fills up, then an additional block will be used as overflow, and its address will be held in the block that the record would have been stored in. Figure 18.3 shows the organization of a hashed direct access file and the calculation of a simplified version of the hashing algorithm based on just three characters of the key.

Improving access. There are a number of ways of improving access to data in files. Files with a random organization can normally only be accessed directly or serially (for example, to copy the file), so it is not possible to read through them sequentially in an order based on key fields. However, it is possible to add two extra fields to each record containing the key values of the next record and the previous record in sequence (a linked list). This makes it possible to read through the records sequentially, but adds a considerable overhead when a record is added or deleted.

A common way of improving access to a file is to add a secondary index. This is a similar approach to that used in adding an index to an index-sequential file. It is used when there is a requirement either to access records in a file based on the values in some field other than the key field, for example to find address records by

Records hashed on first three characters of key field

Khan → Kha
ASCII Values = 75, 104, 97
 $75 + 104 + 97 = 276$
276 divided by 7 leaves a modulo of 3
So Khan will be added in Block 3.

Data records in blocks (for simplicity only the keys are shown)

Block no. in file	Records			
0	Hao			
1	Ford	Farmer		
2	Firmin	Firth		
3	Harris			
4	Hastings	Gomez	Jacobson	
5	Ibrahim	Finch	Fern	Gangar
6	Hanson			

Figure 18.3 Organization of a sample hashed direct access file.

postal code or zipcode, or to provide sequential access to a random file, by building an index of sequential keys. A separate file is created in which the keys are the values from the indexed field (for example, the postal code) in all the records in the main file. Each record also contains either the keys or the block addresses of each of the records that contain that indexed field. This kind of index is known as an *inverted file*. There are various structures that can be used for indexes, such as B-trees, which have different benefits—in terms of speeding up retrieving records—and different disadvantages—in terms of adding an overhead to updates to the file.

18.3.4 File types

We have seen that as well as files that hold data, there can also be files that hold indexes to the data in the main files. Other types of files may be required in a file-based system.

Master files. Master files are the files that hold the essential, persistent data records for the system. In transaction-processing systems the master files are updated with details of transactions that are recorded in transaction files. Master files usually require some kind of direct access so that records can be updated quickly.

Transaction files. Transaction files record business transactions or events that are of interest to the organization and that are used to update records in master files. In a banking system, transactions that take place when customers withdraw cash from an automatic teller machine (ATM) may be recorded in a transaction file. At the end of the day, the transaction file is processed in order to update all the accounts of customers who have withdrawn cash.

Index files. Indexes are used to speed up access to records in master files, as described above. There are many index file structures that can be used, and the choice of index structure will depend on the nature of the data and the type of access required.

Temporary files or work files. During the course of processing data, it may be necessary to create a temporary file that is used to hold the results of one process before it is used in another process. For example, it may be necessary to sort records into a particular order so that a report can be produced, a work file containing the records in the correct order would be produced and then deleted once the report had been created. If you use a PC that prints your work in the background (while you get on with another task), then it is using spool files to hold the printed output that is sent to the printer. These spool files are deleted when they have been printed.

Backup files. Backup files may be direct copies of master files or transaction files that are held on the system and that allow the data to be restored if the originals are destroyed or corrupted. Alternatively, they may be files with a special structure that can be used to reconstruct the data in the system.

Parameter files. Many programs need to store information about items of data of which there is only one record. These are typically system settings, such as the name and address of the company using the software, and configuration information, such as the currency format to be used to print out money values or the format to be used for printing the date. Parameter files hold this kind of information.

In a project that uses files to store data, part of the role of the designer is to choose the appropriate file organization and access method for the storage of all the objects in the system. In most cases, objects will need to be stored and retrieved using some kind of object identifier and direct access files will be required. However, there may be requirements for all the objects of a particular type to be retrieved from the file in sequence, which may indicate the need for an organization that supports sequential access, or for the addition of an index. Some object-oriented languages provide mechanisms for streaming objects out to a serial file. This is the case with both Smalltalk and Java. Java, for example, contains two classes called `ObjectOutputStream` and `ObjectInputStream` that can be used to write objects together with information about their class to a disk file and read them back again.

Many organizations use database management systems to hold data. In their systems, there will be no need for designers to make decisions about the physical file structures used to provide persistent storage for objects. However, there are still situations where files are the appropriate mechanism for storing data in an application. In the next section we present one example.

18.3.5 Example of using files

In the Agate system, one of the non-functional requirements is that the application can be customized for use in different countries with different languages. This means that all the prompts that are displayed in windows, labels on buttons and menus and error and warning messages cannot be written into the classes in the presentation layer as string literals. For example, if a user interface class was implemented in Java using string literals, the line of code to create a cancel button would

look like this:

```
Button cancelButton = new Button ("Cancel");
```

To use the program in French, someone would have to go through finding all the strings like this and translating them. Then there would be two versions of the program, and any changes would have to be made in both. This is unmanageable with just two languages, let alone several. In Java, it is possible to use the class `java.util.Locale` to hold information about the current locale in which an application is running. This includes information about the language, the country and a variant value. For example 'fr' for French, 'FR' for France and 'EURO' to indicate that the country has used the Euro as its unit of currency since January 2002. For an application running in French in Canada, only the language code 'fr' and the country code 'CA' would be required.

Another Java class, `java.util.ResourceBundle`, uses the locale information to hold objects in memory, each of which is associated with a key value. It can load these objects from a file, and the name of the file is made up of the name of the resource and the language, country and variant codes. So for a resource called `UIResources`, designed to hold the values for all the prompts and labels, there could be different versions called `UIResources_fr_FR_EURO`, `UIResources_en_UK`, and `UIResources_en_US`, for France, England and the USA respectively.

When the user interface class is instantiated, it needs to find out its locale and then load the correct resources into memory with a line of Java like this:

```
resources = ResourceBundle.getBundle
    ("UIResources", currentLocale);
```

Then the code to set up the cancel button becomes the following:

```
Button cancelButton = new Button
    (resources.getString("Cancel"));
```

The resource file is made up of records, each of which is on a separate line, with an equals sign to separate the key from the associated string, for example:

```
Cancel = Annuler
OK = OK
File = Fichier
```

for the French version. When the application is deployed, either the installation routine must install the correct resource files on the machines that will be running the user interface classes, or a full set of files must be deployed, and the appropriate one will be chosen at runtime. (See Chapter 19 for more detail about implementation and deployment.)

18.4 Database Management Systems

18.4.1 Files and databases

Files are appropriate for simple programs and for storing data that does not need to be shared and updated by many users. During the 1960s systems were built using files to store data; since the 1970s most large commercial systems have used

databases of some sort to hold their data and, more importantly, database management systems to organize and manage the tasks associated with storing and providing effective access to large volumes of data.

Using files to store data can result in a number of problems.

- As the number of applications grows, the number of different files grows. Some of these files may hold the same data for different applications in different formats and so data is duplicated, taking up unnecessary storage space. This is known as redundancy.
- There is the risk that the updates to data in different applications will not be synchronized. For example, a customer address may be changed in one file but not in another, leaving the data inconsistent.
- Each application must contain its own mechanisms for storing the data in its set of files. If the data changes or the way that it is stored has to be changed, then each program within an application that accesses that data must be amended. This makes it difficult to add new programs to an application that use some of the same data but also need to store additional data.
- As business requirements change, users may want to access the data in new ways, for example to produce a report combining data from different applications. This cannot be implemented without considerable programming effort.

The first step towards resolving these problems is to analyse the storage requirements of different applications across the organization and to build an enterprise database that contains all the data from different applications. Each application then uses a subset of this database for its own requirements. The second step is to use a database management system to organize and manage the data and to decouple the storage mechanisms from the application programs. The aim of using a DBMS is to separate the details of the way that data is stored physically from the way that it is used in application programs. This is achieved by producing a logical model of the data that is independent of the demands of applications and that could be stored in different ways in the database. This is known as the Three-schema Architecture. Figure 18.4 shows the Three-schema Architecture. The external schema represents the ways in which data is used in application programs. The conceptual schema is a logical model of the data and is independent both of the external schema and of the details of how the data will be stored. The physical organization of the data is to be found in the internal schema, which defines the

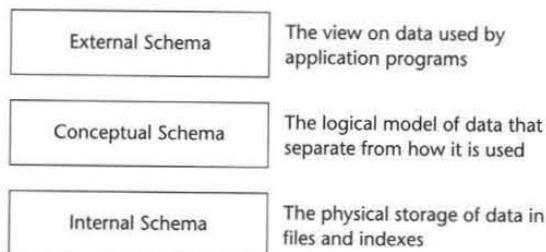


Figure 18.4 The Three-schema Architecture.

files used to store the data. The aim of this approach is to isolate the application programs from the details of how any particular item of data is stored. This is central to the way that relational DBMS work. Design for relational DBMS is described in more detail in Section 18.5.

DBMS provide more than just a means of storing data that can be shared across many applications. They provide tools and features that can be used to manage the data.

- *Data definition language (DDL)*. The DDL is used to specify the data that is held in a database management system and the structures that are used to hold it.
- *Data manipulation language (DML)*. The DML is used to specify updates and retrievals of the data in the DBMS.
- *Integrity constraints*. Constraints can be specified to ensure that the integrity of the data is maintained.
- *Transaction management*. Updates to the database can be specified as transactions in which all of the component updates must succeed, otherwise the entire transaction is *rolled back*, which means that it is not committed to the database.
- *Concurrency*. Many users can simultaneously use the database and update its contents.
- *Security*. Access to the data in the database can be controlled, and permissions granted to different users for different levels of access (for example in SQL, select, insert, update and delete).
- *Tuning of storage*. Tools can be used to monitor the way that data is accessed and to improve the structures in the internal schema in order to make the access more efficient. These changes can then be made without having any impact on the application programs.

As mentioned above, these structures in the internal schema will be files. An important feature of a DBMS is that the kind of file used, the access methods and the indexes that are held on the file are hidden from users of the DBMS (typically application programmers) and can be changed without affecting the programs that use that data.

These changes to the database do not happen automatically. For large systems a Database Administrator (DBA) must be employed to manage the database and to ensure that it is running efficiently. The DBA will be responsible for controlling the data dictionary that defines the conceptual schema of the database, for controlling access to data and for tuning the performance of the database. For smaller systems a DBA will not be needed, but someone will need to be responsible for managing the database.

In summary, the use of a DBMS based on the Three-schema Architecture has a number of advantages over the use of files to store data for an information system.

- The use of a conceptual schema can eliminate unnecessary duplication of data.
- Data integrity can be ensured by the use of integrity constraints and transaction management techniques.
- Changes to the conceptual schema, the logical model, should not affect the application programs, provided the external schema used by the application programs does not have to be changed.

- Changes to the internal schema, the physical storage of the data, have no impact on the conceptual schema and should not affect the application programs, except perhaps positively in enabling them to access data more efficiently. Compromises may have to be made between the needs of different application programs.
- Tools are available for tuning the performance of the database, for the back-up and recovery of data and to control security and access to data by multiple simultaneous users.

However, the use of DBMS may also have disadvantages for organizations that decide to go down this route.

- There is a cost associated with investing in a large DBMS package.
- There is a running cost involved in employing staff to manage the DBMS.
- There will be a processing overhead in converting data from the database to the format required by the application programs.

The most widely used type of DBMS is the relational DBMS. For object-oriented systems one might hope to be able to use an object DBMS. However, in many situations, organizations have an existing relational DBMS and new object-oriented applications must share that enterprise database. A relational database with C++ or Java as the application development language is still the most common platform in use.

18.4.2 Types of DBMS

The three main types of database that we are concerned with here are relational, object and hybrid object-relational.

Relational databases. The idea of relational databases was first suggested by Codd (1970). His proposal was followed by considerable research effort that led to the development of commercial relational database management systems (RDBMS) during the 1970s. However, it was not until 1986 that the American National Standards Institute published the first SQL standard based on this work (ANSI, 1986). SQL (Structured Query Language) is now the standard language for relational databases and provides both DDL and DML capabilities.

Relational databases have a theoretical foundation in set theory and their operations are defined in terms of the *relational algebra*, a mathematical specification of the operations that can be carried out on *relations*. The essence of the relational model is to eliminate redundancy from data and to provide the simplest possible logical representation of that data. This is achieved by means of a series of steps that can be applied in analysing the data and that result in normalized data. This normalized data is held in relations or tables. This process simplifies a complex data structure until it can be held in a series of tables. Each table is made up of *rows* of data. Each row contains attribute values that are organized in *columns*. Each column contains data values of the same attribute type. The data in each row must be distinct and can be uniquely identified by some combination of attribute values in that row. Each attribute value in the table must be atomic, that is, it may not contain multiple values or be capable of being broken down further. Figure 18.5 shows the conventional form for representing tables on paper. In

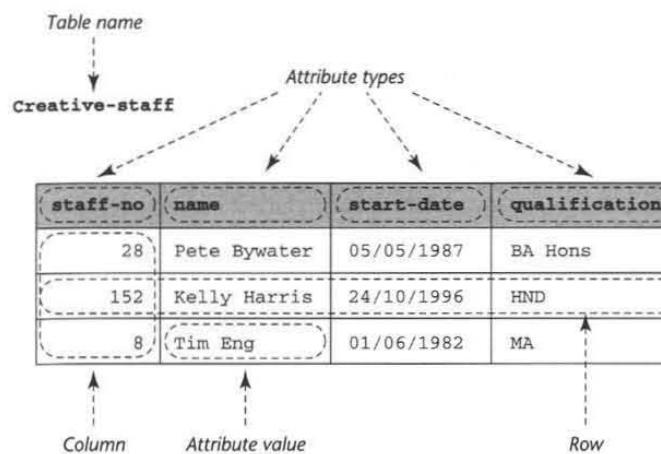


Figure 18.5 Conventional representation of a table.

existing RDBMS, all data structures must be decomposed into this kind of two-dimensional table¹.

The weakness of current implementations of RDBMS lies in the fact that objects in object-oriented systems do not fit easily into this model. They can be broken down into tables, as is shown in Section 18.4, but there is a processing overhead associated with breaking them down and reconstructing them. References to other objects (represented by associations in the class diagram) must also be maintained when an object is stored in a relational database, and restored when it is retrieved. And, even if the associated object is not itself in memory, some mechanism must be created to allow it to be referenced and to be sent messages. Tables in RDBMS are linked to one another by common attribute values (foreign keys), whereas objects are linked to one another by references or pointers. Data in RDBMS is processed in sets, while data in ODBMS must be navigated through, following links from object to object.

Relational DBMS are currently the most widely used type of DBMS. They are based on a sound mathematical theory, they have been developed over a number of years, they are robust and they are efficient and flexible for the kind of data that they are designed to handle. The best known is probably Access; others include Oracle, SQL-Server, DB2, Informix, Ingres, Progress and Sybase. The growth of Linux has also led to the availability of free and open source databases such as MySQL.

Object databases. In contrast, objects in an object-oriented system are not flat, two-dimensional structures. Each object may contain other objects nested within it. For example, a SalesOrder object could contain an Address object that contains its own attributes, and a Collection of OrderLine objects, each of which is made up of two attributes. An example of this is shown in Fig. 18.6 with a SalesOrder object with its class definition in UML notation. Object database

¹ Date and Darwen (1998) argue that the relational model can handle complex data types, but it is the way that relational DBMS have been implemented that leads to their inability to handle these data types.

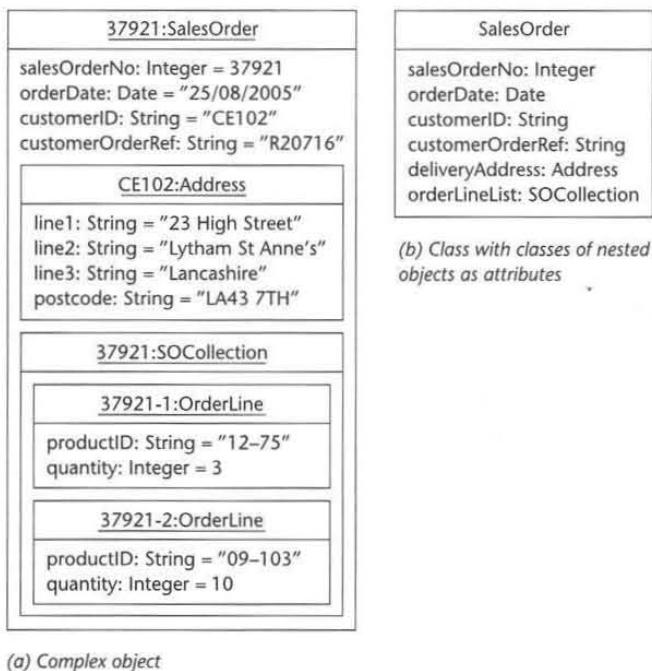


Figure 18.6 Composite SalesOrder object with equivalent UML class.

management systems (ODBMS) have been developed to handle complex objects of this sort. Part of the motivation for the development of ODBMS has been the growth in the number of applications that use complex data structures. These include multimedia applications, in which objects such as sounds, images and video clips are not easily represented in tables, and applications such as computer-aided design packages in which the designer may want to deal with different levels of abstraction, for example treating a subassembly in terms of its behaviour as a subassembly, in terms of the individual chips or in terms of the components such as logic gates that make up those chips. ODBMS provide services that make it possible to store complex objects of this type. Examples of ODBMS include Gemstone/S, Jasmine and ObjectStore.

Object-relational databases. Object-relational databases combine the simplicity and efficiency of relational databases with the ability of object databases to store complex objects and to navigate through the associations between classes. The SQL standard is being updated to allow the relational model to incorporate many features of object-oriented systems such as user definable abstract data types, inheritance and operations. The open source product PostgreSQL is probably the most well-known hybrid DBMS. Oracle now includes some hybrid features. In what follows we shall focus on relational and object databases.

18.5 Designing for Relational Database Management Systems

18.5.1 Relational databases

RDBMS have been in use since the 1970s. They use mature technology and are robust. It is common for an object-oriented system to be built to use a relational DBMS. Relational databases hold data in flat two-dimensional tables, whereas classes may have complex nested structures with objects embedded within other objects. If it is necessary to use an RDBMS to provide the storage for a system built using an object-oriented programming language, then it will be necessary to 'flatten' the classes into tables in order to design the storage structures. When the system requires an instance of a class from the database, it will have to retrieve the data from all the tables that hold parts of that object instance and reconstruct it into an object. When a complex object instance has to be stored, it will have to be taken apart and parts of it will be stored in different tables. The designer of such a system has to decide on the structure of the tables to use to represent classes in the database. It should be emphasized that it is only the attribute values of object instances that are stored in an RDBMS; operations are implemented in the programming language used.

There are two ways in which classes can be converted to tables in a relational database. The first, normalization, is suitable for decomposing complex objects into tables. It is used in systems that are not object-oriented to design the structure of tables in databases. It can also be used during object design to simplify large complex objects that are not cohesive. The second approach is based on a series of rules of thumb that can be applied to classes in a class diagram to produce a set of table structures. In this section, we describe these two approaches and, in Section 18.6, we discuss the impact that this will have on the design of classes in the system.

18.5.2 Data modelling and normalization

In order to store the objects from an object-oriented system in a relational database, the objects must be flattened out. *Normalization* is an approach that is also used to convert the complex structures in business documents into tables to be stored in a relational database. A typical example of its use would be to design a set of tables to hold the data in a sales order like the FoodCo example in Fig. 16.3. How then do we apply normalization? Normalization is based on the idea of *functional dependency*.

For two attributes, A and B, A is functionally dependent on B if for every value of B there is exactly one value of A associated with it at any given time. This is shown as:

$$B \rightarrow A$$

Attributes may be grouped around functional dependencies according to the rules of normalization to produce normalized data structures that are largely free of redundancy. There are five *normal forms* of normalized data. The data is free of redundancy in *fifth normal form*. For practical purposes it is usually adequate to

normalize data into *third normal form*. Normalization is carried out in a number of steps, and we shall apply these to an example from the Agate case study.

Analysis activity during a further iteration has identified a class called `InternationalCampaign`, and the attribute values of two instances of this class are shown in Fig. 18.7. As stated earlier, we may wish to decompose this class into simpler classes because it is not cohesive or we may need to decompose it into table structures for storage using an RDBMS. The same approach is used in both cases. Here we are applying normalization as part of the design for a relational database.

The first step is to remove any calculated values (derived attributes). The attribute `locationCount` is a derived attribute. It is the number of locations associated with any one international campaign and can be calculated when an instance of `InternationalCampaign` is instantiated.

We now create a relation that is said to be in unnormalized form. Each `InternationalCampaign` is uniquely identified in this system by its `campaignCode`. This is the *primary key* attribute. Figure 18.8 shows the data from these instances in a table. Each `InternationalCampaign` is represented in a single row. Note that there are multiple values in some of the columns in each row. These relate to the locations where the campaign will run and the manager in each location.

A table is in *first normal form* (1NF) if and only if all row/column intersections contain atomic values. The table in Fig. 18.8 does not conform to this criterion and must be redrawn as in Fig. 18.9. These multiple values are often known as *repeating groups*.

<code>SMHL:InternationalCampaign</code>	
<code>campaignCode = SMHL</code>	
<code>campaignTitle = Soong Motors</code>	
<code>Helion Launch</code>	
<code>locationCount = 3</code>	
<code>locationsList =</code>	
<code>[</code>	
<code>locationCode = HK</code>	
<code>locationName = Hong Kong</code>	
<code>locationMgr = Vincent Sieuw</code>	
<code>locationMgrTel = ext. 456</code>	
<code>]</code>	
<code>locationCode = NY</code>	
<code>locationName = New York</code>	
<code>locationMgr = Martina Duarte</code>	
<code>locationMgrTel = ext. 312</code>	
<code>locationCode = TO</code>	
<code>locationName = Toronto</code>	
<code>locationMgr = Pierre Dubois</code>	
<code>locationMgrTel = ext. 37</code>	
<code>]</code>	

<code>YPSC:InternationalCampaign</code>	
<code>campaignCode = YPSC</code>	
<code>campaignTitle = Yellow Partridge</code>	
<code>Summer Collection</code>	
<code>locationCount = 2</code>	
<code>locationsList =</code>	
<code>[</code>	
<code>locationCode = HK</code>	
<code>locationName = Hong Kong</code>	
<code>locationMgr = Jenny Lee</code>	
<code>locationMgrTel = ext. 413</code>	
<code>]</code>	
<code>locationCode = NY</code>	
<code>locationName = New York</code>	
<code>locationMgr = Martina Duarte</code>	
<code>locationMgrTel = ext. 312</code>	

Figure 18.7 Example `InternationalCampaign` objects.

InternationalCampaign

<code>campaignCode</code>	<code>campaignTitle</code>	<code>locationCode</code>	<code>locationName</code>	<code>locationMgr</code>	<code>locationMgrTel</code>
SMHL	Soong Motors Helion Launch	HK NY TO	Hong Kong New York Toronto	Vincent Sieuw Martina Duarte Pierre Dubois	456 312 37
YPSC	Yellow Partridge Summer Collection	HK NY	Hong Kong New York	Jenny Lee Martina Duarte	413 312

Figure 18.8 Table for sample International Campaigns.

InternationalCampaign-1

<code>campaignCode</code>	<code>campaignTitle</code>	<code>locationCode</code>	<code>locationName</code>	<code>locationMgr</code>	<code>locationMgrTel</code>
SMHL	Soong Motors Helion Launch	HK	Hong Kong	Vincent Sieuw	456
SMHL	Soong Motors Helion Launch	NY	New York	Martina Duarte	312
SMHL	Soong Motors Helion Launch	TO	Toronto	Pierre Dubois	37
YPSC	Yellow Partridge Summer Collection	HK	Hong Kong	Jenny Lee	413
YPSC	Yellow Partridge Summer Collection	NY	New York	Martina Duarte	312

Figure 18.9 Revised table for International Campaigns without repeating groups.

The `campaignCode` no longer uniquely identifies each row in the table. Each row is identified by a combination of `campaignCode` and `locationCode`. These attributes form a candidate *primary key* for the table.

The data values have been flattened out into a two-dimensional table and could now be stored in a relational database as they are. However, there is redundancy that we want to eliminate from the data. If redundant data is held in the database, there is the risk that values will not be updated correctly. For example, if Martina Duarte's telephone extension number changes, the system must ensure that it is correctly updated in every row in which it appears. This is inefficient and prone to error.

The next step is to convert these relations to *second normal form* (2NF). A relation is in 2NF if and only if it is in 1NF and every non-key attribute is fully dependent on the primary key. Here the attribute `campaignTitle` is only dependent on `campaignCode`, and `locationName` is only dependent on `locationCode`. (These are sometimes called part-key dependencies.) The other

attributes are dependent on the whole primary key. (Remember A is dependent on B if for every value of B there is exactly one value of A associated with it at a given time.) Figure 18.10 shows the creation of two new relations Campaign and Location.

The next step is to convert the tables to *third normal form* (3NF). A relation is in 3NF if and only if it is in 2NF and every attribute is dependent on the primary key and not on another non-key attribute. Campaign and Location are in 3NF, but in InternationalCampaign-2 locationMgrTel is dependent on location Mgr and not on the primary key. Figure 18.11 shows the tables in 3NF with the addition of a new table called LocationManager.

These relations can be shown in a diagram using the notation of entity-relationship diagrams, which are often used to represent the logical structure (conceptual schema) of relational databases. Note that this is not part of the UML notation. Figure 18.12 shows the relations of Fig. 18.11 as an entity-relationship diagram. Some UML CASE tools can also produce entity-relationship diagrams, and some can generate the SQL statements to create the tables.

If we examine the attributes of these relations, we may come to the conclusion that InternationalCampaign was not a very well analysed class in the first place. It should perhaps be a subclass of Campaign. LocationManager appears to be nothing more than CreativeStaff with an association to InternationalCampaign. We also require a new class called Location with an association to InternationalCampaign. One possible design is shown in Fig. 18.13. Note that in this class diagram we have included the collection classes that were introduced during object design (Chapter 14). There is presumably also an association between StaffMember (the superclass of CreativeStaff) and Location, but we should not rely on that to find out which Location an InternationalCampaign is running in, as the member of staff who is LocationManager could move offices.

InternationalCampaign-2

campaignCode	locationCode	locationMgr	locationMgrTel
SMHL	HK	Vincent Sieuw	456
SMHL	NY	Martina Duarte	312
SMHL	TO	Pierre Dubois	37
YPSC	HK	Jenny Lee	413
YPSC	NY	Martina Duarte	312

Campaign

campaignCode	campaignTitle
SMHL	Soong Motors Helion Launch
YPSC	Yellow Partridge Summer Collection

Location

locationCode	locationName
HK	Hong Kong
NY	New York
TO	Toronto

Figure 18.10 2NF tables.

InternationalCampaign-3

campaignCode	locationCode	locationMgr
SMHL	HK	Vincent Sieuw
SMHL	NY	Martina Duarte
SMHL	TO	Pierre Dubois
YPSC	HK	Jenny Lee
YPSC	NY	Martina Duarte

Campaign

campaignCode	campaignTitle
SMHL	Soong Motors Helion Launch
YPSC	Yellow Partridge Summer Collection

Figure 18.11 3NF tables.

LocationManager

locationMgr	locationMgrTel
Vincent Sieuw	456
Martina Duarte	312
Pierre Dubois	37
Jenny Lee	413

Location

locationCode	locationName
HK	Hong Kong
NY	New York
TO	Toronto

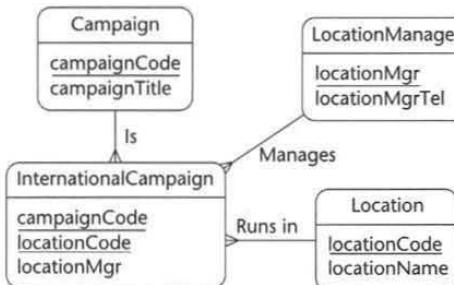


Figure 18.12 Entity-relationship diagram for tables of Fig. 18.11.

The classes in Fig. 18.13 raise a number of questions that should have been addressed during the analysis of requirements. Some of these are listed below.

- Does each campaign have a location? Is this the main location for an international campaign?
- Does each member of staff work in a location?
- Are there different versions of adverts for different locations?
- How are international campaigns costed? What currencies are used?

You can probably think of others. In an iterative lifecycle, it is acceptable to be raising these issues in early iterations. In a traditional waterfall lifecycle, these issues should have been resolved during the analysis stage, but in reality may not have been.

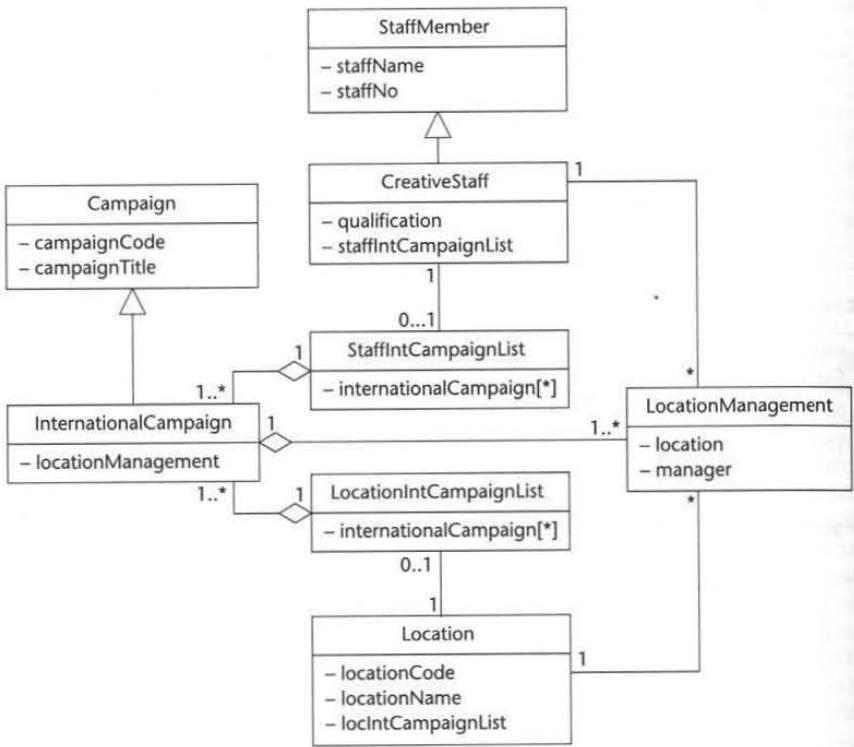


Figure 18.13 Excerpt from design class diagram for `InternationalCampaign`.

18.5.3 Mapping classes to tables

An alternative approach to that provided by normalization is to follow a set of guidelines for how to map the classes and multiplicities in the class diagram to tables in a relational database design. A summary of the patterns that can be applied to this mapping can be found in Brown and Whitenack (1996). The following guidelines are derived from Rumbaugh et al. (1991) and Brown and Whitenack (1996).

- Classes with a simple data structure. These classes become tables.
- Object identifiers become primary keys. A unique identifier is generated for every object and can be used as a primary key in the relational table in which it is held. (Various schemes are available that guarantee a unique id for every object.)
- Classes that contain an instance of another class as an attribute. A separate table should be created for the embedded class. Objects of the embedded class should be allocated a unique object identifier. The object identifier should replace the embedded object in the table for the container class as a foreign key.

■ Classes that contain collections. Allocate an object identifier to the class held in the collection. This class will be represented by a table. Create a separate table that contains two columns. The first holds the object identifiers of the objects that contain the collection; the second holds the object identifiers of the objects that are held in the collection.

- One-to-many associations can be treated like collections.
- Many-to-many associations become separate tables. Create a table that contains two columns. Each row contains a pair of object identifiers, one from each object participating in the association. (These are like two collections.)
- One-to-one associations are implemented as foreign key attributes. Each class gains an extra attribute in which to hold the object identifier of the associated object.

(A *foreign key* is used in relational databases to create the relationships between tables. `locationCode` in the `InternationalCampaign` table in Fig. 18.11 is an example of a foreign key. Objects do not have keys, and this is why object identifiers are allocated to them. It may be possible to use an attribute that will have a unique value in each instance of a class as a foreign key.)

When a relational database is used, collection classes that exist only to provide access to a set of objects of the same class need not be part of the data that is stored in tables. If it is necessary to iterate through every instance of a particular class, this can be done by selecting every row from the table.

Inheritance poses more of a problem. There are three alternative ways of mapping an inheritance hierarchy to relational database tables.

- Only implement the superclass as a table. Attributes of subclasses become attributes of the superclass table and hold null values where they are not used. This approach is most appropriate where subclasses differ from their superclass more in behaviour than in attributes. A `type` attribute is required to indicate which subclass each row represents.
- Only implement the subclasses as tables. The attributes of the superclass are held in all the subclass tables. This only works if the superclass is abstract and there will be no instances of it.
- Implement all the classes (both superclass and subclasses) as separate tables. To retrieve the data for a subclass, both its own table and the table of its superclass must be accessed. Again, a `type` attribute may be required.

The solution that is chosen may depend on the requirements of the particular application or may be constrained by the use that will be made of the data in the database by other applications.

This brings us to a further aspect of relational databases: data is added to and retrieved from them using SQL statements. SQL provides both the DDL and DML for relational databases. Figure 18.14 shows the SQL statements necessary to create the tables of Figs 18.11 and 18.12 in Oracle generated by the CASE tool from the storage class diagram.

Figure 18.15 shows an SQL statement that finds all the international campaigns with the `locationName` 'Hong Kong'. Typically this kind of query would be written as a procedure with a variable in place of the string 'Hong Kong', and the variable would be replaced with a parameter value when the procedure was invoked,

```

CREATE TABLE Campaign (
    campaignCode VARCHAR(4) NOT NULL,
    campaignTitle VARCHAR(50) NULL,
    PRIMARY KEY (campaignCode)
);
CREATE TABLE InternationalCampaign (
    campaignCode VARCHAR(4) NOT NULL,
    locationCode VARCHAR(2) NOT NULL,
    locationMgr VARCHAR(30) NULL,
    PRIMARY KEY (campaignCode, locationCode)
);
CREATE TABLE Location (
    locationCode VARCHAR(2) NOT NULL,
    locationName VARCHAR(20) NULL,
    PRIMARY KEY (locationCode)
);
CREATE TABLE LocationManager (
    locationMgr VARCHAR(30) NOT NULL,
    locationMgrTel INT NULL,
    PRIMARY KEY (locationMgr)
);

```

Figure 18.14 SQL statements to create tables of Figs 18.11 and 18.12.

```

SELECT campaignTitle FROM Campaign c, InternationalCampaign ic,
Location l
WHERE c.campaignCode = ic.campaignCode
AND ic.locationCode = l.locationCode
AND l.locationName = 'Hong Kong'

```

Figure 18.15 SQL statement to find campaigns running in Hong Kong.

allowing the same query procedure to be reused to find campaigns in any location.

There is a design decision to be made in deciding where to place the responsibility for this kind of requirement.

- This SQL statement could be executed and only data for those objects that are required would be returned from the database. This replaces the interaction to select the relevant objects modelled in a sequence diagram with functionality provided by the DBMS.
- Alternatively, the data from all these tables could be retrieved from the database and used to instantiate the objects in the system. Each International Campaign object could then be sent a message to check whether it includes the Location 'Hong Kong'. This will involve it sending a message to each associated Location object. This is more object-oriented, but will take a longer time to execute.
- The third alternative is to retrieve data from each table in turn, as though navigating through the structure of the class diagram, first the International Campaign then each of the Locations for that InternationalCampaign.

This approach requires use of indexes on the tables to make access possible in a reasonable time.

This kind of design decision trades off the pure object-oriented approach against the efficiency of the relational database. In order to retrieve this data into objects in a programming language such as Java, the SQL statements must be embedded in the program. In Chapter 19, we present examples of how this can be done using JDBC (Java Database Connectivity). During design, we have to decide which classes have the responsibility for accessing the database. In Section 18.8 we describe two different approaches to this design decision and show how they can be modelled in UML. However, before we address how we can model the database management responsibilities of the system, we need to consider object DBMS and what they have to offer the designer as an alternative to relational DBMS.

18.6 Designing for Object Database Management Systems

18.6.1 Object databases

Object DBMS differ from current relational DBMS in that they are capable of storing objects with all their complex structure. It is not necessary to transform the classes in the design model of the system in order to map them to storage objects. As you might expect, using an object database maintains the seamlessness that is claimed for object-oriented systems right through to the storage of objects in the database. Designing for an object database will have a minimal impact on the design of the system.

The standard for object databases was originally set by the Object Data Management Group (ODMG) and is currently available in Version 3.0 (Cattell et al., 2000). The standard defines both the Object Definition Language (ODL) and the Object Manipulation Language (OML) for object databases. The ODL is similar to the DDL elements of SQL for relational databases but allows objects to maintain their complex structure: objects can contain other objects, including collections, as attributes. Figure 18.16 shows the ODL definition of the International Campaign, StaffMember and CreativeStaff classes based on Fig. 18.12 but with the StaffIntCampaignList as an embedded attribute of CreativeStaff. Note also that the client contact association with Client is shown as a one-to-one association to illustrate the syntax.

However, in 2001 the ODMG was disbanded and the work that had been done on the standard was submitted to the Java Community Process (JCP) in response to Java Specification Request (JSR) 12, the Java Data Objects (JDO) Specification. JDO provides an abstract API for Java programs to access databases that is independent of the underlying database technology. The mapping between the Java classes and the structures in the database is set up in an XML configuration file. Mapping tools are explained in Section 18.8.4.

Individual ODBMS do not necessarily conform to the ODMG standard. Eaglestone and Ridley (1998) describe both the ODMG standard and the syntax of O₂. Blaha and Premerlani (1998) describe how to map an object-oriented design to ObjectStore. Some object databases include a DML similar to SQL that can be

```

interface InternationalCampaign
(extent international_campaigns
key (campaignCode, locationCode))
{
    attribute String campaignCode;
    attribute String locationCode;
}
interface StaffMember
(extent staff_members
key staffNo)
{
    attribute Short staffNo;
    attribute String staffName;
    attribute Date staffStartDate;
}
interface CreativeStaff: StaffMember
(extent creative_staff
key staffNo)
{
    attribute String qualification;
    attribute List<InternationalCampaign>
staffIntCampaignList
relationship Client isContact;
inverse Client::hasContact;
}

```

Figure 18.16 ODL for InternationalCampaign, StaffMember and CreativeStaff.

used to query data in the database. Statements in this OML are usually embedded in the program code of an object-oriented language. Unlike SQL statements, which are usually built as strings and then executed by a call to a system function or passed to a database server, OML queries are extensions to the programming language that is being used. Figure 18.17 shows a C++ operation to query the collection of InternationalCampaign objects (abbreviated in the figure to IntCampaign to save space) associated with a particular CreativeStaff instance in order to find one that contains a particular campaignCode. The campaignCode would be passed as a parameter to the operation.

Object databases provide much better facilities for navigating the structure of the classes in the class diagram. Figure 18.18 shows an excerpt from a class declaration and implementation in C++ that illustrates how the associations are created between classes using ObjectStore's macros. This excerpt declares an attribute of CreativeStaff called isContact, the type of which is an os_Set of pointers to objects of type Client. The inverse association from Client is called hasContact. This technique makes it straightforward to implement the associations, and it is not necessary to manually insert object identifiers into the classes to handle the associations. This would make the collection classes of Fig. 18.13 unnecessary.

Where an object contains a reference to another object as an attribute, it is possible to directly invoke its public methods or access its public attributes even if the object in question is not in memory. The great benefit of object databases is that they transparently materialize objects from the database into memory without the

```

IntCampaign * CreativeStaff::findIntCampaign ( string campaignCode )
{
    IntCampaign * intCampaignPointer;
    intCampaignPointer = staffIntCampaignList.getValue().query_pick(
        "IntCampaign",
        "campaignCode == this->campaignCode",
        os_database::of(this));
    return intCampaignPointer;
}

```

Figure 18.17 Example query (highlighted in bold) in C++ for ObjectStore.

```

os_relationship_1_m(CreativeStaff, isContact, Client, hasContact,
os_Set<Client*>) isContact;
...
os_rel_1_m_body(CreativeStaff, isContact, Client, hasContact);

```

Figure 18.18 Example association declaration and implementation in C++ for ObjectStore.

need for special action to be taken by the developer. Typically the only requirement is for some part of the application to make a connection to the database and for operations that read from or write to the database to be bracketed by commands to start and finish a transaction. This approach is used, for example, in ObjectStore for Java and C++.

The structure of the class diagram will require minimal changes to be used with an object database. Minor changes may be required. For example, if you use ObjectStore Personal Storage Edition (PSE Pro) with Java, you will find that some of the Java collection classes cannot be made persistent. However, ObjectStore provides persistent equivalents for these classes. For example, wherever you have used a class such as a Hashtable and require it to be persistent, it must be replaced with an instance of the class OSHashtable.

Although object DBMS make it easier to map an object-oriented design straight to a database, there is one area where the seamlessness breaks down. Apart from very simple operations, such as those to insert new values or to do simple arithmetic, ODL and hence object databases do not support the storage of operations in the database. Operations must still be implemented in an object-oriented programming language such as Java or C++.

18.7 Distributed Databases

Before we consider how to design the classes that handle data management for a database that is not object-oriented, it is worth considering the issue of design for distributed databases.

In a simple system, objects can be stored on the local machine in some kind of database, brought into the memory of the machine and sent messages to invoke their operations before being saved back to the database and removed from memory. This is not always possible. The objects that participate in a particular use

case need not be on the same machine as the user interface, indeed they may be distributed on different machines. This is the essence of architectures based on multiple tiers or layers, which we discussed in Chapter 12 and used as the basis of the user interface design in Chapter 17.

There are several ways of communicating between layers on distributed machines, for example Remote Procedure Calls (RPC) in languages such as C or C++ and Remote Method Invocation (RMI) in Java. The object-oriented industry standard, however, is CORBA (the Common Object Request Broker Architecture). CORBA Version 2.0 is defined as a standard by the Object Management Group (OMG, 1995), a different group from the ODMG. More recently, Simple Object Access Protocol (SOAP) has emerged as a contender in this marketplace. SOAP uses XML to encapsulate messages and data that can be sent from one process to another. We explain CORBA here as an example of this kind of communication protocol, because CORBA is platform and language independent.

CORBA separates the interface of a class (the operations that it can carry out) from the implementation of that class. The interface can be compiled into a program running on one computer. An object instance can be created or accessed by name. To the client program it appears to be in memory on the same machine. However, it may actually be running on another computer. When the client program sends it a message to invoke one of its operations, the message and its parameters are converted into a format that can be sent over the network (known as marshalling); at the other end the server unmarshals the data back into a message and parameters and passes it to the implementation of the target object. This object then carries out the operation and, if it returns a value, that value is marshalled on the server, unmarshalled on the client and finally provided as a return value to the client program. The marshalling and unmarshalling process makes it possible to pass objects with a complex structure over the network by flattening them out and reconstructing them at the other end.

CORBA achieves this by means of programs known as ORBs (Object Request Brokers) that run on each machine. The ORBs communicate with each other by means of an Inter-ORB Protocol (IOP). Over the Internet, the protocol used is IIOP (Internet IOP) and Java RMI can run on top of IIOP (RMI/IIOP) or with Java Remote Method Protocol (JRMP). To use this facility, the developer must specify the interface for each class in an Interface Definition Language (IDL). IDL defines the interface of the class in terms of its public attributes and operations. Figure 18.19 shows a sample IDL file for the Location class. The IDL file is then processed by a program that converts the interface to a series of files in the target language or languages. In the case of Java, this program is called IDL2JAVA and produces several files. These files include:

- a file that defines the interface in Java
- a stub file that provides the link between the client program and the ORB
- a file that provides a skeleton for the implementation on the server.

There are a number of other files that may be generated, but these are the important ones. The stub file implements the interface on the client and is compiled into the client program. The skeleton file also implements the interface (but on the host) and is amended by the developer to provide the implementation and compiled on the host.

```
module Agate
{
    interface Location
    {
        attribute string locationCode;
        attribute string locationName;
        void addIntCampaign( in IntCampaign campaign );
        void removeIntCampaign( in string campaignCode );
        int numberOfCampaigns( );
    };
}
```

Figure 18.19 CORBA IDL for Location class.

Systems developed using CORBA can be set up so that the remote objects are located on a named machine and accessed by name. However, CORBA also provides services for locating objects by name when it is not known where they are running. It also includes sophisticated services for locating objects that implement a certain interface and for interrogating an object to determine its interface (operations, parameter types and return types) in order to dynamically invoke its operations. However, for the vast majority of applications, these services are not required.

CORBA is known as *middleware*, as it acts as an intermediary between clients and servers. As such it enables the implementation of a three- or four-tier architecture that isolates the user interface and client programs from the implementation of classes on one or more servers. CORBA also provides interoperability between different languages: a Java client program can invoke operations on a C++ or Smalltalk object that exists on a separate machine. CORBA also makes it possible to encapsulate pre-existing programs written in non-object-oriented languages by wrapping them in an interface. To the client it looks like an object, but internally it may be implemented in a language such as COBOL.

18.8 Designing Data Management Classes

18.8.1 The layered architecture

In designing the boundary classes, one of our aims has been to keep them separate from the business logic and the entity objects.

In designing the presentation layer one of our aims was to isolate the entity classes in the system from the way that they are presented on screen and in reports and documents. We did this in order to maximize the reusability of the classes. Our aim is to do the same with the data storage layer. Classes are less reusable if they are tightly coupled to the mechanisms by which instances are stored in some kind of file system or database. We also want to decouple the entity classes from the business logic.

What are the options for locating the operations that handle the tasks of storing and retrieving objects?

1. We could add operations to each class to enable objects to save and store themselves.

- This reduces reusability. Each class must now contain code to implement the operations that couple the class to the particular data storage mechanism used.
 - This breaches the idea of cohesion. Each business class should contain operations that are relevant to the behaviour of instances of that class. Data storage methods belong in a data storage class.
 - If an object is not currently instantiated, how can we send it a message to invoke an operation to load itself?
2. We can get around this last problem by making the storage and retrieval operations class-scope methods rather than instance-scope methods (static methods in Java or static member functions in C++).
- This still suffers from the first two problems listed above for option 1. The class is less reusable and lacks cohesion.
3. All persistent objects in the system could inherit methods for storage from an abstract superclass—`PersistentObject` for example, rather than `Object` in Java.
- This has the effect of strongly coupling existing classes to the `PersistentObject` superclass, so all business classes end up inheriting from a utility class.
4. Where we have introduced collection classes into the design to manage collections of objects, we could make these collection classes responsible for storing and retrieving object instances.
- This is closer to a solution. The collection classes are design artefacts—not part of the business classes in the system. However, we may wish to reuse the design, and we are coupling it to the storage mechanisms.
5. We could introduce into the system separate classes, whose role is to deal with the storage and retrieval of other classes. This is the database broker approach.
- This solution fits the layered architecture. These classes are part of the data storage layer.
 - The data storage classes are decoupled from the business classes. The business classes will contain nothing that indicates how they are to be stored. The same business classes can be reused unchanged with different storage mechanisms.
6. We could limit the number of new data storage classes to one. Different instances of this class would be created with attributes to hold the names of tables or files that are to be used to store and retrieve instances of their associated classes.
- This parameterized version is more difficult to set up and more difficult to implement.
 - It requires some part of the system outside the database broker class to know what parameters to set for each instance that is created.

Option 5 is the approach that is favoured by most developers of object-oriented systems. It involves the use of a number of patterns. Larman (2005) describes it in some detail as a *persistence framework*, the main feature of which is the use of *database brokers* or *database mappers*, which mediate between the business

classes and the persistent storage and which are responsible for storing and retrieving objects. However, we shall first describe the use of option 3—inheritance from a persistent superclass—before looking at option 5.

18.8.2 PersistentObject superclass

A simple approach to the design of data storage classes is to design an abstract superclass `PersistentObject` that encapsulates the mechanisms for an object of any class to store itself in and retrieve itself from a database. Eriksson and Penker (1998) use this approach in order to keep their example case study application simple and independent of any vendor's DBMS. The `PersistentObject` superclass implements operations to get an object by object identifier, to store, delete and update objects and to iterate through a set of objects. These operations are implemented in terms of two abstract operations, to write and read objects, that must be implemented by each subclass that inherits from the `PersistentObject` superclass. This is shown in Fig. 18.20 (adapted from Eriksson and Penker, 1998). This approach also uses an aspect of option 2, as the `getObject()` operation is a

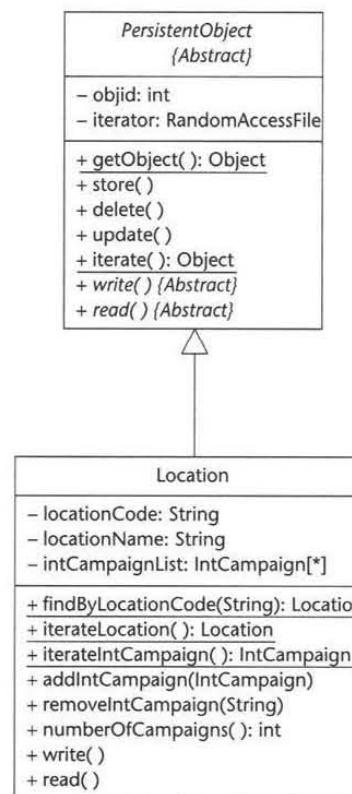


Figure 18.20 Excerpt from class diagram to show inheritance from `PersistentObject`.

class-scope method rather than an instance-scope method (as are the others underlined in the class diagram of Fig. 18.20).

The `PersistentObject` hides some of the detail of the implementation of the data storage from the business objects in the application. However, they must implement the `write()` and `read()` operations and this will limit their reusability.

This approach does have the benefit of limiting the changes that will be made to sequence diagrams. Messages that have been shown being sent to object instances to select an instance or to iterate through a set of instances can be shown as being sent to the class rather than the instances. Figure 18.21 shows an example of this for the use case `Get number of campaigns for location`. We have shown the class `Location` using a constructor `Location()` to make the particular instance `:Location` available. While the object instance is being created in memory for this instance of the collaboration, strictly speaking it already exists as an object and is just being *materialized* from the database.

However, the use of a persistent superclass is unlikely to be robust enough for business applications and a more sophisticated approach, such as the database broker, must be used.

18.8.3 Database broker framework

The database broker framework separates the business objects from the data storage implementation. The classes that provide the data storage services will be held in a separate package.

Our objective here is to separate the data storage mechanisms completely from the business classes. For each business class that needs to be persistent, there will be an associated database broker class that provides the mechanisms to materialize objects from the database and dematerialize them back to the database. A simple form of this is shown in Fig. 18.22 for the `Location` class. The `Location Broker` is responsible for the storage and retrieval of `Location` object instances. In order to ensure that there is only ever one `LocationBroker` instance, we can use the Singleton pattern (see Chapter 15). This means that we use a class-scope operation, but only to obtain an instance of the `Location Broker` that can be used

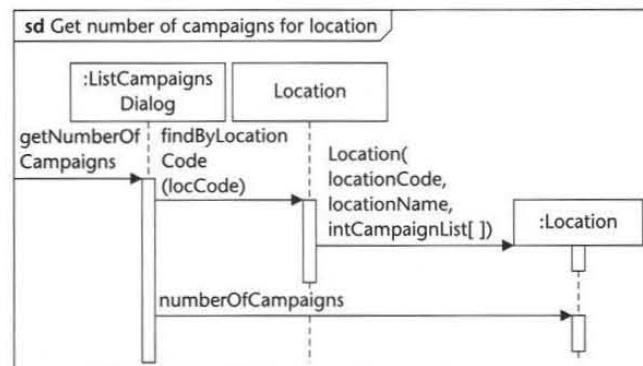


Figure 18.21 Sequence diagram for `Get number of campaigns for location` showing `Location` retrieving (or materializing) an instance.

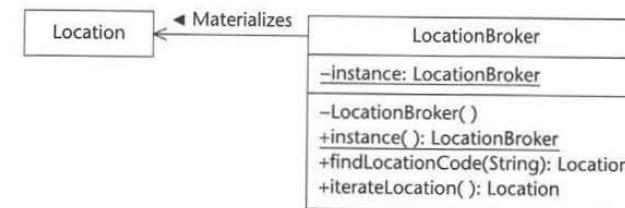


Figure 18.22 Location and `LocationBroker` classes.

subsequently to access the database. The sequence diagram involving the `LocationBroker` is very similar to that of Fig. 18.21, and is shown in Fig. 18.23. (Note that this diagram does not show the creation of the instance of `Location Broker`.)

Each persistent class in the system will require a broker class, so it makes sense to create a superclass that provides the services required by all these broker classes. Larman (2005) suggests two levels of generalization. At the top of his hierarchy is an abstract `Broker` class that provides the operation to materialize an object using its object identifier. This is then subclassed to produce different abstract classes of brokers for different kinds of storage, for example one for a relational database and one for a file system. Finally, the appropriate broker is subclassed into the concrete classes for each persistent class in the system. A simplified version of this inheritance hierarchy is shown in Fig. 18.24.

In the Agate case study, we could use JDBC to link the Java programs to a relational database. This will require the use of classes from the `java.sql` package, in particular `Connection`, which is used to make a connection to the database, `Statement`, which is used to execute SQL statements, and `ResultSet`, into which the results of SQL Select statements are placed (we can then iterate through the `ResultSet` retrieving each row in turn and extracting the values from each column). An appropriate driver will also be required. Figure 18.25 shows the

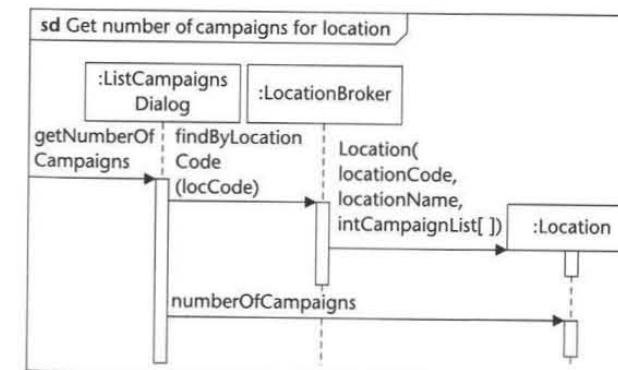


Figure 18.23 Sequence diagram for `Get number of campaigns for location` showing `LocationBroker` retrieving an instance of `Location`.

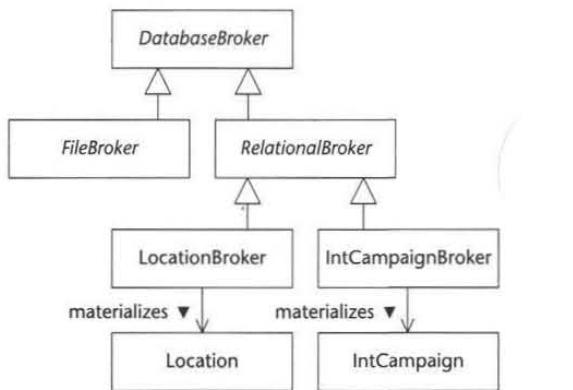


Figure 18.24 Simplified version of inheritance hierarchy for database brokers.

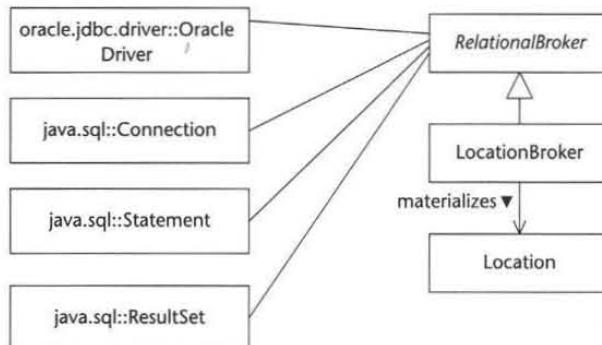


Figure 18.25 RelationalBroker class and classes from other packages.

associations between the `RelationalBroker` abstract class and these other classes. The figure shows the Oracle JDBC driver; to access a database such as Access via a link from JDBC to ODBC, the appropriate driver would be `sun.jdbc.odbc:JdbcOdbcDriver`.

The dependencies between the application classes and those in other packages can be shown using packages in a class diagram, as for the classes in the presentation layer in Chapter 17, Fig. 17.12. This is shown in Fig. 18.26.

This simple framework using database broker classes can be extended to deal with some of the problems that remain.

- The most important problem concerns the way in which persistent objects maintain references to other objects. If the `LocationBroker` retrieves an instance of `Location`, what happens when an operation of the `Location` requires it to send a message to one of its `IntCampaigns`? The `IntCampaign` will not necessarily have been retrieved from the database. The same applies to the many other operations that involve collaboration between objects.

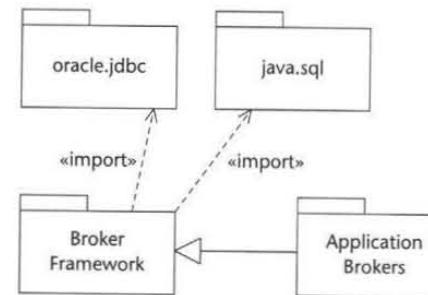


Figure 18.26 Class diagram showing packages for database brokers package.

- The second problem concerns the ability to manage transactions in which a number of objects are created, retrieved from the database, updated and deleted.

Two extensions to the database broker framework can be used to resolve these problems. The first uses the Proxy pattern to provide proxy objects for those objects that have not yet been retrieved from the database. The second uses caches to hold objects in memory and keep track of which have been created, updated or deleted.

The Proxy pattern (Gamma et al., 1995) provides a proxy object as a placeholder for another object until it is required. In this case, we can use proxies for each business class to link to where there is an association with another object or objects. If no message is sent to the associated objects, then the proxy does nothing. If a message is sent then the proxy asks the relevant database broker to retrieve the object from the database and, once it has been materialized, the proxy can pass the message directly to it. Subsequently, messages can be sent directly to the object by the proxy, or the proxy can replace the reference to itself in the object that sent the message with a reference to the real object. For this to work, the proxy must hold the object identifier of the object that it is a placeholder for. When the object itself is retrieved from the database, the object identifier is effectively transformed into a reference to the object itself. The proxy class must also implement the same interface as the real class so that it appears to other objects as if it is the real thing.

Caches can be combined with this approach. The database broker can maintain one or more caches of objects that have been retrieved from the database. Each cache can be implemented as a hashtable, using the object identifier as the key. Either a single cache is maintained and some mechanism is used to keep track of the state of each object, or six caches can be maintained:

- new clean cache—newly created objects
- new dirty cache—newly created objects that have been amended
- new deleted objects—newly created objects that have been deleted
- old clean cache—objects retrieved from the database
- old dirty cache—retrieved objects that have been amended
- old deleted objects—retrieved objects that have been deleted.

As objects are changed, the broker must be notified so that it can move them from one cache to the other. This can be achieved using the Observer–Observable

pattern: the object implements Observable, and the broker inherits from Observer.

When the transaction is complete, the broker can be notified. If the transaction is to be committed, the broker can process each object according to which cache it is in:

- new clean cache—write to the database
- new dirty cache—write to the database
- new deleted objects—delete from the cache
- old clean cache—delete from the cache
- old dirty cache—write to the database
- old deleted objects—delete from the database.

The cache or caches can be used by the proxy object to check whether an object is already available in memory. When it receives a message, the proxy can ask the broker for the object. If it is in a cache, the broker will return a reference to it directly; if it is not in the cache, the broker will retrieve it. Figure 18.27 shows the associations between the broker class, the caches and the proxy.

The collaboration between these classes can be seen in the communication diagram in Fig. 18.28, which represents the following interaction.

A Location is in memory with an IntCampaignProxy as a placeholder for the real IntCampaign object that runs in that Location. In order to print a list of campaigns, the title of the IntCampaign is required.

- 1.1 The Location sends the message getTitle to the IntCampaignProxy.
- 1.2 The IntCampaignProxy checks whether the IntCampaign is in memory.
- 1.3 It is not, so it then requests the object by object identifier from the broker, IntCampaignBroker.
- 1.4 The IntCampaignBroker checks if the object is in a Cache.

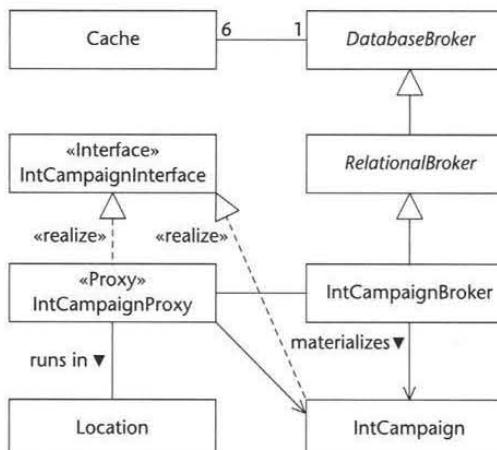


Figure 18.27 Extension of the database broker framework to include caches and proxies.

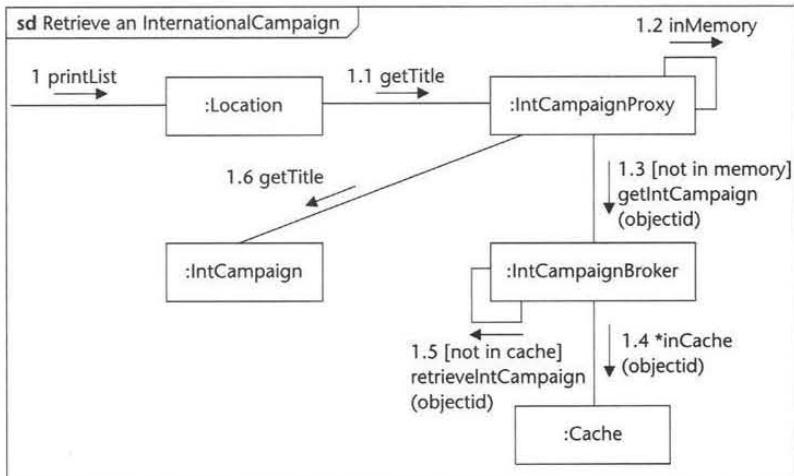


Figure 18.28 Communication diagram showing proxy, broker and cache objects collaborating to retrieve an object instance.

- 1.5 It is not, so the IntCampaignBroker retrieves the object from the database and returns it to the IntCampaignProxy.
- 1.6 The IntCampaignProxy sends the getTitle message to the Int Campaign object and returns the result to the Location.

This may appear to be overkill, but it enables us to maintain our objective of decoupling the business classes from the data storage mechanisms that are provided by the classes in the persistence framework. This should make it possible to migrate the data storage to a different platform without having an impact on the business classes or the application logic, simply by replacing the database broker class with the broker for a different database.

18.8.4 Using a data management product or framework

In the previous two sections, we have considered ways of designing a persistence mechanism yourself. However, this is not necessary, as there are a number of products and frameworks available that provide a persistence mechanism for you. We have included the previous section in order to give an idea of what is involved in developing a persistence framework. We shall consider two commercial products that will handle the mapping of objects to relational database tables, and the frameworks in the Java 2 Enterprise Edition (J2EE) that handle persistent objects.

Object-relational mappings

The standard for object-relational mapping in the Java world is Java Data Objects (JDO), which was mentioned in Section 18.6. A number of products are available that implement this standard. JDO is defined in the JDO Specification (JDOEG, 2003), while Roos (2003) describes how JDO works in a more readable form.

- To produce persistence-capable classes using JDO involves the following steps.
- Write and compile the Java classes that are required to be persistence-capable.
 - Write an XML *persistence descriptor* file to describe various properties of these classes.
 - Run an *enhancer* programme that reads the persistence descriptor and processes the compiled Java classes to add operations that implement the interface `PersistenceCapable`.
 - Run a program that generates the SQL necessary to create the database tables to hold the persistent classes.
 - Create the database tables.
 - Write the application to use a JDO `PersistenceManagerFactory` to provide a `PersistenceManager` to persist and retrieve instances of the enhanced classes. (The `PersistenceManagerFactory` must be passed a set of properties that define the connection to the database that will be used.)

Different JDO implementations work in slightly different ways but all conform to the standard API. The persistence descriptor for a class can be very simple, essentially just defining the class as persistent, or can contain a number of elements that define properties of the way the class is persisted such as the primary key to be used.

JDO handles transactions, caching and queries using JDO Query Language (JDOQL). It can also be used to provide persistence for EJBs in a J2EE application server.

An alternative to JDO in the Java world is the Hibernate framework, which provides similar capabilities and is likely to be integrated into the EJB3 specification.

Application servers

In the last few years there has been a move towards the use of application servers. Java application servers provide an environment in which Enterprise Java Beans (EJBs) can run. EJB is part of the J2EE framework developed by Sun and other interested parties. J2EE provides a number of standards for building enterprise Java systems, primarily systems that will use the web to deliver services. Various software providers have produced commercial and open source implementations of J2EE.

Application servers provide a mechanism for managing distributed objects or components. In the Java world, these objects and components are typically EJBs. EJBs can be either session beans, components that encapsulate a piece of business logic, or entity beans, components that encapsulate one or more entity classes. Application servers provide a way for client applications to communicate with the business logic layer and for the business logic layer to communicate with entity objects. They can provide persistence services for the entity objects, storing them in a database and retrieving them when required. Some application servers come with their own database; others can be run in conjunction with a range of databases from other suppliers. As with CORBA, described in Section 18.7, there is work to be done in defining interfaces and writing classes that implement those interfaces, but it is likely to be significantly less work than developing your own persistence mechanism and, with the use of tools like JDO implementations or Hibernate, the amount of work can be reduced.

Using EJB and an application server also makes it possible to design systems in which the same entity objects support web-based and conventional client applications.

18.9 | Summary

The design of persistent storage for object-oriented systems is not straightforward. For simple systems, it is possible to use files to store the objects. However, commercial systems require a more robust and sophisticated approach so that objects can be shared between applications and users. Database management systems provide the facilities to build robust, commercial-strength information systems and offer a number of advantages. Object DBMS can be used and will have a less significant impact on the design of the classes in the system than if a relational DBMS is used. However, many organizations have an existing investment in a relational DBMS and it may be necessary to build the system to use this database. In this case it is necessary to design tables, either by normalizing object instances or by following a set of guidelines for mapping classes and associations to tables. To decouple the business objects from the persistent storage mechanism, a persistence framework can be designed that can be extended to handle the resolution of object identifiers into references to real objects and that can use caches to manage transactions involving multiple objects.

The design of the persistent data storage mechanisms should ideally be carried out in conjunction with the object design activities of Chapter 14. If an object DBMS is being used, it may provide mechanisms by which to implement collection classes to handle associations (as discussed in Section 14.5), and class diagrams and interaction diagrams should reflect this. On the other hand, if a relational DBMS is being used, then the broker classes will play the role of collection classes and provide operations such as `findFirst()`, `getNext()` and `findObject()`, shown for example in Fig. 14.10. If the object design is carried out before the data management design, then the collection classes that are produced as part of an implementation-independent design model can be converted to database broker classes at a later stage.

Benefits can be gained from using persistence frameworks and tools to develop the storage mechanisms in a system rather than designing your own.

Review Questions

- 18.1** Give one example each of a persistent and a transient object.
- 18.2** Explain the difference between different types of file organization and file access.
- 18.3** Of the different kinds of record type listed in Section 18.3.1 suggest which would be most appropriate for storing complex nested objects. Explain the reasons for your choice.
- 18.4** Outline the advantages and disadvantages of using a database management system over developing an application using files.
- 18.5** What is the key difference between a relational DBMS and an object DBMS?

- 18.6** List in your own words the three steps used in going from an unnormalized relation to a relation in third normal form.
- 18.7** What are the three ways of mapping the classes in an inheritance hierarchy to tables?
- 18.8** What is meant by OML and ODL?
- 18.9** What is the difference between ODBC and CORBA as ways of connecting to databases?
- 18.10** Explain what is meant by (i) a broker and (ii) a proxy.

Case Study Work, Exercises and Projects

- 18.A** Find out what you can about localization mechanisms in a programming language or environment such as Java or .NET. What use do they make of files?
- 18.B** Normalize the data in the Agate invoice in Fig. 6.1. (Remember to remove the calculated values first.)
- 18.C** Normalize the data in the FoodCo sales order entry screen of Fig. 16.3. (Remember to remove the calculated values first.)
- 18.D** Use the guidelines in Section 18.4.2 to decide on the tables necessary to hold the classes of Fig. 14.7.
- 18.E** Find information about a relational DBMS and an object DBMS and write a short report comparing the features they offer.
- 18.F** Extend the sequence diagram of Fig. 17.18 to show the use of a proxy class and database broker.
- 18.G** Redraw your answer to 18.F as a communication diagram.
- 18.H** Draw communication diagrams similar to Fig. 18.28 to show what happens (i) when the `IntCampaign` is already in memory, and (ii) when it is in one of the caches.

Further Reading

Codd's (1970) paper on relational databases was reprinted in the 25th Anniversary issue of the *Communications of the ACM*, which is more likely to be available in a library (1983, 26(1) pp. 64–69). This 25th Anniversary issue is well worth looking at for other papers by some of the greats of computer science.

Korth and Silberschatz (2001) provide a good overview of database theory. Howe (2001) explains normalization in detail.

Loomis (1995) deals with background to object databases and the functionality they offer, whereas Eaglestone and Ridley (1998) present the ODMG standard and provide a worked example case study using `O2`.

Roos (2003) explains Java Data Objects. The JDO 2.0 standard is being developed under the auspices of JSR 243 and information is available on the Sun Microsystems website (<http://java.sun.com/products/jdo/index.jsp>).

Java 2 Enterprise Edition (J2EE) provides a range of tools for developing scalable distributed systems using Java. In particular, Enterprise Java Bean (EJB) entity beans are used to provide persistent storage for objects, and can provide a mapping between classes and tables in a relational database. Alur et al. (2003) explain how to use a number of design patterns with J2EE to get the best from the framework it provides.

There are links to the web pages of the standards bodies—ANSI, ISO, ODMG and OMG—and to the web pages of various database providers in the website for this book.

CHAPTER

Agate Ltd Case Study—Design

Agate Ltd

A5.1 Introduction

In this chapter we show how part of the Analysis Model presented in Chapter A4 has been modified by the activities of design. The design activities have been concerned with finalizing the software architecture, designing the entity classes, their attributes, operations and associations, designing the boundary classes and the human-computer interaction, designing the mechanisms used for data storage, and designing the control classes. These activities have been explained in Chapters 12 to 18.

The following sections include:

- package diagrams to illustrate the overall software architecture
- class diagrams to illustrate the classes in the design model
- sequence diagrams to illustrate the interaction between instances of classes
- a state machine for the control of the user interface.

A5.2 Architecture

The architecture of the system (shown in Fig. A5.1) has been designed to use Java Remote Method Invocation (RMI) for communication between the client machines and the server¹. Control classes have been split into two layers. First, there are the control classes that reside on the client machines (in the package Agate Control Client) and manage the interaction between users and the

¹ To meet the non-functional requirements relating to the distribution of the system, we will need a more complex architecture than this. The eventual solution will probably involve Java 2 Enterprise Edition (J2EE) and Enterprise Java Beans (EJB), and will require the use of application server software. For now we are presenting a design that is not so dependent on an application server, the design for which is beyond the scope of this book.

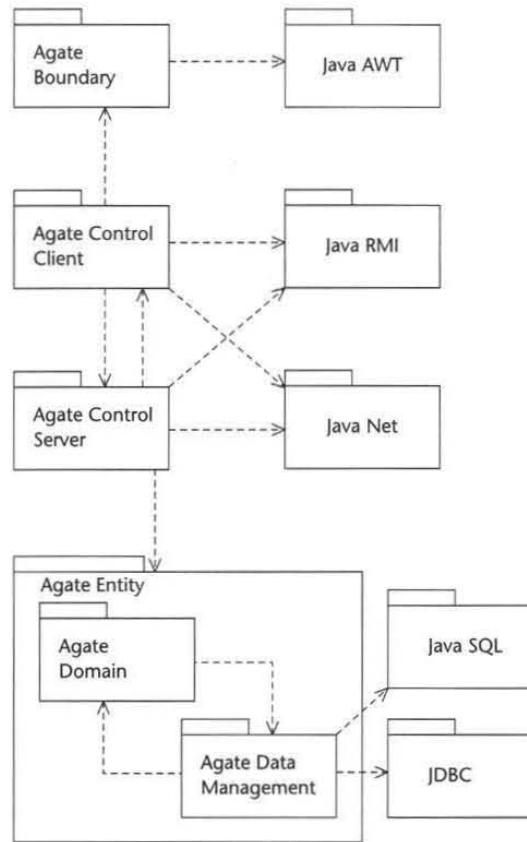


Figure A5.1 Package diagram for software architecture.

boundary classes. These control classes are essentially those that were designed in Chapter 17. Second, there are control classes that reside on the server. These control classes handle the interaction between the business logic of the application and the entity classes (and the associated data management classes). This helps to decouple the layers: the only communication between the clients and the server will be the communication between the client and server control classes, using RMI.

Not all control classes will have versions on both the clients and the server. For example, the `ListClients` and `ListCampaigns` classes in Figs 17.36 and 17.37 could just exist on the server, where they will have more immediate access to the entity and data management classes. One consequence of this will be visible in the sequence diagrams, where these two classes will no longer be passed references to the boundary class as a parameter, but will return their results to the control class on the client machine, which will set the values in the boundary class. This is shown in Figs A5.11 and A5.12.

On the server, we are using JDBC, and we will map the classes to relational database tables. A design based on the Broker pattern will be used to handle this.

A5.3 Sample Use Case

For the purpose of this case study chapter we are going to present the design of one use case, Check campaign budget, for which the boundary and control classes were designed in Chapter 17.

Figure A5.2 shows the design of the user interface for this use case. In the first iteration, we are not concerned with adding the extensions to the use case that handle printing of the campaign summary and campaign invoice.

A5.4 Class Diagrams

The packages on the architecture diagram have been named in a way that will allow us to use the Java package notation for classes. So, for example, the boundary classes will be in the package `Agate::Boundary`. This is the first package that we are illustrating here, and the classes we are concerned with are shown in Fig. A5.3.

The boundary class `CheckCampaignBudgetUI` will implement the two interfaces `CampaignLister` and `ClientLister`. Note that some of the operations that were included in the class `CheckCampaignBudgetUI`, such as `getSelectedClient()`, have been moved into the interfaces, as it is thought that they will apply to any class that implements these interfaces.

Because the control class `CheckCampaignBudget` will now be split, the version that resides on the client machines (now called `CheckCampaignBudgetClient`) must be able to respond to the messages `addCampaignName()` and `addClientname()`. We have used interfaces for this, because they have to be sent messages remotely by the control classes on the server. This is shown in Fig. A5.4. Note also that this class will need to hold a reference to the version of itself that exists on the server. We have not shown the full package name in the

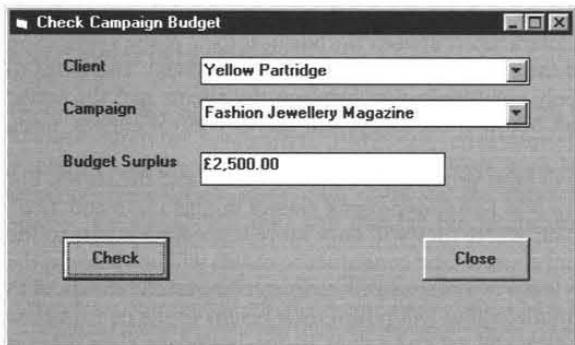


Figure A5.2 Prototype user interface for Check campaign budget.

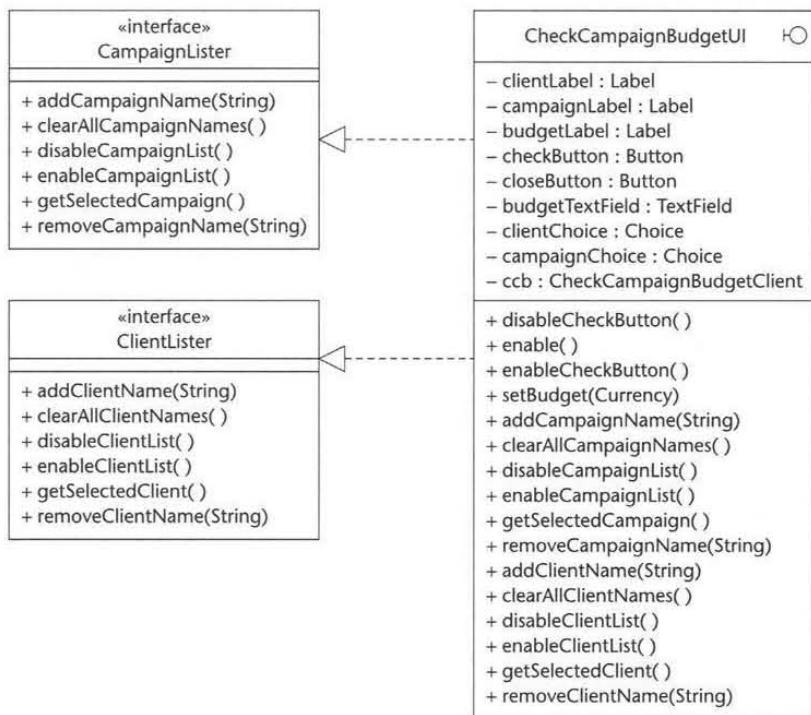


Figure A5.3 Relevant classes in the package `Agate::Boundary`.

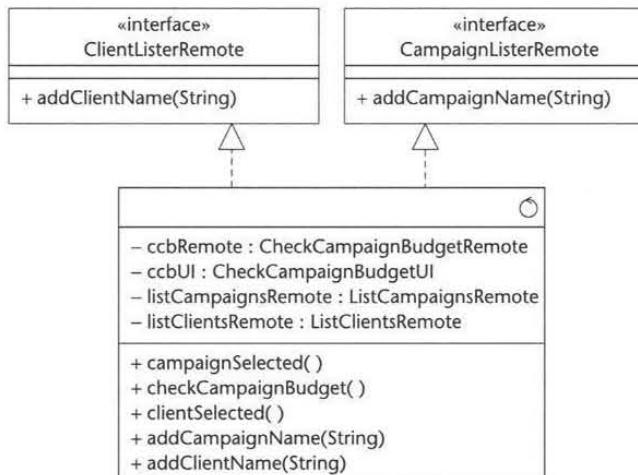


Figure A5.4 The class `Agate::Control::Client::CheckCampaignBudgetClient`.

class diagram, but the attribute `ccbRemote` will in fact be an instance of `Agate::Control::Server::CheckCampaignBudgetRemote`. In fact, there will be an instance of `Agate::Control::Server::CheckCampaignBudget` Server on the server, and for the object on the client to communicate with it via RMI it will have to implement the interface `Agate::Control::Server::CheckCampaignBudgetRemote`. If `ListCampaigns` and `ListClients` only exist on the server, then they will also be in the same package and will implement the interfaces `ListCampaignsRemote` and `ListClientsRemote`.

All the classes that communicate via RMI will need to inherit from the Java RMI package. Rather than being subclasses of the default Java class `Object`, they will need to be subclasses of `java.rmi.server.UnicastRemoteObject`.

In Fig. A5.5 we have shown the control classes that reside on the server and the remote interfaces that they must implement. Although we have not shown the full package names, the references to `ClientListerRemote` and `CampaignListerRemote` are to the interfaces in the package `Agate::Control::Client`, shown in Fig. A5.4.

The entity classes that collaborate in this use case are `Client`, `Campaign` and `Advert`. They are shown in a first draft design in Fig. A5.6. However, this design will only work for the kind of application where all the objects are in memory. We

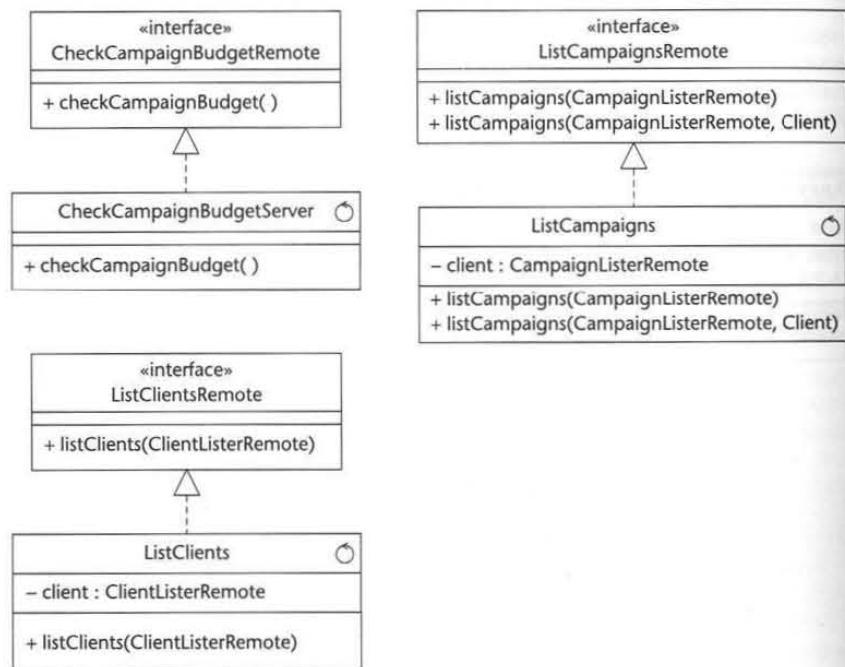


Figure A5.5 Relevant classes in the package `Agate::Control::Server`.

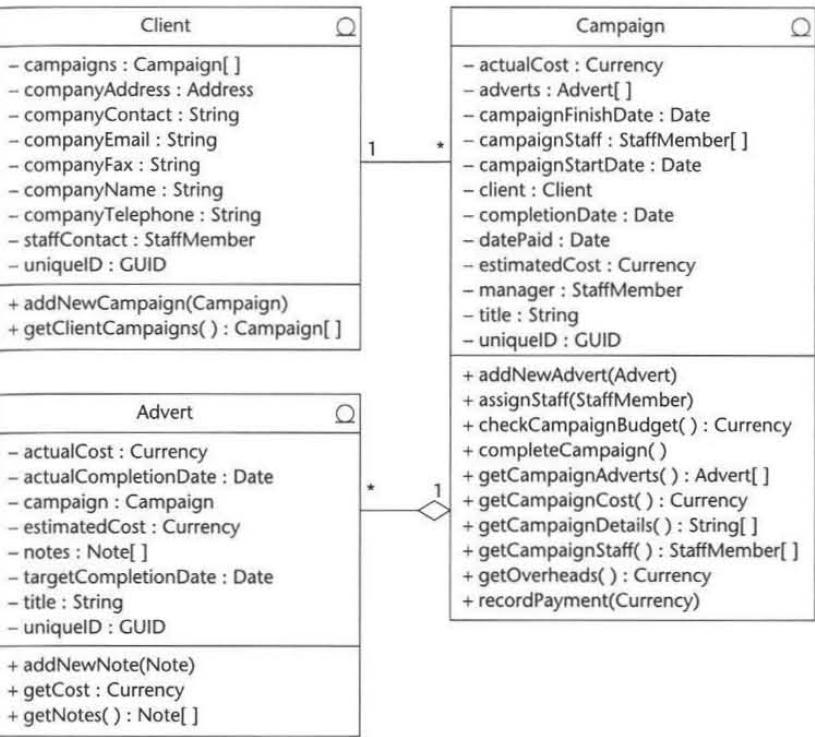


Figure A5.6 First draft design of some classes in the package `Agate::Entity::Domain`.

need to be able to deal with the process of materializing instances of these classes from the database and, when required, materializing their links with other object instances or collections of object instances. For example, when a particular `Client` is materialized, we do not necessarily want to re-establish its links with all its `Campaigns` and the instance of `StaffMember` that is its `staffContact`. The Broker pattern, which we discussed in Chapter 18, is a way of making it possible to materialize the objects that are linked to other objects only when they are required.

In order to achieve this, we can replace the references to the arrays of linked objects with references to the various subclasses of Broker, for example `ClientBroker`, `CampaignBroker` and `AdvertBroker`. Since these are still private attributes, they cannot be referred to directly by other objects and their values can only be obtained by calling one of the operations of the object in which they are contained. The result of this is shown in Fig. A5.7.

The broker subclasses could use the Singleton pattern (see Chapter 15). If this is done, then the design of the operations to return sets of whatever objects they are acting as brokers for will have to be carefully designed to handle concurrent requests from different clients. Alternatively, there could be multiple instances of brokers, and they could be created and destroyed as required, or there could be a pool of brokers available in the server, and when an object needs a broker of a

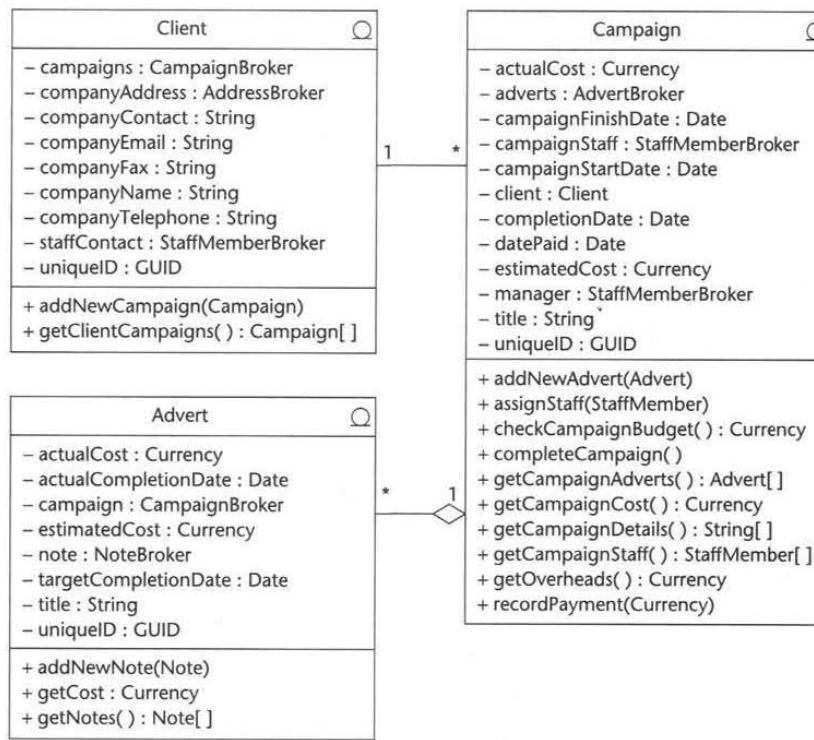


Figure A5.7 Second design of some classes in the package Agate::Entity::Domain.

certain type, it would request one from the pool. Figure A5.8 shows the brokers that we are interested in for this use case. We have not used the Singleton pattern in this design.

These brokers will also be used directly by the control classes, for example, when they need to obtain a list of all the objects of a particular class in the system. The brokers have been shown with attributes in which to hold references to the objects necessary for connecting to the database and issuing queries. We have also assumed that having obtained a list of results, a broker may store it internally in a collection class and allow client objects to iterate through the list of results using an enumerator².

The brokers will be in the package Agate::Entity::DataManagement, together with any other necessary classes to handle the connection to the database. (In this design we are not using proxies or caches, in order to keep it relatively simple.)

The final piece of design necessary to enable the interaction of this use case realization to take place concerns how the control objects on the client machine will obtain references to control objects on the server. For this, we shall use the

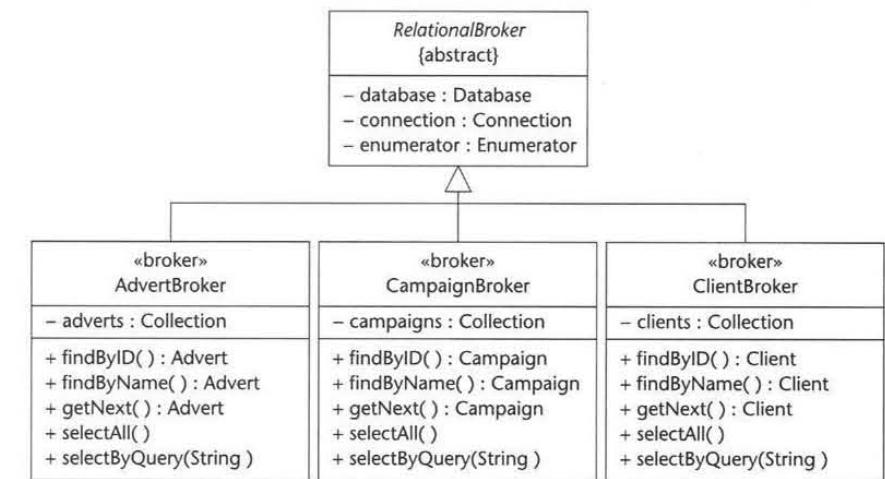


Figure A5.8 Broker classes in the package Agate::Entity::DataManagement.

Factory pattern. A Factory class creates instances of other classes and returns a reference to the new instance to the object that requested it. This is shown in Fig. A5.9.

So an instance of the control class CheckCampaignBudgetClient on the client machine will request a Factory on the server to provide it with a reference to an instance of CheckCampaignBudgetServer. The Factory will create this instance and pass back a reference to it via the RMI connection with the client. From that point onwards, the client object can make direct requests to the control object on the server. When it is finished with it, it can destroy it, or ask the Factory to destroy it.

In a more sophisticated design, the Factory could hold a pool of already instantiated control classes ready for use. When a client requests an instance of a particular control class, the Factory will take one from the pool if it is available. When the client is finished with the instance, the Factory can put it back into the pool. We are not using pooling in this design, but it is an approach that is commonly used to improve the performance of servers to prevent delays while instances are created and destroyed on demand.

In this design the control class on the server has only one method. This class, Agate::Control::Server::CheckCampaignBudgetServer, could be designed

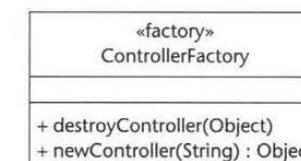


Figure A5.9 Factory class in the package Agate::Control::Server.

2 A mechanism for working through a collection dealing with each object in turn.

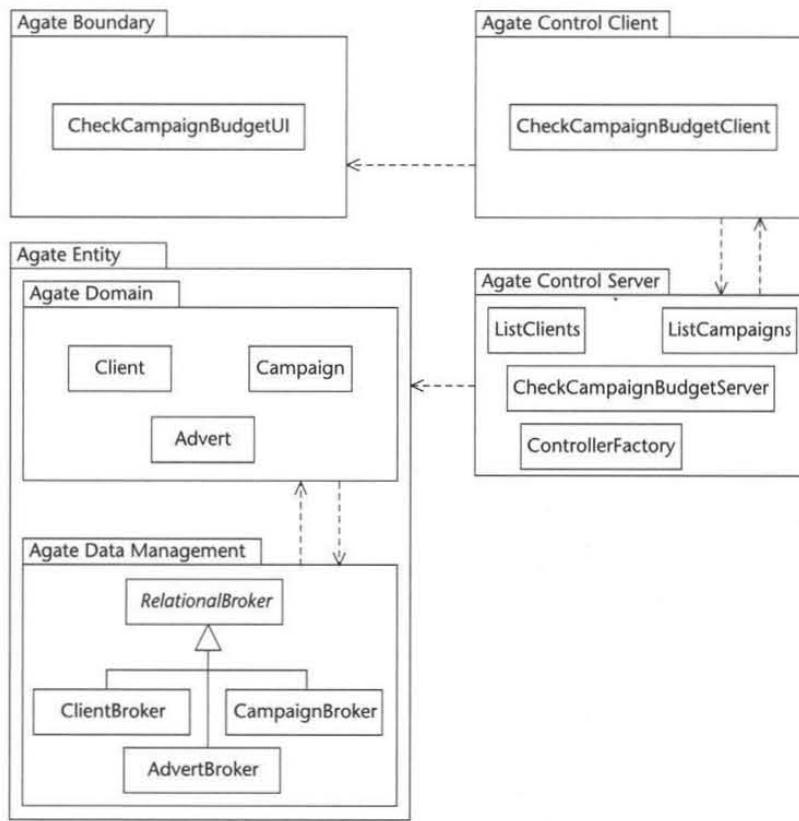


Figure A5.10 Package diagram showing classes.

to hold the business logic for checking the budget of a campaign, but we have taken the decision to leave the responsibility for calculating whether or not the budget is overspent in the Campaign class. There is a case for giving this responsibility to the control class; then, if the business logic changes, it only has to be updated in the control class. However, this makes the entity objects little more than data stores.

Figure A5.10 shows the package diagram with the classes (but not the interfaces) from Figs A5.3 to A5.9. Note that we have not used value objects, which we discussed in Chapter 12 and showed in Fig. 12.24

A5.5 Sequence Diagrams

Figures A5.11 to A5.13 show the sequence diagrams from Chapter 17 revised to take account of the splitting of the control objects and the addition of the Factory class. The package names of objects are also shown.

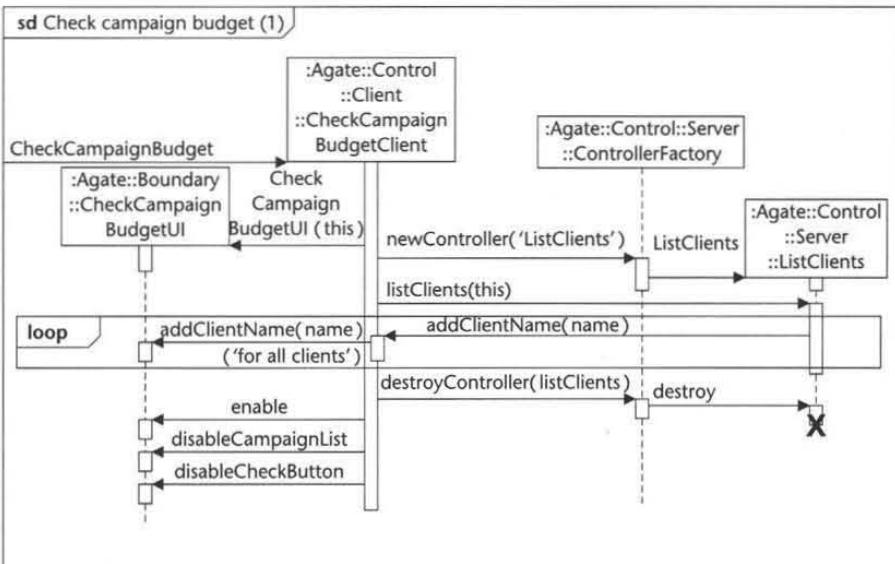


Figure A5.11 First sequence diagram for Check campaign budget.

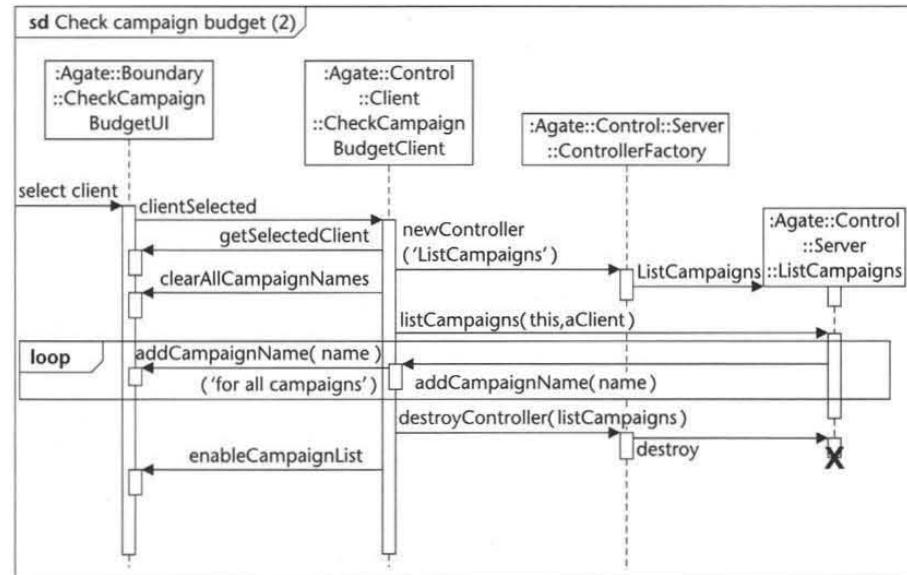


Figure A5.12 Second sequence diagram for Check campaign budget.

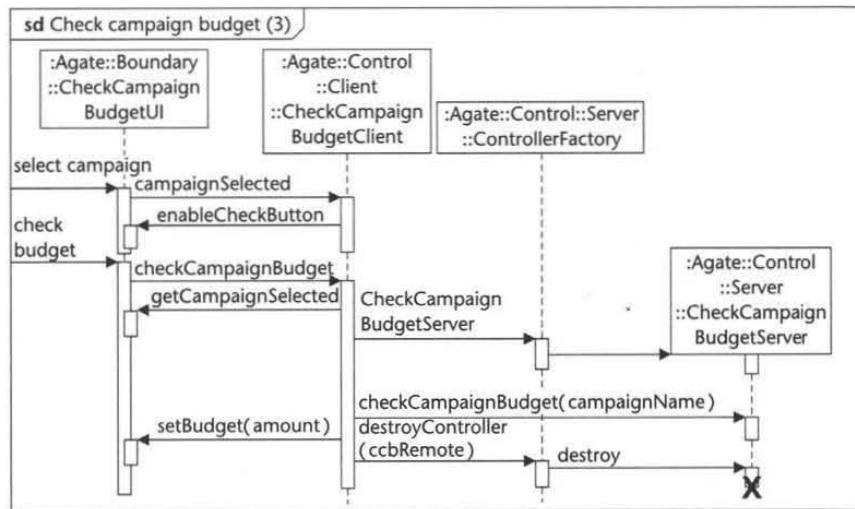


Figure A5.13 Third sequence diagram for Check campaign budget.

Although we show the control class on the client as able to directly connect to the instance of `ControllerFactory` on the server, in reality it would have to request a reference to this object from a naming service or registry on the server, for example a running instance of the Java `rmiregistry`.

In Fig. A5.14 we show the interaction between the control class, the brokers and the entity classes. Note how the broker classes perform the tasks involved in retrieving instances or sets of instances from the database.

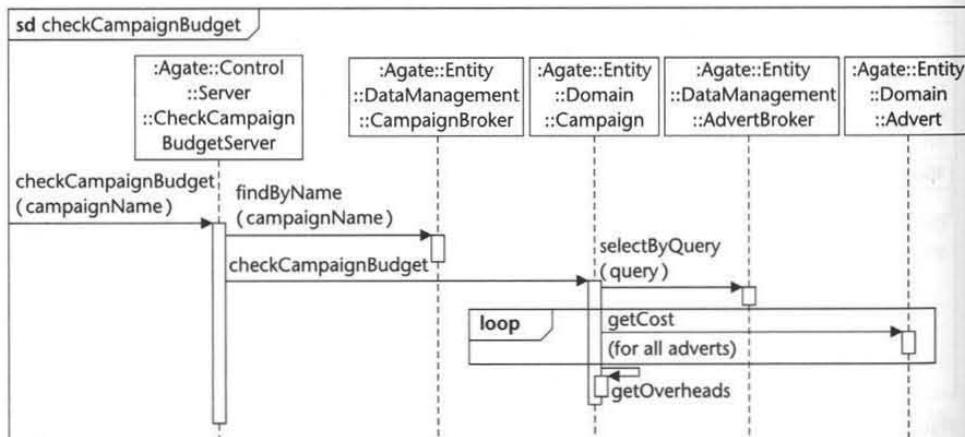


Figure A5.14 Sequence diagram for the operation `checkCampaignBudget()`.

We have used a simple approach for obtaining the adverts linked to a particular campaign, by having the broker return an array of `Adverts`. As mentioned above, this could return an enumerator so that the control class could iterate through the collection of `Adverts`.

A string named `query` has been passed to the `selectByQuery()` operation of the `AdvertBroker`. The exact format of this will depend on how the object-relational database mapping is set up. If the `uniqueID` attributes are used in the database as foreign keys, then the SQL statement will be something like:

```
SELECT * FROM adverts WHERE adverts.campaignID = '123456789';
```

and the ID of the particular campaign is added into the query string before it is passed to the broker.

A5.6 Database Design

Figure A5.15 shows the SQL to create the tables to map to the classes in Fig. A5.7. The indexes are required to ensure that it is possible quickly to retrieve all the campaigns linked to a client or all the adverts linked to a campaign. A character field has been used to hold the unique ID for each object. We are assuming that some mechanism will be used to generate these, but have not detailed it here. An alternative would be to use long integer values.

A5.7 State Machines

Figure A5.16 shows the event-action table for the state machine of Fig. A5.17. This state machine is the same as the one shown in Chapter 17.

A5.8 Activities of Design

The activities in the design workflow are shown in the activity diagrams of Figs A5.18 and A5.19.

In order to keep the diagram simple, we have shown the flow of activities in Fig. A5.19 without dependencies on the products that are used and created. Although we have shown a flow through the activities from top to bottom, there will inevitably be some iteration through this workflow even within a major iteration.

```

CREATE TABLE Clients
  (VARCHAR(30) uniqueID PRIMARY KEY NOT NULL,
  VARCHAR(30) companyAddress,
  VARCHAR(40) companyContact,
  VARCHAR(30) companyEmail
  VARCHAR(30) companyFax,
  VARCHAR(50) companyName NOT NULL,
  VARCHAR(30) companyTelephone,
  VARCHAR(30) staffContactID);
CREATE INDEX client_idx ON Clients (staffContactID, companyName);
CREATE TABLE Campaigns
  (VARCHAR(30) uniqueID PRIMARY KEY NOT NULL,
  FLOAT actualCost,
  DATE campaignFinishDate,
  DATE campaignStartDate,
  VARCHAR(30) clientID NOT NULL,
  DATE completionDate,
  DATE datePaid,
  FLOAT estimatedCost,
  VARCHAR(30) managerID,
  VARCHAR(50) title);
CREATE INDEX campaign_idx ON Campaigns (clientID, managerID, title);
CREATE TABLE Adverts
  (VARCHAR(30) uniqueID PRIMARY KEY NOT NULL,
  FLOAT actualCost,
  DATE actualCompletionDate,
  VARCHAR(30) campaignID NOT NULL,
  FLOAT estimatedCost,
  DATE targetCompletionDate,
  VARCHAR(50) title);
CREATE INDEX advert_idx ON Adverts (campaignID, title);

```

Figure A5.15 SQL to create tables for the classes Client, Campaign and Advert.

Current State	Event	Action	Next State
-	Check Campaign Budget menu item selected	Display CheckCampaignBudgetUI. Load client dropdown. Disable campaign dropdown. Disable check button. Enable window	1
1	Client selected	Clear campaign dropdown. Load campaign dropdown. Enable campaign dropdown	2
2, 3, 4	Client selected	Clear campaign dropdown. Load campaign dropdown. Clear budget textfield. Disable check button	2
2	Campaign selected	Clear budget textfield. Enable check button	3
3	Check button clicked	Calculate budget. Display result	4
3, 4	Campaign selected	Clear budget textfield	3
4	Check button clicked	Calculate budget. Display result	4
1, 2, 3, 4	Close button clicked	Display alert dialogue	5
5	OK button clicked	Close alert dialogue. Close window	-
5	Cancel button clicked	Close alert dialogue	H*

Figure A5.16 Event-action table for Fig. A5.17.

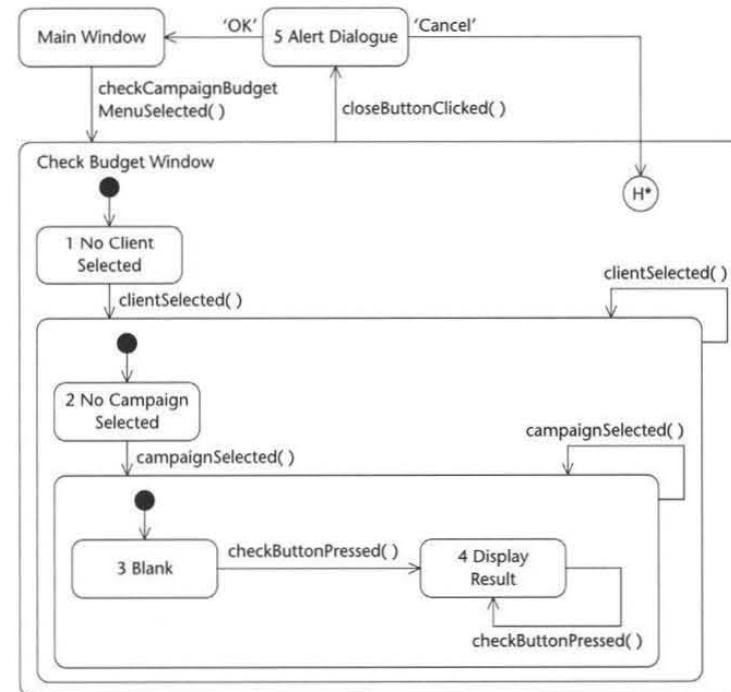


Figure A5.17 State machine for control of the user interface in Check campaign budget.

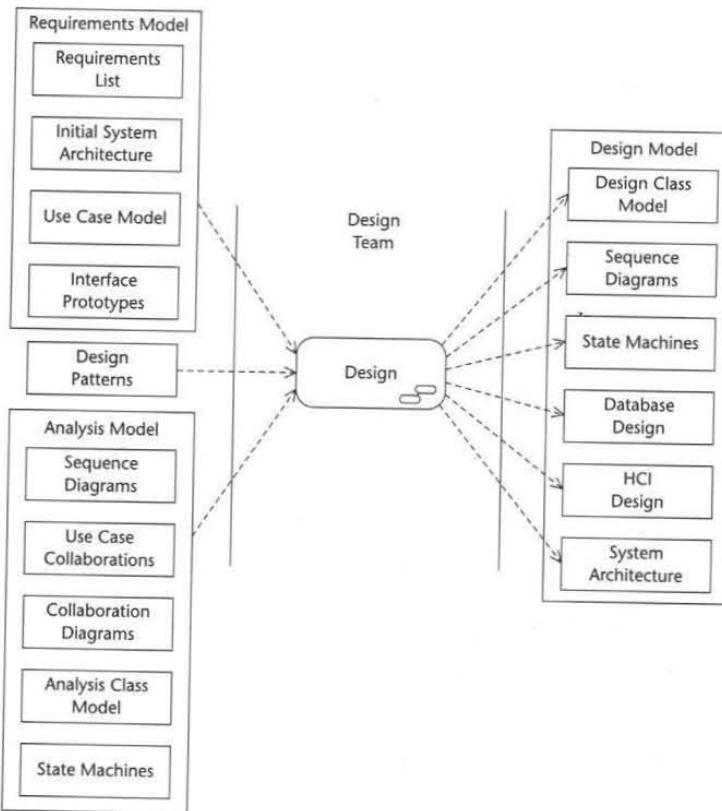


Figure A5.18 Activity diagram for the design workflow.

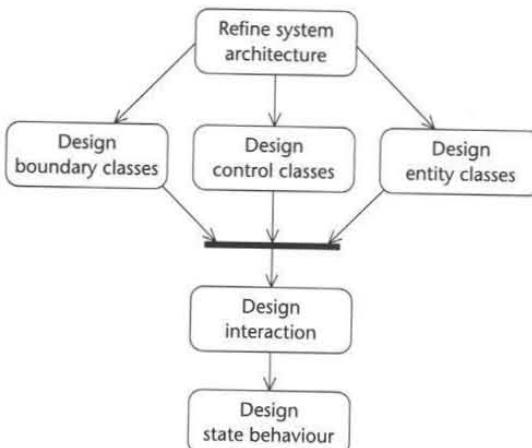


Figure A5.19 Detailed activity diagram for the design workflow.

Implementation

Learning Objectives

In this chapter you will learn

- how to draw component diagrams
- how to draw deployment diagrams
- the tasks that must be carried out in testing a new system
- how to plan for data to be converted from an existing system
- the alternative ways of introducing a new system into an organization
- the tasks that are undertaken as part of system maintenance and review.

19.1 Introduction

Implementation might be considered outside the scope of analysis and design. However, in projects that use rapid application development techniques, the distinction between different roles tends to break down (see Chapter 22 on the book's website). Analysts in particular may have a role during implementation in dealing with system testing, data conversion and user training. In other organizations staff may be employed in specialist roles to carry out these tasks. In this chapter we address some of the issues concerned with the implementation of systems and the role of analysts and designers in this process. We focus on the various UML diagrams that can be used to plan and document the implementation of the software. We also cover more general aspects of system implementation that are independent of the development approach that is taken.

A range of different software packages are required for implementation, for example, languages and database management systems. It is important to keep track of the relationship between different elements of the system such as source code files, object code files and software libraries. It is also important to maintain standards in the software development process: classes, objects and variables should be named in ways that make their purpose clear to others and make it

possible to trace from analysis through design to code; programs should be self-documenting and well-structured. (This is all covered in Section 19.2.)

UML provides two diagrams that can be used to document the implementation of a system. *Component diagrams* are used to document dependencies between the different elements of the system (Section 19.3). They can also be combined with *deployment diagrams* (Section 19.4) to show how the software components relate to the physical architecture of the system. For a large system these diagrams may be an unwieldy way of documenting the implementation and it may be simpler to use tables of information using a spreadsheet.

The testing of a new system is an essential part of implementation and includes testing of individual components, subsystems and the complete system (Section 19.5). A major task when a new system is introduced is to take data from an existing system or systems and transfer it into the new system (Section 19.6). The existing data may be held on paper or in a computerized information system that is being replaced. Temporary staff may have to be employed during the changeover to the new system. They and the existing staff will require training in how to use the system and user documentation will have to be produced (Section 19.7). There are four different strategies for the introduction of a new system into an organization. These different approaches are appropriate to different circumstances, and each has its own advantages and disadvantages (Section 19.8). Finally, we address the maintenance issues that arise after a system has been implemented (Section 19.9).

19.2 Software Implementation

19.2.1 Software tools

The implementation of a system will require a range of tools. Ensuring that these are available in compatible versions and with sufficient licences for the number of developers who will be using them is part of the project management role. The Rational Unified Process (Kruchten, 2004) adds an Environment Discipline to those of the generic Unified Process. The purpose of the Environment Discipline is to support the development organization with appropriate processes and tools. Many such tools have been designed and developed to make the work of the system developer easier. In this section we describe each in turn.

Modelling tools

Modelling tools (sometimes known as computer-aided software engineering (CASE) tools) allow the analysts and designers to produce the diagrams that make up their models of the system. Modelling tools were discussed in detail in Section 3.6. There are now many modelling tools that support UML notation. If they have been implemented to use the UML XML Metadata Interchange format (XMI), it should be possible to exchange models between different vendors' tools. The repository for the project should also be maintained using the modelling tool to link the textual and structured descriptions of every class, attribute, operation, state and so on to its diagrammatic representation.

To ensure that the implementation accurately reflects the design diagrams, it may be possible to generate code in one or more programming languages from the

models in the modelling tool. Modelling tools exist that generate code for languages such as Visual Basic, C++, C# and Java. Some support the generation of SQL statements to create relational database tables to implement data storage, and the generation of EJBs and XML schemas. Some modelling tools provide support for reverse engineering from existing code to design models. When this is combined with code generation it is known as round-trip engineering.

Compilers, interpreters, debuggers and run-time support

Whatever the language being used, some kind of compiler or interpreter will be required to translate the source code into executable code. C++ must be compiled into object code that can be run on the target machine. Smalltalk is interpreted, each command is translated as the program executes. Java is compiled into an intermediate bytecode format and requires a run-time program to enable it to execute. For applets, this run-time program is provided in the web browser, otherwise it is provided by the program called simply java or java.exe. C# can be compiled into bytecode in Microsoft Intermediate Language (MSIL) format for .NET applications. Developers also require the use of debugging tools that allow them to analyse problems in the code by stopping execution at breakpoints and examining the contents of variables.

Visual editors

Graphical user interfaces can be extremely difficult to program manually. Since the advent of Visual Basic, visual development environments have been produced for a wide range of languages. These enable the programmer to develop a user interface by dragging and dropping visual components onto forms and setting the parameters that control their appearance in a properties window. All the user interface examples in Chapters 6, 16 and 17 were produced in this way.

Integrated development environment

Large projects involve many files containing source code and other information such as the resource files for prompts in different human languages discussed in Chapter 18. Keeping track of all these files and the dependencies between them, and recompiling all those that have changed as a project is being built is a task best performed by software designed for that purpose. Integrated development environments (IDEs) incorporate a multi-window editor, mechanisms for managing the files that make up a project, links to the compiler so that code can be compiled from within the IDE and a debugger to help the programmer step through the code to find errors. An IDE may also include a visual editor to help build the user interface and a version control system to keep track of different versions of the software. Some also include modelling tools that either allow the developer to model in UML or provide UML visualizations of the code.

Configuration management

Configuration management tools keep track of the dependencies between components and the versions of source code and resource files that are used to produce a particular release of a software package. Each time a file is to be changed, it must be checked out of a repository. When it has been changed it is checked in again as a new version. The tool keeps track of the versions and the changes from one

version to the next. When a software release is built, the tool keeps track of the versions of all the files that were used in the build. To ensure that an identical version can be rebuilt, other tools such as compilers and linkers should also be under version control.

Some such tools are simple and easily available such as CVS and RCS, while others are for large-scale distributed projects and require full-time administrators. Web interfaces are available for some, and these make it possible to check items in and out over the Internet for work on Open Source software or distributed development.

There are standard protocols for version control software, which make it possible for users of editors, IDEs and CASE tools to check items out and in from within the tool.

Class browsers

In an object-oriented system, a browser provides a visual way of navigating the class hierarchy of the application and the supporting classes to find their attributes and operations. Smalltalk-80 was the first language to provide this kind of browsing capability. Some IDEs now provide it. The Java Application Programming Interface (API) is documented in HTML and can be browsed with a web browser.

Component managers

Chapter 20 discusses how software reuse can be achieved through the development of reusable components. Component managers provide the user with the ability to search for suitable components, to browse them and to maintain different versions of components.

DBMS

A large-scale database management system will consist of a considerable amount of software. If it supports a client-server mode of operation, there will be separate client and server components as well as all the tools discussed in Section 18.2.6. To use ODBC or JDBC will require software installed on the client. For any database, special class libraries or Java packages may be required on the client either during compilation or at run-time or both. ObjectStore PSE Pro includes a post-processor that is used to process Java class files to make them persistent. Using object-relational mapping tools such as those based on JDO (see Chapter 18) also requires the use of post-processors that change compiled classes.

CORBA

An ORB is required in order to use CORBA. It will include the IDL compiler that takes interface definitions in IDL and produces the interface, stub and skeleton files necessary to use CORBA.

Application containers

With the growth of web-based applications, much software now runs in *containers* of some sort. For web applications that are not part of larger enterprise applications, this may be a web container such as Tomcat, while large-scale enterprise applications are now often developed to run in application servers such as IBM's WebSphere or BEA's WebLogic. These provide a framework within which classes can be implemented as EJBs and deliver the business logic of the system.

Testing tools

Automated testing tools are available for some environments. What is more likely is that programmers will develop their own tools to provide harnesses within which to test classes and subsystems according to company standards. Section 19.3 covers testing in more detail.

Installation tools

Anyone who has installed commercial software on a Windows PC or a Mac or used a package manager on Linux will have experienced one of these tools, which automate the creation of directories, the extraction of files from archives and the setting up of parameters or registry entries. To do this they maintain the kind of information that can be modelled using component and deployment diagrams (see Sections 19.3 and 19.4). In our experience, uninstallation tools do not work as well!

Conversion tools

In most cases data for the new system will have to be transferred from an existing system. Whereas once the existing system was usually a manual system, most projects nowadays replace an existing computerized system, and data will have to be extracted from files or a database in the existing system and reformatted so that it can be used to set up the database for the new system. There are packages that provide automated tools to extract data from a wide range of systems and format it for a new system.

Documentation generators

In the same way that code can be generated from the diagrams and documents in a CASE tool, it may be possible to generate technical and user documentation. In Windows there are packages that can be used to produce files in Windows Help format. Java includes a program called javadoc that processes Java source files and builds HTML documentation in the style of the API documentation from special comments with embedded tags in the source code. C# uses special XML tags embedded in comments to generate documentation.

19.2.2 Coding and documentation standards

Even one person developing software on his or her own is likely to find at some point that they cannot remember the purpose of a class, an attribute or an operation in a program. On any project in which people collaborate to develop software, agreed standards for the naming of classes, attributes, operations and other elements of the system are essential if the project is not to descend into chaos. (See also Chapter 5 and Section 13.8.)

Naming standards should have been agreed before the analysis began. In this book we have tried to conform to a typical object-oriented standard.

- Classes are named with an initial capital letter. Words are concatenated together when the class name is longer than one word. Capital letters within the name show where these words have been joined together. For example *SalesOrder Proxy*.

- Attributes are named with an initial lower case letter. The same approach is taken as for classes by concatenating words together. For example `customer OrderRef`.
- Operations are named in the same way as attributes. For example `getOrder Total()`.

There are other standards. In C++ one convention is to use *Hungarian* notation: all member variable (attribute) names are prefixed by an abbreviation that indicates the type of the member variable, for example `b` for a Boolean, `i` for an integer, `f` for a float, `btn` for a button and `hwnd` for a handle to a window object. This can be particularly useful in languages that are not strongly typed, like Smalltalk, as it helps to enforce the consistent use of the same variable for the same purpose by different developers.

Consistent naming standards also make it easier to trace requirements through from analysis through design to implementation. This is particularly important for class, attribute and operation names.

Not everything in a program can be deduced by reading the names of classes, attributes and operations. Beveridge (1996) in a book on Java programming gives five reasons for documenting code.

- Think of the next person. Someone else may be maintaining the code you have written.
- Your code can be an educational tool. Good code can help others, but without comments complicated code can be difficult to understand.
- No language is self-documenting. However good your naming conventions, you can always provide extra help to someone reading your code.
- You can comply with the Java coding standards. Your documentation will be in the same hypertext format as the Java API documentation.
- You can automate its production. The javadoc program is discussed below. It generates HTML from your comments.

(The comments about Java apply just as well to other languages.) Standards should be enforced for the way that comments are added to a program. This should include a block at the start of each class source file (or header file in C++) that describes the purpose of the class and includes details of its author and the date on which it was written. The amendment history of the source file can be included in this block. Every operation should begin with a comment that describes its purpose. Any obscure aspect of the code should be documented with a comment. If you are developing in Java, you can use the javadoc conventions to generate HTML documentation for your classes. You can embed HTML tags and javadoc tags in the comments, and javadoc will also use its own special tags to add information about the author and version to the HTML. The javadoc tags include:

- `@see classname`—‘See also’ hypertext link to the specified class
- `@version text`—‘Version’ entry
- `@author name`—‘Author’ entry.

C# uses XML tags embedded in comments in the code to generate documentation. The recommended tags include:

- `<see cref="classname" />`—Link to the specified class
- `<seealso cref="classname" />`—Cross-reference to a `<see>` link
- `<summary>text</summary>`—Summary description.

As well as technical documentation, there will be a need for user documentation that will be required for training users and for them to refer to once the system is in use. Standards need to be agreed with the users for the format of this documentation, which may be produced by analysts or by specialist technical authors.

19.3 Component Diagrams

We have already introduced component diagrams in Chapter 8. The notation and use of component diagrams has changed in UML 2.0. A distinction has been made between components and *artefacts*. Artefacts are new in UML 2.0 and are used in deployment diagrams (see Section 19.4). They are used to represent development artefacts that were previously represented by components. In UML 2.0 components are specifically used to represent modular software units with a well-defined interface. They can be logical or physical components, so component diagrams can be used to model either the abstract, logical view of the components in a system or subsystem or the actual physical components that are deployed. Cheesman and Daniels (2001) make a clear distinction between different forms of component.

- *Component Specification*—specification of a modular software unit in terms of its behaviour (interfaces)
- *Component Implementation*—deployable implementation of a component specification
- *Installed Component*—a copy of an implementation in a runtime environment
- *Component Object*—an instance of an installed component that actually carries out the specified behaviour.

They also define a *Component Interface* as the set of behaviours that can be offered by a component object.

Dependencies between components can be shown in a component diagram, as in Fig. 19.1. However, they are more commonly shown using the notation used in Fig. 8.10, in which the required interface of one component is ‘wired up’ to the

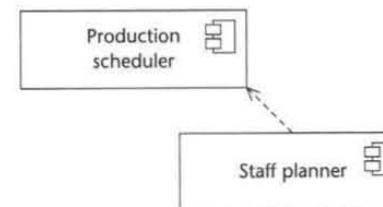


Figure 19.1 Dependency between high-level components.

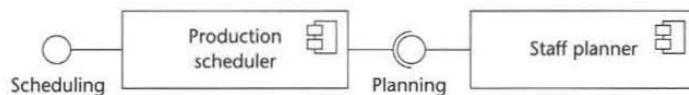


Figure 19.2 Wiring connection between required and provided interfaces.

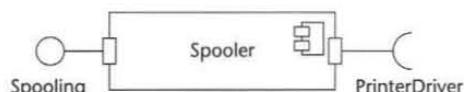


Figure 19.3 Component with ports.

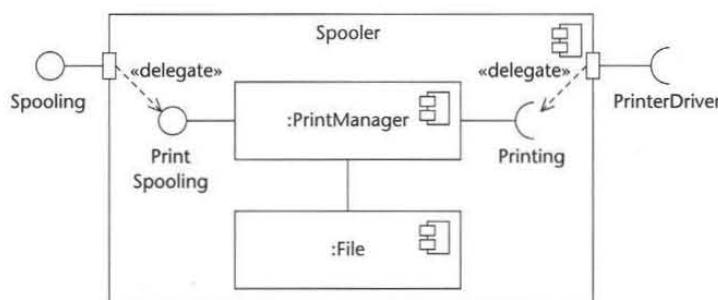


Figure 19.4 Delegation of interfaces to subcomponents within a component.

provided interface of another component. An example of this is shown in Fig. 19.2, where the Production scheduler provides a Scheduling interface and requires a Planning interface, which is provided by the Staff planner component. These wiring connections can also be shown as dependency arrows between the required and provided interfaces, but the ball and socket view is clearer.

The interfaces of components can be shown connected to ports. Ports are small rectangles drawn on the edge of the component. Where an interface is connected to a port, it means that the component delegates responsibility for the behaviour associated with that interface to a subcomponent or object within it. This is shown in Fig. 19.3, and Fig. 19.4 illustrates the internal structure of a component with the delegation dependencies and the connections between the internal subcomponents.

19.4 Deployment Diagrams

The main implementation diagram provided by UML is the *deployment diagram*. Deployment diagrams are used to show the configuration of run-time processing elements and the software components, artefacts and processes that are located on them. They are made up of *nodes* and *communication paths*. Nodes are typically used to show computers, and the communication paths show the network and

protocols that are used to communicate between nodes. Nodes can be used to show other processing resources such as people or mechanical resources. Nodes are drawn as 3D views of cubes or rectangular prisms, and the simplest deployment diagrams show just the nodes connected by communication paths as in Fig. 19.5.

Deployment diagrams can show either types of machine or particular instances as in Fig. 19.5, where *swift* is the name of a PC. Deployment diagrams can be shown with artefacts within the nodes to indicate their location in the run-time environment. Figure 19.6 shows that the *AgateClient.jar* artefact will be deployed onto PC Clients and *AgateServer.jar* will be deployed onto a server.

The relationship between components and artefacts can be shown using a «manifest» dependency. Figure 19.7 shows this for the *Agate Client* component and the *AgateClient.jar* artefact.

Nodes can represent *execution environments* as well as processing *devices*, and can be stereotyped as either in order to distinguish between them. An execution environment is an environment into which executable components can be deployed as artefacts. A typical example would be a J2EE application server. Artefacts can have *deployment specifications* associated with them. A deployment

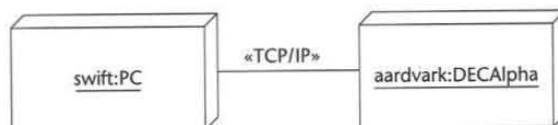


Figure 19.5 Simple deployment diagram.

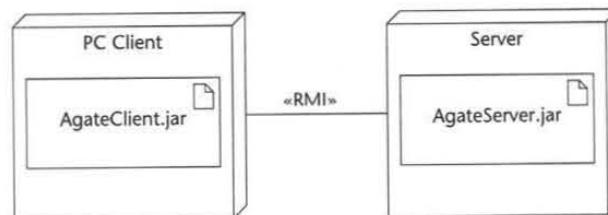


Figure 19.6 Deployment diagram with artefacts.

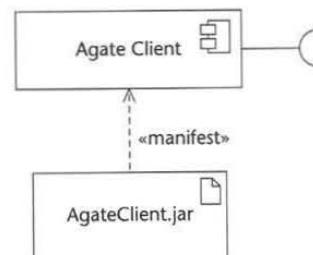


Figure 19.7 Relationship between a component and an artefact.

specification is a set of properties that define the parameters necessary for the execution of an artefact on a node. A *deployment descriptor* is an artefact that represents a deployment specification. Figure 19.8 illustrates these diagram elements. Note that in Figs 19.5 and 19.6 the nodes shown are node types, whereas in Fig. 19.8, an instance of a node is shown. The same conventions are used as for classes and objects, with a colon and underlining to indicate the instance.

If you try to show all the artefacts of a system in deployment diagrams, the diagrams are likely to become very large or difficult to read. They can serve the purpose of communicating information about the location of key components to other members of the team or to users. Indeed, most computer professionals will have drawn an informal diagram like this at some time in their working lives to show where different parts of a system are to be located. Deployment diagrams show the physical architecture of the system.

If you intend to use component and deployment diagrams to illustrate general principles about the way that the new system will be structured, then they are fine as a diagramming technique. However, if the aim of drawing these diagrams is to provide a complete specification of the dependencies between components at compile-time and run-time and the location of all software components as artefacts in the implemented system, then this may be one of those cases where a picture is not worth a thousand words. Even for a relatively simple system that involves multiple nodes, keeping track of all these dependencies and documenting which components have to be on which machines is not a trivial task. For large systems, it is next to impossible. For most systems, this information may be easier to maintain in a tabular format, and a spreadsheet, a database or a configuration management tool may be the best way of doing this.

Component diagrams can be replaced by a table that shows a list of all the software components down the rows and the same list across the top of the columns. It may be best to keep up to three tables (depending on the language used) for compile-time, link-time and run-time dependencies. For each case where a com-

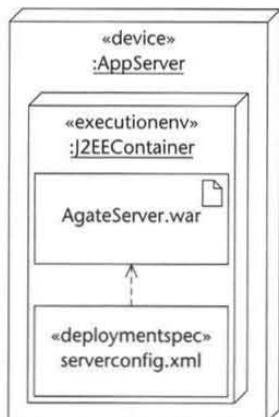


Figure 19.8 Device containing an execution environment with an artefact and a deployment specification.

Campaign database—compilation time dependencies

	JDBC sun.jdbc.*	Campaign .java	Campaign Broker.java	Campaign Proxy.java
JDBC sun.jdbc.*				
Campaign. java				
Campaign Broker.java	✓		✓	
Campaign Proxy.java		✓		✓

Figure 19.9 Excerpt from example table to show artefact dependencies.

Campaign database—run-time locations

	Client PC	Database server
jdbc-odbc.jar	c:\jdbc	
agateclient.jar	c:\agate\client\lib	
SQL*Net		✓
OCI Listener		✓
...		

Figure 19.10 Excerpt from example table to replace deployment diagram.

ponent is dependent on another, place a mark where the row of the dependent component intersects with the column of the component on which it is dependent. A simple example of this is shown in Fig. 19.9.

In the same way, deployment diagrams can be replaced by a table that lists components down the rows and either types of machines or particular instances across the top of the columns. A mark is entered in the row-column intersection for every component that has to be on a particular machine or type of machine. If the exact location of components in a directory structure is important, then that location can be entered into the table. This is shown in Fig. 19.10. Later this will form the basis of the information required for installing software onto users' machines for testing and eventual deployment.

Some configuration management and software build tools also make it possible to store this kind of information and use it to automate the building and deployment of software.

19.5 Software Testing

19.5.1 Who carries out the testing?

One view of testing is that it is too important to be left to the programmers who have developed the software for the system. This is not meant as a criticism of

programmers but reflects the fact that it is important that testing is carried out by someone whose assessment of the software will be objective and impartial. It is often difficult for programmers to see the faults in the program code that they have written. An alternative view is provided by Extreme Programming (XP) (Beck, 2004). XP is an approach to rapid application development in which programmers are expected to write test harnesses for their programs before they write any code. Every piece of code can then be tested against its expected behaviour and if a change is made can easily be retested. XP is explained in Chapter 21.

Some organizations employ specialist software testers. The following paragraph is an excerpt from an advertisement in the British computer press for a post as a tester.

A leading financial institution has an opportunity for a systems tester to work on a business critical project. Testing throughout the project lifecycle, you will liaise closely with developers and team leaders to implement test cases and organize automated testing scripts. All testing is organized within a fully automated environment. With a background in testing you will have strong business acumen ...

However, not all organizations can afford the luxury of specialist testers. Often the analysts who carried out the initial requirements analysis will be involved in testing the system as it is developed. The analysts will have an understanding of the business requirements for the system and will be able to measure the performance of the system against functional and non-functional requirements.

The people responsible for testing will use their knowledge of the system to draw up a test plan. This will specify what is to be tested, how it is to be tested, the criteria by which it is possible to decide whether a particular test has been passed or failed and the order in which tests are to take place. Based on their knowledge of the requirements, they will also draw up sets of test data values that are to be used.

The other key players in the process of testing new software are the eventual users of the system or their representatives. Users may be involved in testing the system against its specification, and will almost certainly take part in final user acceptance tests before the system is signed off and accepted by the clients. If a use-case-driven approach to testing is used, the use cases are used to provide scenarios to form the basis of test scripts.

19.5.2 What is tested?

In testing any component of the system, the aim is to find out if its requirements have been met. One kind of testing seeks to answer the following questions.

Does it do what it's meant to do?

Does it do it as fast as it's meant to do it?

This is equivalent to asking 'Never mind how it works, what does it produce?' and is known as *black box* testing because the software is treated as a black box. Test data is put into it and it produces some output, but the testing does not investigate how the processing is carried out. Black box testing tests the quality of performance of the software. It is also necessary to check how well the software has been designed internally. This second kind of testing seeks to answer the following question.

Is it not just a solution to the problem, but a *good* solution?

This is equivalent to asking 'Never mind what it's for, how well does it work?' and is known as *white box* testing because it tests the internal workings of the software and whether the software works as specified. White box testing tests the quality of construction of the software. In a project where reusable components are bought in, it may not be possible to apply white box testing to these components, as they may be provided as compiled object code. However, some suppliers will supply source code as well as compiled code, and there is a growing Open Source movement that makes this possible. As an aside, some organizations require, as part of their software contracts, that source code is placed in *escrow*. This means that a copy of the source code is lodged with a third party, usually a lawyer or a trade association, so that it is available to the client if the software company goes out of business. By this means the client ensures that they will be able to maintain and enhance the software even if its original developers are no longer able to.

Ideally, testers will use both white box and black box testing methods together to ensure:

- completeness (black box and white box)
- correctness (black box and white box)
- reliability (white box)
- maintainability (white box).

However, the aim of any kind of testing is always to try to get the software to fail—to find errors—rather than to confirm that the software is correct. For this reason the test data should be designed to test the software at its limits, not merely to show that it copes acceptably with routine data.

Testing can take place at as many as five levels:

- unit testing
- integration testing
- subsystem testing
- system testing
- acceptance testing.

In an object-oriented system, the units are likely to be individual classes. Testing of classes should include an initial *desk check*, in which the tester manually walks through the source code of the class before compilation. The class should then be compiled, and the compilation should be clean with no errors or warnings. To test the running of a class the tester will require some kind of test program (the term *harness* is often used) that will create one or more instances of a class, populate them with data and invoke both instance operations and class operations. If pre-conditions and post-conditions have been specified for operations, as suggested in Chapter 10, then the operations that have been implemented will be tested to ensure that they comply with the pre-conditions and that the post-conditions are met when they have completed. State machine diagrams can be used to check that classes are conforming to the behaviour in their specification.

It may be difficult to test classes in isolation. For the reasons that are discussed in Chapter 20 on reuse, most classes are coupled in some way to other classes in the system. Unit testing merges into integration testing when groups of classes are tested together. The obvious test unit at this point is either the use case to test the

system from the user's perspective or the component to test the correct working of components and the interaction between components. The interaction between classes or components can be tested against the specification of the sequence diagrams, timing diagrams and communication diagrams. User interface classes and data management classes will also have to be tested in conjunction with the classes in the application logic layer. If scenario-based design has been used (see Chapter 16), then the scenarios can form the basis for testing scenarios in which a use case can be tested against a typical business situation.

Use cases that share the same persistent data should be tested together. This kind of testing should check that applications work correctly when multiple clients are accessing the database and that transactional database updates are carried out correctly. This is one form of subsystem testing in which the subsystems are built around different business functions that make use of the same stored data.

If significant changes are made to a system, then some of the tests must be run again to ensure that the changes have not broken existing functionality. This is *regression testing*.

Testing is sometimes described as taking place at three levels.

Level 1

- Tests individual modules (e.g. classes or components).
- Then tests whole programs (e.g. use cases).
- Then tests whole suites of programs (e.g. the Agate application).

Level 2

- Also known as Alpha testing or verification.
- Executes programs in a simulated environment.
- Particularly tests inputs that are:
 - negative values when positive ones are expected (and vice versa)
 - out of range or close to range limits
 - invalid combinations.

Level 3

- Also known as Beta testing or validation.
- Tests programs in live user environment:
 - for response and execution times
 - with large volumes of data
 - for recovery from error or failure.

A final stage of testing is *user acceptance testing*, during which the system is evaluated by the users against the original requirements before the client signs the project off. Documentation produced during requirements capture and analysis will be used to check the finished product, in particular use case scenarios and non-functional requirements.

19.5.3 Test documentation

Thorough testing requires careful documentation of what is planned and what is achieved. This includes the expected outcomes for each test, the actual outcomes

Test no.	23		
Purpose	Test correct addition of campaign and adverts.		
Step no.	Test description	Test data	Expected result
23.1	Create a new Campaign		Campaign added to database. Campaign Estimated Cost is set to £0.00
23.2	Add Advert 1 to Campaign	Advert Estimated Cost = £500.00	Advert added to database. Campaign Estimated Cost is set to £500.00
23.3	Add Advert 2 to Campaign	Advert Estimated Cost = -£500.00	Advert not added to database. Negative value rejected. No change to Campaign Estimated Cost. Error message displayed
23.4	Add Advert 2 to Campaign	Advert Estimated Cost = £300.00	Advert added to database. Campaign Estimated Cost is set to £800.00
23.5	Set Advert 1 Completed	Advert Actual Cost = £400.00	Campaign Estimated Cost is set to £700.00. Actual Cost is set to £400.00

Figure 19.11 Excerpt from test plan for Agate.

and, for any test that is failed, details of the retesting. Figure 19.11 shows part of a test plan for the Agate case study. It shows details of each test and its expected outcomes. The results of the actual tests will be documented in a separate, but similar format, with columns to show the actual result of each instance of each test and the date when each test was passed, and to document problems. Many organizations have standard forms for these documents or may use spreadsheets or databases to keep this information. The advantage of using a spreadsheet or database is the ability to produce reports that show what percentage of tests are complete. If requirements are held in a database, it is possible to link requirements to the tests that show whether they have been met and thus to provide a mechanism for tracing through from the original requirements to functionality in the finished system.

Testers should also watch out for unexpected results. Interaction between different operating systems can often cause unanticipated problems with different conventions for newline characters or case sensitivity of filenames. Problems such as these should be reported as bugs and recorded in a fault reporting package for action by the developers.

19.6 Data Conversion

Data from existing systems will have to be entered into a new system when it is introduced. The organization may have a mixture of existing manual and computerized systems that will be replaced by the new system. The data from these systems must be collated and converted into the necessary format for the new

system. The timing of this will depend on the implementation strategy that is used (see next section), but it is likely to be a costly task, involving the use of staff time, the employment of temporary staff or the use of software to convert data from existing computer systems. These costs should have been identified in any cost benefit analysis that was carried out at the inception of the project.

If data is being collated from existing manual systems, it may be necessary to gather it from different sources. Data may be stored in different files, on index cards, in published documents, such as catalogues, or in other paper-based systems. If this data is going to be entered manually into the new system, by users keying it in, then the designers should draw up paper forms that can be used to collate the information so that it is all in one place when it is keyed in. Some data will only ever be entered when the system is started up, for example codes that are used in the system and will not be altered. Special data entry windows may be required for this kind of one-off activity.

Data from existing computer systems will have to be extracted from existing files and databases and reformatted to be usable with the new system. This provides an opportunity to clean up the data: removing out-of-date records and tidying up the values that are stored. Address and telephone number attributes of existing systems are likely to have been abused or misused by users. The work of converting the data may be done by using special programs written by the developers of the system, by employing consultants who specialize in this kind of work or by using commercial software that is capable of reading and writing data in a variety of formats; some packages are capable of working out the format of an unknown file.

The tasks involved in data conversion can be summarized as follows:

- creating and validating the new files, tables or database
- checking for and correcting any format errors
- preparing the existing data for conversion:
 - verifying the existing data for correctness
 - resolving discrepancies between data items from different sources
 - collating data in special forms for input
 - obtaining specially written programs to convert and enter the data
- importing or inputting the data
- verifying the data after it has been imported or input.

All the converted data may have to be ready for entry into the new system to meet a tight deadline, or it may be possible to enter it over a period of time. It is best to convert relatively static data such as product information and customer details first and leave dynamically changing files of information such as orders or other business transactions until last. It may be that only open orders should be taken over into the new system. If historic data is kept in the old system, then this system may have to be maintained for a period of time after the start of the new system. The implementation strategy will determine the timescale for conversion.

Always carry out a trial data conversion exercise before doing it for real. One of the authors was recently involved in converting data from one stock and manufacturing system to another. In both packages (from the same supplier) the Part ID field was case-sensitive. However, in that same supplier's data conversion software

all alphabetic characters in Part IDs were converted to upper case. Because upper and lower case letters were used to distinguish major and minor subassemblies of the same product, this caused problems that would have delayed the implementation if the error in the data conversion program had not been identified well before the planned implementation date.

19.7 User Documentation and Training

19.7.1 User manuals

As well as preparing the technical documentation for the system, analysts or specialist staff in the role of technical authors will be involved in producing manuals for end-users. The technical documentation will be required by the system manager and other staff responsible for running the system and by staff who have to maintain the system. Ordinary users of the system, who will be using it to carry out their daily work tasks, require a different kind of documentation.

Users will require two kinds of manual. During training they will need training materials that are organized around the tasks that they have to carry out with the new system. Online computer-based training materials can be developed so that users learn the tasks in a staged way. These may be in the form of self-study tutorials that users can work through independently of any formal training that is provided.

The users will also need a reference manual that they can refer to while they are using the system. The reference manual should be a complete description of the system in non-technical language. Many software companies employ technical authors to write manuals in language that users can understand. The manual should be organized for ease of use. This involves the author understanding how the user will carry out their tasks and the kind of problem that they will face. The manual should be organized around the users' tasks, and should be supplemented with a comprehensive index based on the terms that the users will be familiar with, rather than the technical terms used by the system developers. Particular attention should be paid to exceptional circumstances and not just routine tasks.

The reference manual may be replicated in the online help so that the users can refer to it while they are using the system. However, it should also be available as a paper manual that the users can refer to if there is a problem with the system, or on a CD-ROM that can be loaded onto a separate machine.

19.7.2 User training

Temporary staff and existing staff will have to be trained in the tasks that they will carry out on the new system. Specialist trainers or analysts are likely to be involved in the design of the training programme, the development of training materials, the planning of the training sessions and the delivery of the training itself.

Training programmes should be designed with clear learning objectives for the trainees. They will be using the system and it is important that the training is practical and geared to the tasks that they will be performing. If it is too theoretical or technical, they will not find it useful. Training should be delivered 'just in time'—when the users need it—as they will forget much of what they are told within a

short space of time, so training delivered even a few weeks before it is required is likely to be wasted. Online computer-based training using video and audio materials that users can refer to when they need it is likely to be of most use. If formal training sessions are used, then trainees should be given learning tasks to take away and carry out in their workplace. This implies that they will be allocated adequate time for training—skimping on training in order to save money is likely to be counter-productive. Staff will not get the best out of the system and are likely to become frustrated if they do not understand how to work the system. It is often worth following up after users have started using a new system to check that they are using it correctly and identify needs for refresher training.

19.8 Implementation Strategies

There are four main strategies for switching over to the new system:

- direct changeover
- parallel running
- phased changeover
- pilot project.

Figure 19.12 shows three of these changeover strategies in diagram form. Each of them has its advantages and disadvantages.

Direct changeover means that on an agreed date users stop using the old system and start using the new system. Direct changeover is usually timed to happen over a weekend to allow some time for data conversion and implementation of the new system. This does not mean that everything happens in a couple of days, as preparatory work will have been carried out in advance. The advantages and

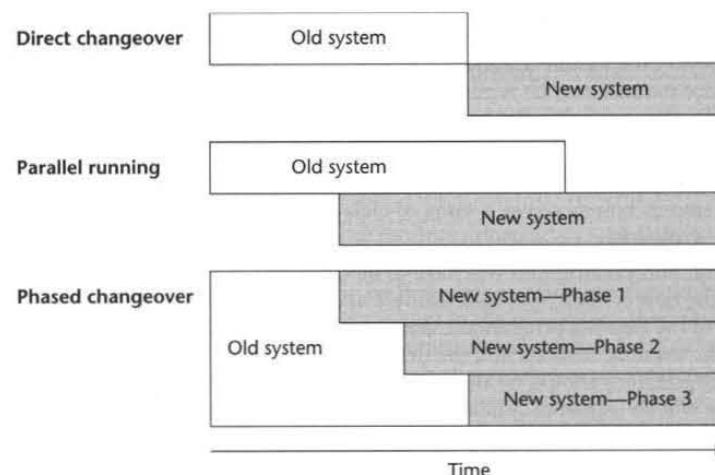


Figure 19.12 Changeover strategies.

disadvantages of this approach are:

- + The new system will bring immediate business benefits to the organization, so should start paying for itself straightaway.
- + It forces users to start working with the new system, so they will not be able to undermine it by using the old system.
- + It is simple to plan.
- There is no fallback if problems occur with the new system.
- Contingency plans are required to cope with unexpected problems.
- The plan must work without difficulties for it to be a success.

Direct changeover is suitable for small-scale systems and other systems where there is a low risk of failure, such as the implementation of established package software.

Parallel running allows the existing system to continue to run alongside the new system. The advantages and disadvantages of this approach are:

- + There is a fallback if there are problems with the new system.
- + The outputs of the old and new systems can be compared—so testing can continue.
- There is a high cost as the client must pay for two systems during the overlap period, and this includes the staffing necessary to maintain information in the old system as well as the new.
- There is a cost associated with comparing the outputs of the two systems.
- Users may not be committed to the new system as it is easier to stick with the familiar system.

Parallel running should be used in situations where there is a high level of risk associated with the project and the system is central to the business operations of the organization.

In a *phased changeover*, the system is introduced in stages. The nature of the stages depends on the subsystems within the software, but introduction into one department at a time may be appropriate. The advantages and disadvantages are:

- + Attention can be paid to each individual subsystem as it is introduced.
- + If the right subsystems can be chosen for the first stages, then a fast return on investment can be obtained from those subsystems.
- + Thorough testing of each stage can be carried out as it is introduced.
- Disaffection and rumour can spread through the organization ahead of the implementation if there are problems with the early phases.
- There can be a long wait before the business benefits of later stages are achieved.

Phased changeover is suitable for large systems in which the subsystems are not heavily dependent on one another. The different phases can be introduced on a geographical basis or by department.

A variation on phased changeover is the use of a *pilot project* approach. This involves trialling the complete system in one department or on one site. The decision on extending the system to the rest of the organization depends on the

success of the pilot scheme. The pilot project can be used as a learning experience, and the system can be modified on the basis of the practical experience in the pilot project. As such, pilots are suitable for smaller systems, or packaged software, as it is unlikely that a large-scale system will be developed and then introduced in a way that makes its full-scale implementation dependent on the success of a pilot.

Although we have included information about implementation strategies at this late stage in the book, it is important to stress that liaison with operational staff about a new system and planning for its implementation should begin early in the development lifecycle. Adopting an iterative approach, in which incremental deliveries of software may occur from the start of the construction phase, means that the necessary infrastructure to run and support the new system has to be in place before the transition phase. It also means that planning for implementation of the new system must be based on a gradual migration from existing manual processes or an existing system.

New systems often mean new or changed business processes, and changing business processes may mean changes to organizational structures in the functional business departments within the organization. Staff may be assigned to change management roles in order to ensure that process and organizational changes are planned and that they deliver the benefits that were claimed for them when the project was justified in cost-benefit terms.

19.9 Review and Maintenance

19.2.1 The next steps

The work of analysts, designers and programmers does not stop after a system has been implemented. There is a continuing requirement for staff to work on the new system. First, it is important that the organization reviews both the 'finished' product and the process that was undertaken to achieve it. This may be for contractual reasons, in order to check that the product meets requirements. However, there is a growing recognition of the need for organizations to learn from experience and to record and manage the organizational knowledge that results from this learning. If there were any problems during the lifetime of the project, then these should be reviewed and conclusions drawn about how they might be avoided in the future. The amount of time spent on different tasks during the project can be used as the basis for metrics to estimate the amount of time that will be required for similar tasks in future projects. Second, it is unlikely that the system will be working perfectly according to the users' requirements, and further work will have to be done. Third, in an object-oriented project, the design should be reviewed to identify candidate components for future reuse, although as we suggest later in this chapter, planning for reuse should begin in the early stages of a project. Identifying reusable software components is not an activity to be left until the completion of the project. This subject is covered in Chapter 20.

19.2.2 The review process and evaluation report

The review process will normally be carried out by the systems analysts who have been involved in the project from the start, although it is possible to involve outside

consultants in the process for an impartial view. They will normally be supported by representatives of users and user management. The various stakeholders who have invested time, money and commitment in the project will all have an interest in the content of the evaluation report. The report can be very detailed or can provide an overview evaluation—like everything else in the project, there will be a cost associated with producing it. The report's authors should consider the following areas.

Cost benefit analysis. The evaluation should refer back to criteria that were set for the project at its inception. It may not be possible to determine whether all the benefits projected in the cost benefit analysis have been achieved, but most of the costs of development, installation, data conversion and training will have been incurred and can be compared with the projections.

Functional requirements. It is important to check that the functional requirements of the system have been met. Clearly, this is something that should have been taking place throughout the lifetime of the project, but a summary can now be produced. Any actions that were taken to reduce the functional requirements, perhaps to keep the project within budget or on schedule, should be documented for future action under the heading of maintenance. If large areas of functionality were removed to bring the project in on schedule or within budget, a new project should be considered. Major bugs should similarly be documented if they have emerged since the implementation of the new system.

Non-functional requirements. The system should be reviewed to ensure that it meets the targets for non-functional requirements that were documented during requirements analysis. It is now possible to assess whether quantitative objectives for learnability, throughput, response times or reduction of errors have been achieved.

User satisfaction. Both quantitative and qualitative evaluations of the users' satisfaction with the new system can be undertaken, using questionnaires or interviews or both. The results should be treated carefully, as users can pick on quite minor problems or be influenced in their views of the finished product by experiences during the project.

Problems and issues. As stated above, this is an important part of the evaluation process. Problems that occurred during the project should be recorded. These problems may have been technical or political, and it is important to handle the political issues with tact. Including a section criticizing unco-operative users or obstructive user management will mean that some readers take no notice of the rest of the report. Solutions to problems should also be included in the report, as should an indication of who should be learning from this part of the process.

Positive experiences. It is all too easy to focus on the negative aspects of a completed project. It is worth recording what parts of the project went well and to give credit to those responsible.

Quantitative data for future planning. The evaluation report provides a place in which to record information about the amount of time spent on different tasks in the project, and this information can be used as the basis for drawing up future project plans. The quantitative data should be viewed in the light of the problems and issues that arose during the project, as the amount of time spent on a difficult task that was being tackled for the first time will not necessarily be an accurate predictor of how much time will be required for the same task in the future.

Candidate components for reuse. If these have not already been identified during the project itself, then they should be identified at this stage. There will be different issues to be addressed, depending on whether the project has been carried out by in-house development staff or external consultants. For in-house projects, the reuse of software components should be viewed as a process of recouping some of the investment made and being able to apply those reusable elements of the system in future projects. For projects undertaken by external consultants, it may highlight legal issues about who owns the finished software that should have been addressed in the contract at the start of the project.

Future developments. Any requirements for enhancements to the system or for bugs to be fixed should be documented. If possible, a cost should be associated with each item. Technical innovations that are likely to become mature technologies in the near future and that could be incorporated into the system in an upgrade should also be identified.

Actions. The report should include a summary list of any actions that need to be undertaken as a result of carrying out the review process, with an indication of who is responsible for carrying out each such action and proposed timescales.

The end of a project provides a good opportunity for managers of the development staff to review performance on the project as part of a separate review process, the results of which are not necessarily in the public domain. This may feed into staff appraisal procedures and can affect the payment of bonuses, promotion prospects and staff development issues. The latter can include the choice of the next project for a member of staff to work on, their training needs or opportunities to manage or act as a mentor to less experienced staff.

19.2.3 Maintenance activities

Very few systems are completely finished at the time that they are delivered and implemented, and there is a continuing role for staff in ensuring that the system meets the users' requirements. As well as maintenance of the system, there will be a need for support of users: providing initial and on-going training, particularly for new staff; improving documentation; solving simple problems; implementing simple reports that can be achieved using SQL or OQL without the need for changes to the system software; documenting bugs that are reported; and recording requests for enhancements that will be dealt with by maintenance staff. In large organizations these tasks are often handled by a helpdesk that supports all the organization's systems, and it may be appropriate for a member of the development team to join the helpdesk staff either temporarily or permanently. Whether or not this happens, helpdesk or support staff will need to be provided with training so that they can support the new system.

Maintenance involves more significant amendments to a system once it is up and running. Maintenance may be required for a number of reasons.

- There will almost certainly be bugs in the software that will require fixing. The use of object-oriented encapsulation should mean that it is easier to fix bugs without creating knock-on problems in the rest of the system. It is sometimes suggested that bug-fixing involves spending as much time fixing bugs that were introduced by the previous round of maintenance as it does in fixing bugs in the original system.

- In an iterative lifecycle, parts of the system may be in use while further development is undertaken. Subsequent iterations may involve maintaining what has already been developed.
- Users request enhancements to systems virtually from day one after implementation. Some of these will be relatively simple, such as additional reports, and may be dealt with by support staff, while others will involve significant changes to the software and will require the involvement of a maintenance team. Often these user requests will reflect the fact that until the system is running and users have a chance to see what it can do, it is difficult for them to have a clear idea of what their requirements are.
- In some cases, changes in the way that the business operates or in its environment, for example new legislation, will result in the need for changes to the system.
- There is a growing expectation that systems will be built as a collection of loosely coupled services (service-oriented architecture), and that business processes will be assembled by linking these services together with a business process management tool. The reason for this is to allow the business users to change their processes and link the services together in new ways. Such changes may result in the need to make changes to the implementation of services.
- Similarly, changes in the technology that is available to implement a system may result in the need for changes in that system.
- Disasters such as fires that result in catastrophic system failure or loss of data may result in the need for maintenance staff to be involved in restoring the system from data back-ups. Procedures for handling disastrous system failure should be put in place before disasters take place.

In each of these cases, it is necessary to document the changes that are required. In the same way as it is necessary to have a system in place during a project for handling users' requests for changes to the requirements (a change control system), it is necessary to have a system for documenting requests for changes and the response of the maintenance team. This should include the following elements.

Bug reporting database. Bugs should be reported and stored in a database. The screen forms should encourage users to describe the bug in as much detail as possible. In particular, it is necessary to document the circumstances in which the bug occurs so that the maintenance team can try to replicate it in order to work out the cause.

Requests for enhancements. These should describe the new requirement in a similar amount of detail. Users should rate enhancements on a scale of priorities so that the maintenance team can decide how important they are.

Feedback to users. There should be a mechanism for the maintenance team to feed back to users on bug reports and requests for enhancements. Assuming that bugs affect the agreed functionality of the system, users will expect them to be fixed as part of the original contract or under an agreed maintenance contract. The maintenance team should provide an indication of how soon each bug will be fixed. Enhancements are a different matter. Depending on the contractual situation, enhancements may be carried out under a maintenance contract or they may be subject to some kind of costing procedure. Significant enhancements may cost large amounts of money to implement. They will require the same kind of

assessment as the original requirements. They should not be left to maintenance programmers to implement as they see fit, but should involve managers to authorize the expenditure and analysts and designers to ensure that the changes fit into the existing system and do not have repercussions on performance or result in changes to subsystems that affect others. This process itself may incur significant costs just in order to work out how much an enhancement will cost to implement. Significant enhancements should therefore be regarded as mini projects in their own right and involve the professional skills of project managers, analysts and designers.

Implementation plans. The maintenance team will decide how best to implement changes to the system, and this should be carried out in a planned way. For example, significant additions to a class that affect what persistent data is stored in the database will require changes to the database structure and may also require all existing instances of that class to be processed in order to put a value into the new attribute. This will probably have to take place when the system is not being used, for example over a weekend. Enhancements to the system may fall into one of four categories: those to be made at no cost; those that will be made at a cost to be agreed with the client; those that will be held over until a major upgrade to the software is made and that will be part of a future version; and those that cannot or will not be made in the foreseeable future.

Technical and user documentation. Amendments to a system must be documented in exactly the same way as the original system. Diagrams and repository entries must be updated to reflect the changes to the system. If this is not done, then there will be a growing divergence between the system and its technical documentation; this will make future amendments all the more difficult, as the documentation that maintenance analysts consult will not describe the actual system. Clearly, user documentation, training and help manuals as well as online help must all be updated.

In large organizations with many systems, staff in the information systems department may spend more time on maintenance of existing systems than they do on development of new systems. There is a growing movement for organizations to *outsource* their maintenance. This means handing over the responsibility for maintenance of a system to an external software development company under a contractual agreement that may also involve the provision of support. Some companies now specialize entirely in maintaining other people's software.

The other impact of the growth of the maintenance task is that the time will come when a system is such a burden in terms of maintenance that a decision must be made about its replacement. The review process that takes place at this stage may lead to the inception of a new project and to the systems development life-cycle starting again.

19.10 Summary

The implementation of a new system involves a large number of different types of software package that are used to produce and to support the finished system. Component diagrams and deployment diagrams are the two UML implementation diagrams that can be used to document the software components and their

location on different machines in the system. For large, complex installations, these diagrams may become unwieldy, and a table format in a spreadsheet or database may be easier to maintain.

Analysts and designers may have a role during the implementation stage in maintaining system and user documentation and in providing user training. They may also plan and carry out testing, plan for data conversion from existing systems and assist the project management in planning the appropriate implementation strategy for the system. In larger organizations, these tasks may be carried out by staff in specialist roles.

After a new system has been implemented, it is customary and advisable to carry out a post-implementation review. This will result in the production of an evaluation report to stakeholders that will measure the success of the project and identify issues and problems from which the organization should learn. Typically the evaluation report will include the following sections:

- Cost benefit analysis
- Functional requirements
- Non-functional requirements
- User satisfaction
- Problems and issues
- Quantitative data for future planning
- Opportunities for reuse
- Future developments
- Actions.

New systems rarely work exactly as expected, and maintenance must be carried out in order to ensure that the system is bug-free and meets users' requirements. Procedures must be put in place for the maintenance team (project manager, analysts, designers and programmers) to document the process and the changes that are made.

Review Questions

- 19.1** List the different categories of software packages that may be used in developing a system.
- 19.2** What development packages have you used and which categories do they fall into?
- 19.3** What is the difference between an artefact and a component?
- 19.4** Draw a component diagram to show two components wired together with required and provided interfaces.
- 19.5** Draw a deployment diagram with artefacts to show the run-time dependency between a Java class file, the java.exe run-time program and the Java classes in a zip file.
- 19.6** Draw a deployment diagram to show how a web browser and web server are located on different machines and the communication protocol they use.

- 19.7** List five tests that you would carry out on the FoodCo use case Start Line Run.
- 19.8** List the sections that you would include in a post-implementation evaluation report and explain the content of each section.
- 19.9** What is the difference between maintenance work carried out to fix bugs and work carried out to add requested enhancements to a system?
- 19.10** Why should decisions about enhancements not be left to maintenance programmers?
- 19.11** What tasks do maintenance staff undertake?

Case Study Work, Exercises and Projects

- 19.A** Draw a component diagram for a program that you have written.
- 19.B** Find out about the library system in your school, college, university or local library. Draw a deployment diagram to show the physical hardware architecture.
- 19.C** The FoodCo system will have a database on a central server and PCs in different offices that access that server. Draw a deployment diagram to show this architecture.
- 19.D** There are commercial packages to automate the installation of software such as InstallShield for Windows and RPM for Red Hat Linux. Investigate how these packages maintain information about what artefacts must be installed on a machine.
- 19.E** Read the user manual or online help for a software package that you use. What examples can you find of good practice and bad practice in what has been written? How would you improve it?
- 19.F** What would you include in the screen layouts for users to fill out for bug reports and enhancement requests? Produce a draft design for each of these two layouts.

Further Reading

Implementation takes us on into areas that are outside the scope of this book. There are a number of excellent books on Java implementation, for example Deitel and Deitel (2004). Wutka's book on Java 2 Enterprise Edition (2001) gives a clear idea of the techniques involved in building J2EE systems. Orfali and Harkey (1998) explain CORBA and the alternatives. There are now many books available on .Net, for example Fedorov (2002).

Many CASE tools now offer a wide range of functionality, including integration with configuration management tools, built-in IDEs for programming, code generation and reverse engineering, object-relational mapping tools, software metrics and deployment tools. Vendor websites are the best starting point for understanding what is on offer.

Reusable Components

Learning Objectives

In this chapter you will learn

- why reuse is regarded as important
- some of the problems with reuse
- how to plan systems development to achieve reuse of components
- the different techniques for obtaining and managing reuse
- how to design a reusable component.

20.1 Introduction

In the system lifecycle, implementation is followed by maintenance, when the new system has any remaining bugs removed and enhancements made to it. Using object-oriented technology does not make the problems of removing bugs and enhancing systems go away, but it does add a possible further stage to the lifecycle—the reuse stage. In this chapter we look at how object-oriented software can be reused. In particular we look at the idea of *componentware*—software packaged into components that can be reused as part of other systems.

In Section 20.2 we discuss approaches to developing component-based software: we raise issues concerning planning for reuse during the project and consider how to package software to make it reusable. We discuss an example of industrial strength componentware and present an example from the case studies used in this book which demonstrates how software can be designed and packaged for reuse (Section 20.4). There is a growing interest in services as a way of creating reusable components, and in Section 20.5 we introduce the idea of service-oriented architecture. Chapter 22 on the book's website addresses the project management issues associated with planning for reuse.

20.2 Why Reuse?

In Chapter 4 reusability was discussed as one of the reasons for adopting object-oriented development techniques and programming languages, in Chapter 8 inheritance and composition were discussed as two techniques that facilitate the development of reusable components and in Chapter 13 we highlighted reusability as one of the characteristics of a good object-oriented design. Reusable software has been one of the objectives of developers for many years. Using top-down functional decomposition of designs in languages such as Fortran or C, the development of reusable libraries of functions has made it possible for programmers to save time and effort by reusing others' work. The growth of Visual Basic as a programming language was aided by the availability of controls that could be bought off the shelf and incorporated into applications to provide functionality that would be difficult for the less experienced programmer to develop—and in any case, why re-invent the wheel? Object-oriented languages have always been promoted as significantly enabling the reuse of software and, when Java was released in the mid 1990s, part of the hype surrounding the language was the ability to download and reuse services as Applets over the Internet. Why then is reuse regarded as so important?

The arguments for reuse are partly economic, partly concerned with quality and partly about business flexibility.

- If some of the requirements of a project can be met by models or software components that have been developed on a previous project or are bought in from an outside supplier, then the time and money spent producing those models or code is saved, although the saving will be partly offset by the cost of managing a catalogue of reusable models or code or of paying to buy them from elsewhere. In 2005 Standard Life, a British mutual insurance company, claimed to have achieved savings of £2M by creating a culture of reuse and the supporting tools. From a collection of 246 reusable services, 123 had been reused in 253 separate instances of reuse.
- If a developer can reuse a design or a component that has been tested and proved to work in another application, then there is a saving in the time spent to test and quality assure the component. Jacobson et al. (1997) cite IBM as an example of a company that has invested in software reuse and that has reuse support centres that maintain a library of 500 zero-defect components in Ada, PL/X and C++. In 2004, IBM adopted Logidex from LogicLibrary as its component repository.
- From the point of view of business users of information systems, reuse is not such a critical issue (although it has an impact on cost). In a dynamic business world, organizations need to be able to respond to a changing environment by changing business processes. The ability to recombine information system components or services in new ways to create new business processes or adapt existing ones is important to business stakeholders.

Developers of object-oriented systems are often end-users of reusable components, when they use packages, libraries, classes or controls in their chosen development environment. However, object-oriented systems have not achieved the level of reuse that was expected of them in terms of generating reusable components that

can be applied again within the same organization. There are a number of reasons for this; some are technical and some are concerned with organizational culture.

- *Inappropriate choice of projects for reuse.* Not all organizations or projects within those organizations are necessarily suitable to take advantage of or act as sources of reusable components.
- *Planning for reuse too late.* If reuse is appropriate, it is something that needs to be planned for even before a project starts, not an afterthought. By the time a project has been completed, it is likely that anything that might have been reusable will have been designed in such a way that it cannot easily be extracted from the rest of the system. To achieve reuse, the organization needs to be structured to support it, with the people and tools in place to make it possible.
- *The level of coupling between different classes in an object-oriented design.* Many people have thought of classes as the unit of reuse in object-oriented developments. However, when we come to design classes for different systems, it may be possible to identify similar classes that could be developed in a way that makes them of use in more than one system, but, more often than not, the implementations of these classes will include attributes and associations that tie them into other classes in the particular application of which they are a part. This means that reuse needs to be at a more coarse-grained level than classes, typically components or services.
- *The platform dependence of reusable components.* Most components are built to run on a particular platform. Although Java provides a measure of platform independence at run-time and CORBA makes it possible to connect together components on different platforms, interoperability can still be an issue. One of the attractions of the web services standards is that they allow services on different platforms to be connected together using a standard protocol.
- *The lack of standards for reusable components.* This has changed recently with developments in the technology of repositories in which to store components and services and with the introduction of standards such as the Object Management Group's Reusable Asset Specification (RAS) for managing software and metadata assets and the W3C's SOAP (Simple Object Access Protocol) for web services. In the following sections, we will address each of these issues.

20.2.1 Choice of project

Not all projects are necessarily suitable for the development of reusable components. The two main factors that influence this are the nature of the business within which the software development is taking place and the maturity of the organization's object-oriented development.

Jacobson et al. (1997) identify four kinds of software business, which they suggest are suitable candidates for developing reusable components. In all of these they talk of the organization developing a Reuse-driven Software Engineering Business (RSEB).

- Organizations where creating an RSEB improves the business processes within the organization: large organizations with a considerable information systems infrastructure and a portfolio of projects to support business activities.

- Organizations producing hardware products that contain embedded software: they cite Hewlett-Packard and Ericsson as examples of this type of organization.
- Consultancy companies and software houses that develop software for external clients that have outsourced their information systems development; particularly those which target particular vertical markets (companies in the same kind of business).
- Developers of software products, such as Microsoft, where reusable components can be applied across a large product range and where end-users can also benefit from the interoperability of software through mechanisms such as DCOM (Distributed Component Object Model).

Small, one-off projects in small organizations are unlikely to bring significant benefits from building reuse into the software development lifecycle. Large organizations with a significant portfolio of systems are beginning to see benefits from reuse through the adoption of a service-oriented architecture. By wrapping existing systems behind a service façade and building new applications as a set of loosely coupled services with a Business Process Management (BPM) layer above them that combines the services to support business processes, they are achieving both reuse and flexibility.

If we consider our two case study companies, we have to ask whether they would fall into any of the categories listed. Although both of them are developing systems to support their business activities, are they large enough and with enough potential projects to justify taking a reuse-driven approach to these projects? Building the organizational structures to support reuse costs money, and that expense is only justified if it can be recouped by savings on other projects. In both cases, they do not have a developed information systems department and we would probably have to say that they are not going to benefit from developing an RSEB.

If, however, an outside consultancy company is doing the development for both Agate and FoodCo, then there are a number of areas where reuse may be applicable. A software company looking at the two systems would identify areas such as managing information about staff that are common to both. If the software company specializes in a vertical market—media and advertising or food manufacturing—then there are going to be parts of the software systems that can be reused elsewhere. Indeed, if the software company has already introduced a reuse-driven approach to its business, the systems for Agate and FoodCo could be developed from a range of existing components tailored to the specific requirements of these companies.

20.2.2 Organizational structure

Jacobson et al. (1997), based on experience at Hewlett-Packard, describe organizations as typically going through six stages of development of a reuse culture. At each stage some benefit is to be gained, but it is unlikely that an organization can leap from a situation in which there is no reuse of design models or software taking place to one in which there is a complete organizational culture of reuse and the structures and tools are in place to support the consistent reuse of components in a way that brings the kind of business benefits that were mentioned earlier. The six stages are shown in Fig. 20.1.

Stage of Reuse	Description
None	No code reuse takes place; everything is developed from scratch
Informal code reuse	Developers trust each other enough to begin to reuse each other's code in order to save time on development
Black-box code reuse	Particular pieces of code are engineered for reuse, and all developers are encouraged or required to use them to ensure a consistent approach and reduce maintenance costs
Managed workproduct reuse	An organizational structure is developed to manage reusable code, to maintain versions, to document functionality and to train developers
Architected reuse	In order to ensure that components work together, a common architecture is designed and applied to all development processes
Domain-specific reuse-driven organization	The organization's software development is geared to the production of reusable components for the business domain, and the culture and structure of the organization supports this approach

Figure 20.1 Jacobson's six stages of reuse.

Allen and Frost (1998) argue that, despite moves to client-server and three-tier architectures, most software development organizations still have an application mind-set, and software is developed for individual applications, even within the same organization, without regard for reuse. (Many professional developers would counter this with the argument that they have always developed reusable libraries of code.) This may even be a step backwards from the developments of the 1970s and 1980s when the growth of corporate databases under the central control of database administrators meant that at least the data model was likely to be consistent across the whole organization. The development of web services and interest in Service Oriented Architecture is rekindling the interest in software reuse in large organizations.

To gain the benefits of an RSEB requires an incremental process of change within the organization, involving technical champions to argue the technical case and develop the software architecture, management champions who believe in the business benefit and will provide the support and investment to allow the change to take place, pilot projects to kick-start the process, education and training of developers to enable them to understand the process and the development of support structures within the organization. Of these, the first is the most critical: to achieve effective reuse, the elements of the software architecture must be common across different systems.

One of the most significant requirements for support structures is that if developers are to use reusable components in their code they need some way of finding out what components are available and what their specifications are. This requires software tools to manage a repository of components and staff to maintain the components in the repository and to document them.

Allen and Frost (1998) place a repository at the centre of their model of the development process for reusable components. Figure 20.2 shows this with the two complementary processes: sowing reusable components during development and harvesting reusable components for reuse in other projects. The Select Perspective has evolved since then into three workflows concerned with supplying, managing and consuming components and is described in more detail in Section 20.3.1.

20.2.3 Appropriate unit of reuse

In Chapter 8 we talked about components as the unit of reuse, and we have used the term throughout this chapter so far. However, we have not yet defined what we mean by a component in this context.

If we again consider the case studies, there is a need for a `Client` or `Customer` class both in the Agate system and in the FoodCo system. During analysis, these two classes may look very similar, but, as we move into design, the associations between these classes and others in their system will be resolved into specific attributes. The Agate `Client` class will have attributes to link it to `Campaigns` while the FoodCo `Customer` class will be linked to `SalesOrders`. If the development of both systems is being carried out by the same software company, then it requires a novel style of project management and organization to recognize this commonality in two different projects. If the commonality is recognized, then there is no guarantee of successful reuse unless a suitable architecture is developed that will support the reuse of the common elements of the `Client` class and allow them to be tailored to the requirements of the individual systems.

For example, either a `Client` class can be obtained from elsewhere and subclassed differently for each project, or a `Client` class can be written that is domain-neutral and then subclassed for each different project. This inheritance-based approach also helps to solve a problem that is common with software that is tailored to the needs of different customers of the software house: it clearly separates those parts of the class that are common to all users from those that have been tailored to specific needs. This helps with the installation of upgrades and prevents the changes made for one customer being implemented for all customers.

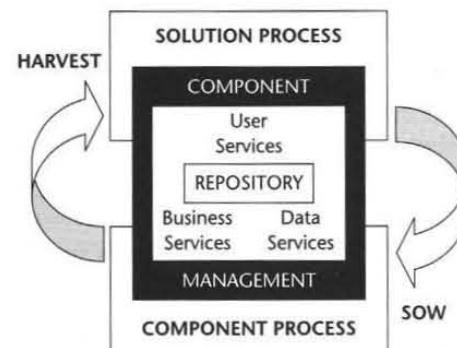


Figure 20.2 The SELECT Perspective service-based process (adapted from Allen and Frost, 1998).

However, even if we can reuse the `Client` class in both applications by extending its functionality through inheritance, there are going to be other aspects of the `Client` class that we may or may not want to take through into another system. These include control classes and the business logic associated with the management of clients, related boundary classes and the mechanisms that manage the persistent storage of instances of `Client` in some kind of database. So can we reuse the class on its own?

Allen and Frost (1998) argue that the class is the wrong level of granularity at which to apply reuse. They argue that reuse should take place at the level of components rather than classes. They define a component as follows.

A component is an executable unit of code that provides physical black-box encapsulation of related services. Its services can only be accessed through a consistent, published interface that includes an interaction standard. A component must be capable of being connected to other components (through a communications interface) to form a larger group.

(Allen and Frost, 1998, p.4)

However, this view of a component as executable code limits the types of reuse that can be made of a component. Jacobson et al. define a component as follows.

A component is a type, class or any other workproduct that has been specifically engineered to be reusable.

(Jacobson et al., 1997, p.84)

This definition is more useful, as it does not limit the developer to only considering executable code for reuse. The intermediate products of the development lifecycle—use case model, analysis model, design model, test model—can all be considered as candidates for reuse. There are two outcomes from this view.

- First, we may choose to reuse as components subsystems that provide more functionality than just a single class. For example, again both Agate and FoodCo have requirements to manage information about employees. Rather than developing two separate systems that handle staff, grades, staff development and so on, we may aim to produce a single reusable subsystem that can be used in both companies' systems.
- Second, we may choose to reuse intermediate products. For example, if we have an existing subsystem to manage staff, we could reuse some of the use cases and related boundary, control and entity classes, but choose to leave out others. Tracing through from the analysis model to the design and test models of the system, we should also be able to reuse elements of these models.

Essentially, we are dealing here with the difference between black-box and white-box reuse. Allen and Frost are suggesting a black-box model in which the contents of the components are not visible to the consumer; Jacobson et al. are suggesting a white-box model in which the internals of the component are visible, giving more flexibility to the consumer about how they make use of the component.

Jacobson et al. also make the point that there are different mechanisms for reusing components. In Chapter 8 we discussed the use of inheritance and composition as mechanisms for reuse. However, using inheritance to subclass existing classes is not the only mechanism for reuse and, if the class is not the unit of reuse,

then other mechanisms must be used. Jacobson et al. suggest the following:

- inheritance
- the «include» relationship between use cases
- extensions and extension points in use cases and classes
- parameterization, including the use of template classes
- building applications by configuring optional components into systems
- generation of code from models and templates.

The last two are development processes rather than specific design structures and make reuse easier to achieve. To this list we must add the use of loosely coupled services along the lines of web services, which are the basis of service-oriented architectures.

20.2.4 Component standards

If we are talking about black-box reuse, then the potential for reuse depends on the software mechanisms for reusable components. If we want to consider white-box reuse, then the potential depends on the mechanisms for exchanging software models. In the latter case, UML is clearly a candidate for exchangeable, reusable software models, especially if modelling tool vendors implement the XMI (XML Metadata Interchange). In the former case, then we are dependent on the developers of programming languages and software development infrastructure to deliver appropriate tools to the development community to enable them to develop reusable components.

A number of programming languages and development environments provide mechanisms by which developers can package software into components. Figure 20.3 lists some of these. This table shows that the search for ways of promoting reuse through some kind of modular architecture is not new in the software development industry. Reuse has been an objective that has driven the design of programming languages and has informed the development of programming styles. However, the potential for developing reusable components has been increased recently by five factors:

- the development of CORBA as a standard for interoperability of components written in different languages and running on different platforms
- the promotion of Java as an object-oriented language with relatively straightforward mechanisms for producing software in packages to deliver different services
- the growth of the Internet and the World Wide Web, which has made it possible for people to make their software components easily available to a wide marketplace of potential consumers
- the spread of web services accessed using the SOAP protocol over networks, which may be private within an organization or public for business to business exchanges
- the availability of the Reusable Asset Specification (RAS), which provides a standard format for packaging up the artefacts associated with a reusable component, including specification, design, code, executables and metadata that makes it possible to search for components in a catalogue.

Language or development environment	Mechanism for component reuse
Microsoft Visual Basic	.vbx files—Visual Basic Extensions .ocx files
Microsoft Windows	.ole files—Object Linking and Embedding DDE—Dynamic Data Exchange .dll files—Dynamic Link Libraries COM—Common Object Model DCOM—Distributed Common Object Model
CORBA	.idl files—Interface Definition Language IOP—Inter-ORB Protocol
Java	.jar files—Java Archive packages JavaBeans
Microsoft .NET	MSIL—Microsoft Intermediate Language CLR—Common Language Runtime
Web Services	SOAP—Simple Object Access Protocol WSDL—Web Service Description Language UDDI—Universal Description, Discovery and Integration

Figure 20.3 A sample of languages and development environments with mechanisms for reuse.

The platform independence of Java and CORBA makes them different from the other languages and environments shown in Fig. 20.3.

However, web services provide even greater interoperability between components running on different platforms. The Universal Description, Discovery and Integration (UDDI) standard provides a way for potential clients of services to search for services in a registry and connect to them dynamically at run-time. The descriptions of the services are provided in Web Services Description Language (WSDL), an XML language, which defines the input and output parameters of the services and their location on the network. The use of web services makes it possible to create components in different languages that can interoperate and do not require the use of a CORBA Object Request Broker (ORB) in order to do this. Many businesses are also wrapping functionality of legacy systems in web services to make it possible to reuse their functionality in conjunction with new systems.

.NET also defines extensions to Microsoft's Portable Executable (PE) format so that metadata is stored with the bytecode in Microsoft Intermediate Language (MSIL) executables, allowing them to provide information about the services they offer in response to requests in the correct format. However, this does not provide the same degree of interoperability as languages such as Java, which are platform-independent. Whether the MONO project, which is porting the Microsoft run-time environment to other platforms such as Linux, will change this remains to be seen.

The Reusable Asset Specification (RAS) from OMG promises to make it easier to support reuse within an organization by standardizing the way in which information about reusable components and the executables for the components can be stored in a repository by producers and located in a catalogue by potential consumers. At the time of writing there are only a few products that support

the standard, notably Logidex from LogicLibrary, which has been adopted by IBM Software Division as its component repository. This is potentially a growth area.

However, the existence of CORBA, Java, web services, .NET and RAS does not guarantee that effective reuse of components will take place. A strategy needs to be put in place within the organization to ensure that reuse is built into the systems development lifecycle.

20.3 Planning a Strategy for Reuse

In some organizations, reuse may just be about making use of reusable components from elsewhere, using the kinds of mechanisms that are listed in Fig. 20.3. In others, reuse will be about the kind of organizational change that we discussed in Section 20.2.2. In the rest of this section we describe two approaches to the introduction of a reuse strategy and then in Section 20.4 give an example of a commercially available reusable component package.

20.3.1 The SELECT Perspective

Allen and Frost (1998) first described the SELECT Perspective approach to the development of reusable components. At the level of practical techniques, this includes guidelines for the modelling of business-oriented components and for wrapping legacy software in component wrappers. They distinguish between reuse at the level of component packages, which consist of executable components grouped together, and service packages, which are abstractions of components that group together business services. The focus of this approach is to identify the services that belong together and the classes that implement them. Service classes in a single package should have a high level of internal interdependency and minimal coupling to classes in other packages.

The Select Perspective has been updated by Apperley et al. (2003), who define the process in terms of three workflows.

- *Supply*—delivers and maintains components. Participants negotiate requirements with consumers, and design, construct and test the components. This may entail the wrapping of legacy systems as services.
- *Manage*—deals on the one hand with acquiring, certifying and publishing components and on the other with locating and retrieving components for reuse.
- *Consume*—delivers solutions to stakeholders based on components produced by the suppliers. Ensures component specifications are aligned to business needs.

These three workflows are shown in Figs 20.4, 20.5 and 20.6.

20.3.2 Reuse-driven Software Engineering Business (RSEB)

Jacobson et al. (1997) describe an approach to developing reusable software components that is rooted in Jacobson's OOSE and Objectory (Jacobson et al., 1992) and that uses the notation of UML Version 1.0. The approach is based on practical experience within Ericsson and Hewlett-Packard.

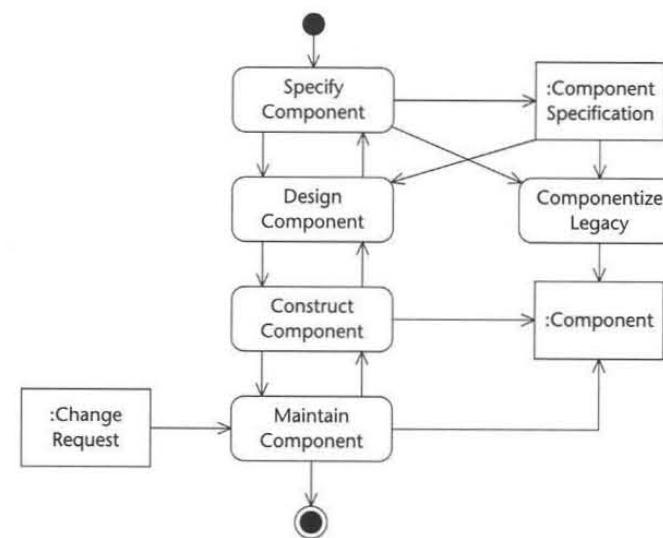


Figure 20.4 Select Perspective Supply workflow (adapted from Apperley et al., 2003).

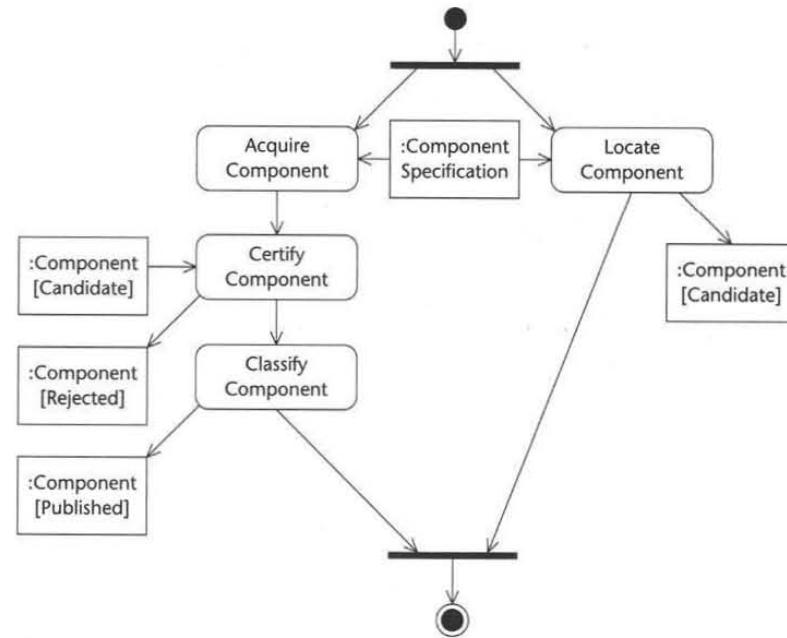


Figure 20.5 Select Perspective Manage workflow (adapted from Apperley et al., 2003).

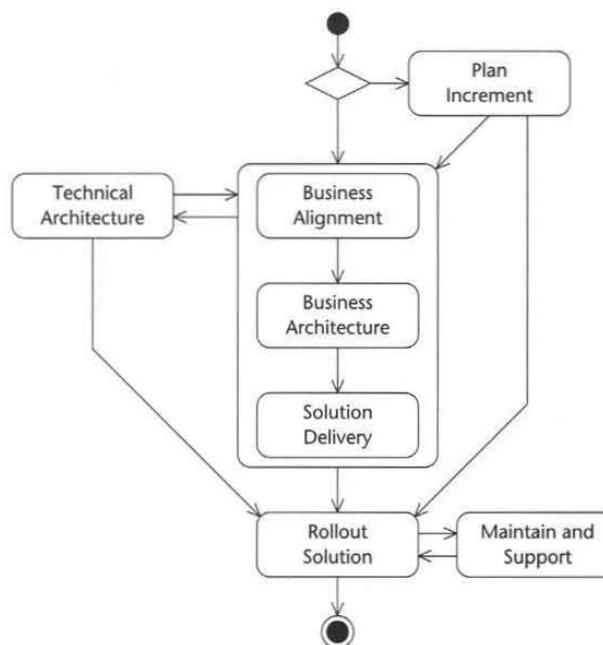


Figure 20.6 Select Perspective Consume workflow (adapted from Apperley et al., 2003).

Unlike Allen and Frost, who consider components as executables or as packages of executables designed to deliver a particular service, Jacobson et al. consider reuse in terms of any of the work products of systems development. This means that models that are produced before the finished program code are candidates for reuse, and that artefacts other than classes, for example use cases, can be reused. However, the key point of this approach is that the design of systems to make use of reusable components requires an architectural process right from the start. And that means changing the way the business operates.

In order to transform the business into a reuse business, Jacobson et al. also draw on another book from the same stable. Jacobson et al. (1995) explain an approach to business process re-engineering that is based on OOSE and Objectory. The task of developing a reuse business is a re-engineering task that can be modelled using object-oriented business engineering and that leads to the development of systems to support the RSEB. Jacobson et al. suggest that the end result is a business consisting of the following competence units: Requirements Capture Unit, Design Unit, Testing Unit, Component Engineering Unit, Architecture Unit and Component Support Unit. These competence units are groupings of staff with particular skill-sets and the business data and documents for which they are responsible.

The emphasis in RSEB is to design an architecture for systems that supports reuse from the start. This is done through three engineering processes: Application Family Engineering, Component System Engineering and Application System

Engineering.

- *Application Family Engineering (AFE)* is an architectural process that captures the requirements for a family of systems and turns them into a layered architecture, consisting of an application system and a supporting component system.
- *Component System Engineering (CSE)* is the process of focusing on the requirements for the component system and developing the use cases, analysis models and design for reusable components to support application development.
- *Application System Engineering (ASE)* is the process of developing the requirements for applications and developing the use cases, analysis models and design to produce application software that makes use of the reusable component systems developed by CSE.

The lifecycle for this kind of project is an iterative one. The engineering processes can run concurrently, with the emphasis changing as the project progresses. Model elements in the component systems are exposed to those in the application systems through façades. The Façade pattern is explained in Section 20.5. Jacobson et al. use the Façade pattern to organize components in packages and then expose those components to other packages. (We take a slightly different approach in Section 20.5.)

20.4 Commercially Available Componentware

Until recently, most commercially available components took the form of utilities or graphical user interface components. The best example of this is the wide variety of controls that are available for use with Microsoft Visual Basic. Originally these were supplied as add-ins in the form of .vbx files, which could be included in the Visual Basic toolbar in the same way as the built-in controls, or in the form of OLE (Object Linking and Embedding) objects, which allowed the functionality of other software packages such as word-processors to be embedded in applications. With the introduction of ActiveX, based on the Microsoft COM architecture, these add-ins are now available as .ocx files. If you look through the catalogue of a good software supplier that sells development tools, you will find pages of ActiveX controls that can be used in your applications and that provide the developer with the possibility of building a wide range of different functions into their software without having to reinvent the wheel. Examples include:

- serial communications
- computer-aided design
- project management including Gantt charts
- spreadsheets
- scientific charts
- barcode reading and printing.

The use of standardized mechanisms to access the functionality of these controls has meant that other software companies can also write interfaces to them, although they are mainly used in Visual Basic or Visual C++ and C#.

For applications written in Java, however, there is another mechanism that can be used: the JavaBean. JavaBeans (or just Beans) are components written in Java that typically combine a graphical element with the functionality to support it. Beans support the Component–Container model in the Java Abstract Windowing Toolkit (AWT), which means that Beans that encapsulate access to particular domain objects can be added into applications in the same way as other visual components can be added to the applications. In the Java 2 Platform Enterprise Edition (J2EE), Enterprise Java Beans (EJBs) are the standard way of implementing business logic in a Java-based application server, and provide entity beans to encapsulate domain objects and their business logic as well as session beans that act in a way similar to control objects in the MVC model.

Most of these add-in controls provide generic capabilities rather than reusable components for particular types of business operations. Suppliers are now producing reusable components for business functions. Most notable among these developments is IBM's SanFrancisco project.

The SanFrancisco project provides distributed, server-based components for different types of business processes. SanFrancisco uses a layered architecture (shown in Fig. 20.7). The Foundation layer provides a programming model to support distributed transactions, and uses a set of distributed object services and utilities written entirely in Java. It also provides an optional GUI framework written using JavaBeans. The Common Business Objects layer implements general purpose business objects, together with the facts and rules required for any business application. This includes business objects such as company, business partner, address and calendar. Four components are provided in the Core Business Processes layer at present. These are:

- general ledger
- accounts receivable and accounts payable
- warehouse management
- order management.

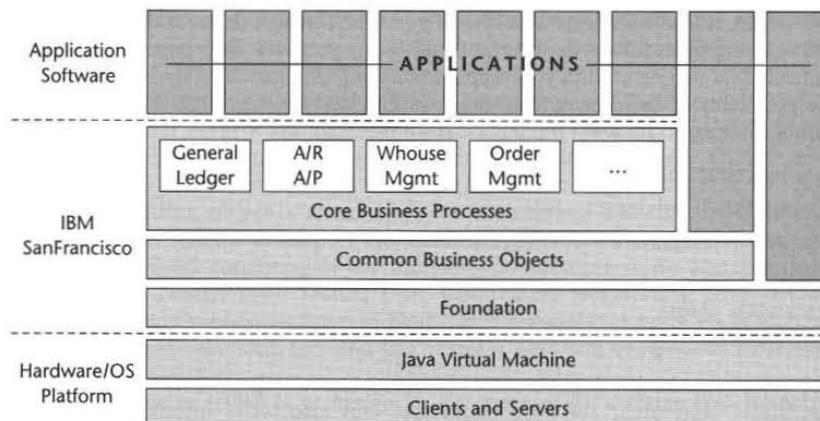


Figure 20.7 Layered architecture of the SanFrancisco project (adapted from IBM, 1998).

These have been built using design patterns, many of which were discovered as the project developed, and provide support for electronic commerce. IBM has integrated the SanFrancisco project into its WebSphere range of products for e-business development and now makes them available to business partners as Business Components for WebSphere Application Server.

20.5 Case Study Example

A common feature of many applications is the need to control the access of members of staff to the different programs that make up a system. A non-functional requirement both for Agate and for FoodCo is to restrict access of staff to the use cases that they are permitted to use. This requirement can be summarized as follows.

Each program in the system will be represented by a use case in the use case diagram. One or more actors will be associated with each use case, and each actor may be associated with more than one use case. A member of staff will fill the role of one or more actors, and each actor will be represented by one or more members of staff. Each actor will be given access rights to use specific use cases (programs). A member of staff may only use those use cases (programs) for which one of the actor roles they fill has been given an access right.

This non-functional requirement in the context of the main systems can be viewed as the basis for functional requirements in a security subsystem. This subsystem is a potential candidate for the development of a reusable component. Figure 20.8 shows the use cases for this subsystem. It can be modelled in a class diagram in the

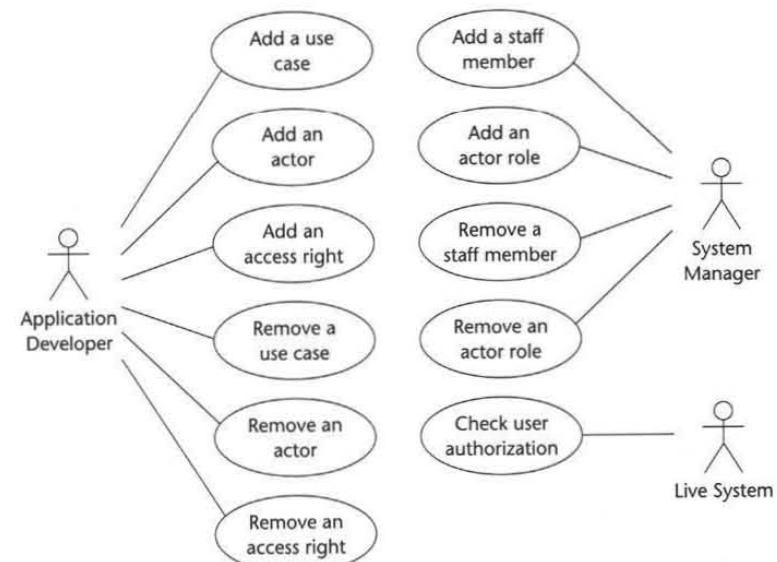


Figure 20.8 Use case diagram for security requirement.

same way as the business classes that meet the functional requirements of the system. Figure 20.9 shows the initial domain class diagram for this requirement. Two association classes, *ActorRole* and *AccessRight*, have been included in the class diagram, as it was initially thought that there might be some data associated with the creation of links between instances, for example the date that an access right was given, or the type of an access right (Read/Write/Update). However, further discussion with users and analysis of the requirements indicates that this is likely to make the subsystem more complicated than it needs to be, so they have been removed from Fig. 20.10, which shows the analysis class diagram.

There are many design alternatives for this part of the system. The particular design alternative that we choose will affect the detailed design of this subsystem. If we look at it as though we are the software company developing software for both Agate and FoodCo, some of the alternatives are as follows.

- Do we design this subsystem with a set of boundary and control classes to support the use cases in Fig. 20.8?
- Can we reuse the existing *StaffMember* class in the business domains of Agate and FoodCo (and any other companies with similar systems)? We do not really want to have to set up data about staff members in two places.
- What happens if we do reuse the *StaffMember* class in the business domain and then want to use this security subsystem to support a system that does not have *StaffMember* as an entity class?
- If this security subsystem is to be implemented for all the application software we develop, then we are going to have to make some classes in the software (presumably the control classes) aware of the interface to this subsystem. How do we extend these control classes: rewrite them, extend them or subclass them?
- How do we provide persistent data storage for this subsystem? Does it have its own storage mechanisms or will it use the same storage mechanisms as whatever application it is supporting?

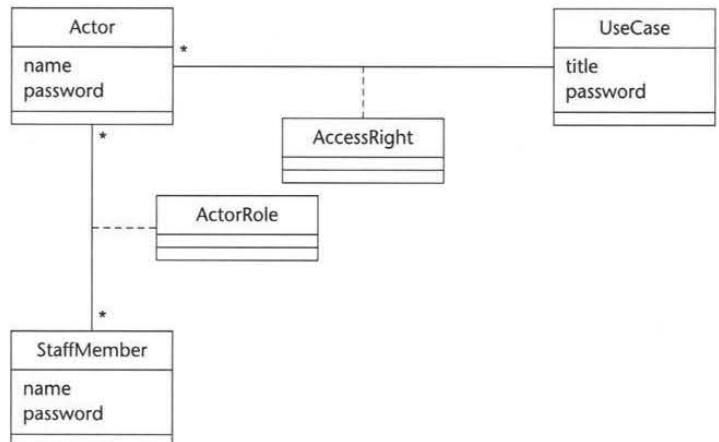


Figure 20.9 Initial domain class diagram for security requirement.

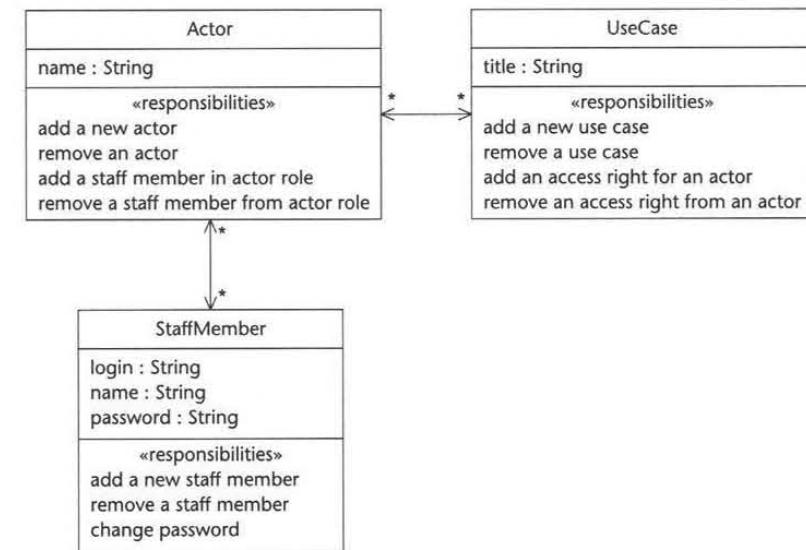


Figure 20.10 Analysis class diagram for security requirement with responsibilities.

- What parts of this subsystem are we going to make visible to other applications? Are we going to make all the classes visible or are we going to provide a single interface?

We might choose to design the system so that when a user starts running the application, they are prompted for their name or login and a password. Alternatively, if they are required to log into a network anyway, the software could obtain the user's login from the network operating system. Each time a user starts a new use case in the main application, the application will need to check with the security classes whether that user is authorized to use that particular use case.

The security requirement is not part of the business requirements of the domain applications, and we want to reuse the security software in other applications, so it makes sense to separate these classes from the rest of the software and put them in a package of their own. The security classes will require their own boundary classes, to allow the actors shown in Fig. 20.8 to carry out the use cases. These will run on client computers and will be in a separate package within the overall security package. They will have dependencies on other packages that provide these services, such as the Java AWT. We have created two packages for control classes, one for classes that will run on the clients and control the boundary classes and one for control classes that will run on the server. These control classes will have a dependency on the core security classes. Figure 20.11 shows these package dependencies. We have also shown a package to represent a business application that will be using the services of the security package to authenticate users. It is arguable whether this should have a dependency on the server control classes or on some kind of client package that hides the implementation. Whatever approach we take, we want to provide a clean interface to the functionality of the security subsystem.

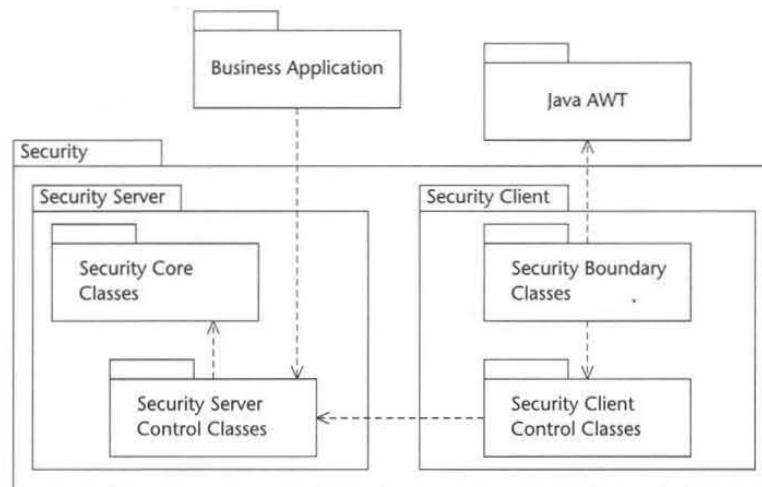


Figure 20.11 Package diagram showing security classes and dependencies.

for developers to use. It should be possible to design and implement a separate security client, which uses the interface to the security server control classes. Also, programmers should have a straightforward application programming interface (API) to the authorization service—the use case *Check user authorization*.

One way of doing this would be to replace the control classes in the *Security Server Control Classes* package with a single control class. This will make it easier for developers to reuse the package, and application programmers only need to know the API of this one class. However, that would lead to a single class with no attributes and a large number of operation implementations. An alternative approach is to leave the control classes as they are and to create a Façade class that has the operations from the control classes within the subsystem but does not contain their implementation.

This approach is based on a design pattern, called the Façade pattern. (See Chapter 15 for more on design patterns.) The Façade pattern is a Gang of Four structural pattern, and is described by Gamma et al. (1995) in the following terms.

Intent

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

...

Applicability

Use the Façade pattern when

- you want to provide a simple interface to a complex subsystem ...
- there are many dependencies between clients and the implementation classes of an abstraction. Introduce a façade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
- you want to layer your subsystems ...

(Gamma et al., 1995, pp. 185–186)

The structure of the Façade pattern is shown in Fig. 20.12. We could use this structure to add a single class, called *SecurityManager*, which provides the API to the functionality in the security package. Or we could add two separate Façade classes, one for the management of the security subsystem (adding staff members etc.), and one for the authorization service used by business applications. This is shown in the class diagram in Fig. 20.13. We have added other operations that will be required in order to support the use cases for maintaining the information in the subsystem, for example to list all the actors for a particular use case.

The control classes in the *Security Server Control Classes* package can probably be designed to be Singletons (see Section 15.4.2). Figure 20.14 shows the control classes in this package.

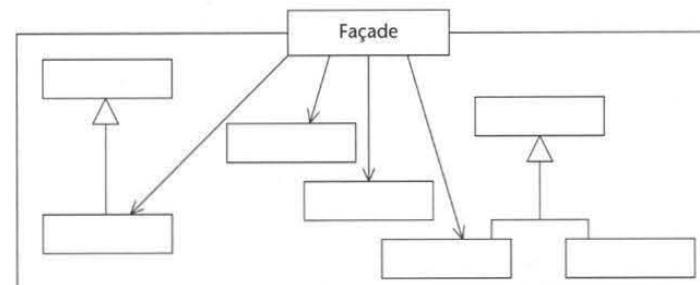


Figure 20.12 Structure of Façade pattern (adapted from Gamma et al., 1995, p.187).

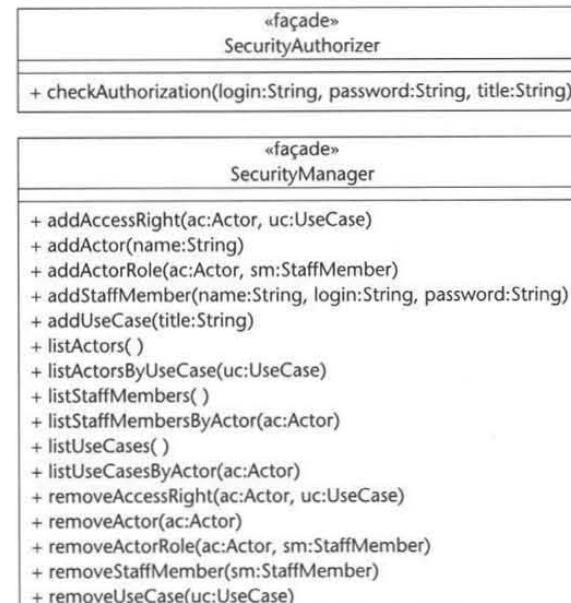


Figure 20.13 Class diagram showing Façade classes.



Figure 20.14 Class diagram showing control classes.

In order to make the security package as reusable as possible, either it needs to make use of whatever data storage mechanisms are used in the application with which it is supplied or it needs to have its own mechanism for persistent storage. The simplest approach is to provide the security package with its own persistence mechanism. We can use an object-oriented database management system such as ObjectStore PSE Pro for Java to provide a persistence mechanism without having to worry about brokers and proxies. (Alternatively, given the relatively small volumes of data that are involved, it is possible for the persistence to be provided by using a system of files. If the data is to be stored in simple files, then it should be encrypted before storage. An encryption package could be added.)

In the design in Fig. 20.15 we have added collection classes to provide entry points from the control classes to the lists of Actors, UseCases and Staff Members. We have also added hashtables as collection classes to implement the associations between the classes. Adding a link between a UseCase and an Actor means adding the Actor to the UseCase.actors hashtable and adding the UseCase to the Actor.usecases hashtable.

The collection classes support operations that have been added to the façade class and the control classes. The intention of this is that it should be easy to display in a dialogue box a list of all the Actors that currently have AccessRights to a particular UseCase etc. They are also necessary to check the authorization of staff members for particular use cases by working through the list of actors for a particular staff member, and for each actor checking whether that actor has an access right to the use case.

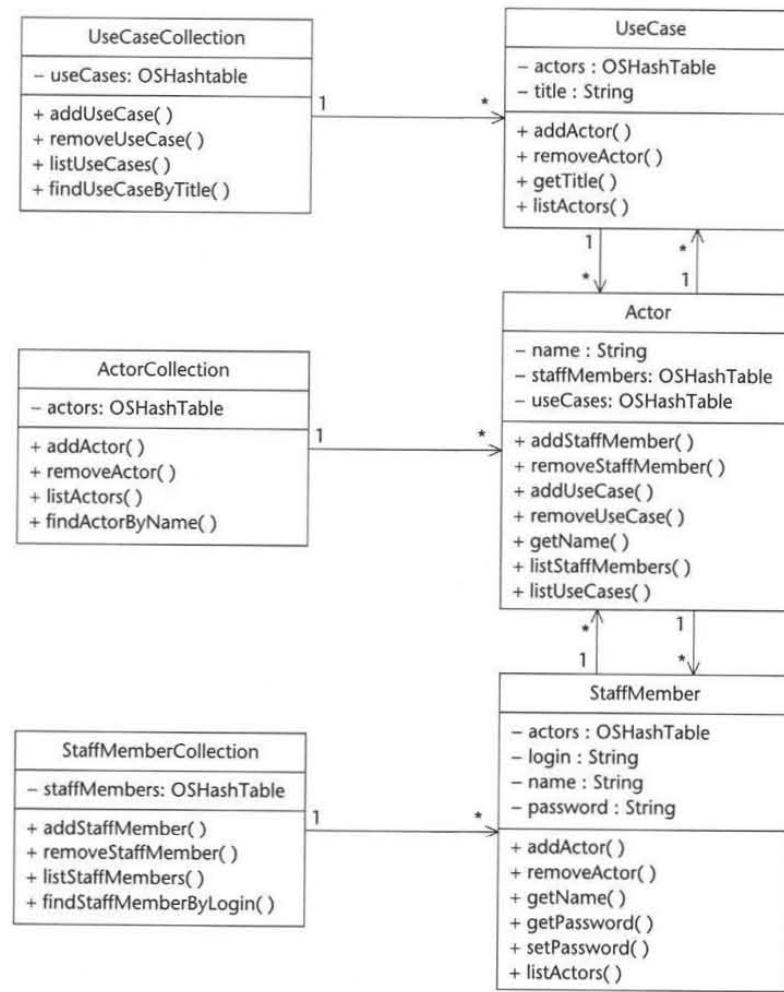


Figure 20.15 Design class diagram showing collection classes.

If we use ObjectStore then we shall need to base the collection classes on the persistent `OSHashTable` class that is provided with ObjectStore. Figure 20.16 shows the dependency between Security package and the package that contains the `OSHashTable` (`com.ODI.util`).

All the operations that update the database will require a reference to an ObjectStore database and must take place in the context of a transaction. The dependencies on the ObjectStore Database and Transaction classes (in `com.ODI` and `com.ODI.imp` respectively) are also reflected in the dependencies of Fig. 20.16.

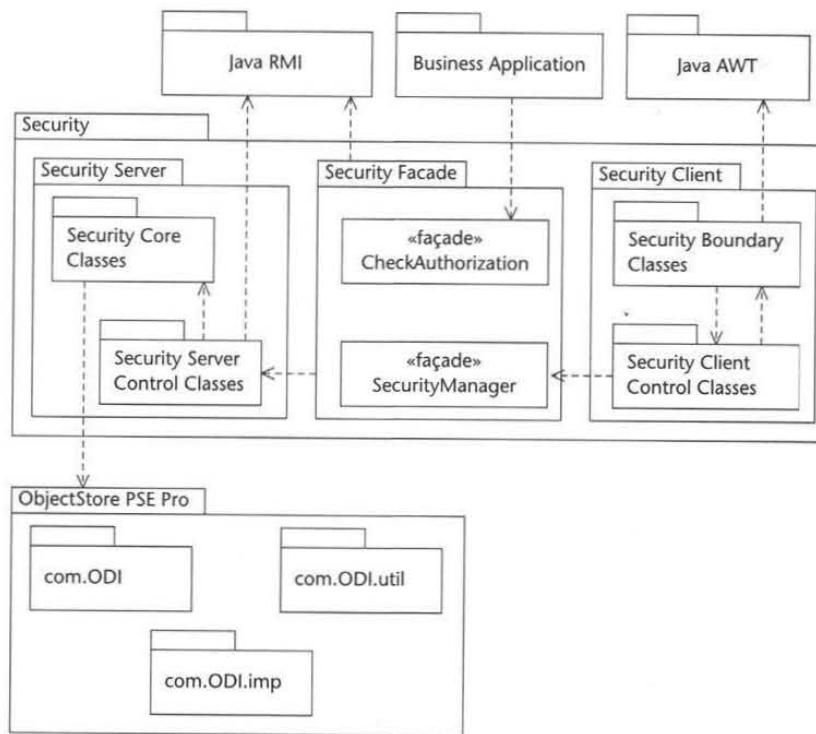


Figure 20.16 Package diagram showing dependencies of the Security package.

If we use Java Remote Method Invocation (RMI) to allow client packages to connect to the Security package, then we also require the dependency on the Java RMI package, which is also shown in Fig. 20.16. We have also shown the façade packages in a separate Security Façade package.

Clearly, the security package itself should not be accessible to unauthorized users, so there will be a requirement for the security package to use its own services

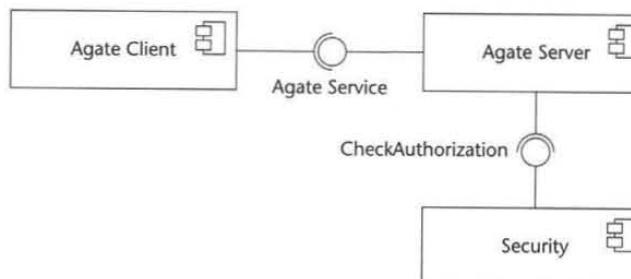


Figure 20.17 Component diagram showing dependency on the Security component.

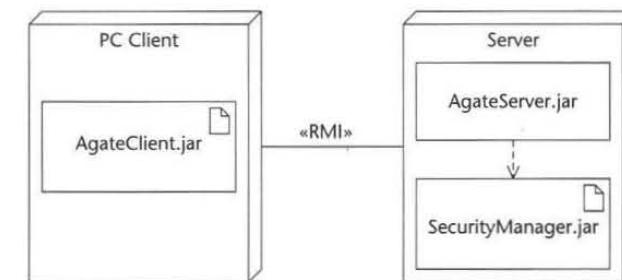


Figure 20.18 Deployment diagram showing the `SecurityManager.jar` artefact.

to restrict access to the developers of the system, who will set up the associations between use cases and actors, and to the system manager who will authorize staff within the company to use particular use cases by linking them to specific actor roles.

The Security component will provide an interface to the Agate Server component that allows it to check security authorizations, as shown in Fig. 20.17.

The security package could be implemented in Java, compiled and stored in Java Archive (.jar) files. This can be shown in a deployment diagram, as in Fig. 20.18, which shows the dependency of the Agate server classes (also stored in a .jar file) on the `SecurityManager.jar`. This component can be used in both the Agate and FoodCo projects and in other future projects.

20.6 Summary

Adopting object-oriented software development is not on its own a sufficient cause for a business to benefit from the reusability that is claimed as one of the benefits of object-orientedness. Businesses thinking of taking advantage of reusability in object-oriented software need to take into account a number of factors.

- Is the organization the right kind to be able to develop reusable components as well as use them?
- How will the organization systematically move from a situation of little or no reuse to one in which reuse is built into the design of systems from the start?
- What organizational support systems and software tools are required to enable reuse?

Given a commitment to the introduction of a reuse business, an organization needs to re-engineer its software development operations to provide the structures and support for a reuse culture and to train developers in this approach. For reuse to be effective, it should not be left until the maintenance stage but should be planned into the project from the outset. The architecture of the systems should be designed from the start to support reuse.

The SELECT Perspective is one methodology that focuses on techniques and project planning to achieve reuse. The RSEB of Jacobson et al. is another approach, and one that advocates an approach based on re-engineering the

business and designing architectures for reuse. This approach is based on three processes, which produce a layered architecture, separating the component layer from the application layer, engineer the component layer and build the application layer on top of the component layer.

The introduction of CORBA, Java and web services makes it easier for developers to produce and distribute reusable software components. The Reusable Asset Specification provides a standard for packaging up the artefacts that comprise a reusable component. Commercially available components include the products of IBM's SanFrancisco project, which provides business components for general ledger, for accounts receivable and payable and for warehouse and order management. The Façade pattern provides a means for hiding the complexity of components from application programmers by means of classes that handle the API to the classes in the component.

Review Questions

- 20.1** What are the benefits of reusable components?
- 20.2** What are some of the obstacles to reuse?
- 20.3** Give Jacobson et al.'s definition of a 'component'.
- 20.4** Name three mechanisms for creating reusable components in different programming languages.
- 20.5** What are the three workflows in the revised Select Perspective (Apperly et al., 2003)?
- 20.6** How does Allen and Frost's definition of a component differ from that of Jacobson et al.?
- 20.7** What are the three processes in Jacobson et al.'s approach to reuse, and what is meant by each?
- 20.8** What is the purpose of the Façade pattern?

Case Study Work, Exercises and Projects

- 20.A** Describe the mechanisms that are available in an object-oriented language with which you are familiar for creating reusable components.
- 20.B** A data encryption package is required to be added to the security package to provide services to encrypt different data types, such as the Integer, String and Date classes. Draw a class diagram (with packages and classes) to show new classes CryptInteger, CryptString and CryptDate. Access to Encryption and Decryption control classes is to be provided by a single Façade class called EncryptionManager. Include these in your diagram in suitable packages.
- 20.C** Redraw the package diagram in Fig. 20.13 to include the encryption package and add any new dependencies necessary.
- 20.D** Draw a deployment diagram for FoodCo showing the security packages as artefacts, some of which will be on client machines and some of which will be on a server.

20.E Complete the design of the security package and implement it in an object-oriented language such as Java, Smalltalk, C# or C++. (This is suitable for a coursework assignment or small project.)

Further Reading

Allen and Frost (1998) introduced the SELECT Perspective, while Apperly et al. (2003) have brought it up to date. Jacobson et al. (1997) present a different view of developing a reuse-driven software development process, which builds on the Objectory and object-oriented business re-engineering approaches of Jacobson's other books. It also uses UML as a notation.

Cheesman and Daniels (2001) also provide a straightforward process for specifying component-based systems.

For a comparison of the mechanisms involved in CORBA and DCOM (and some other approaches to distributed systems), Orfali and Harkey (1998) provide a clear and readable coverage, which includes detailed instructions on how to implement systems which use these techniques.

Daniels et al. (2004) provide an introduction to building web services using Java and explain the related specifications such as UDDI and WSDL.

To find the wide range of software add-ins that are available as ActiveX controls, look at the website of a supplier such as www.devdirect.com.

Software Development Processes

Learning Objectives

In this chapter you will learn

- how ‘software development process’ relates to ‘method’ and ‘methodology’
- why methodologies are used
- some characteristics of different methodologies
- the difference between hard and soft methodologies
- some of the issues in choosing an appropriate methodology.

21.1 Introduction

Our focus so far has been on describing the techniques used during the analysis and design of an object-oriented information system. Underpinning this is an implicit assumption that projects move from one task to another, sometimes in sequence, sometimes looping back to repeat a task if it requires further work, sometimes choosing between alternative ways of proceeding, but always with a particular purpose in mind, and always under someone’s control. In practice, developers need guidance on the many questions that arise when the techniques are applied in a real-life situation. The use of a set of modelling or documentation standards such as that provided by UML has an important part to play, but is not enough on its own. There is a need for a higher level view of the system development process, which organizes and manages the many different activities. This need is partly met by systems development methodologies.

In this chapter we describe what is meant by ‘process’, ‘method’ and ‘methodology’, and explain why methodologies are so widely used and why they continue to evolve (Section 21.2). Next, some representative methodologies are described, including the USDP, on which we based the development process followed in this book (Section 21.3), Participatory Design (Section 21.4), the Dynamic Systems Development Method (Section 21.5) and Extreme Programming (Section 21.6).

We then introduce some of the considerations in selecting a methodology that is suited both to the context and to the kind of project to be carried out (Section 21.7). Finally, we explain the distinction between the so-called soft and hard views of systems development (Section 21.8).

21.2 Process, Method and Methodology

The techniques of software system development must be organized into an appropriate development process if they are to work together. For example, once an analyst has constructed communication diagrams for the main use cases, should the next steps be to convert these into sequence diagrams and write operation specifications, or should he or she now concentrate on preparing a class diagram and developing inheritance and composition structures? All of these tasks need to be completed at some point, but how is the analyst to know which is more appropriate at this specific point in the project? UML itself contains nothing that helps to make this decision.

The organization of tasks is not contained within the techniques themselves, and must be described at a higher level of abstraction. The *method* or *process* of a project refers to the particular tasks carried out and how they are organized over the project lifecycle.

The words ‘method’ and ‘methodology’ are used interchangeably by many authors, but their meanings actually differ in a significant way. A method is a step-by-step description of the steps involved in doing a job. Since no two projects are exactly alike, any method is specific to one project. A methodology is a set of general principles that guide a practitioner or manager to the choice of the particular method suited to a specific task or project. In object-oriented terms, we could say that a methodology is a type while a method is its instantiation on a particular project.

To complicate the picture further, the term ‘process’ has come to be used as a synonym for methodology, as well as for method. For example, in both the USDP and the IBM Rational Unified Process, the term ‘process’ has an all-embracing meaning that includes both the particular development activities and techniques used on a given project, and the overall framework that describes what the tasks are, how they are carried out and how they are organized. This seems surprising, since the authors who promote this confusion of terminology are all committed to object-orientation and would presumably not confuse an instance with its class, nor a class with its type.

Wherever possible, we use the terms method and methodology as described above, to distinguish between the process of a single project and the general principles or framework on which that process is based.

Software developers, project managers and their bosses all need to be able to think at different levels of abstraction, depending on the task in hand. In order to run a current project plan, a manager must think at the level of method. In order to plan and organize for the next project, he or she must also be able to think at a higher level, that of methodology. Figure 21.1 summarizes some of the different levels of abstraction involved in software development.

Increasing level of abstraction	Example of application	Typical product
Task	Developing a first-cut class diagram for FoodCo	A specific version of the FoodCo class diagram
Technique	Description of how to carry out a technique, e.g. UML class modelling	Any UML class diagram
Method	Specific techniques used on a particular project (e.g. FoodCo use cases, class model, communication diagrams, etc.) that lead to a specific product	FoodCo's product costing system
Methodology	General selection and sequence of techniques capable of producing a range of software products	A range of object-oriented business applications

Figure 21.1 Increasing level of abstraction in software development.

21.2.1 Methodology

A methodology in the domain of IS must cover a number of aspects of the project, although coverage varies from one to another. Avison and Fitzgerald (2002) describe a methodology as a collection of many components (not in the everyday sense of the word). Typically, each methodology has procedures, techniques, tools and documentation aids that are intended to help the system developer in his or her efforts to develop an information system. There is usually also a lifecycle or structure that contains and organizes the procedures. Finally, there is some kind of underlying philosophy that captures a particular view of the meaning and purpose of information systems development. (Note that, according to this view, process is merely one aspect of a methodology.)

Checkland (1997), in a conference address that discussed the potential contribution of the wider systems movement to information systems development, gave a more general definition that captures well the notion of methodology as a guide to method. In his view, a methodology is a set of principles that in any particular situation has to be reduced to a method uniquely suited to that particular situation.

To give some examples of these aspects:

- The UML class diagram is a technique, and so is operation specification.
- IBM Rational Rose is a tool. So, too, is the IBM Rational Unified Process software, which provides information and guidance about the process.
- The activity represented by 'find classes by inspecting the use case descriptions' is an aspect of process. So is the advice that an analyst is usually the best person to write test plans.
- The advice that 'operation specifications should not be written until the class model is reasonably stable' is an aspect of structure, as it identifies a constraint on the sequence in which two steps should be performed.
- Analysis and design can (reasonably) be viewed as distinct procedures.
- The statement 'object-oriented development promotes the construction of software which is robust and resilient to change' is an element of a systems development philosophy.

A package that contains enough information about each of these aspects of the overall development process is fit to be named a methodology. Many attempts have been made to capture the essence of methodology for software development, and the resulting methodologies are almost as varied as are the projects themselves. In practice, methodologies vary widely in philosophy, in completeness of definition or documentation, in coverage of the lifecycle and even in the type of application to which they are best suited.

One example of an attempt to define a metamodel for software development is the OMG's Software Process Engineering Metamodel (SPEM). This uses UML notation for entities that include *roles*, *activities*, *products* and *phases*. The software process followed on a specific project is modelled at the least level abstract level and could be represented by instances of real workers, activities and actual modelling products. At the next more abstract level come specific methodologies such as RUP (in SPEM they are called processes). The abstract features of methodologies are themselves modelled by SPEM in UML, as a process metamodel, and these in turn are derived from the still more abstract Meta Object Facility (MOF). The assumptions behind SPEM are in close agreement with the definitions given earlier in this section, although the concerns of the authors of SPEM differ in at least one respect. Part of the underlying rationale of SPEM, as also of MDA, is to move towards automating the process of software development. In order to achieve this, it is necessary to define activities, products and so on with rather more rigour than has usually been needed in the past.

Some authors believe that the idea of a published methodology is misleading, if it is taken to mean that following a methodology leads inevitably to a good product. For example, Daniels (an early and influential contributor to UML) argued in a conference presentation (1995) that an IS methodology is more a means for learning the process of systems development than a recipe for practising it. He compared a methodology to a ladder that, once used, can be thrown away (the metaphor is borrowed from Wittgenstein). We are safe to 'throw the ladder away', not because we do not need one, but because, having once climbed it, we know how to build one of our own that is better suited to our needs. In Daniels's view, the skill and judgement of an experienced developer count for much more than the prescriptive steps of a methodology, however comprehensive the latter may be.

Note that Daniels's position expresses two key principles of the Agile Manifesto: '**Individuals and interactions** over processes and tools' and '**Responding to change** over following a plan' (the Agile Manifesto was introduced in Chapter 3). As we shall see in the following sections, the recent agile movement has much in common with earlier efforts to build software in a way that is responsive to changing circumstances and to individuals.

We believe that there is much in common between the various approaches we describe in the following sections, despite their very different historical antecedents, and some very real differences in approach. The common features could be summarized as follows.

- Avoid getting bogged down in documentation.
- Be flexible in approach when circumstances change.
- Devolve control to project team level as far as possible.

- Maintain creativity; do not just follow a rigid, dogmatic process.
- Involve users in a way that shares ownership of both the problem and the solution.
- Maximize the productivity of the team.

This seems like such obvious common sense that it may be hard to understand why common sense on its own is not enough for software development to be successful. However, as we saw in Chapter 2, most reasons for a project to run into difficulty lie in the human aspects, and managing these will always require great skill and care.

21.2.2 Why use a methodology?

Over many decades, IS methodologies have been developed and introduced specifically to overcome those problems of software development projects that were perceived to be important at the time. However, to date, no methodology has been wholly successful in fulfilling its objectives, partly because computing is a highly dynamic field, and the nature of both projects and their problems is constantly changing. In a changing world, it is unlikely that yesterday's solution will ever completely solve today's problems.

Nevertheless, many advantages have been claimed for the use of a methodology, including the following.

- The use of a methodology helps to produce a better quality product, in terms of documentation standards, acceptability to the user, maintainability and consistency of software. We believe this to be generally true, even if only the minimum necessary set of documentation is maintained—as is recommended by DSDM, for example.
- A methodology can help to ensure that user requirements are met completely. This does not necessarily imply that all requirements must be captured, documented and agreed in advance of any development work.
- Use of a methodology helps the project manager, by giving better control of project execution and a reduction in overall development costs.
- Methodologies promote communication between project participants, by defining essential participants and interactions and by giving a structure to the whole process.
- Through the standardization of process and documentation, a methodology can also encourage the transmission of know-how throughout an organization.

If all these claims could be fulfilled in practice, the benefits would be clear. However, the evidence is mixed, partly because of the many dissimilarities between organizations, and also between the different types of project within any one organization. The picture in the UK at present is that approximately two thirds of businesses use some form of IS methodology. Even where this actually means little more in practice than an in-house set of standards for documentation and procedures (as it certainly is in at least some cases), this still shows a consistent level of faith in the utility of a methodology of some kind.

21.3 The Unified Software Development Process

We have already briefly introduced the main principles that underlie the USD^P (often now just called the UP) in Chapters 3 and 5. In the following section, we present a more detailed picture of the methodology.

Philosophy and principles. This part of the UP should be familiar to anyone who has read much of the rest of this book, since the development process that we have followed is partly based upon, and broadly consistent with, the UP. In particular, the UP is a use-case driven, architecture-centric, iterative and incremental process. These terms can be explained briefly as follows.

The starting point for all modelling is some sort of interaction, called a use case, between a user and the software system under consideration. This interaction is the beginning of the modelling activity and also the fundamental unit from which later models are derived. Use cases are thus important in several different ways. Each use case is a thread that links a series of models from requirements to implementation; it is also a unit of delivery that has practical significance to users; it is a constant reminder to the systems developers that only the users' requirements really matter.

In the UP, the resulting software architecture is an essential theme in modelling from the earliest stages of a project. This is reflected in the stereotyping of the classes that contribute to realizing a use case as boundary, control and entity classes.

Phases and workflows. Figure 21.2 repeats a diagram shown earlier in Chapter 5. This illustrates the relationship between the phases, iterations and workflows of the

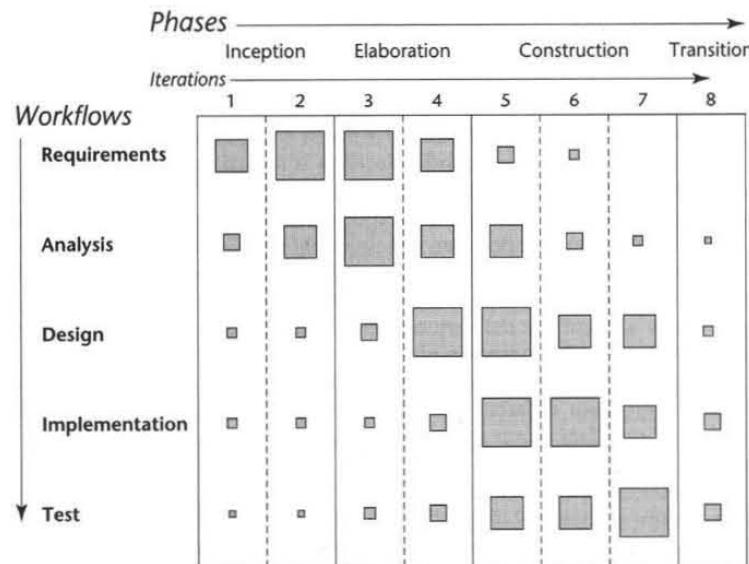


Figure 21.2 Phases, iterations and workflows in the Unified Process.

UP. We do not need to dwell on workflows (requirements, analysis, design and so on) here, beyond noting that they are made up of activities, since these are the main subject of the greater part of this book. We explain the UP view of activities a little later in this section. In the paragraphs that immediately follow, we explain the phases of the UP and relate these to the activities that are carried out within each phase.

While an activity is something that has particular meaning for the developers who carry it out, a *phase* is considered primarily from the perspective of the project manager. He or she must necessarily think in terms of milestones that mark the progress of the project along its way to completion. In the UP, for example, two key milestones are Initial Operational Capability (which marks the close of the Construction phase) and Product Release (which marks the close of the Transition phase).

Phases are sequential. A project passes through each phase in turn and then (usually) moves on to the next. The end of a phase is a decision point for the project manager. When each phase is complete, those in charge must decide whether to begin the next phase or to halt development at that point. The focus of the manager's attention shifts as the project progresses from one phase to the next.

Within each phase, the activities are carried out in an iterative manner that could be summed up in a very simplistic way as follows:

- Do some investigation, model the requirements, analyse them, do some design, do some coding, test the code, then repeat the whole process.

There is no set rule that states how many iterations should be conducted within a phase; this is a matter for the project management team to judge, depending on the project characteristics and the available resources. (There is more information about managing object-oriented projects on the book website.)

Within each phase, the workflows are essentially the same. All four phases include the full range of workflows from requirements to testing, but the emphasis that is given to each workflow changes between the phases. In the earlier phases, the emphasis lies more on the capture, modelling and analysis of requirements, while in the later phases the emphasis moves towards implementation and testing.

During the inception phase, the essential decision is that of assessing the potential risks of the project in comparison with its potential benefits. This judgement of project viability (or otherwise) during the inception phase resembles the feasibility stage of a waterfall lifecycle. The decision will probably be based partly on a similar financial assessment (typically some sort of cost benefit analysis). One principal difference at this early stage is that the viability of a UP project is much more likely to be judged partly also on the delivery of a small subset of the requirements as working software. During the inception phase, the main activities are thus requirements capture and analysis, followed by a small amount of design, implementation and testing. Another major difference is that, even at this early stage, there is the likelihood of iteration. That this is even possible is due to the fact that the development approach is object-oriented.

During the elaboration phase, attention shifts to the reduction of cost uncertainties. This is done principally by producing a design for a suitable system that demonstrates how it can be built within an acceptable timescale and budget. As the emphasis shifts towards design, the proportion of time spent on design activities

increases significantly. There is a further small increase in the time spent on implementation and testing, but this is still small in relation to the analysis and design activity.

The construction phase concentrates on building, through a series of iterations, a system that is capable of satisfactory operation within its target environment. Implementation and testing rapidly become core activities in this phase, with a move further away from design and towards testing as each iteration gives way to the next.

Finally, the transition phase concentrates on achieving the intended full capability of the system. This deals with any defects or problems that have emerged late in the project. It could also include system conversion, if an older system is being replaced (see Chapter 19).

Workers and activities. The UP differentiates between the real people who are involved with any project, such as users, analysts, managers and customers, and the more abstract *worker*. This term denotes someone who plays a specified part in carrying out an activity. Some examples of workers are: use-case specifier, system architect, component engineer and integration tester. There need not be a direct one-to-one mapping between people and workers. An employee may play the part of different workers at different times, and, conversely, a group of people could represent a single worker engaged on an activity.

Most UP activities can be partially defined in terms of the workers who carry them out, and the artefacts that either serve as inputs or are produced as outputs. Figure 21.3 illustrates this for the activity *Analyse a use case* (this can be compared with the process that we follow in Chapter 7).

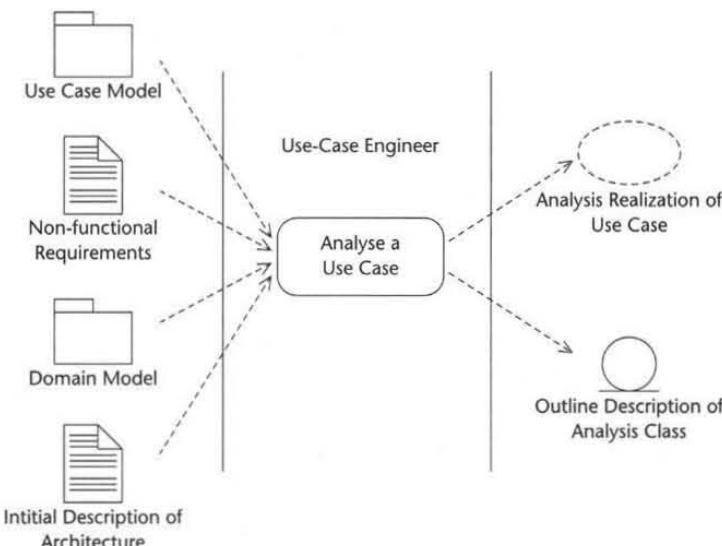


Figure 21.3 Inputs and outputs of the activity *Analyse a use case* (adapted from Jacobson et al., 1999).

As we mentioned above, a workflow can be seen as a flow of activities. Since each activity can be related to a worker who will carry it out, we can identify which workers will need to participate in the project. Figure 21.4 shows the analysis workflow broken down into its constituent activities.

Artefacts. By now, it should be reasonably clear what the main artefacts are in the UP. These will clearly include models, such as the Use Case Model or the Design Class Model, and products, such as an implementation package or subsystem. However, Jacobson et al. (1999) define an artefact very broadly as almost any kind of information that is created, used or amended by those who are involved in developing the system. This means that the term also covers small-scale things, such as a class specification or an operation signature, and transient things, such as notes and prototypes.

Summary. At the time of its release, the UP represented the most mature object-oriented methodology that had yet been released. In large part, this is due to its ancestry. Rosenberg and Scott (1999) comment that the Booch method provided good methods for detailed design and implementation, OMT had particular strengths in its tools for exploring the problem space and Objectory was particularly strong in its approach to defining the solution space, but that all three ingredients are necessary for overall success. The UP strove to bring these together.

Many aspects of the UP follow almost inevitably from its basis in object-orientation. For example, no object-oriented approach would be taken seriously if it did not follow an iterative lifecycle.

However, in at least one respect the UP is distinct from most other object-oriented methodologies that preceded it, and we regard this as probably its greatest weakness. As methodologies go, the UP is large and complex. There will inevitably be a significant learning curve involved wherever the UP is adopted within an organization. The same is also true for its proprietary sibling, the IBM Rational Unified Process. However, IBM RUP differs significantly in the level of documentation, tool support and other guidance available to the adopter. These make the latter an industrial strength methodology, which continues to evolve today.

Our experience suggests that complex methodologies tend to be fully adopted only in organizations with sufficient resources to provide thorough training, and with the culture to impose relatively strict discipline. Many software development organizations take from a methodology only what is easiest to implement. As a

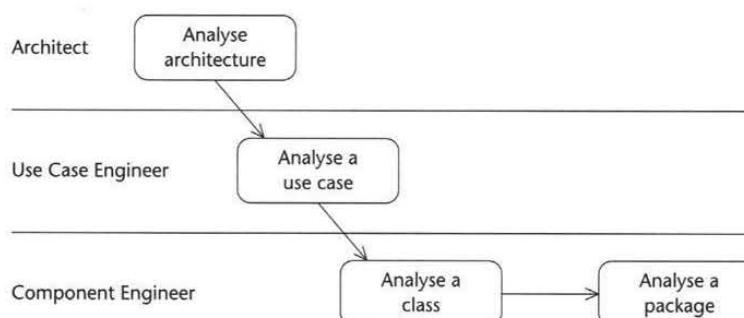


Figure 21.4 The analysis workflow in the UP (adapted from Jacobson et al., 1999).

historical example, the similarly complex SSADM (Structured Systems Analysis and Design Method) was mainly successful in the UK public sector, where it was adopted as a Government standard. Some organizations that claimed to follow SSADM actually used its notation and little else.

The complexity of the UP undoubtedly derived from a natural desire to retain the best features of the three contributing methodologies. But this is unlikely to encourage the adoption of the UP in any complete form. In some cases, as with SSADM, this may amount to no more than the use of UML and some form of iterative lifecycle. Such a response would be both unfortunate and unnecessary. The development approach that we have advocated in this book provides one example of a subset of the UP, but many other approaches are possible that adhere to the spirit of the UP without slavishly following its every detail.

21.4 | Participatory Design

Participatory Design (PD) is the name given to a collection of approaches to information systems development that share a guiding ethos more than they share any particular tools or techniques. Sometimes known as co-operative design, PD should perhaps be seen as a movement rather than as a methodology. In this, as in a number of other ways, it resembles the later Agile Alliance. The common ethos of PD is based on the assumption that active involvement of users in the design and development activity is critical to the success of an information system. This is because a successful design for an information system is said to rely just as much on knowledge and understanding of the work that is to be supported as it does on knowledge of the possibilities and limitations of the available technology. In practice, most approaches to PD assume that users will be actively involved throughout the development lifecycle, rather than in just the design activity. The approach has distinctly Scandinavian roots and can be related back to the workplace democracy movement of the 1960s. This had a particular impact on industrial relations, and subsequently also on information systems development theory and practice, in the Scandinavian countries.

We include the approach in this chapter because of its many points of contact with the development approach advocated in this book.

- Some approaches to PD, particularly that of Kyng and his collaborators (see, for example, Bødker et al., 1993) emphasize the use of use scenarios. In many respects these resemble the use cases of UP and the stories of XP, although there are differences in granularity (the use scenario corresponds more closely to a set of related use cases or stories). In Chapters 6 and 7 we introduced a use-case-driven approach to requirements modelling and analysis.
- PD approaches usually emphasize prototypes and storyboards, often low-technology mock-ups drawn on paper or cardboard, for requirements capture. This has some similarity with the planning game in XP. In Chapter 6 we presented storyboards and prototypes, both hand-drawn and computer-displayed, as important elements in establishing requirements and exploring alternative approaches to their realization.
- The typical PD lifecycle is experimental and iterative in nature, in recognition of the fact that design is always a learning experience for all participants. In

Chapter 5, and in most subsequent chapters of this book, we have stressed the significance of an iterative approach to object-oriented development.

- The active participation of users in design and development is fundamental to all PD approaches, usually throughout the project lifecycle. An iterative approach to object-oriented development naturally encourages the involvement of users at many stages of the lifecycle. In the following section, we introduce DSDM as a methodology that is thoroughly compatible with object-oriented development. Like PD, DSDM insists on the active involvement of users in the project team, and so, too, do XP and the agile methods in general.

Many styles of participative design have been proposed and practised, both in Western Europe and in North America. We try to give an understanding of the movement as a whole through a focus on only one example, the co-operative design approach developed by Morten Kyng at the University of Aarhus in Denmark. This is shown graphically in Fig. 21.5.

The diagram illustrates a possible sequence of activities within a project and also shows the relative degree of responsibility for each activity that lies with users and with developers. Most activities are shared to some extent between the two, but we can see that some rely chiefly on the contributions of developers (for example,

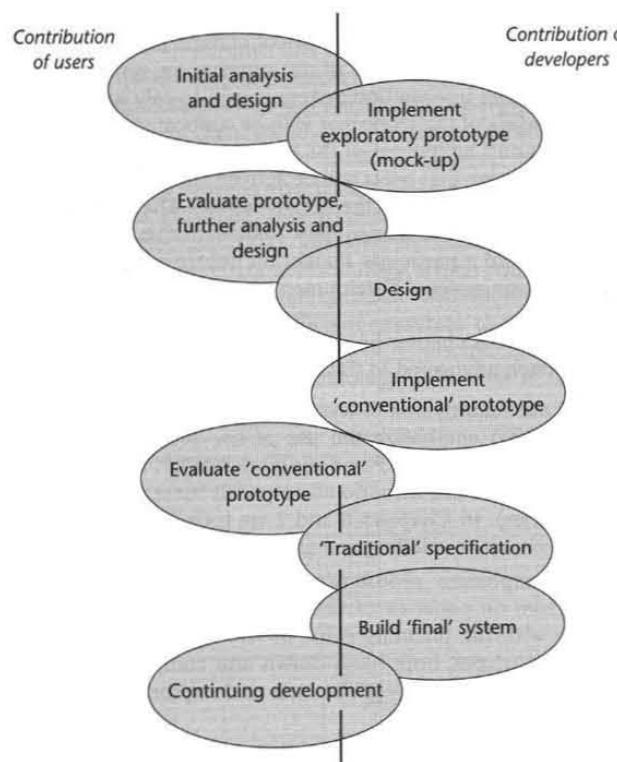


Figure 21.5 Illustrative lifecycle for co-operative design (adapted from Kyng, 1995)

building the prototypes) while others rely primarily on the contribution of users (for example, evaluating the prototypes).

While Fig. 21.5 might seem at first sight to suggest a waterfall lifecycle, a more careful reading reveals that in fact the underlying lifecycle is iterative. Two cycles of prototyping are shown followed by a final system build, but this can just as easily be interpreted as three cycles of iterative development. The approach lends itself equally well to any number of iterations and also to incremental approaches to delivery. The contribution of PD approaches to RAD, and to DSDM in particular, is quite clear.

Part of the background to PD is the belief among its proponents that the models built during the analysis and design of a proposed system fall into two general categories.

The first category includes representations of the work that is done by people who will become users of the system. In the Agate case study, the use case Assign staff to work on a campaign falls into this category. A problem can occur with models that fulfil this purpose (of representing work). Sometimes a user's lack of familiarity with the modelling language can cause misunderstanding about what a model is meant to convey. For this reason it is important that users themselves play a major role in constructing the models.

The second category includes representations of the system that is being designed. In the Agate case study, the sequence diagram for the same use case falls into this category. A similar problem can occur with models that fulfil this purpose (of representing the system that is being designed). Only a user can really tell whether a given design fully takes into account all requirements, together with the constraints and limitations that are imposed by work practices, working environment, technology and so on. Thus the activities of system design also require active user participation.

In practice, since it is important for both users and developers that their mutual knowledge and understanding should grow as the project progresses, it makes sense for both groups to share responsibility for the whole project from inception to completion.

21.5 | Dynamic Systems Development Method

The Dynamic Systems Development Method (DSDM) is a management and control framework for rapid application development (RAD). The distinction between RAD and prototyping is sometimes blurred. A RAD approach aims to build a working system rapidly while a prototyping approach also builds rapidly, but usually only produces an incomplete system, often aimed at identifying or confirming requirements. Since both approaches aim to build software quickly, similar development environments are used. Indeed, one approach to prototyping continues the development of a prototype incrementally until it becomes a working system. In effect this is a RAD development approach.

RAD became popular in the early 1990s, largely due to the perceived deficiencies of the traditional waterfall approach. For example, business managers were frustrated by the time taken to deliver a working system and the inflexibility of waterfall approaches to requirements change. Yet iterative approaches can also

be problematic, as it is sometimes difficult to cease the iterations when they become unproductive.

The DSDM consortium was formed in 1994 to produce an industry standard definition of the RAD process, and the first version was released in January 1995. This defines the structure and controls to be used in a RAD project but does not specify a development methodology. As a result, DSDM may be used comfortably alongside object-oriented development techniques, although it does not require them.

One of the key innovations of DSDM is its perspective on project requirements. Instead of seeing the requirements as fixed, and then attempting to match resources to the project, DSDM fixes the resources for the project (including the time available for completion) and then sets out to deliver only what can be achieved within these constraints.

DSDM is based upon nine underlying principles (DSDM Consortium/Stapleton, 2003).

- Active user involvement is regarded as imperative. Many other approaches effectively restrict user involvement to requirements acquisition at the beginning of the project and acceptance testing at the end of the project. In DSDM users are members of the project team and include one known as an 'Ambassador' user.
- DSDM teams are empowered to make decisions. A team can make decisions that refine the requirements and possibly even change them without the direct involvement of higher management.
- The focus is on frequent product delivery. A team is geared to delivering products in an agreed time period and it selects the most appropriate approach to achieve this. The time periods are known as timeboxes and are normally kept short (2 to 6 weeks). This helps team members to decide in advance what is feasible. Products can include analysis and design artefacts as well as working systems.
- The essential criterion for acceptance of a deliverable is fitness for business purpose. DSDM is geared to delivering the essential functionality at the specified time.
- Iterative and incremental development is necessary to converge on an accurate business solution. Incremental development allows user feedback to inform the development of later increments. The delivery of partial solutions is considered acceptable if they satisfy an immediate and urgent user need. These solutions can be refined and further developed later.
- All changes made during development are kept reversible. If the iterative development follows an inappropriate development path then it is necessary to return to the last point in the development cycle that was considered appropriate. Changes are limited within a particular increment.
- Requirements are initially agreed at a high level. Once requirements are fixed at a high level they provide the objectives for prototyping. The requirements can then be investigated in detail by the DSDM teams to determine the best way to achieve them. Normally the scope of the high-level requirements is not changed significantly.

- Testing is integrated throughout the lifecycle. Since a partially complete system may be delivered, it must be tested during development, rather than after completion. Each software component is tested by the developers for technical compliance and by user team members for functional appropriateness.
- A collaborative and co-operative approach between all stakeholders is essential. The emphasis here is on the inclusion of all stakeholders in a collaborative development process. Stakeholders not only include team members, but others such as resource managers and the quality assurance team.

It is easy to see common ground between DSDM and the UP-based development process that we have followed through this book. In particular, we have shared DSDM's emphasis on iterative and incremental development, its stress on product delivery and its focus on fitness for business purpose.

21.5.1 The DSDM lifecycle

The DSDM lifecycle has five phases:

- feasibility study
- business study
- functional model iteration
- design and build iteration
- implementation.

The relationships between the phases are shown graphically in Fig. 21.6 and each is described in more detail below. Note that the last three are actually iterative processes, but for the sake of clarity this is not shown explicitly in Fig. 21.6.

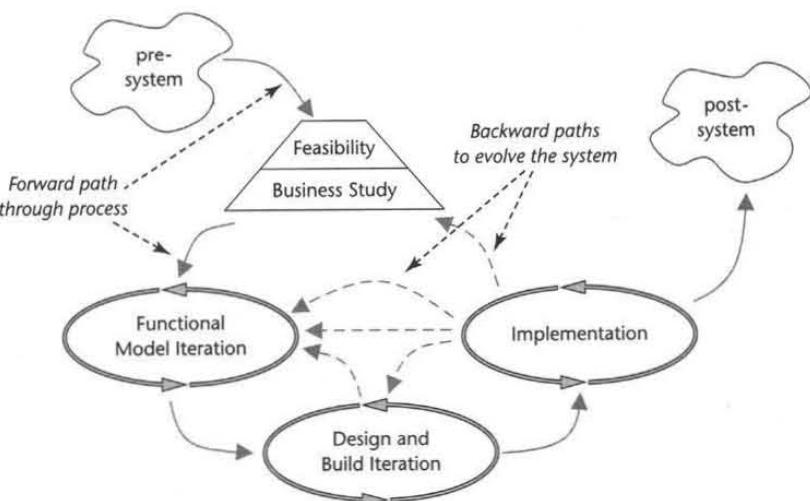


Figure 21.6 Simplified DSDM lifecycle (adapted from DSDM Consortium, 2004).

The feasibility study phase determines whether the project is suitable for a DSDM approach. It typically lasts only weeks, whereas the feasibility stage may last months on a traditionally run project. The study should also answer questions such as the following.

- Is the computerized information system technically possible?
- Will the benefit of the system be outweighed by its costs?
- Will the information system operate acceptably within the organization?

The *business study* phase identifies the overall scope of the project and results in agreed high-level functional and non-functional requirements. Maintainability objectives are set at this stage and these determine the quality control activities for the remainder of the project. There are three levels of maintainability:

- maintainable from initial operation
- not necessarily maintainable when first installed but this can be addressed later
- short life-span system that will not be subject to maintenance.

Where the third level of maintenance is chosen, care should be taken to ensure that the system is discontinued at the end of its allotted time, otherwise it may become subject to maintenance requests that are difficult to service.

The *functional model iteration* phase is concerned with the development of prototypes to elicit detailed requirements. The intention of DSDM is to develop prototypes that can ultimately be delivered as operational systems, so these must be built to be sufficiently robust for operational use and also to satisfy any non-functional requirements such as performance. When completed, the functional model comprises high-level analysis models and documentation, together with prototypes that are concerned with detailed functionality and usability. During the functional model iteration the following activities are undertaken.

- The functional prototype is identified.
- A schedule is agreed.
- The functional prototype is created.
- The functional prototype is reviewed.

The *design and build iteration* phase is concerned with developing the prototypes to the point where they can be used operationally. The distinction between the functional model iteration and the design and build iteration is not clear-cut and both phases can run concurrently. The activities for the design and build iteration phase are very similar to those described above for the functional model iteration phase.

The *implementation* phase deals with the installation of the latest increment, including user training. At this point it is important to review the extent to which the requirements have been met. If they have been fully satisfied the project is complete. If some non-functional requirements have yet to be addressed, the project may return to the design and build iteration phase. If some element of functionality was omitted due to time constraints, the project may return to the functional model iteration phase. If a new functional area is identified, the project may return to the business study phase. The return flows of control are shown with dashed arrows in Fig. 21.6. Implementation comprises the following iterative

activities:

- producing user guidelines and gaining user approval
- training users
- implementing the system
- reviewing the business requirements.

21.5.2 Timeboxing

DSDM uses *timeboxing* to fix the resource allocation for a project, or for a part of a project. This planning technique limits the time available for the refinement of a product. Overall, a DSDM project has a fixed completion date and this defines a timebox for the project as a whole. Smaller timeboxes are identified within this, each with a set of prioritized objectives. Each timebox produces one or more deliverables that allow progress and quality to be assessed. Examples of deliverables include requirements and design artefacts, software increments, documentation and so on.

Within a timebox, the team members have three major concerns. First, they must carry out any investigation needed to determine the direction they should take for that part of the project. Second, they must develop and refine the specified deliverables. Third, they must consolidate their work prior to the final deadline.

Timeboxing relies on prioritizing requirements, and this can be difficult to do. One way of doing this is to apply the set of rules known as the *MoSCoW* rules (for Must ... Should ... Could ... Won't).

Must have requirements are crucial. If these are omitted the system will not operate. In DSDM the set of *Must have* requirements is known as the minimum usable subset.

Should have requirements are important but if necessary the system can still operate without them.

Could have requirements are desirable but provide less benefit to the user.

Won't have this time around requirements can reasonably be left for development in a later increment.

All requirements may matter, but not to the same extent. If the full set cannot be addressed within the scope of a timebox, the MoSCoW categorization can be used to focus the requirements in an appropriate way.

21.5.3 Project selection in DSDM

Most publications on DSDM (for example, DSDM Consortium/Stapleton, 2003) include some advice on how to choose projects that are suitable for a DSDM approach. This advice can be distilled down to a simple set of rules.

- The system should not be too algorithmically complex, and its main functionality should be visible at the user interface. This is understandable if we consider the key requirement for all DSDM projects that there should be active user involvement. In effect, this rule seeks to test whether the system is focused on the direct provision of useful functionality to business users. Without that, user involvement would be much less relevant.

- There should be a real business need for early delivery of the software. This is also understandable if we consider the DSDM emphasis on frequent product delivery. It is hard to see how the team could continue to take this seriously if there were no pressure to complete.

One other condition applies more to the organization as a whole than to the individual project or to the team. It is a prerequisite that senior managers must give their unequivocal support to the use of DSDM, since it relies on each project team having a great deal of freedom about how to interpret the requirements for the system on which they are working. The team's energy and enthusiasm will soon be stifled if their independence turns out not to be a reality.

21.6 eXtreme Programming

The approach known as eXtreme Programming (XP) is a novel combination of elements of best practice in systems development. It was first publicized by Beck (2004) and incorporates a highly iterative approach to development. It has become well known in a relatively short period of time for its use of *pair programming*, though it encompasses various other important ideas. Pair programming involves writing the program code in pairs and not individually. At first sight it would appear that this approach would significantly increase the staffing level and hence the cost of developing an information system. The advocates of XP claim otherwise.

Beck (2004) identifies the four underlying principles of XP as communication, simplicity, feedback and courage.

- *Communication*. Poor communication is a significant factor in failing projects. XP highlights the importance of good communication among developers and between developers and users.
- *Simplicity*. Software developers are sometimes tempted to use technology for technology's sake rather than seeking the simplest effective solution. Developers justify complex solutions as a way of meeting possible future requirements. XP focuses on the simplest solution for the immediate known requirements.
- *Feedback*. Unjustified optimism is common in systems development. Developers tend to underestimate the time required to complete any particular programming task. This results in poor estimates of project completion, constant chasing of unrealistic deadlines, stressed developers and poor product quality. Feedback in XP is geared to giving the developers frequent and timely feedback from users and from test results. Work estimates are based on the work actually completed in the previous iteration.
- *Courage*. The exhortation to be courageous urges the developer to throw away code that is not quite correct and start again, rather than trying to fix the unfixable. Essentially the developer has to abandon unproductive lines of development, despite the personal emotional investment in work done.

XP argues that embracing change is an important key to systems development and that development staff are motivated by the production of quality work.

Requirements capture in XP is based on *user stories* that describe the requirements. These are written by the user and form the basis of project planning and the

development of test harnesses. A user story is very similar to a use case, though some proponents of XP suggest that there are key differences in granularity. A typical user story is about three sentences long and does not include any detail of technology. When the developers are ready to start work they get detailed descriptions of the requirements by sitting face to face with their customer. Beck (2004) describes the systems development process as being driven by the user stories in much the same way that the UP is use case driven.

XP involves the following activities.

- The *planning game* involves quickly defining the scope of the next release from user priorities and technical estimates. The plan is updated regularly as the iteration progresses.
- The information system should be delivered in small releases that incrementally build up functionality through rapid iteration.
- A unifying *metaphor* or high-level *shared story* focuses the development.
- The system should be based on a *simple design*.
- Programmers prepare unit tests in advance of software construction and customers define acceptance tests.
- The programme code should be restructured to remove duplication, simplify the code and improve flexibility—this is known as refactoring, and is discussed in detail by Fowler (1999).
- *Pair programming* means two programmers write code together on one workstation.
- The code is owned collectively and anyone can change any code.
- The system is integrated and built frequently each day. This gives the opportunity for regular testing and feedback.
- Normally staff should work no more than forty hours a week.
- A user should be a full-time member of the team.
- All programmers should write code according to agreed standards that emphasize good communication through the code.

The XP approach is best suited to relatively small projects—say with no more than ten programmers. XP relies on clear communicative code and rapid feedback. If circumstances preclude either of these, then XP is not the most appropriate approach.

One key feature of XP is that the code itself is its own design documentation. This runs counter to some aspects of the approach suggested in this book. We have suggested that requirements are best analysed and suitable designs produced through the use of visual models using UML. Nonetheless, XP offers many interesting insights into how software development projects can be organized and managed.

21.7 Issues in Choosing a Methodology

The introduction of any methodology to an organization is not a trivial matter. There are many costs, some quite difficult to estimate. Staff must be trained in the techniques, structure and management of the new methodology, documentation

Box 21.1 Methodology and Student Projects

We include here some remarks about the role of methodology in student projects. Supervisors of projects at undergraduate and post-graduate levels usually require students to follow some explicit methodology. It is also often a condition for professional accreditation of a degree that students should show that they can choose and apply an appropriate methodology.

The requirement is not trivial. The driving force that led to the existence of so many methodologies is the constant struggle by software developers to learn how to avoid past errors, and how to repeat past successes. But what does this mean in practice? What must a student do in order to claim that he or she has ‘followed an appropriate methodology’? Most of the methodologies described in this chapter simply do not fit the context of an assessed student project without at least some adaptation. This is for several reasons.

- There is typically only one developer, and he or she is usually quite inexperienced.
- There is often no real client, and no real business need for the product.
- Even if there is a real client, access may be limited when compared to a commercial project.
- Access to the project supervisor may also be limited when compared to a project manager in a commercial setting.
- The student must usually substitute self-study for formal training and mentoring in processes and tools.

It is hardly surprising that students are sometimes confused about how to cope with this set

of challenges. Our suggestions for resolving this confusion are threefold.

First, we take the general view that methodologies, like software systems, should be fit for purpose. In other words, the methodology chosen for a project should be one that at least appears likely to be helpful in achieving the project aims. For example, a project expected to use software components in its implementation should make use of techniques fit for specifying components.

Second, all methodologies are capable of some adaptation, and very few need be adopted in their entirety. For example, students working alone may benefit from following some aspects of XP, but they will certainly find pair programming to be an irrelevance. Furthermore, there are more marks to be gained by producing explicit analysis and design models that document decisions made along the way, than by slavishly following the principle that the code can serve as its own documentation.

Finally, a methodology is as much a means for learning how to do systems development as it is a recipe. This is particularly true for a student project, where it is taken for granted at the outset that the developer has only limited experience. Every student project is intended partly as a learning experience, and this applies as much to the methodology as to any other aspect of the project. For this reason, it is not really important whether the ‘right’ methodology is chosen at the outset. Instead, the student should reflect on his or her chosen methodology during the course of the project, and, at the end, should write up the lessons learned from this reflective evaluation.

must be purchased and software licences must be obtained for CASE tools that support the methodology. The indirect, hidden costs are often underestimated. Productive time is lost during training, and for some time after the change there may be a reduction in productivity and an ongoing need for support from external consultants. This is true whether or not the organization already uses a methodology. Even with careful evaluation before a decision is made, followed by careful planning of the change, it is often still prudent to conduct a full-scale trial of a new methodology on a pilot project, which must also be chosen carefully. It would be unwise to risk the failure of a critical system, yet a pilot project must be sufficiently complex to put the new methodology to a thorough test.

The choice of the ‘right’ methodology is also fraught with difficulties, as there are now many hundreds to choose from, and these differ radically in their philosophies, their coverage of the lifecycle and their suitability to particular application domains. Many factors affect the appropriateness of a methodology, including type of project (large, small, routine or mission-critical), application domain (e.g. real-time, safety-critical, user-centred, highly interactive, distributed or batch-mode) and nature of the IS development organization.

One very influential thinker on the management of software development is Humphrey (1989), whose ‘process maturity framework’ has now evolved into the Software Engineering Institute’s Capability Maturity Model Integration (Software Engineering Institute, 2000). This model suggests that organizations evolve through stages of maturity, which necessarily follow each other in a pre-defined sequence. By analogy, a butterfly must first pass through the stages of egg, then caterpillar, then chrysalis, and it would simply make no sense for one to attempt to fly before it emerged as an adult. The logic for software development is that there is little point in introducing practices too far beyond the organization’s current level of maturity.

Humphrey originally described five stages. First comes an ‘initial’ level, where development activities are chaotic, and each developer uses ad hoc procedures that they have probably devised themselves. There are no common standards and no dissemination of good practice through the organization, so the success of any project depends solely on the skill and experience of the development team. At this level there is no point in introducing any methodology, since management have neither the skill nor the structures required to control it. Instead, the focus should be on moving to the next ‘repeatable’ level, where an organization has adopted simple development standards and project management procedures. These allow successes to be repeated on later projects and the organization can benefit from a methodology, since management procedures are capable of enforcing its application. However, while individual managers may repeat their successes, there is no clear understanding of which specific factors led to each success. It is unlikely that success can be generalized to different kinds of project or application, and the flexibility of the organization is still limited. A prescriptive methodology that defines all steps in some detail is more likely to be successful.

An organization at the next, ‘defined’ level has its own definition of the software process and is able to standardize activities throughout the organization. A methodology can now be introduced more readily and is likely to produce greater benefits, since the organization already has a culture of working to defined procedures. But staff still adapt much more readily to a methodology that is in harmony with their current ways of working. The next step is typically to introduce a metrics programme (see the book website for more information about metrics). If successful, this can lift the organization to the ‘managed’ level—but few organizations are yet at this level. Only a tiny handful have reached the final ‘optimizing’ level, where there is a capability for continuous improvement in all activities (corresponding to the general management approach called ‘Total Quality Management’).

21.8 Hard versus Soft Methodologies

This chapter would not be complete without some mention of the critical debate that turns on the distinction between *hard* and *soft* methodologies, and that

sometimes divides both the profession and the academic community into two opposing camps. The distinction has emerged principally from the broad systems movement, and, while there is no one precise definition of the difference, it is summarized in Fig. 21.7.

To summarize, 'hard' is usually taken to mean objective, quantifiable or based on rational scientific and engineering principles. In contrast, 'soft' involves people issues and is ambiguous and subjective. The methodologies discussed in this chapter derive mainly from the hard tradition. However, influence of a soft approach can be discerned, particularly in Participative Design, and also to a lesser extent in techniques such as user stories in XP and the use case technique, since these aim at eliciting the practical, context-based requirements of individual users.

On the whole, those methodologies that might be characterized as principally soft in their orientation tend to focus more on making sure that the 'right' system is developed, than on how to actually develop the system. Their intellectual antecedents are diverse. For example, Checkland's influential Soft Systems Methodology (SSM) (Checkland, 1981; Checkland and Scholes, 1990; Checkland and Holwell, 1998) originally grew out of a failed attempt to apply hard systems engineering techniques to messy, ill-defined problem situations. The resulting methodology is grounded in a set of philosophical ideas about the nature of systems at a conceptual level. The techniques of SSM, such as the conceptual model and the rich picture, provide ways of exploring and agreeing the characteristics of the problem situation, before any attempt is made to define a specific system that will help users to meet their goals.

This is very different from the approach taken by most 'hard' methodologies. These tend to assume that the purpose and nature of the organization can, to a large extent, be taken for granted, and that every project begins with an identified need for an information system to solve some recognized problem.

Hard systems view	Soft systems view
The activity of IS development is all about building a technical system that is made only of software and hardware.	An IS also comprises the social context in which the technical system (software and hardware) will be used.
Human factors are chiefly important from the perspective of the software's usability and acceptability. Politics and group behaviour are only an issue for project managers.	A new IS impacts on interpersonal communication, social organization, working practices and much more, so human and social factors are paramount.
Organizations exist only to meet rational objectives, through the application of rational principles of business management. It is possible to be both rational and objective about the requirements for a new system.	Organizations are made up of individuals with distinct views and motivations, so any picture of requirements is subjective. It is not always possible even to reach a consensus. In practice, this means that the powerful decide, not the wise.
When requirements are uncertain or unclear, it is up to management to decide. Setting objectives is a principal role of management, and others should follow their lead.	If management has not accommodated the full range of views in the organization, encouraging managers to decide on the requirements may be completely counter-productive.

Figure 21.7 Some underlying assumptions of hard and soft systems approaches.

One attempt to reconcile this contradiction argues that soft and hard methodologies cover different parts of the lifecycle. In this view, a soft methodology is more useful in the earlier stages of the lifecycle, particularly when there is uncertainty about the goals or strategy of the organization as a whole. A hard approach will be more appropriate once any initial uncertainties and ambiguities have been resolved (insofar as this is possible), since the emphasis then shifts to a specific project with relatively clear goals and boundaries.

This has led to the suggestion that, in certain situations, hard and soft methodologies can complement each other and can be used together to help overcome some of the perennial difficulties in systems development. Flynn (1998) proposes a 'contingency framework' shown in Figure 21.8, which aims at helping to select an appropriate methodology for a specific organizational context.

'Requirements uncertainty' is the extent to which requirements are unknown or subject to debate or disagreement, and also whether they are expected to change during development. For example, a new system intended primarily to automate an existing manual system may have relatively low requirements uncertainty. Agate's Campaign Management system might fall into this category. 'Process uncertainty' refers to the degree of doubt about the best way to build the proposed system. For example, it may be very difficult to decide on the best process if a new system will use untried technology, with unpredictable effects on the organization's employees. A project intended to introduce electronic commerce to an organization with no experience of it might fall in this category. The term 'Linear Model' refers to a sequential lifecycle model like the waterfall model.

A project is rated along both dimensions, and this helps to indicate an appropriate development approach. For example, where both the requirements and process are clear from the outset, a linear model of development is recommended, which in practice might mean using a fully featured methodology with clear tasks and phases, such as the UP. It might also mean the procurement of a ready-made solution. Where the process uncertainty is low but the requirements are very unclear, an agile approach such as DSDM might be recommended. At the other extreme, where everything is unclear, a soft approach is recommended, so that the character of the problem can be clarified before any further action is taken.

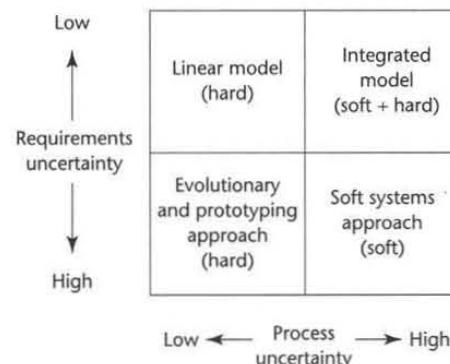


Figure 21.8 The contingency framework can be used to help select an appropriate methodology for a given organizational context (adapted from Flynn, 1998).

In seeking to merge together a soft and a hard methodology, the development team is really trying to devise a unique method suited to the project. This implicitly recognizes the complementary nature of their strengths and weaknesses.

21.9 Summary

We began this chapter by considering how the concept of methodology differs from method. This is an important distinction, since the development approach is an important factor in the success or failure of a project, and selection of an appropriate methodology is a necessary precursor to choosing the specific method to be followed. Many methodologies have been developed over the years, stemming from different traditions, and each in some way attempting to counter a perceived shortcoming in contemporary rivals. The 1990s were a prolific time for the spread of object-oriented methodologies, which have comprehensively replaced the older structured methods in many organizations today. Among the object-oriented methodologies, the UP and its derivatives look set to be the survivors. Over the same period, there was also much research into the possible merging of hard and soft methodologies. This was aimed at meeting a wider range of demands, and thus improving the overall success rate of IS development projects, but so far it has had relatively little impact. Recently, a similar quest has been taken up by proponents of the Agile Alliance. This differs in some important respects from earlier methodology wars. Methodologies such as DSDM and XP aim to complement object-oriented development rather than to replace it outright. As a result, it appears likely that they will continue to evolve alongside UML for the foreseeable future.

Review Questions

- 21.1** What is the difference between ‘methodology’ and ‘method’?
- 21.2** Distinguish between ‘task’ and ‘technique’, and give some examples of each.
- 21.3** What does it mean to say that XP is ‘agile’?
- 21.4** Explain the key elements in the philosophy of the DSDM.
- 21.5** In what ways does the participative design approach agree with object-oriented approaches, such as the UP?
- 21.6** How does the full UP approach differ from the simplified approach followed in this book?
- 21.7** Name the five levels of the Capability Maturity Model.
- 21.8** Distinguish between the hard systems view and the soft systems view.
- 21.9** Why might a methodology based on a hard systems approach be unsuccessful in a situation where the goals of the organization are unclear?
- 21.10** What general advantages are claimed for using a methodology?
- 21.11** What might be the disadvantages of using an inappropriate methodology?

Case Study Work, Exercises and Projects

Do some research in your library or on the Internet, and collect material that describes four or more different systems development methodologies. Try to make these as different from each other as possible, for example by choosing one that is structured (e.g. SSADM), one that is object-oriented (e.g. UP), one based on a soft systems view (e.g. SSM) and one from another tradition (e.g. PD). Then use the following questions as a basis for comparison.

- 21.A** What techniques are used by each methodology? In particular, what aspects of the system do they represent and what techniques do they use to do this?
- 21.B** To what extent does each methodology cover the full project lifecycle, from project selection through to implementation and maintenance?
- 21.C** How far do you think each methodology can be adapted to suit differing projects or circumstances?
- 21.D** Can you find a statement that gives an underlying philosophy for each methodology? If not, is it possible to identify the intellectual tradition from which the methodology has been derived?

Further Reading

The DSDM Consortium/Stapleton (2003) is a very readable introduction to this methodology, and also includes as an appendix a reprint of the Fowler and Highsmith (2001) article that launched the Agile Alliance.

Larman (2003) is another readable introduction to agile development, with chapters devoted to XP, the Unified Process and other agile methodologies.

Fitzgerald et al. (2002) is a comparative review of some traditional methodologies (not discussed in this chapter) and some not-so-traditional methodologies that include RUP, DSDM, XP, SSM. There is also some coverage of the role of the CMM and a helpful discussion of the theoretical background to methodologies and their use.

Avison and Fitzgerald (2003), now in its third edition, remains a key text for issues in understanding and selecting software development methodologies.

Vidgen et al. (2002)—though mainly devoted to the development of web information systems, which is beyond the scope of this book—include a useful summary of systems thinking and SSM, and there is also some coverage of PD.

Flynn (1998), although older than other texts mentioned here, remains a good introduction to the many traditions behind today’s methodologies, and is also still worthwhile reading for its treatment of the issues that cause difficulty in choosing the ‘right’ way to do systems development.

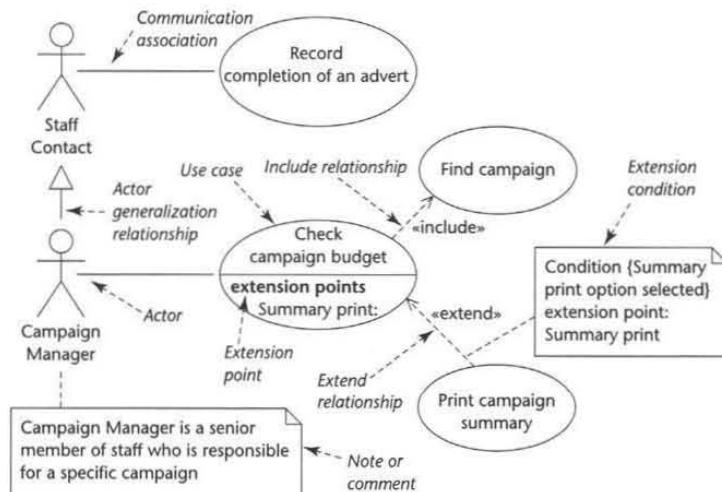
For more detailed information about SSM, readers should ideally consult the primary sources: Checkland (1981) and Checkland and Scholes (1990).

Jacobson et al. (1999) is still the definitive work on the UP, while Kruchten (2004) and Kroll and Kruchten (2003) are both good sources on RUP.

A**APPENDIX**

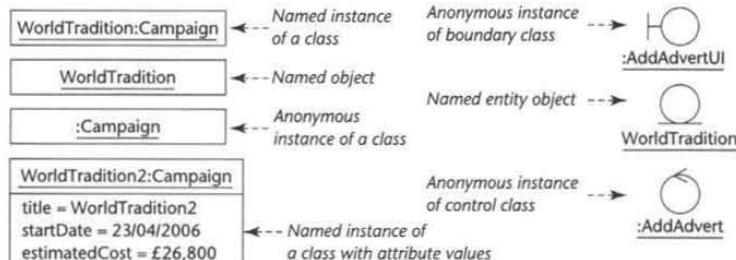
Notation Summary

Use Case Diagram

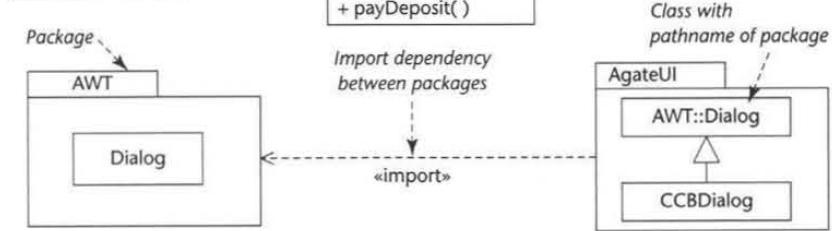
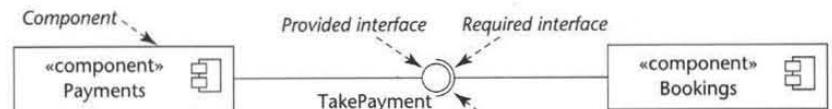
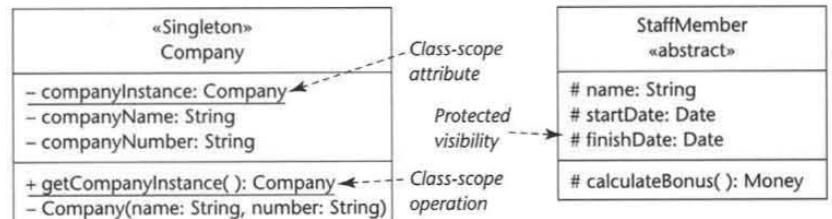
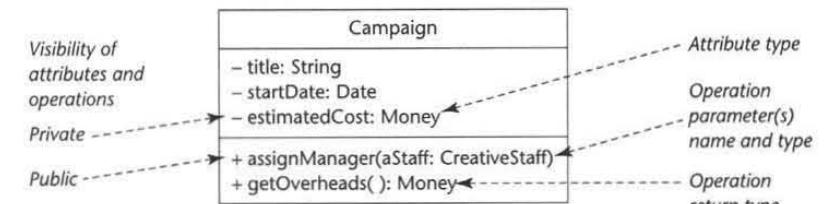
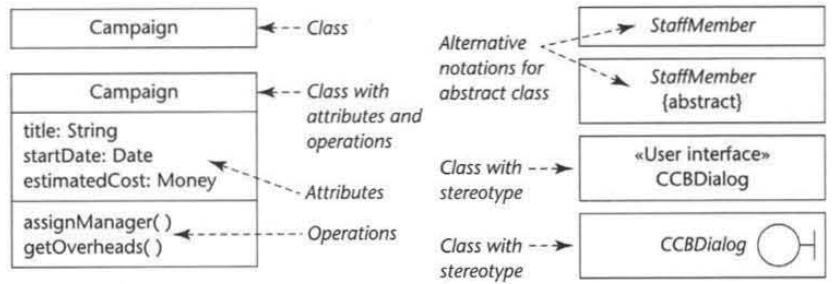


Static Structure Diagrams

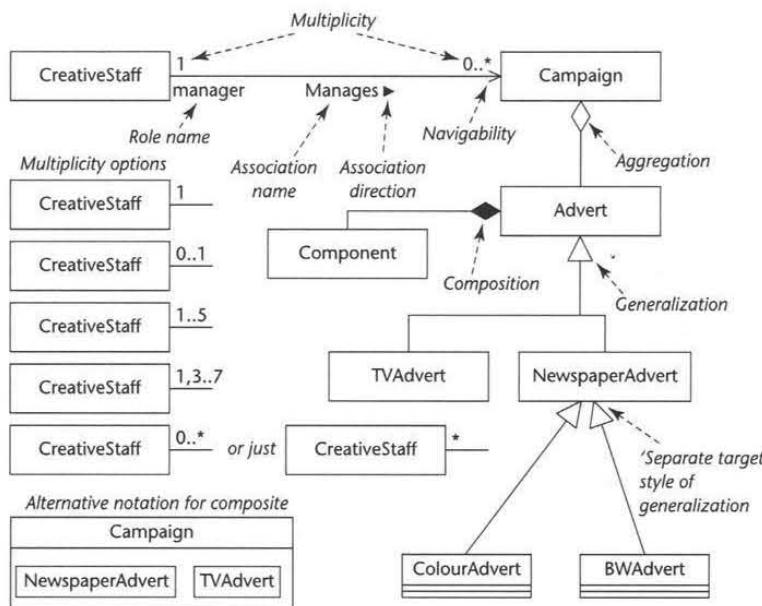
Object instance notation



Class notation

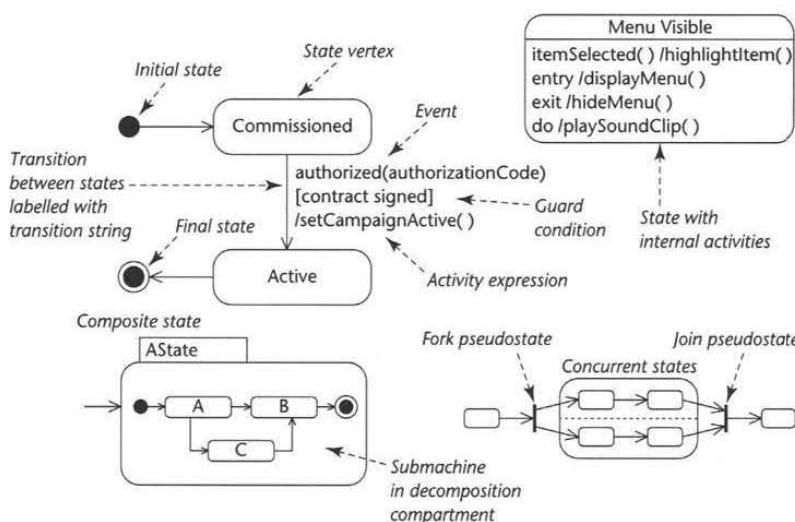


Associations

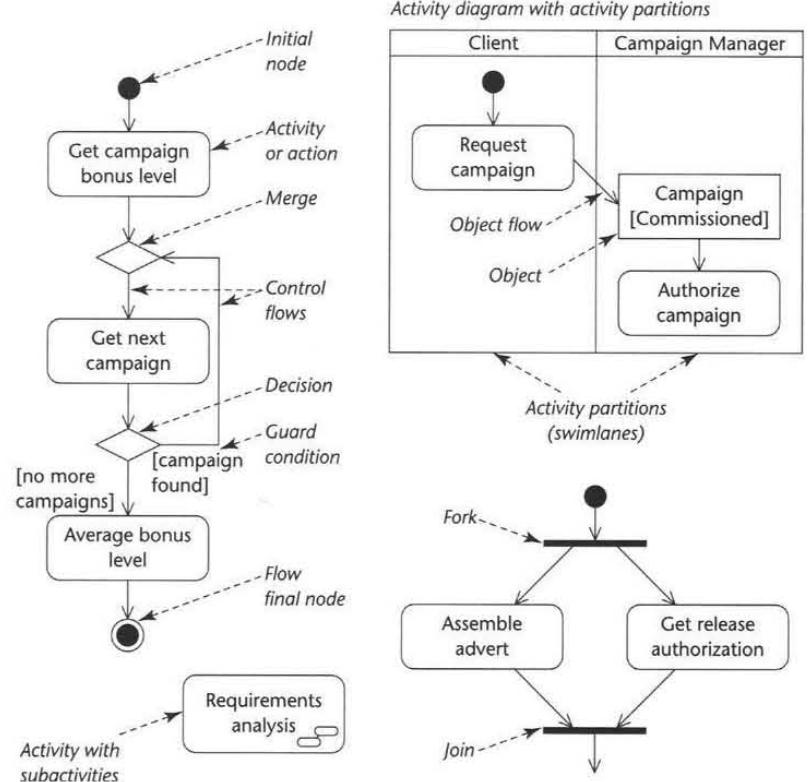


Behaviour Diagrams

State Machine

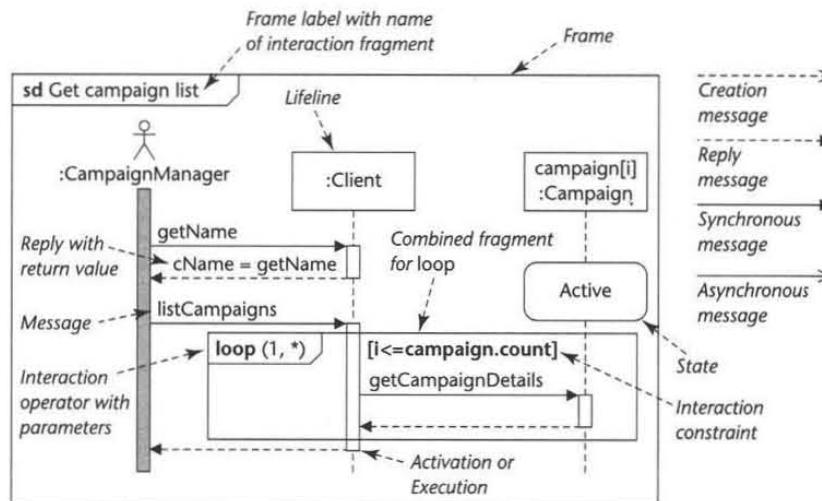


Activity diagram

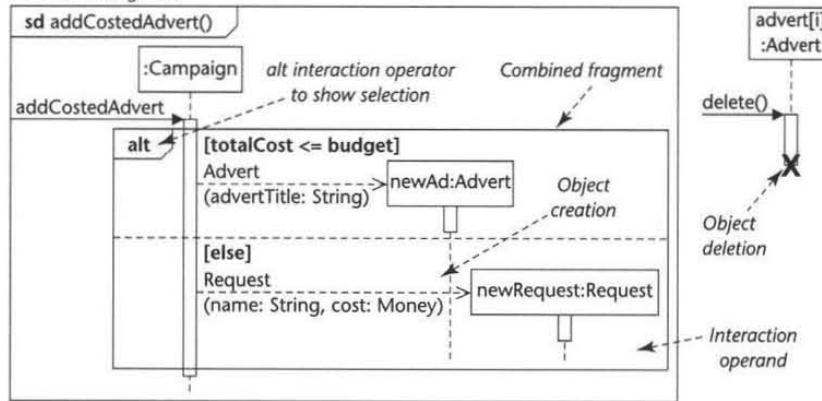


Interaction Diagrams

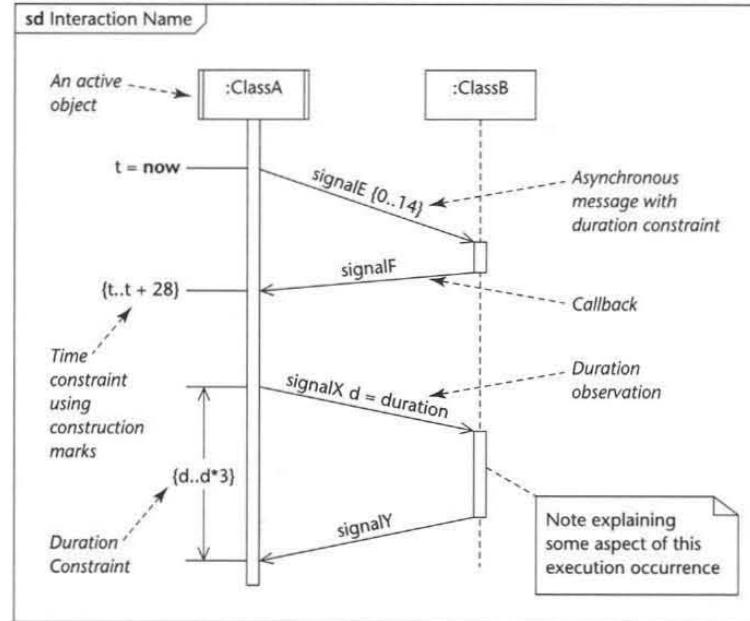
Sequence diagram



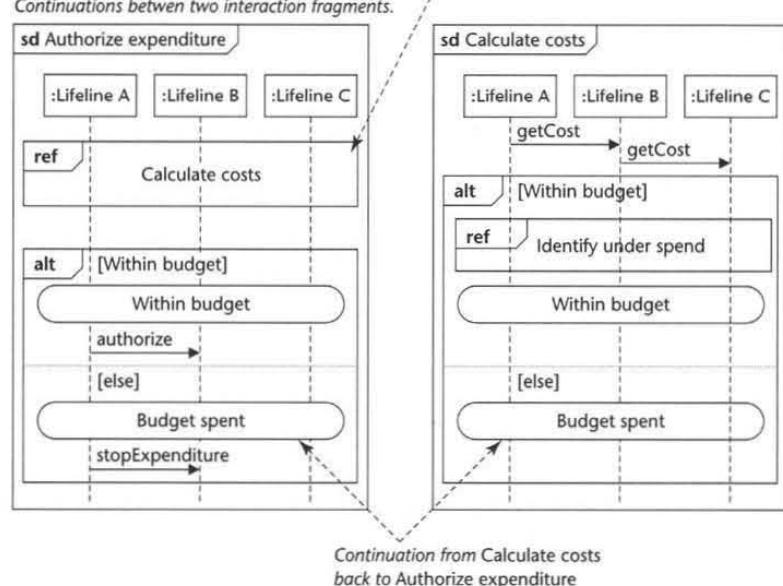
Interaction fragment



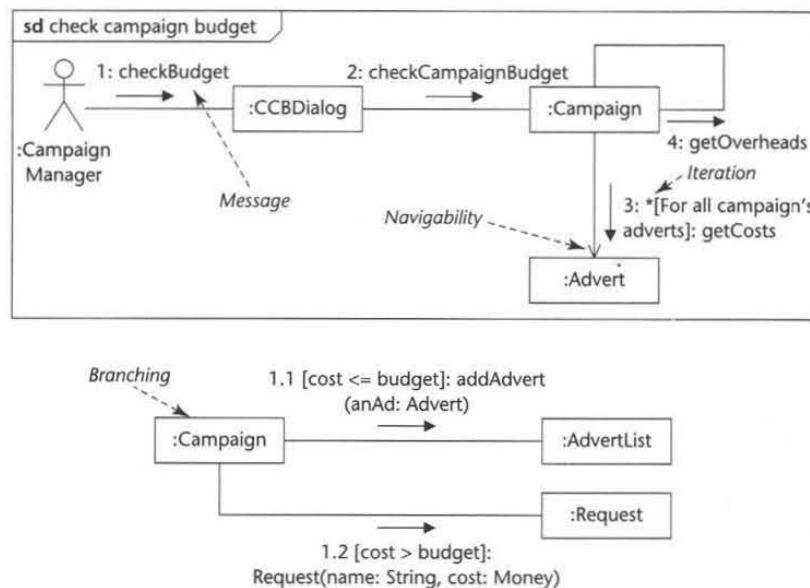
Interaction sequence diagram with duration and time constraints.



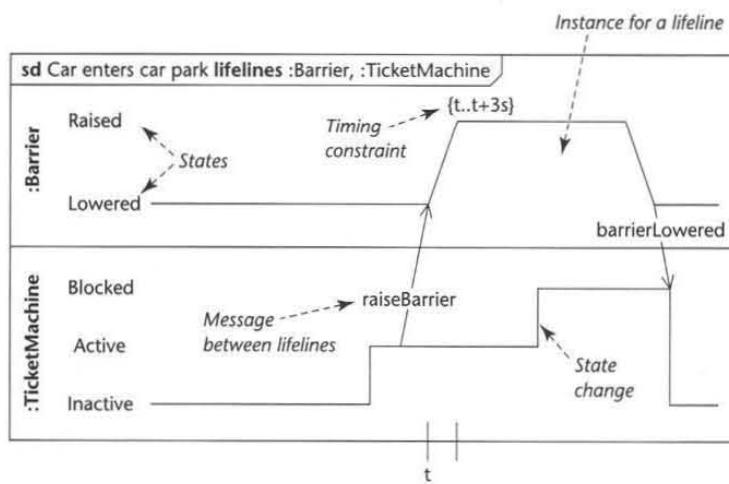
Continuations between two interaction fragments:
sd Authorize expenditure → sd Calculate costs



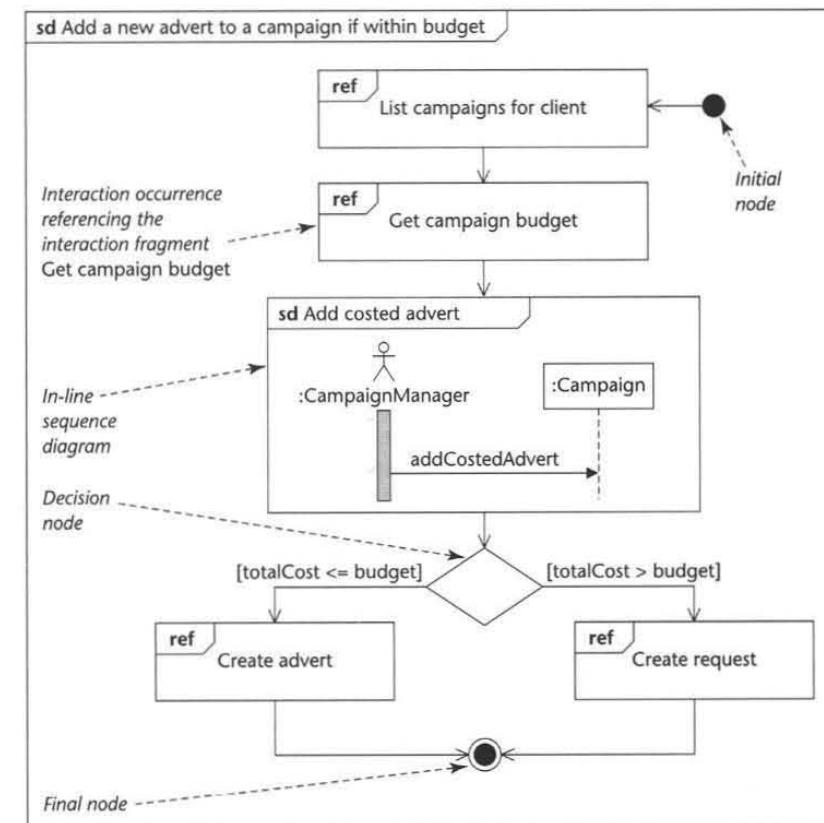
Communication diagram



Timing diagram

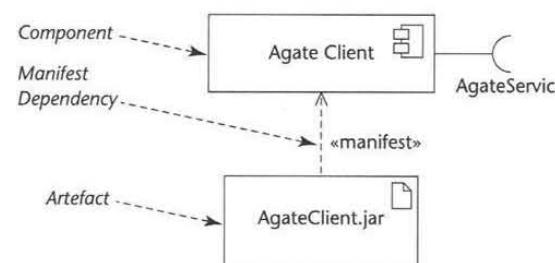
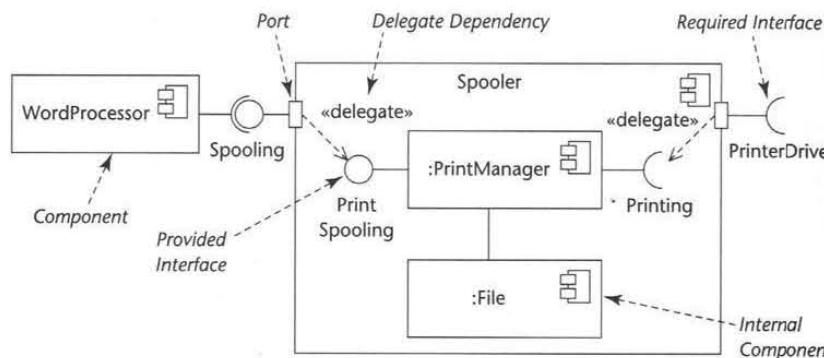


Interaction overview diagram

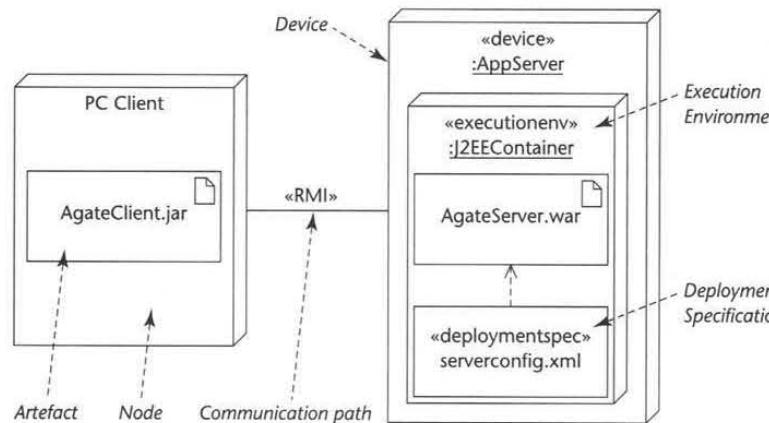


Implementation Diagrams

Component diagram



Deployment diagram



Selected Solutions and Answer Pointers

In this section we give solutions to a selection of the review questions in the chapters, and also pointers on possible approaches to some of the end-of-chapter case study work, exercises and projects.

Answers to Selected Review Questions

- 1.3** Even if something isn't a system (or might not be one), thinking of it as one still gives useful insights.
- 1.5** Feedback is sampling one or more outputs of a system for comparison with a control value. Feed-forward is sampling a system input, usually before it enters the system. The control value may be an output, input or an internal measure of system performance.
- 1.7** A management support system provides information that helps managers to make decisions. Most use feedback or feed-forward to monitor the performance of that part of the organization for which the manager is responsible.
- 1.9** Business goals and strategy are typically defined first, as they provide the context. An IS strategy identifies applications that can help to meet business goals, and an IT strategy identifies IT needed to develop and run the applications. Each informs its predecessor about what can realistically be achieved. The process is iterative.
- 1.10** One of the simplest definitions says that information is data with a structure and a meaning derived from the context in which it is used.
- 2.1** They differ in their view of the problems because their view of the meaning and purpose of IS development also differs (see also Review Question 1.6).
- 2.3** The simplest definition is 'fitness for purpose'. But it can be hard to identify and define the purpose unambiguously. A more practical alternative is 'meeting all user requirements, both stated and implied'.
- 2.6** The system may address irrelevant problems. It may not fit the way that people work. It may be unsuitable for its environment. It may be out of date before delivery. Political difficulties may lead to delay or cancellation.

- 2.7** A stakeholder has an interest in a project because they are (or will be) affected by its progress or by its results.
- 3.1** Some advantages of the traditional waterfall lifecycle:
- Teams with specialized skills can be assigned to tasks in particular phases.
 - Progress can be evaluated at the end of each phase.
 - Attendant risk can be controlled and managed.
- 3.2** Some disadvantages of the traditional waterfall lifecycle:
- Real projects rarely follow a simple sequential lifecycle.
 - Iterations are almost inevitable.
 - The lapsed time between inception and delivery is frequently too long.
 - It is unresponsive to changes in the technology or requirements.
- 3.5** Prototyping is not necessarily concerned with delivery of a working system, whereas an incremental approach delivers a working system in successive increments. Note that in the Unified Software Development Process an increment can produce any lifecycle product.
- 3.7** Syntactic correctness is concerned with using the notation (e.g. UML) correctly, consistency relates to producing a set of models or diagrams that are consistent with each other and completeness refers to producing models that are completely defined.
- 3.8** The term requirements traceability refers to the capability of tracking each requirement to all the systems development deliverables, from requirements models to program code that relates to it.
- 3.9** A diagram may be syntactically correct and complete and consistent with other diagrams and models, but it may not relate accurately or completely to the user requirements, the most important criterion for any diagram or model.
- 4.2** Semantics is the study of meaning. In object-oriented development it is used to denote the meaning that an element has for the user. (User may include the modeller or developer, not just the end-user of the software). An object's semantics include its purpose, description, relationships and behaviour seen from a user perspective.
- 4.3** Other parts of a system only see an object's interface (services it can perform and operation signatures). Internal details including data are hidden and can only be accessed by a message that contains a valid signature.
- 4.4** Polymorphism means that when one message is sent to objects of different types, each has an appropriate, but different, implementation for its response. The object that sends the message need not know which type of object is addressed, since the operation signature is the same. One way of implementing polymorphism is through inheritance and overriding.
- 4.6** A subclass inherits all characteristics of its superclass and other ancestors (some may be overridden, but are still technically inherited). Each subclass is different from its ancestors in at least one way.
- 5.2** Icons, two-dimensional elements, paths and strings.
- 5.4** To promote communication between team members in a project. To communicate over time to other people who will work on the system. To communicate good practice and experience.

- 5.7** A rectangle with rounded corners.
- 5.8** Activity edges or control flows.
- 5.10** The start node (a filled black circle) and the final node (a filled black circle within another circle).
- 5.13** An arrow between an object and an action.
- 6.1** Examples of functional requirements are: the need for a process to be run that allocates staff to lines based on their skills and experience, and on holidays and sick leave; printing out an allocation list; amending the allocation list. Examples of non-functional requirements include: printing the allocation list by 12.00 noon; the need to handle 200 operatives' details.
- 6.4** Use cases are produced to model the functionality of the system from the users' point of view and to show which users will communicate with the system. They show the scope of the system.
- 6.6** An essential use case documents the interaction between user and system in a way that is free of technological and implementation details, and a real use case describes the concrete detail of a use case in terms of its design.
- 6.10** The use cases represent the functions of the business rather than functions of the computer system, and the actors represent people and organizations outside the business interacting with those functions.
- 7.2** A purely graphical model does not have the precision to cover every aspect of system behaviour and structure. A purely textual model would be too large, too complex and extremely difficult to understand and to maintain.
- 7.3** An attribute is a characteristic of a class (every person has a height). An attribute value is a characteristic of an instance (this author is 1.75m tall).
- 7.4** An element's stability is the relative infrequency of change in its description. Instances may be created, destroyed or updated frequently, but it is unlikely that the class description will change as often.
- 7.6** Multiplicity denotes the range of values for the number of objects that can be linked to a single object by a specific association. It is a constraint because it limits the behaviour of a system. If a client can have only one staff contact, it should not be possible to link a second.
- 7.10** A link is a connection between two objects. 'Changing' a link (say by substituting another object at one end) is equivalent to destroying the link and creating a new one. (Think about two objects tied with a single length of string. In a substitution, there is a moment when neither one nor the other is connected—unless you tie on the second before untying the first, but an object link cannot do this).
- 7.12** A communication diagram shows only those classes that collaborate to provide the functionality of a particular use case (or operation); the links that are shown are those that are required for that purpose. A class diagram typically shows all the classes in a particular package and all the associations between them.
- 8.1** Use of a component saves time and work. A friend of one of the authors once said 'Have you ever wondered how much it would cost to make your own light bulb?'

- 8.3** Objects are well encapsulated, and object structures can be designed this way. The hierarchic nature of generalization abstracts out the more general features of a class. Hierarchic organization of models helps the developer to find components easily when they are needed. Composition encapsulates whole structures within a composite object.
- 8.4** A component of a composition cannot be shared with another composition. The component has a coincident lifetime with the composition (although a component can be explicitly detached before the composition is destroyed).
- 8.5** This is a basis for polymorphism. The superclass operation defines the signature, but each subclass has a different method that implements the behaviour (see Chapter 10).
- 8.6** An abstract class has no instances and exists only as a superclass in a hierarchy. It provides a generalized basis for concrete subclasses that do have instances.
- 8.12** An antipattern documents unsuccessful attempts at solving problems and suggests how the failed solution may be adapted to solve the problem successfully.
- 9.1** Communication diagrams discourage both using a large number of messages between two objects and having too many parameters for each message as these are clumsy to represent on the diagram.
- 9.2** Small self-contained classes are easier to develop, test and maintain.
- 9.3** Sequence diagrams have a time dimension (normally vertically down the page) while communication diagrams do not. Communication diagrams show the links between objects, which are not shown on sequence diagrams.
- 9.5** A lifeline represents the existence of the entity being modelled (e.g. an object) during the interaction represented in an interaction diagram. It may be used in sequence diagrams, communication diagrams, timing diagrams or interaction overview diagrams (within interaction fragments).
- 9.6** An execution occurrence indicates which operation is executing at a particular stage in an interaction represented in a sequence diagram.
- 9.9** Sequence numbers are written in a nested style in a communication diagram to represent nested procedural calls.
- 9.11** Complex interactions may be represented using UML by:
 - hiding some of the detail of the interaction using interaction occurrences referencing other interaction fragments. Common interaction fragments may be referenced from several other sequence diagrams.
 - using lifelines to represent groups of objects and their interaction or to represent subsystems. These lifelines refer to another interaction fragment, which contains the hidden detail.
 - using interaction overview diagrams to show the overall flow of control with in-line interaction fragments and interaction occurrence to hide the detail of the interaction.
- 9.16** Timing diagrams are used to show how state changes occur over time. They are of particular value when specifying how time constraints affect the interaction between lifelines.

- 10.1** Operation specifications confirm the user's view of the logical behaviour of a model. They also specify what the designer and programmer must produce to meet the users' requirements.
- 10.2** Decision tables are particularly suited to representing decisions with complex multiple input conditions and complex multiple outcomes, where the precise sequence of steps is either not significant or is not known.
- 10.4** An algorithm defines the step-by-step behaviour of an operation. A non-algorithmic approach defines only pre-conditions and results.
- 10.5** Non-algorithmic methods of operation specification emphasize encapsulation.
- 10.9** OCL expressions have:
 - a context within which the expression is valid (for example, a specified class)
 - a property within the context to which the expression applies (for example, an attribute of the specified class)
 - an operation that is applied to the property (for example, a mathematical expression that tests the value of the attribute).
- 11.2** A guard condition is evaluated when a particular event occurs and only if the condition is true does the associated transition fire.
- 11.3** All the guard conditions from a state should be mutually exclusive so that for each set of circumstances there is only one valid transition from a state. If they are not mutually exclusive more than one transition may be valid and the behaviour of the state machine is indeterminate.
- 11.4** A composite state contains substates and is used to represent complex state behaviour in a hierarchical fashion. A composite state may contain a single decomposition compartment with a single submachine or it may contain several decomposition compartments, each with a submachine. In the latter case the composite state has concurrent substates.
- 11.5** If an object is occupying a composite state that has more than one submachine then it will occupy a substate in each of the submachines in the composite state. Consequently it will occupy more than one substate at any one time while it is in the composite state.
- 11.6** The substates in a composite state are nested and may be nested to any arbitrary depth. In most cases nesting substates to one or two levels down will model the state behaviour adequately. Only if the composite state contains orthogonal submachines will these nested substates be concurrent.
- 11.11** The typical symptoms that a state machine has not been drawn to model state changes include the following.
 - Most transitions are fired by state completion.
 - Many messages are sent to 'self', reflecting code reuse rather than actions triggered by events.
 - States do not capture state-dependent behaviour associated with the class.
- 12.2** An architectural view is a representation of a particular system or part of a system from a particular perspective. An architectural viewpoint is a template that describes how to create and use an architectural view.

- 12.3** The four views are the *logical view*, the *implementation view*, the *process view* and the *deployment view*. The additional view is the *use case view*.
- 12.7** An enterprise architecture links the design of the business to the information systems that are needed to support that business.
- 12.10** Open layered architectures are more difficult to maintain because each layer may communicate with all lower layers, hence increasing the degree of coupling in the architecture. A change to one layer may ripple to many layers.
- 12.11** A closed layered architecture may require more processing as messages have to be passed through intervening layers.
- 12.13** The main differences between the MVC and the layered architecture include the update propagation mechanism and the separation of the presentation layer into the View and Controller components in the MVC.
- 12.14** A broker decouples subsystems by acting as an intermediate messaging-passing component through which all messages are passed. As a result a subsystem is aware of the broker and not directly in communication with the other subsystems. This makes it easier to move the subsystems to distributed computers.
- 13.3** User wants report etc.–analysis. Selection of business objects etc.–logical design. Size of paper etc.–physical design.
- 13.5** Seamlessness means that the same model (class diagram) is used and successively refined throughout the project.
- 13.10** Functional, efficient, economical, reliable, secure, flexible, general, buildable, manageable, maintainable, usable, reusable.
- 13.14** The facilities typically offered by a DBMS are:
 - different views of the data by different users
 - control of multi-user access
 - distribution of the data over different platforms
 - security
 - enforcement of integrity constraints
 - access to data by various applications
 - data recovery
 - portability across platforms
 - data access via query languages
 - query optimization.
- 14.1** We elaborate user interface and application control classes; we add mechanisms to support data management. The class diagram is also updated with the types and visibility of attributes and operations and to show how associations are designed.
- 14.2** Private, public, protected or package visibility.
- 14.3** Attributes should be designated private to enforce encapsulation.
- 14.7** Collection classes can be used to hold the object identifiers of the linked objects at the many end of an association. Collection classes provide collection-specific behaviour for manipulating the collection.

- 14.8** A collection class of object identifiers should be included in a class if it is not used by another class and it does not increase the complexity of the class unduly.
- 14.10** A derived attribute should be used to reduce the processing time for one or more transactions so that user response times are appropriate. However, it should be noted that this may increase the processing time of other transactions in order to ensure that the data integrity is maintained.
- 15.2** The main aspects of changeability are maintainability, extensibility, restructuring and portability.
- 15.3** The class constructor in the Singleton pattern is private so that it can only be accessed by the class-scope `instance()` method. This ensures that the Singleton class has total control over its own instantiation.
- 15.4** The Singleton pattern ensures that only one instance of a class exists and provides system-wide access to that instance.
- 15.7** A pattern catalogue is group of largely unrelated patterns, which may be used together or independently. A pattern language is a group of related patterns that relate to a particular problem domain.
- 16.3** Dialogue metaphor describes interaction in terms of conversation between user and system involving different kinds of communication. Direct manipulation metaphor represents objects of interest to the user as objects on the screen that they can manipulate through the use of the mouse. Dialogue follows sequence determined by system. Direct manipulation is event-driven, and user can determine sequence of events.
- 16.6** User may hit Return key without thinking and delete the Client in error.
- 16.9** Possible advantages: structured–aids management of projects, applies standards that aid communication, forces consideration of all aspects of HCI design; ethnographic–analyst gets detailed understanding of context of system, active user involvement, social and political factors taken into account; scenario-based–helps to think through possible alternative routes in use cases, can be used to justify design decisions, valuable for testing programs. Possible disadvantages: structured–can be bureaucratic; ethnographic–can be time-consuming; scenario-based–generates large volume of documentation.
- 17.2** A horizontal prototype deals with only one layer of the system architecture, usually the user interface. A vertical prototype takes one subsystem and develops it through each layer.
- 17.6** We use state machine diagrams to model the lifetime of instances of business classes.
- 17.8** A list of states, for each state the valid events that can cause a transition from that state, the state that each transition leads to, and any operations associated with the transition into the new state.
- 17.11** Java `EventListener` only handles changes to interface objects. MVC deals with changes to Model objects. The Java `Observer` and `Observable` interfaces provide MVC mechanisms.

- 18.3** Tagged data, with a tag for the class of each object and the attribute of each value within each object. This way, it is possible to reconstruct any object from the data in the file without having to hard code the structure of every possible complex object. (This is the approach used by the Java ObjectOutputStream and ObjectInputStream classes, and by XML and SOAP.)
- 18.6**
1. Remove repeating groups. Ensure all row column intersections contain only atomic values.
 2. Make sure every attribute is dependent on the whole primary key. Create a separate table for part-key dependencies.
 3. Ensure every attribute is dependent on the primary key and not on another non-key attribute. Create a separate table for non-key dependencies.
- 18.8** Object Manipulation Language and Object Definition Language.
- 19.3** Artefact represents development artefacts, typically physical files that are deployed on a system, whereas a component represents modular software units with a well-defined interface and can be a logical or physical component of the system.
- 19.7** Possible tests would be to test validation of date, test validation of start time, check validation of job number, line number, product etc. More detailed validation of time could be to check that it is within a certain amount of time of current time—if not, then a warning should be displayed.
- 19.8** Review of cost benefit analysis. Summary of functional requirements met and amended. Review of achievement of non-functional requirements. Assessment of user satisfaction. Problems and issues with the system. Extract of quantitative data for future planning. Identification of candidate components for reuse. Possible future developments. Actions required. (See Section 19.9.2 for the detail.)
- 19.10** Because analysts or designers will have the wider view of the system and can ensure that changes fit in and do not have a detrimental impact on other subsystems.
- 20.1** Saving time and money in developing the components, saving time and money in testing the components, and creating components that can be recombined to create flexible business processes.
- 20.3** ‘A component is a type, class or any other work product that has been specifically engineered to be reusable.’ (Jacobson et al., 1997)
- 20.6** It considers a component to be an executable unit of code rather than a type or class. It specifies that a component should have an interface, and that it should be capable of being connected together with other components via its interface.
- 21.1** A methodology is essentially a set of principles. A method is an instantiation of the principles in a given situation.
- 21.2** A task is something you do in a particular project. Tasks have products. A technique specifies how to carry out a task. A task might be ‘Analyse the requirements for a use case’. One technique for doing this would be the UML collaboration diagram.

- 21.4** A full statement of DSDM would repeat the nine principles—active user involvement, focus on fitness for business purpose, and so on. But perhaps the key feature is the use of timeboxing combined with the MoSCoW approach to prioritizing requirements.

Answer Pointers for Selected Case Study Work, Exercises and Projects

- 1.B** Some main subsystems are: online sales, retail shops, supplies, deliveries, systems support and accounts, and more. Some of the control mechanisms involve supplier re-orders, the product catalogue, network performance and security. Most have some human activities and automated support. One example of feedback includes online shoppers, watching the progress of their orders. The market researcher uses feed-forward (what attracts customers to web pages).
- 1.C** Main business aim: say, ‘To establish FoodCo as an independent branded name supplying a range of high quality food products to consumers’. Subsidiary aims: diversification of customer base; achievement of international recognition and sales. Each will be translated into measurable objectives, for example as a basis for the selection of information systems development projects.
- 2.C** Stakeholders should include many of the following. Good if you listed them all, excellent if you thought of some not shown below (provided you can justify their inclusion).
Patients and potential patients, patients’ relatives, ambulance drivers, paramedics, control room operators, accident and emergency staff, supervisors of professional stakeholders, managers who control affected budgets, policy level managers, taxpayers (or purchasers of medical insurance policies if this is how the system is funded), general medical practice staff, politicians (particularly if the service is publicly funded), members of the general public (who make the emergency calls), other road users.
- 3.A** An incremental development can be justified for the following reasons.
 - Useful increments can be delivered quite quickly. For example, staff management and material tracking could be implemented initially.
 - Users can gain experience with the systems.
 - Risk is minimized.
 - Requirements can be refined in light of the initial increments.
- 4.A** The human activity system referred to is the application domain for the proposed IS. Other systems might include the project team, the analyst’s department, the business planning system and the wider (political and cultural) system of the organization. Formal and informal structures of communication and relationships are the interfaces. Other installed software and hardware systems may be important (consider the discussion in Box B1.2).

- 4.D** You should have equivalents for most of the following. The names are not significant at this stage, nor is an exact match in the way that you have grouped concepts together.

Factory, Product, ProductRange, PackedProduct, SaladPack, VegetablePack, CookedProduct, Sauce, Pickle, SandwichTopping, Ingredient, Customer, Supermarket, Brand, Farm, Supplier, Employee, Consumer.

- 5.A** Some kinds of information systems can be used to model the real world in order to try out ideas. For example, decision support systems typically model some aspect of a business and allow staff and managers to ask 'What if?' questions: 'What would happen to demand for a product if the price was increased by 10%' or 'If we targeted a particular area with a mailshot, what kind of response to our product could we expect, based on what we know about the population of that area?' However, a customer in an information system is not a model of the customer, it is a set of data values that describe attributes of the customer. Also, some things in information systems are the real-world objects. An invoice in a sales order processing system is the real invoice; it is not a model. In object-oriented systems, there is sometimes a belief that the operations of objects are things that those objects do to themselves. (Rumbaugh et al. (1991) suggest operations for a Bicycle class, like move and repair.) Typically the operations of objects are actually operations that we want the system to carry out on those objects, and we package them in the class as a way of organizing the design of the software system.

- 5.B** Designing cars, designing aircraft (models to use in wind tunnels), architecture and town planning, packaging design for products.

- 6.B** Here are some of the use cases that should be in the diagram with the actor in brackets. Check Staff Availability (Production Planner), Enter Details of Staff Illness (Production Planner), Print Availability Lists (Production Planner). There is a need for some means of entering details of staff holidays. The decision about who does this will affect the scope of the system. It could be done by the staff themselves and authorized online by the factory manager, or this process could be done on paper and only authorized holidays entered by the production planners.

- 7.A** The following are sample descriptions for two of the use cases.

Record employee leaving the line

Normally employees are recorded as leaving the line when they clock off at the end of a working shift. Although there are breaks in the operation of the line during a shift these are not normally recorded as employees leaving the line. Date, time and location are recorded.

Stop run

When the production line stops for a routine reason, e.g. for a break, to restock or to reload equipment, the time the run stopped is recorded and a reason is recorded. The line supervisor or chargehand can do this.

- 7.B** For the use case realization for Record employee leaving the line, you should have a collaboration involving Employee, Supervisor, ProductionLine, ProductionLineRun and EmployeeAbsence, as well as a boundary class and a control class.

- 8.C** Possible subclasses include TelevisionAdvert, RadioAdvert, MagazineAdvert, PosterAdvert, LeafletAdvert. We could introduce another layer of hierarchy by grouping NewspaperAdvert and MagazineAdvert under PrintMediaAdvert, and TelevisionAdvert and RadioAdvert under BroadcastMediaAdvert. (You may have chosen equally valid alternative names.)

- 8.E** Some generalization and composition can be justified by the inclusion of the following classes: Operative, RoutineBreak, AbsenceRecord.

- 9.A** The sequence diagrams should be derivable and consistent with the communication diagrams produced during use case realization when you answered Exercise 7.B. However, you will be adding more detail in terms of message signatures and message types.

- 9.B** Variations in allocations of responsibility will depend upon how much responsibility the control class has and how much is devolved to the entity classes or the boundary classes. At one extreme the control class orchestrates all the functionality of the use case, at the other the control class delegates the complete control of the use case to one of the entity classes. A good design will lie between these extremes.

- 10.A** One of the more complex (and therefore one that is well worth trying) would be ProductionLineRun.start(). Preconditions for this operation should be suggested by your use case description.

- 10.D** Most decision tables can be converted easily into Structured English with either case or nested-if. For very simple tables (two outcomes) if-then-else may be enough.

- 11.A** The events that affect ProductionLine include start run, end run, detect problem, pause run. The possible states for ProductionLine include Idle, Running, ProblemInterrupted and Paused.

- 12.D** An example would be Popkin System Architect. When it is started, it displays a framework diagram. One of the available frameworks is for Zachman, and clicking on the cells in the six by six matrix takes the user to a diagram of the appropriate type.

- 12.E** The FoodCo Production Control System could contain the subsystems Employee Management, Product and Production Line Management.

- 13.A** Examples include sequence of entry, branching points (where the user has a choice), repetition of entries (can the user enter more than one holiday at the same time?), commands that the user might need to use while entering data (but not whether they use a menu, function keys or control keys).

- 13.D** In Windows there are many standards, for example: the use of function keys, particularly in combination with Alt and Ctrl keys; the standards for the appearance of menus, for example, menu entries followed by dots ('...') when the menu entry leads to a dialogue box; the positioning of certain buttons in dialogue boxes ('OK' and 'Cancel').

- 14.A** All attributes should be designated private and operations public. Choose data types that reflect the domains from which the attribute values are selected.
- 14.B** The one-way associations are Line-LineFault. The two-way associations are Supervisor-ProductionLine and Line-LineRun.
- 15.B** The ProductionLine class could use the state pattern with the state subclasses Idle, Running, ProblemInterrupted, Waiting. This use of the state pattern reduces the complexity of the ProductionLine class but may increase the storage and processing requirements for the application as a whole.
- 16.A** See answer pointer to 13.D.
- 16.C** Something along these lines ...
- First, Rik runs off the three availability lists to show who is available for work the following week. He then starts with operatives who are available all week. For each operative, he views their record on screen, looking at their skills and experience, the line they are currently working on, and how long they have been on that line. He allocates each operative in turn to a line and a session in one of the factories.
- This does not provide any detail of actual interaction with the system.
- 17.A** If you are expecting to develop for a windowing environment, you will need a dialogue window as a minimum. Depending on how you handle looking up information, for example a list of valid reasons for stopping the line, you may need separate windows in which to display these look-ups. However, in the factory environment you may want to use a simpler device with an LCD screen for display and a sealed keyboard with dedicated keys. A PC with a mouse may be unnecessary and unsuitable in a messy environment. However, we are assuming a windowing environment for the other exercises in this chapter.
- 18.B** You should end up with the following tables (or similar names): SalesOrder, OrderLine, Customer and Product.
- 18.F** This will be similar to Figure 18.23, with a ClientBroker class to handle the operation to find each Client.
- 19.B** Many libraries now use a web browser to access catalogue services. If this is the case, then your deployment diagram will include the client machines (PCs, Apple Macs or workstations), the web server and probably another machine running the library software. Library staff may access the system from simple terminals for use cases to issue and return books. They will connect directly to the machine running the software, not via the web server. (The actual configuration will depend on your particular system.)
- 19.E** Issues to consider are as follows. Is the manual organized around the tasks a user carries out? Has it got an index? Can you find the terms in the index that you, the user, know, or does it use computer jargon? Does it show screen shots? Are they the same as actual screens or windows in the version you are using? (You should be able to think of other criteria.)

- 19.F** Possible inclusions for bug reports: User name, telephone no., building, room, address etc. Date and time bug occurred. Type of machine on which bug occurred. Operating system of machine. Software in which bug occurred. Other software running at the same time. Program/window/function being used at time of bug. Any error messages displayed for the user. What the user expected to happen. What actually happened. What the user did (key strokes, mouse clicks on buttons or menus etc.) immediately beforehand.
- 20.C** You could either include the encryption package within the security package or have it as a separate package. The Security Core Classes will need its services and will have a dependency on it. Will it need any kind of user interface classes, for example, if it requires setting up with some kind of parameters? If it does, it will presumably need somewhere to store these parameters, in which case it will also need data storage services.

Glossary

Abstract class a class that can have no instances; a superclass that acts only as a generalized template for its instantiated subclasses.

Abstract operation an operation that is not implemented in the class in which it appears (usually an abstract superclass), but that will be implemented in a subclass.

Abstraction a simplified representation that contains only those features that are relevant for a particular task; the act of separating out the general or reusable parts of an element of a system from its particular implementation.

Action a step in an activity, used to represent where work is taking place; an executable statement, commonly associated with a state or transition in a state machine.

Activation the execution of an operation, represented in interaction sequence diagrams as a long thin rectangle.

Activity an activity is a group of one or more actions that may execute as a result of a triggering event.

Activity diagram a diagram that shows activities and actions to describe workflows.

Activity edge a flow between elements in an activity.

Activity final node the final node in an activity that terminates the actions in that activity.

Activity partition a column or row in an activity diagram used to show an area of responsibility for the actions in it. Sometimes referred to as a swimlane.

Actor an external entity of any form that interacts with the system. Actors may be physical devices, humans or information systems.

Adornment an element attached to another model element, for example a stereotype icon or a constraint.

Aggregation a whole-part association between two or more objects or components, where one represents the whole and the others parts of that whole.

Algorithm a description of the internal logic of a process or decision in terms of a structure of smaller steps. The structure may consist of steps in sequence, selection between alternatives and iteration.

Analysis class stereotype one of three specialized kinds of class (boundary, control and entity classes (*q.v.*)) that feature in analysis class diagrams. These represent a separation of concerns that forms the basis of the architecture recommended for most models developed following UP guidelines (*cf* stereotype).

Antipattern documents unsuccessful attempts at providing solutions to certain recurring problems but includes reworked solutions that are effective.

Artefact a model element used in deployment diagrams representing a file or collection of files containing source code or an executable. (Artifact in US spelling.)

Association a logical connection between two instances. Associations are commonly found between classes, although in some circumstances a class can have an association with itself. An association describes possible links between instances, and may correspond either to logical relationships in the application domain or to message paths in software.

Association class a class that is modelled in order to provide a location for attributes or operations that properly belong to an association between other classes.

Association instance another name for a link (*q.v.*).

Asynchronous message a message that does not cause the invoking operation to halt execution while it awaits the return of control.

Attribute an element of the data structure that, together with operations, defines a class. Describes some property of instances of the class.

Behavioural state machine see *state machine*.

Boundary class a stereotyped class that provides an interface to users or to other systems.

Business rule see *enterprise rule*.

Capta data that has been selected for processing because of its relevance to a particular purpose.

Cardinality the number of elements in a set; contrast with *multiplicity* (*q.v.*).

Choice pseudostate allows the splitting of a transition into multiple exit transitions and is a dynamic conditional branch. The guard conditions are evaluated when the transition reaches the choice pseudostate.

Class a descriptor for a collection of objects that are logically similar in terms of their behaviour and the structure of their data.

Class diagram a UML structure diagram that shows classes with their attributes and operations, together with the associations between classes.

Class Responsibility Collaboration (CRC) CRC cards provide a technique for exploring the possible ways of allocating responsibilities to classes and the collaborations that are necessary to fulfil the responsibilities

Class-scope an attribute that occurs only once and is attached to the class, not to any individual object. A class-scope operation is accessed through the class (*i.e.* prefixed with the class name) not through an object. Model elements that are of class scope are underlined in class diagrams.

Cohesion a measure of the degree to which an element of a model contributes to a single purpose.

Collaboration the structure and links between a group of instances that participate in a behaviour. The behaviour can be that of an operation, a use case or any other behavioural classifier.

Collaboration diagram a UML 1.X term for a communication diagram; see *communication diagram*.

Collection class provides collection-specific behaviour to maintain a collection. Used when designing associations with a many multiplicity to hold collections of object identifiers.

Combined fragment a combined fragment contains one or more interaction operands (or compartments) and is enclosed in a frame; each operand contains an interaction fragment; the meaning of a combined fragment is determined by its interaction operator.

Common Object Request Broker Architecture (CORBA) a mechanism to support the construction of systems in which objects, possibly written in different languages, reside on different machines and are able to interact by message passing.

Communication diagram shows an interaction between lifelines (e.g. objects) and the context of the interaction in terms of the links between the lifelines.

Communication path a path between nodes in a deployment diagram that allows communication between the nodes, usually stereotyped with the protocol for the communication.

Component a replaceable part of a system defined primarily in terms of the interfaces that it provides and the interfaces that it requires in order to operate. Components may be executable software modules that have a well-defined interface and identity; they may also be complex model elements that will be realized as free-standing software modules.

Component-Based Development (CBD) an approach to software development that focuses on the use of replaceable components. CBD differs from other development approaches, in particular, through its emphasis on the assembly of software systems from pre-existing components.

Component diagram a diagram that shows the organization of and dependencies among components.

Composite structure diagram a diagram that shows the internal structure of a composite. For instance, a composite structure diagram may be used to show the elements of a collaboration and how the collaboration as a whole interacts with the rest of the system.

Composition a strong form of aggregation with a lifetime dependency between each part and the whole. No part can belong to more than one composition at a time and, if the composite whole is deleted, its parts are deleted with it.

Concrete class a class that may have instances.

Concurrent states if an object may be in two or more states at the same time, then these states are concurrent states.

Constructor operation an operation that creates a new instance of a class.

Context (of a pattern) the circumstances in which a particular problem occurs.

Contract a black box description of a service (of a class or subsystem) that specifies the results of the service and the conditions under which it will be provided.

Control class a stereotyped class that controls the interaction between boundary classes and entity classes.

Control flow a flow between actions in an activity diagram that indicates the flow from one to another.

Coupling relates to the degree of interconnectedness between design components and is reflected by the number of links and the degree of interaction an object has with other objects.

Critical Path Analysis (CPA) a diagrammatic technique for analysing the dependencies between project tasks and determining those tasks that must be completed on time if the project itself is to be completed on time.

Data raw facts, not yet identified as relevant to any particular purpose.

Decision a node in an activity diagram where a flow branches into multiple alternative flows.

Deep history pseudostate shows that a region in a composite state will resume at the last active substate in each of the nested submachines within that region, no matter how deeply nested they are.

Degree another word for *multiplicity* (*q.v.*).

Dependency a relationship between two model elements, such that a change in one element may require a change in the dependent element.

Deployment diagram A diagram that shows the run-time configuration of processing nodes (*q.v.*) and the artefacts that are located on them.

Deployment specification an artefact deployed to a node in a deployment diagram in order to configure other artefacts deployed to the same node.

Design constraint a constraint that limits the design options that may be used. Common design constraints include cost and data storage requirements.

Destructor operation an operation that destroys an instance of a class.

Device a node representing a processor in a deployment diagram.

Domain model an analysis class model that is independent of any particular use cases or applications, and that typically contains only entity objects. A domain model may serve as a basis for the analysis and design of components that can be reused in more than one software system.

Encapsulation hiding the internal details of an element (typically a class or component) from the view of other elements, so that each can be maintained or modified without affecting the operation of other parts of the system.

Enterprise (or business) rule a statement that expresses business constraints on the multiplicity of an association; for example, an order is placed by exactly one customer.

Entity class a stereotyped class that represents objects in the business domain model.

Entry pseudostate shows an exceptional entry point into a submachine state.

Event an occurrence that is of significance to the information system and may be included in a state machine.

Exception a mechanism for handling errors in object-oriented languages.

Execution or Execution occurrence see *activation*.

Execution environment a type of node in a deployment diagram representing an implementation environment, container or platform in which an artefact is deployed.

Extend relationship a relationship between use cases where one use case extends or adds new actions to another. Written as a stereotype «extend».

Extension point the point in a use case where an extension occurs, based on the satisfying of some condition.

Exit pseudostate an exit pseudostate shows an exceptional exit point from a submachine state.

eXtreme Programming (XP) an approach to systems development that focuses on producing the simplest coding solution for application requirements. One characteristic is the use of pair programming, with code being written by two developers working at a single workstation.

Final pseudostate a final pseudostate is a notational convenience used to indicate the final state in a state machine.

Flow final node the final node in a flow within an activity diagram that terminates that flow but leaves other flows unaffected.

Forces (of a pattern) the particular issues that must be addressed in resolving a problem.

Fork a node in an activity diagram where a flow is split into multiple parallel flows.

Frame a labelled rectangle that represents the boundary of a diagram; frames may be used for all diagram types but may be omitted where the boundary of the diagram is clear; they are an integral part of the notation for interaction diagrams and should be used for these.

Functional requirement a requirement that specifies a part of the functionality required by the user.

Generalization the abstraction of common features among elements (for example, classes) by the creation of a hierarchy of more general elements (for example, superclasses) that encapsulate the common features.

Guard condition a Boolean expression associated with a transition that is evaluated at the time the event fires. The transition only takes place if the condition is true. A guard condition is a function that may involve parameters of the triggering event and also attributes and links of the object that owns the state machine.

Implementation diagram a generic term for the UML diagrams used in modelling the implementation of a system.

Include relationship a relationship between use cases where one use case includes the actions described in another use case. Written as a stereotype «include».

Incremental development development and delivery of software in incremental stages, where each increment provides some useful functionality. Some initial analysis scopes the problem and identifies major requirements. These are then reviewed and those that deliver most benefit to the client become the focus of the first increment. The installation of the first increment provides feedback to the team, which informs the second increment and so on.

Information facts that have been selected as relevant to a purpose and then organized or processed in such a way that they have meaning for that purpose.

Inheritance the mechanism by which object-oriented programming languages implement a relationship of generalization and specialization between classes. A subclass instance automatically acquires features of its superclasses.

Initial node the entry point to an activity diagram.

Initial pseudostate an initial pseudostate is a notational convenience used to indicate the starting state in a state machine. A state machine may not remain in its initial pseudostate.

Instance a single classifier, for example an object. Usually called an instance in the context of its membership of a particular class or type (also object instance).

Instance diagram a UML diagram similar in form to a class diagram, but which contains object instances instead of classes, links instead of associations and may show attribute values (also known as an object diagram).

Instance value (of an attribute) the value of an attribute that is taken by a particular object at a particular time.

Integrity constraint ensures that an information system holds data that is mutually consistent and is manipulated correctly. Referential integrity ensures that object identifiers in one object refer only to objects that exist. Dependency constraints ensure that attribute dependencies, where one attribute may be calculated from other attributes, are maintained consistently. Domain integrity ensures that attributes hold only permissible values.

Interaction defines the message passing between lifelines (e.g. objects) within the context of a collaboration to achieve a particular behaviour.

Interaction constraint a Boolean expression that must be true before the interaction fragment in the combined fragment region it guards can execute.

Interaction diagram an umbrella term for sequence, communication, timing and interaction overview diagrams.

Interaction occurrence a frame that does not show the detail of an interaction fragment, but refers to another sequence diagram that does show this detail. It provides a mechanism to hide complex detail and it may be included in a sequence diagram or an interaction overview diagram.

Interaction operand a compartment in a combined fragment; contains an interaction fragment.

Interaction operator determines the meaning of a combined fragment; for example, the interaction operator 'loop' specifies that the combined fragment is specifying an iteration.

Interaction overview diagram a variant of an activity diagram incorporates interaction diagrams. An interaction overview diagram focuses on the overview of flow of control in an interaction where the nodes in the diagram are interactions or interaction occurrences.

Interface that part of the boundary between two interacting systems across which they communicate; the set of all signatures for the public operations of a class, package or component.

Interface class a system interacts with its actors via its interface or boundary classes.

Invariant an aspect of a UML model expressed as a formal statement that must always remain true. For example, the value of a derived attribute `totalCost` may need always to be equal to the total of all `cost` attribute values.

Join a node in an activity diagram where multiple parallel flows are merged into a single flow.

Junction pseudostate allows the merging or splitting of transitions. When there are several exit transitions and only one entry transition, this is known as a static conditional branch.

Knowledge a complex structure of information, usually one that allows its possessor to decide how to behave in particular situations.

Legacy system a computerized information system, probably in use for a long time and built with technologies that are now outmoded (perhaps also using different technologies and/or development approaches at different times)—but that continues to deliver benefit to the organization.

Lifecycle (of a project) the phases through which a development project passes from the inception of the idea through the completion and use of the product and its eventual decommissioning.

Lifeline a lifeline represents the existence of a modelling element (e.g. object, subsystem) in an interaction diagram.

Link a connection between objects; an instance of an association.

Merge a node in an activity diagram where alternative flows created at a decision node are merged back to a single flow.

Message a request to an object that it provide some specified service, either an action that it can carry out or some information that it can provide. A message will invoke an operation or service.

Message passing a metaphor for the way that objects interact in an object-oriented system by sending each other messages that request services, or request or supply information. Since objects interact only through the messages they exchange, their internal details can remain hidden from each other.

Method the implementation of an operation.

Methodology typically comprises an approach to software development (e.g. object-orientation), a series of techniques and notations (e.g. the Unified Modelling Language—UML) that support the approach, a lifecycle model (e.g. spiral incremental) to structure the development process, and a unifying set of procedures and philosophy.

Model Driven Architecture (MDA) an OMG initiative and an approach to developing systems in which a platform-independent architectural model (*q.v.*) is produced and transformed into a platform-specific model (*q.v.*) from which code can be generated.

Modular construction an approach that aims to build component-based systems that are easy to maintain, modify or extend. Modular construction relies on modules that are essentially decoupled subsystems, with their internal details encapsulated.

Multiplicity a constraint that specifies the range of permitted *cardinalities* (*q.v.*), for example in an association role or in a composite class. For example, an association may have a multiplicity of between 1 and 5 (written as 1..5), and a particular instance of that association may have a cardinality of 3.

Node a physical computational resource used by a system at run-time, typically having processing capability and memory in a deployment diagram. Also an element in an activity diagram.

Non-functional requirement a requirement that relates to system features such as performance, maintainability and portability.

Normalization a technique that groups attributes based upon functional dependencies according to several rules to produce normalized data structures that are largely redundancy free.

Object a single thing or concept, either in a model of an application domain or in a software system, that can be represented as an encapsulation of state, behaviour and identity; a member of a class that defines a set of similar objects.

Object diagram see *instance diagram*.

Object flow flow between an object and an action or activity in an activity diagram, that indicates that the object is used or its state changed by the action or activity.

Operation an aspect of the behaviour that defines a class; an element of the services that are provided by a class; a specification of an element of system functionality that will be implemented as a method of an object.

Operation signature determined by the operation's name, the number and type of its parameters and the type of the return value, if any. Polymorphically redefined operations have the same signature.

Package a mechanism for grouping UML elements, usually classes, into groups. Packages can be nested within other packages.

Pattern an abstract solution to a commonly occurring problem in a given context.

Platform-independent model (PIM) a model produced in model-driven architecture (*q.v.*) that is independent of a development or deployment environment.

Platform-specific model (PSM) a model produced in model-driven architecture (*q.v.*) that is generated from a platform-independent model (*q.v.*) for a particular deployment environment.

Polymorphism the ability of different methods to implement the same operation, and thus to respond to the same message in different ways that are appropriate to their class. For example, objects of different subclasses in an inheritance hierarchy may respond differently to the same message, yet with a common meaning to their responses.

Post-condition part of an operation specification or used in a protocol state machine; those conditions that must be true after the operation has executed or a transition has fired—in other words, the valid results of the operation or the completion of the transition.

Pre-condition part of an operation specification or used in a protocol state machine; those conditions that must be true before the operation can execute in an operation specification or before a transition may fire in a protocol state machine.

- Primary operation** an operation to create or destroy an instance of a class, or to get or set the value of an attribute.
- Processing node** a node in a deployment diagram.
- Procedural call** see *synchronous message*.
- Property** a feature or characteristic of a UML element, usually one for which there is no specific UML notation.
- Protocol state machine** see *state machine*.
- Prototype** a system or partially complete system that is built quickly to explore some aspect of the system requirements. It is not intended as the final working system.
- Query operation** an operation that returns information but causes no change of state within a model or a software system.
- Realize relationship** a relationship between two elements where one is in some sense an implementation of the other, although it may not necessarily have the same structure. Commonly used to show that a class supports an interface. Written as a stereotype «realize».
- Refactoring** to refactor program code is to restructure and simplify it so that duplication is removed and flexibility is enhanced.
- Relation** a group of related data items organized in columns and rows, also known as a table.
- Reply** A reply is a return of control to the object that originated the message that began the activation.
- Repository** the part of a modelling tool environment that handles the storage of models, including diagrams, specifications and definitions.
- Responsibility** a high-level description of behaviour a class or component exhibits. It primarily reflects the services that it can offer to other objects or components, and also the knowledge or information that is available to that class or component, either stored internally or requested via collaboration with other elements.
- Reuse** assembly of all or part of a new software system from elements, components or abstractions that already exist. Reuse can refer to binary software, designs or specifications, or to more abstract features such as elements in a generalization hierarchy.
- Shallow history pseudostate** shows that a region in a composite state will resume at the last active substate at the level of the shallow history pseudostate.
- Sequence diagram** or interaction sequence diagram, shows an interaction between objects arranged in a time sequence. Sequence diagrams can be drawn at different levels of detail and also to meet different purposes at several stages in the development lifecycle.
- Service** a useful function (or set of functionality) that is carried out by an object, component or subsystem when requested to do so by another object or component.
- Signal** see *asynchronous message*.

- Software architecture** describes the subsystems and components of a software system and the relationships between the components.
- Specialization** the other face of generalization; an element (for example, a class) is said to be specialized when it has a set of characteristics that uniquely distinguish it from other elements. Distinguishes subclasses from their superclass.
- Stakeholders** anyone who is affected by the information system. Stakeholders include not only users and development team members, but also resource managers and the quality assurance team, for example.
- State machine** a state machine models states and state-dependent behaviour for a modelling element (e.g. object, subsystem, port, interface) and for interactions. There is a distinction between protocol and behavioural state machines in that protocol state machines only show all the legal transitions with their pre- and post-conditions. Behavioural state machines include activity expressions to show the actions that result from triggering events.
- State** the state of an object is determined by values of some of its attributes and the presence or absence of certain links with other objects. It reflects a particular condition for the object and normally persists for a period of time until a transition to another state is triggered by an event.
- Static conditional branch** see *junction pseudostate*.
- Stereotype** a specialized UML modelling element. The stereotype name is contained within matched guillemets «...». For example, an interface package is a stereotype of a package.
- Subclass** a specialized class that acquires general features from its ancestor superclasses in a generalization hierarchy, but that also adds one or more specialized characteristics of its own.
- Subsystem** a part of a system that can be regarded as a system in its own right.
- Superclass** a generalized class that is an abstraction of the common characteristics of its subclasses in a generalization hierarchy.
- Synchronous message** or procedural call, causes the invoking operation to suspend execution until control has been returned to it.
- Synchronizing operation** an operation that ensures that those attribute values which are dependent upon each other (e.g. may be calculated from each other) have consistent values.
- System** an abstraction of a complex interacting set of elements, for which it is possible to identify a boundary, an environment, inputs and outputs, a control mechanism and some process or transformation that the system achieves.
- Table** group of related data items organized in columns and rows. Used to store data in relational databases.
- Task** a specific activity or step in a project.
- Technique** a method for carrying out a project task.
- Template collaboration** a parameterized collaboration which may be implemented (that is, instantiated) with different classes or objects.
- Timing diagram** shows the state changes for one or more lifelines, typically those with important time-dependent behaviour; messages between lifelines may be shown.

Transaction an elementary exchange, say of an item of *capta* (*q.v.*) or of a unit of value.

Transition the movement from one state or activity to another, triggered by an event. A transition may start and end at the same state.

Type a stereotype of class that is distinct from an implementation class; a type is defined by attributes and operations but, since it is a pure specification, may not have methods. Classes that represent application domain concepts are in fact types. An object may change type dynamically during system execution, and may thus appear at different times to belong to different classes.

Usability requirement user requirement that describes criteria by which the ease of use of the system can be judged.

Use case describes, from a user's perspective, a behaviourally related set of transactions that are normally performed together to produce some value for the user. Use cases can be represented graphically in a use case diagram, each use case being described in the data dictionary. Use cases may be modelled at varying degrees of abstraction; essential use cases, the most abstract, are technologically and implementation independent, whereas real use cases describe how the use case actually operates in a particular environment.

Use case realization a set of model elements that show the internal behaviour of the software that corresponds to the use case—usually a collaboration.

User requirement something that users require a software system to do (functional requirement); alternatively, a standard for the performance of a system (non-functional requirement).

User story in eXtreme Programming requirements are captured as user stories. A user story is similar to a use case.

Vertex a node in a state machine diagram.

Visibility UML modelling elements (e.g. attributes or operations) may be designated with different levels of accessibility or visibility. Public visibility means that the element is directly accessible by any class; private visibility means that the element may only be used by the class that it belongs to; protected visibility means that the element may only be used by either the class that includes it or a subclass of that class; and package visibility means that an element is visible to objects in the package.

Wrapper or object wrapper, used to integrate object-oriented and non-object-oriented systems by encapsulating the non-object-oriented system with an object-oriented style of interface.

Bibliography

- Adams, D., *The Restaurant at the End of the Universe*, London: Pan Books, 1980.
- Agile Alliance, The Agile Manifesto, 2002, at <http://www.agilemanifesto.org/>.
- Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. and Angel, S., *A Pattern Language: Towns, Buildings, Construction*, New York: Oxford University Press, 1977.
- Allison, B., O'Sullivan, T., Owen, A., Rice, J., Rothwell, A. and Saunders, C., *Research Skills for Students*, London: Kogan Page, 1996.
- Allen, C. P., *Effective Structured Techniques: From Strategy to CASE*, Hemel Hempstead, UK: Prentice-Hall, 1991.
- Allen, P. and Frost, S., *Component-Based Development for Enterprise Systems: Applying the SELECT Perspective™*, Cambridge: Cambridge University Press; SIGS Books, 1998.
- Alur, D., Crupi, J. and Malks, D., *Core J2EE Patterns: Best Practices and Design Strategies* (2nd Ed.), Upper Saddle River, NJ: Prentice Hall and Sun Microsystems Press, 2003.
- Ambler, S. W., *The Object Primer: Agile Model-Driven Development with UML 2.0* (3rd Ed.), Cambridge: Cambridge University Press, 2003.
- ANSI, *American National Standard for Information Systems: Database Language SQL*, ANSI X3.135-1986, New York: ANSI, 1986.
- Apperly, H., Hofman, R., Latchem, S., Maybank, B., McGibbon, B., Piper, D. and Simons, C., *Service- and Component-Based Development: Using the Select Perspective and UML*, Harlow: Addison-Wesley/Pearson Education, 2003.
- Apple Computer Inc., *Macintosh Human Interface Guidelines*, Reading, MA: Addison-Wesley, 1996.
- Avison, D. and Fitzgerald, G., *Information Systems Development: Methodologies, Techniques and Tools* (3rd ed), Maidenhead: McGraw-Hill, 2002.
- Barker, C., 'London Ambulance Service gets IT right', *Computing*, 25 June, 1998.
- Bass, L., Clements, P. and Kazman, R., *Software Architecture in Practice* (2nd Ed.), Upper Saddle River, NJ: Software Engineering Institute; Addison-Wesley, 2003.
- BBC, 'Barclays admits new security breach', *BBC News Online*, 2 August, 2000, at <http://news.bbc.co.uk/1/hi/business/862294.stm>.
- BCS, *BCS Code of Conduct*, 2001, at <http://www.bcs.org/bcs/aboutbcs/codes/conduct>.
- Beck, K., *Extreme Programming Explained: Embracing Change*, Reading, MA: Addison-Wesley, 2004.
- Beck, K. and Cunningham, W., 'A laboratory for teaching object-oriented thinking', *Proceedings of OOPSLA '89*, pp. 1–6, 1989.
- Bell, D., *The Coming of Post-Industrial Society*, New York: Basic Books, 1973. Cited in Webster (1995).
- Bellin, D. and Simone, S. S., *The CRC Card Book*, Reading, MA: Addison-Wesley, 1997.
- Bennett, S., Skelton, J. and Lunn, K., *Schaum's Outline of UML* (2nd Ed.), Maidenhead: McGraw-Hill, 2005.
- Beveridge, T., 'Java Documentation' in *Java Unleashed*, Indianapolis: Sams.net Publishing, 1996.
- Bishop, J., *Java Gently* (3rd Ed.), Harlow: Addison Wesley/Pearson Education, 2001.
- Blaha, M. and Premerlani, W., *Object-Oriented Modelling and Design for Database Applications*, Upper Saddle River, NJ: Prentice-Hall, 1998.
- Boehm, B. W., *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice-Hall, 1981.
- Boehm, B. W., 'A Spiral Model of Software Development and Enhancement', in Thayer, R. H. (Ed.), *Tutorial: Software Engineering Project Management*, Los Alamitos, CA: IEEE Computer Society Press, 1988.

- Boehm, B. W., Get Ready for Agile Methods, with Care, *IEEE Computer*, 35, (1), pp. 64–69, 2002.
- Booch, G., *Object-Oriented Analysis and Design with Applications* (2nd Ed.), Menlo Park, CA: Benjamin/Cummings, 1994.
- Booch, G., Rumbaugh, J. and Jacobson, I., *The Unified Modeling Language User Guide*, Reading, MA: Addison-Wesley; ACM Press, 1999.
- Booth, P., *An Introduction to Human-Computer Interaction*, Hove: Lawrence Erlbaum Associates, 1989.
- Brown, K. and Whitenack, B. G., 'Crossing Chasms: A Pattern Language for Object-RDBMS Integration' in Vlissides, J. M., Coplien, J. O. and Kerth, N. L. (Eds.), *Pattern Languages of Programme Design 2*, pp. 227–238, Reading, MA: Addison-Wesley, 1996.
- Brown, W. J., Malveau, R. C., McCormick, H. W. and Mowbray, T. J., *AntiPatterns: Refactoring Software, Architectures and Projects in Crisis*, New York, NY: John Wiley, 1998.
- Browne, D., *STUDIO: Structured User-interface Design for Interaction Optimization*, Hemel Hempstead: Prentice-Hall, 1994.
- Budgen, D., *Software Design*, Reading, MA: Addison-Wesley, 1994.
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., *Pattern Oriented Software Architecture Volume 1*, Chichester: John Wiley, 1996.
- Bødker, S., Gronbaek, K., and Kyng, M., 'Cooperative design: Techniques and experiences from the Scandinavian scene', in Schuler, D. and Namioka, A. (Eds.), *Participatory Design: Principles and Practices*, Hillsdale, NJ: Erlbaum, 1993.
- Carroll, J. M. (Ed.), *Scenario-Based Design: Envisioning Work and Technology in System Development*, New York: John Wiley, 1995.
- Cattell, R. G. G., Barry, D. K., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T. and Velez, F. (Eds.), *The Object Data Standard: ODMG 3.0*, San Francisco: Morgan Kaufmann, 2000.
- Chaffey, D., *E-Business and E-Commerce Management* (2nd Ed.), Hemel Hempstead: Prentice Hall, 2004.
- Checkland, P., *Systems Thinking, Systems Practice*, Chichester: John Wiley, 1981.
- Checkland, P., unpublished presentation given at *Systems for Sustainability: People, Organisations and Environments*, 5th International Conference of the United Kingdom Systems Society, Milton Keynes: The Open University, July 1997.
- Checkland, P. and Holwell, S., *Information, Systems and Information Systems: Making Sense of the Field*, Chichester: John Wiley, 1998.
- Checkland, P. and Scholes, J., *Soft Systems Methodology in Action*, Chichester: John Wiley, 1990.
- Cheesman, J. and Daniels, J., *UML Components: A Simple Process for Specifying Component-Based Software*, Upper Saddle River, NJ: Addison-Wesley, 2001.
- Coad, P. and Yourdon, E., *Object-Oriented Analysis* (2nd Ed.), Englewood Cliffs, NJ: Yourdon Press; Prentice-Hall, 1990.
- Coad, P. and Yourdon, E., *Object-Oriented Design*, Englewood Cliffs, NJ: Yourdon Press; Prentice-Hall, 1991.
- Coad, P. with North, D. and Mayfield, M., *Object Models: Strategies, Patterns and Applications* (2nd Ed.), Upper Saddle River, NJ: Yourdon Press; Prentice-Hall, 1997.
- Cockburn, A., *Writing Effective Use Cases*, Reading, MA: Addison-Wesley, 2000.
- Codd, E. J., 'A Relational Model for Large Shared Data Banks', *Communications of the ACM*, 13, (6), pp. 377–387, 1970. (Also 26, (1), pp. 64–69, 1983.)
- Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F. and Jeremaes, P., *Object-Oriented Development: The Fusion Method*, Englewood Cliffs, New Jersey: Prentice-Hall International, 1994.
- Collins, T., 'Banking's Big Brother option', *Computer Weekly*, 19 November, 1998a.
- Collins, T., 'MPs lambast NHS for double project fiasco', *Computer Weekly*, 12 December, 1998b.
- Collins, T., 'Lords inquiry to follow up Chinook campaign', *Computer Weekly*, 8 March, 2001.
- Connor, D., *Information System Specification and Design Road Map*, Englewood Cliffs, NJ: Prentice-Hall, 1985.
- Constantine, L., 'The Case for Essential Use Cases', *Object Magazine*, May, 1997.
- Cook, S. and Daniels, J., *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Hemel Hempstead: Prentice-Hall, 1994.
- Coplien, J. O., *Advanced C++: Programming Styles and Idioms*, Reading, MA: Addison-Wesley, 1992.
- Coplien, J. O., 'A Generative Development Process Pattern Language' in Coplien, J. O. and Schmidt, D. C. (Eds.), *Pattern Languages of Program Design*, Reading MA: Addison-Wesley, 1995.
- Coplien, J. O., *Software Patterns*, New York: SIGS Books, 1996.
- Coplien, J. O. and Schmidt, D. C. (Eds.), *Pattern Languages of Program Design*, Reading, MA: Addison-Wesley, 1995.
- Cunningham, W., 'The CHECKS Pattern Language of Information Integrity', in Coplien, J. O. and Schmidt, D. C., (Eds.), *Pattern Languages of Program Design*, Reading, MA: Addison-Wesley, 1995.
- Daniels, G., Davis, D., Nakamura, Y., Simeonov, S., Boubez, T., Graham, S. and Brittenham, P., *Building Web Services with Java: Making Sense of XML, Soap, WSDL and UDDI*, Indianapolis, IN: Que, 2004.
- Date, J., unpublished presentation given at *Migrating to Object Technology*, TattOO '95, Leicester: De Montfort University, January 1995.
- Date, C. J., *An Introduction to Database Systems* (8th Ed.), Reading, MA: Addison-Wesley, 2003.
- Date, C. J. and Darwen, H., *Foundation for Object/Relational Databases: The Third Manifesto*, Reading, MA: Addison-Wesley, 1998.
- de Champeaux, D., *Object-Oriented Development Process and Metrics*, Upper Saddle River, NJ: Prentice Hall, 1997.
- Deitel, H. M. and Deitel, P. J., *Java: How to Program*, Upper Saddle River, NJ: Prentice-Hall, 2004.
- DeMarco, T., *Structured Analysis and System Specification*, Upper Saddle River, NJ: Yourdon Press; Prentice-Hall, 1979.
- DeMarco, T., *Controlling Software Projects*, Englewood Cliffs, NJ: Yourdon Press, 1982.
- DeMarco, T. and Boehm, B., The Agile Methods Fray, *Software Technologies*, June, 2002.
- DiCarlo, L. 'Amazon.com: Bookworm Turned Internet Empire', *Forbes Magazine online*, 13 April, 2004, at www.forbes.com/ebusiness/2004/04/13/cx_id_0413fastest.html.
- Disability Rights Commission, *Code of Practice: Rights of Access—Goods, Facilities, Services and Premises*, London: The Stationery Office, 2002.
- Dix, A., Finlay, J., Abowd, G. and Beale, R., *Human-Computer Interaction*, Hemel Hempstead: Prentice-Hall, 1998.
- Douglass B. P., *Real-Time UML Second Edition: Developing Efficient Objects for Embedded Systems*, Reading, MA: Addison-Wesley; ACM Press, 1999.
- Drummond, H., 'The politics of risk: trials and tribulations of the Taurus project', *Journal of Information Technology*, 11, pp. 347–357, 1996.
- DSDM Consortium, The DSDM Lifecycle, 2004, at <http://www.dsdm.org/en/about/lifecycle.asp>.
- DSDM Consortium/Stapleton, J. (Ed), *DSDM: Business Focused Development* (2nd Ed.), Harlow: Addison Wesley/Pearson Education, 2003.
- Eaglestone, B. and Ridley, M., *Object Databases: An Introduction*, Maidenhead: McGraw-Hill, 1998.
- e-GIF, *e-Government Interoperability Framework* (Version 6.0), Office of the e-Envoy, 2004.
- Ellwood, W., 'Seduced by Technology', *New Internationalist*, December, 1996.
- Eriksson, H-E. and Penker, M., *UML Toolkit*, New York: John Wiley, 1998.
- Fedorov, A., *A Programmer's Guide to .Net*, London: Addison-Wesley, 2002.
- Fitzgerald, B., Russo, N. L. and Stolterman, E., *Information Systems Development: Methods in Action*, Maidenhead: McGraw-Hill, 2002.

- Flowers, S., 'IS Project Risk—The Role of Management in Project Failure', *Proceedings of BIT '97*, Manchester Metropolitan University, November 1997.
- Flynn, D., *Information Systems Requirements: Determination and Analysis* (2nd Ed.), Maidenhead: McGraw-Hill, 1998.
- Fowler, M., *Analysis Patterns: Reusable Object Models*, Reading, MA: Addison-Wesley, 1997.
- Fowler, M., *Refactoring: Improving the Design of Existing Code*, Boston: Addison Wesley Professional, 1999.
- Fowler, M., *UML Distilled: A Brief Guide To The Standard Object Modeling Language* (3rd Ed.), Reading, MA: Addison-Wesley, 2003.
- Fowler, M., *The New Methodology*, 2004, at <http://www.martinfowler.com/articles/newMethodology.html>.
- Fowler, M. and Highsmith, J., *The Agile Manifesto, Software Development*, August, 2001.
- Gabriel, R., *Patterns of Software: Tales from the Software Community*, New York: Oxford University Press, 1996.
- Gaines, B. R. and Shaw, M. L. G., 'Dialogue engineering' in Sime, M. E. and Coombs, M. J. (Eds.), *Designing for Human-Computer Communication*, London: Academic Press, 1983.
- Gain, D., *Software Quality Assurance: From Theory to Implementation*, Harlow: Addison Wesley/Pearson Education, 2003.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. M., *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1995.
- Garland, J and Anthony, R., *Large-Scale Software Architecture: A Practical Guide Using UML*, Chichester: John Wiley, 2003.
- Gilb, T., *Principles of Software Engineering Management*, Wokingham: Addison-Wesley, 1988.
- Goldsmith, S., *A Practical Guide to Real-Time Systems Development*, Hemel Hempstead: Prentice Hall, 1993.
- Goodland, M. with Slater, C., *SSADM Version 4: A Practical Approach*, Maidenhead: McGraw-Hill, 1995.
- Goss, E. P., Buelow, V. and Ramchandani, H., 'The Real Impact of a Point-of-Care Computer System on ICU Nurses', *Journal of Systems Management*, pp. 43–47, January/February, 1995.
- Graham, I., Henderson-Sellers, B. and Younessi, H., *The OPEN Process Specification*. Harlow: Addison-Wesley, 1998.
- Greenbaum, J. and Kyng, M., *Design at Work: Cooperative Design of Computer Systems*, Hillsdale, NJ: Erlbaum, 1991.
- Hammersley, M. and Atkinson, P., *Ethnography: Principles in Practice* (2nd Ed.), London: Routledge, 1995.
- Harel, D., 'Statecharts: a visual formalism for complex systems', *Science of Computer Programming*, 8, pp. 231–274, 1987.
- Harel, D., 'On visual formalisms', *Communications of the ACM*, 31, (5), pp. 514–530, 1988.
- Harel, D. and Politi, M., *Modeling Reactive Systems with Statecharts: The Statemate Approach*, New York, NY: McGraw-Hill, 1998.
- Harrison, N., Foote, B. and Rohnert, H., *Pattern Languages of Program Design 4*, Addison Wesley, 2000.
- Hart, A., *Knowledge Acquisition for Expert Systems*, London: Kogan Page, 1989. (New edition 1997 published by Chapman & Hall.)
- Hatley, D. J. and Pirbhai, I. A., *Strategies for Real-Time Systems Design*, New York: Dorset House, 1987.
- Hay, D., *Data Model Patterns Conventions of Thought*, New York: Dorset House, 1996.
- Health and Safety Executive, *Work with Display Screen Equipment*, Publication No. L26, Sudbury: HSE Books, 2003.
- Hicks, M., J., *Problem Solving in Business and Management*, London: Chapman & Hall, 1991.

- Hoffer, J. A., George, J. F. and Valacich, J. S., *Modern Systems Analysis and Design* (4th Ed.), Upper Saddle River, NJ: Prentice Hall, 2005.
- Hopkins, T. and Horan, B., *Smalltalk: An Introduction to Application Development using VisualWorks*, Hemel Hempstead: Prentice-Hall, 1995.
- Horrocks, I., *Constructing the User Interface with Statecharts*, Harlow: Addison-Wesley, 1999.
- Howe, D. R., *Data Analysis for Data Base Design* (3rd Ed.), Oxford: Butterworth-Heinemann, 2001.
- Humphrey, W. S., *Managing the Software Process*, Wokingham: Addison-Wesley, 1989.
- Hyde, M., 'Boo rises from the ashes', *Guardian*, 3 November, 2000.
- IBM, *IBM-Rational Unified Process*, 2005, at <http://www-306.ibm.com/software/rational/index.html> or <http://www-130.ibm.com/developerworks/rational/products/rup/>.
- IBM, *IBM SanFrancisco Base*, Rochester, MN: IBM, 1998.
- IEEE, *Standard 1471-2000*, 'IEEE Recommended Practice for Architectural Description of Software-Intensive Systems', IEEE, 2000.
- Jackson, M., *Principles of Program Design*, London: Academic Press, 1975.
- Jacobson, I., Booch, G., and Rumbaugh, J., *The Unified Software Development Process*, Reading, MA: Addison-Wesley; ACM Press, 1999.
- Jacobson, I., Christerson, M., Jonsson, P. and Övergaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Wokingham: Addison-Wesley, 1992.
- Jacobson, I., Ericsson, M. and Jacobson, A., *The Object Advantage: Business Process Reengineering with Object Technology*, New York, NY: ACM Press, 1995.
- Jacobson, I., Griss, M. and Jonsson, P., *Software Reuse: Architecture, Process and Organization for Business Success*, Harlow: Addison-Wesley, 1997.
- JDOEG (Java Data Objects Expert Group), *Java Data Objects JSR12 Version 1.0.1*, Santa Clara, CA: Sun Microsystems, 2003.
- Jeffries, R., *Essential XP: Documentation*, XProgramming.com, 2001, at <http://www.xprogramming.com/xpmag/expDocumentationInXP.htm>.
- Kendall, K. E. and Kendall, J. E. *Systems Analysis and Design* (6th Ed.), Upper Saddle River, NJ: Prentice Hall, 2005.
- Koestler, A., *The Ghost In The Machine*, London: Hutchinson, 1967.
- Korth, H. F. and Silberschatz, A., *Database System Concepts* (5th Ed.), New York: McGraw-Hill, 2001.
- Kroll, P. and Kruchten, P., *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*, Boston: Addison-Wesley, 2003.
- Kruchten, P., *The Rational Unified Process: An Introduction*, Reading, MA: Addison-Wesley, 2004.
- Kyng, M., 'Making Representations Work', *Communications of the ACM*, 38, 9, pp. 46–55, September, 1995.
- Larman, C., *Agile and Iterative Development: A Manager's Guide*, Boston, MA: Addison Wesley/Pearson Education, 2003.
- Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd Ed.), Upper Saddle River, NJ: Prentice-Hall, 2005.
- Laudon, K. C. and Laudon, J. P., *Management Information Systems* (7th Ed.), Upper Saddle River, NJ: Prentice-Hall, 2002.
- Loomis, M. E. S., *Object Databases: The Essentials*, Reading, MA: Addison-Wesley, 1995.
- Lorenz, M. and Kidd, J., *Object-Oriented Software Metrics: A Practical Guide*, Englewood Cliffs, NJ: Prentice-Hall, 1994.
- Maciaszek, L. A., *Requirements Analysis and Design* (2nd Ed.), Harlow: Addison Wesley/Pearson Education, 2005.
- Maguire, M., *RESPECT User Requirements Framework Handbook*, European Usability Support Centres, 1997.
- Martin, D., Riehle, D. and Buschmann, F. (Eds.), *Pattern Languages of Program Design 3*, Reading, MA: Addison-Wesley, 1998.

- McBride, N., 'Business Use of the Internet: Strategic Decision or Another Bandwagon?', *European Management Journal*, 15, pp. 58–67, 1997.
- Melton, A. (Ed.), *Software Measurement*, London: Thompson Computer Press, 1995.
- Metsker, S., *The Design Patterns Java Workbook (Workbook S.)*, Indianapolis, IN: Addison Wesley Professional, 2002.
- Meyer, B., *Object Success: A Manager's Guide to Object Orientation, its Impact on the Corporation, and its Use for Reengineering the Software Process*, Hemel Hempstead: Prentice-Hall, 1995.
- Meyer, B., *Object-Oriented Software Construction* (2nd Ed.), Upper Saddle River, NJ: Prentice-Hall PTR, 1997.
- Meyer, B., *Object-Oriented Software Construction*, Hemel Hempstead: Prentice-Hall International, 1988.
- Meyer, B., 'Design by Contract', in *Advances in Object-Oriented Software Engineering*, Mandrioli, D. and Meyer, B. (Eds), London: Prentice-Hall, 1991.
- Microsoft Corporation, *The Windows Interface Guidelines for Software Design*, Redmond, WA: Microsoft Press, 1997.
- Muller, P.-A., *Instant UML*, Birmingham: Wrox Press, 1997.
- Newkirk J. and Martin R., *Extreme Programming in Practice*, Upper Saddle River, NJ: Addison-Wesley, 2001.
- OGC, *Common Causes of Project Failure: OGC Best Practice Guide*, Office of Government Commerce, 2005, at http://www.ogc.gov.uk/sdtkdev/new_content/OGCCausesProjectFailure.pdf.
- OMG, *The Common Object Request Broker Architecture and Specification; Revision 2.0*, Framingham, MA: Object Management Group Inc, 1995.
- OMG, *UML 2.0 OCL Specification*, Framingham, MA: Object Management Group Inc, 2003.
- OMG, *UML 2.0 Superstructure RFP (ad/00-09-02)*, Framingham, MA: Object Management Group Inc, 2000.
- OMG, *Unified Modeling Language: Infrastructure Version 2.0*, Framingham, MA: Object Management Group Inc, 2004a.
- OMG, *Unified Modeling Language: Superstructure Version 2.0*, Framingham, MA: Object Management Group Inc, 2004b.
- OMG, *Unified Modeling Language: Version 2.0*, Framingham, MA: Object Management Group Inc, 2004c.
- Oppenheim, A. N., *Questionnaire Design, Interviewing and Attitude Measurement*, London: Continuum International, 2000.
- Orfali, R. and Harkey, D., *Client/Server Programming with JAVA and CORBA* (2nd Ed.), New York, NY: John Wiley, 1998.
- Page-Jones, M., *The Practical Guide to Structured Systems Design* (2nd Ed.), Englewood Cliffs, NJ: Prentice-Hall, 1988.
- Peters, T., *Thriving on Chaos: Handbook for a Management Revolution*, London: Macmillan, 1988.
- Police G., *Using the Rational Unified Process for Small Projects: Expanding upon eXtreme Programming*, Rational Software Corporation, 2001.
- Philips, T., 'A Slicker System For The City', *The Guardian On-Line*, 3 April, 1997.
- Porter, M., *Competitive Strategy*, New York: Free Press, 1985.
- Preece, J. with Rogers, Y., Sharp, H., Benyon, D., Holland, S. and Carey, T., *Human-Computer Interaction*, Wokingham: Addison-Wesley, 1994.
- Pressman, R., *Software Engineering: A Practitioner's Approach* (6th Ed.), Maidenhead: McGraw-Hill, 2004.
- Ranger, S., 'IT still not delivering for business', *Web Active Magazine*, 2005, at <http://www.webactivemagazine.co.uk/news/1161398>.
- Rayport, J. F. and Jaworski, B. J., *Introduction to E-Commerce*, London: McGraw-Hill (ISE), 2003.
- Riehle, D. and Zullighoven, H., 'Understanding and Using Patterns in Software Development', *Theory and Practice of Object Systems*, 2, (1), pp. 3–13, 1996.
- Rifkin, J., *The End of Work*, New York: Putnam Publishers, 1995. Cited in Ellwood, W., 'Seduced by Technology', *New Internationalist*, December, 1996.
- Roberts, G., *The Users' Role in Systems Development*, Chichester: John Wiley; Central Computer and Telecommunications Agency, 1989.
- Robinson, B. and Prior, M., *Systems Analysis Techniques*, London: International Thomson Publishing, 1995.
- Roos, R., *Java Data Objects*, London: Addison-Wesley, 2003.
- Rosenberg, D. with Scott, K., *Use Case Driven Object Modeling with UML: A Practical Approach*, Upper Saddle River, NJ: Addison-Wesley, 1999.
- Rosson, M. B. and Carroll, J. M., 'Narrowing the Specification-Implementation Gap in Scenario-Based Design', in Carroll, J. M. (Ed.), *Scenario-Based Design: Envisioning Work and Technology in System Development*, New York: John Wiley, 1995.
- Royce, W. W., Managing the development of large software systems, in *Proc. WESTCON*, San Francisco CA, 1970.
- Rumbaugh, J., 'Models through the development process', *Journal of Object-Oriented Programming*, 10, (2), pp. 5–8, 14, 1997.
- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W., *Object-Oriented Modeling and Design*, Englewood Cliffs, NJ: Prentice-Hall International, 1991.
- Rumbaugh, J., Jacobson, I. and Booch, G., *The Unified Modeling Language Reference Manual*, Reading, MA: Addison-Wesley; ACM Press, 1999.
- Sachs, P., 'Transforming Work: Collaboration, Learning and Design', *Communications of the ACM*, 38, (9), pp. 36–44, 1995.
- SAGA, *Standards and Architectures for e-Government Applications* (Version 2.0), Berlin: German Federal Ministry of the Interior, 2003.
- Sanders, J. and Curran, E., *Software quality: a framework for success in software development and support*, Wokingham: Addison-Wesley, 1994.
- Sauer, C., *Why Information Systems Fail: A Case Study Approach*, Maidenhead: McGraw-Hill, 1993.
- Schmidt D., Stal M., Rohnert H., and Buschmann, F., *Pattern Oriented Software Architecture: Patterns for Concurrent and Networked Objects: Volume 2*, Chichester: John Wiley, 2000.
- Schneider, K., 'Bug delays £25m court case system', *Computer Weekly*, 22 December, 1997.
- Selic B., Gullekson G., and Ward P. T., *Real-Time Object-Oriented Modeling*, New York, NY: John Wiley 1994.
- Shackel, B., 'Human factors and usability', in Preece, J. and Keller, L. (Eds.), *Human-Computer Interaction: Selected Readings*, Hemel Hempstead: Prentice-Hall, 1990.
- Shlaer, S. and Mellor, S., *Object-Oriented Systems Analysis: Modeling the World in Data*, Englewood Cliffs, NJ: Prentice-Hall, 1988.
- Shlaer, S. and Mellor, S., *Object Lifecycles: Modeling the World in States*, Englewood Cliffs, NJ: Prentice-Hall, 1992.
- Shneiderman, B. and Plaisant, C., *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (4th Ed.), Reading, MA: Addison-Wesley, 2004.
- Skidmore, S., Mills, G. and Farmer, R., *SSADM Models & Methods*, Manchester: NCC Blackwell, 1994.
- Software Engineering Institute, *Capability Maturity Model Integration*, Carnegie Mellon University, 2000, at <http://www.sei.cmu.edu/cmmi>.
- Sommerville, I., *Software Engineering* (4th Ed.), Reading, MA: Addison-Wesley, 1992.
- Sommerville, I., *Software Engineering* (7th Ed.), Reading, MA: Addison-Wesley, 2004.
- Soni, D., Nord, R. and Hofmeister, C., 'Software Architecture in Industrial Applications', in *Proceedings of the 17th International Conference on Software Engineering*, pp. 196–207, Seattle, WA: ACM Press, 1995.
- Sowa, J. F. and Zachman, J. A., 'Extending and Formalizing the Framework for Information Systems Architecture', *IBM Systems Journal*, Vol. 31, No. 3, pp. 590–616, 1992.
- Standish Group, *The Chaos Report* (1994), Standish Consulting Group, 1995, at http://www.standishgroup.com/sample_research/chaos_1994_1.php.

- Standish Group, *Extreme CHAOS* (2001), Standish Consulting Group, 2001, at http://www.standishgroup.com/sample_research/PDFpages/extreme_chaos.pdf.
- Stapleton, J., *Dynamic Systems Development Method*, Harlow: Addison-Wesley, 1997.
- Sun Java Centre, 2005, at <http://java.sun.com>.
- Symons, V., 'Evaluation of information systems: IS development in the processing company', *Journal of Information Technology*, 5, pp. 194–204, 1990.
- Tanenbaum, A. S., Day, W. and Waller, S., *Computer Networks* (4th Ed.), Englewood Cliffs, NJ: Prentice Hall, 2002.
- Texel, P. P., and Williams, C. B., *Use Case Combined with Booch/OMT/UML: processes and products*, Upper Saddle River, NJ: Prentice Hall, 1997.
- The Open Group, *TOGAF 8.1*, 2002, at <http://www.opengroup.org/architecture/togaf8-doc/arch/toc.htm>.
- Turban, E. and Aronson, J. E., *Decision Support Systems and Intelligent Systems* (6th Ed.), Upper Saddle River, NJ: Prentice Hall, 2001.
- Vidgen, R., Avison, D., Wood, B. and Wood-Harper, T., *Developing Web Information Systems: From Strategy to Implementation*, Oxford: Butterworth-Heinemann, 2002.
- Vlissides, J. M., Coplien, J. O. and Kerth, N. (Eds.), *Pattern Languages of Program Design 2*, Reading, MA: Addison-Wesley, 1996.
- Wall Street Journal*, 22 September, 1998, reprinted in *The Guardian Editor*, 26 September, 1998.
- Ward, P. and Mellor, S., *Structured Development for Real-Time Systems, Vol 1: Introduction & Tools*, Englewood Cliffs, NJ: Yourdon Computing Press, 1985a.
- Ward, P. and Mellor, S., *Structured Development for Real-Time Systems, Vol 2: Essential Modelling Techniques*, Englewood Cliffs, NJ: Yourdon Computing Press, 1985b.
- Ward, P. and Mellor, S., *Structured Development for Real-Time Systems, Vol 3: Implementation Modelling Techniques*, Englewood Cliffs, NJ: Yourdon Computing Press, 1986.
- Warmer, J. and Kleppe, A., *The Object Constraint Language: Getting Your Models Ready for MDA* (2nd Ed.), Boston: Pearson Education, 2003.
- Webster, F., *Theories Of The Information Society*, London: Routledge, 1995.
- Weir, C. and Daniels, J., 'Software Architecture Document', in *Proceedings of Object Technology 98*, Oxford, 1998.
- Whiteside, J., Bennett, J. and Holtzblatt, K., 'Usability engineering: our experience and evolution', in Helander, M. (Ed.), *Handbook of Human-Computer Interaction*, Amsterdam: North-Holland, 1988.
- Whitten, J., Bentley, L. and Barlow, V., *Systems Analysis and Design Methods* (3rd Ed.), New York, NY: Irwin, 1994.
- Wirs-Brock, R., Wilkerson, B. and Wiener, L., *Designing Object-Oriented Software*, Englewood Cliffs, NJ: Prentice-Hall International, 1990.
- Wutka, M., *Special Edition Using Java 2 Enterprise Edition (J2EE): With JSP, Servlets, EJB 2.0, JNDI, JMS, JDBC, CORBA, XML and RMI*, Indianapolis, IN: Que, 2001.
- Yourdon, E., *Modern Structured Analysis*, Englewood Cliffs, NJ: Prentice-Hall, 1989.
- Yourdon, E., *Object-Oriented Systems Design: An Integrated Approach*, Englewood Cliffs, NJ: Prentice-Hall International, 1994.
- Yourdon, E., *Structured Walkthroughs*, Englewood Cliffs, NJ: Yourdon Press, 1985.
- Yourdon, E. and Constantine, L. L., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Englewood Cliffs, NJ: Yourdon Press; Prentice-Hall, 1979.
- Zachman, J. A., 'Framework for Information Systems Architecture', *IBM Systems Journal*, Vol. 26, No. 3, pp. 276–292, 1987.
- Zuboff, S., *In The Age Of The Smart Machine: The Future of Work and Power*, Oxford: Heinemann, 1988.

Index

- about this book
 acknowledgements xvii
 background xi
 case studies xii
 changes in third edition xvi
 exercises for readers xiii
 guided tour xix–xx
 main subjects 2
 pathways through the book xiv–xv
 structure of the book xiii–xix
 systems of interest 14–15
 target readership xi–xii
 transferable skills xv
 UML, latest version of xv
 website and support materials xv
- abstract classes 648
 requirements model, refinement of 231–2
- abstract operation 648
 abstraction 648
 increase in object-orientation 84
 levels in software development 603–4
- abstraction mechanisms
 requirements model, refinement of 247
 in UML 222–3
see also generalization; composition; encapsulation
- academic study of IS 2
- Access 509
 access
 access control in security subsystem as reusable component 591–2
 direct access to files 502
 to files 500–3
 implications of access to information 42–3
 improving access to files 502
 index-sequential access to files 500–2
 to information, implications of 42–3
 serial access to files 500
- accuracy in algorithm design 412
- actions 648
 actions list
 implementation 572
 activation 648
see also execution 648
- active objects 269
- ActiveX 589
- activity 648
 activity diagrams 113–19, 648
 operations, specification of 294–5
- activity edge 116, 648
- activity-expressions 308–9
- activity final node 648
- activity partitions 117, 118, 648
- actors (system users) 648
 in requirements capture 146
 and use cases 145–54
- ADA 84, 578
- ADL (Architecture Description Language) 342
- adornment 648
- AFE (Application Family Engineering) 589
- Agate Ltd 91–4, 160–70, 213–21, 328–37, 536–50
- business activities in current system 93
- computer systems existing at 92
- design considerations
 activities of design workflow 550
 architecture 536–8
 class diagrams 538–44
 database design 549
 sample use case 538
- sequence
 diagrams 544–8
 state machines 547, 549
- introduction to Agate Ltd 91–2
- requirements for new system, summary of 94
- requirements model 160–70, 213–21
- activity diagrams 169–70
- actors 161–2
- analysis class diagram, assembly of 218–20
- class diagram, further refinement of 335–6
- detail levels of the model, improvements in 328
- further analysis 328–37
- glossary of specialist terms 167–8
- improving understanding of the domain 328
- initial architecture 168–9
- operation specifications 330–5
- refinement of, improving level of detail 328
- requirements analysis, activities of 220–1
- requirements analysis, details highlighted 336–7
- requirements and use case list 160–1
- sequence diagrams 329–30
- state machines 330
- use case descriptions 163–5
- use case modelling (and elaboration) 165–7
- use case realizations 213–18
- software architecture 366–8
- UK office staff at 92

aggregation 235–6, 236–7, 648
 Agile methods 61–2, 171, 605–6
 Algol 84
 algorithms 648
 accuracy in algorithm design 412
 algorithmic (procedural) approach to operation logic 282, 289–95
 computational complexity in algorithm design 412
 control structures in algorithms 282, 290
 costs in algorithm design 412
 design of, constraints on 412
 detailed design of, factors for consideration in 412–13
 flexibility in algorithm design 412
 implementation costs in algorithm design 412
 object model, fine tuning in algorithm design 413
 performance constraints in algorithm design 412
 platform capabilities in algorithm design 412
 aliases of patterns 419
 alternative courses (paths) in use cases 147–8
 Amazon.com 31
 Americans With Disabilities Act (1990) 460
 analysis and design difference between 371, 372–4
 qualities and objectives of 377–86
 analysis class diagram 171–2
 assembly of 207–8
 analysis class stereotypes 176–9, 648
 analysis patterns 245–6
 answer pointers 635–47
 antipatterns 229, 648
 API (Application Programming Interface) 345, 519, 532, 554–6, 600
 Apple Macs 92, 376

application containers 554
 application domain, concepts in 190
 application of design patterns 436
 application servers 532–3
 architecture
 architectural models 342–4
 architectural styles 348–63
 for persistence 496–8
 see also system architecture
 ArcStyler (InteractiveObjects) 346
 artifacts 610, 649
 ASCII values 502
 ASE (Application System Engineering) 589
 ASP (Active Server Pages) 498
 assembly language 84
 association 649
 association class 649
 association instances (links) 182–3, 649
 associations 182–4
 design of 404–9
 finding and adding 202–3
 many-to-many associations 407–8
 object state and 183–4
 one-to-many associations 405–7
 one-to-one associations 404–5
 reality in 199–200
 significance of 183
 summaries of notation 628
 asynchronous messages 267, 268, 649
 ATM (Automatic Teller Machine) 503
 attributes 649
 adding and locating 200–2
 attribute declarations 397–8
 reality in 198–9
 in requirements analysis 180–2
 background reading 132–3
 backup files 504
 BASIC 84
 basic concepts

notation for state machines 306–11
 notation in interaction sequence diagrams 253–62
 object-orientation 68, 69–83
 B2B (business-to-business) 23
 B2C (business-to-customer) 23
 behaviour specifications 146
 behavioural patterns 419, 429–35
 behavioural state machines 324–5, 649
 black box testing 562
 Blaha 519
 blaming users for problems 35
 Boo.com 39, 43, 44
 Borland C 375
 bottom-up approach to finding generalization 234
 boundary classes 176, 177, 473, 475, 480, 649
 design of 483–91
 business and user interface classes, separation of 464–5
 case study work, exercises and projects 492–3
 class diagram, adding boundary classes 481
 class libraries 472
 classes, design of 469–73
 composite design pattern 483
 further reading 493
 horizontal prototyping 466
 inheritance hierarchy of boundary classes 464, 481
 interaction, designing with sequence diagrams 473–81
 interface, modelling using state machines deep history indicator 487–8

event-action table, preparation of 489–91
 high-level requirements, description of 484–5
 main user tasks, description of 484–5
 state machine, drawing up 485–9
 tasks involved 484
 user interface behaviour, description of 485
 user interface rules, definition of 485
 interface design patterns, use of 482–3
 learning objectives 463
 observer design pattern 483
 outside-in development of applications 467
 packages and 463–4
 presentation layer architecture 464–6
 report design 480–1
 review questions 492
 strategy design pattern 483
 throwaway prototyping 466
 user interface, prototyping 466–9
 vertical prototyping 466
 BPEL4WS (Business Process Execution Language for Web Services) 154
 BPM (Business Process Management) 580
 BPMN (Business Process Modelling Notation) 154
 branching 262–3
 broker architecture for distributed systems 360–1
 budget constraints 33–4
 bug reporting databases 573
 buildable design 381
 business and user interface classes, separation of 464–5
 business modelling 49, 50
 with use case diagrams 153–4
 business rules 184–6, 649
 business strategies, identification of 19–20
 business study phase in DSDM 615, 616
 C# 34, 84, 87, 375, 471, 472, 553, 555, 557, 589
 C++ 84, 87, 229, 375, 419, 421, 497, 508, 520, 522, 523, 524, 553, 556, 578
 call triggers 305
 callback 267
 candidate components for reuse 572
 capta 649
 capturing knowledge of successful solutions 415–16
 cardinalities 649
 allowed (multiplicity) 184–6
 CASE (Computer Aided Software Engineering) 143, 389
 benefits of using 65
 code generators in 64–5
 completeness in 63
 consistency in 63
 construction in 64–5
 development problems, avoidance of 62–5
 difficulties of using 65
 encyclopaedias in 63
 layering in 63–4
 maintenance tools in 65
 model and technique support in 62–4
 navigation to linked diagrams in 63
 performance analysis in 64
 report generation in 64
 repositories in 63
 syntactic correctness in 63
 system simulation in 64
 tool repository 197
 tool support in requirements capture 150, 153
 tools 111, 279, 472, 552, 554, 555, 620
 traceability in 64
 case studies

Agate Ltd 91–4, 160–70, 213–21, 328–37, 536–50
 business activities in current system 93
 computer systems existing at 92
 design considerations activities of design workflow 550
 architecture 536–8
 class diagrams 538–44
 database design 549
 sample use case 538
 sequence diagrams 544–8
 state machines 547, 549
 introduction to Agate Ltd 91–2
 requirements for new system, summary of 94
 requirements model 160–70, 213–21
 activity diagrams 169–70
 actors 161–2
 analysis class diagram, assembly of 218–20
 class diagram, further refinement of 335–6
 detail levels of the model, improvements in 328
 further analysis 328–37
 glossary of specialist terms 167–8
 improving understanding of the domain 328
 initial architecture 168–9
 operation specifications 330–5
 refinement of, improving level of detail 328
 requirements analysis, activities of 220–1
 requirements analysis, details highlighted 336–7

case studies
 Agate Ltd
 requirements model
 (*continued*)
 requirements and use
 case list 160–1
 sequence
 diagrams 329–30
 state machines 330
 use case
 descriptions 163–5
 use case modelling
 (and elaboration)
 165–7
 use case
 realizations 213–18
 UK office staff at 92
 fact-finding techniques,
 tasks covered 142–3
 FoodCo Ltd 95–102
 current thinking at 98
 employees 96
 information systems
 at 98–101
 information systems
 team meeting 102
 introduction 95–8
 organization
 structure 96
 payroll and costing 100
 problems (and possible
 solution, with IS
 implications) 97
 problems in product
 costing 100–1
 costing
 activities 99–101
 production control, line
 operations 99–100
 report of consultants on
 information
 systems 99
 system proposed for
 development 102
 security subsystem as
 reusable
 component 591–9
 access control 591–2
 class diagram showing
 collection classes 597
 class diagram showing
 control classes 596
 collection classes 596–7
 design
 alternatives 592–3

Façade pattern 594–5
 interface 593–4, 599
 persistence
 mechanism 596
 security classes, separate
 packaging for 593–4
 security package
 dependencies 594,
 597–8
 security server control
 classes 594, 596
 use case descriptions,
 development of
 155–7
 case study work, exercises
 and projects
 boundary classes, design
 of 492–3
 control, specification
 of 327
 data management
 design 534
 design patterns 438
 detailed design 414
 development problems,
 avoidance of 66
 development problems,
 information
 systems 45–6
 HCI (Human-Computer
 Interaction) 461–2
 implementation 576
 information systems
 24–5
 modelling concepts 127
 object interaction 281
 object-orientation 89–90
 operations, specification
 of 302
 requirements
 analysis 209–12
 requirements
 capture 158–9
 requirements model,
 refinement of 247–8
 reusable
 components 600–1
 software development
 processes 625
 system architecture 369
 systems design 393
 Caseman system 39
 categories of classes 197–8
 CBD (Component-Based
 Development) 650

change
 changeability in design
 patterns 419–20
 developers' perspective
 on 33
 management of in systems
 design 373–4
 recommendation in
 prototyping 55
 triggers 305
 user requirements and
 timescales 32–3
Chaos Reports 27, 30–1
 characteristics
 of good dialogues 446–9
 of object-oriented
 languages 87–8
 of systems already in
 use 129–30
 CHECKS Pattern Language
 of Information
 Integrity 419
 Child Support Agency,
 UK 27
 choice pseudostates 315–17,
 649
 class diagrams 649
 adding boundary
 classes 481
 communication diagrams
 and 194–6
 derived from
 communication
 diagrams 191, 194–6
 in requirements
 analysis 179–203
 showing collection
 classes 597
 showing control
 classes 596
 classes 649
 abstract classes
 requirements model,
 refinement of 231–2
 association class 649
 boundary classes 176, 177,
 473, 475, 480, 649
 business and user interface
 classes, separation
 of 464–5
 class and object 71–2, 74
 class as descriptor for
 collection of objects 191
 class browsers 554
 class cohesion 384
 class design 122, 124–5
 keeping them to a
 minimum 409
 mapping classes to
 tables 516–19
 class names 191
 class notation 627
 class-scope 649
 class
 specification 397–402
 class (static)
 operations 421
 collection classes 596–7
 design of boundary
 classes 469–73
 duplication of 198
 dynamic interaction
 among 189
 entity classes 176, 177–8,
 651
 generality in class 75, 76
 identification of 189–94,
 196–200
 application domain,
 concepts in 190
 class diagrams derived
 from communication
 diagrams 191, 194–6
 class names 191
 collaboration, further
 refinement in 193
 collaboration,
 initial 190–1
 collaboration, refinement
 in 191–3
 communication diagram
 frame 193
 communication
 diagrams 190–1, 192,
 193, 194
 design issues 191–2
 dynamic interaction
 among classes 189
 first steps 189
 identification of objects
 in collaboration
 189–90
 requirements
 analysis 189–94,
 196–200
 use case description,
 extended
 version 189–90
 and instances, relative
 stability of 179–80
 interface class 654
 Java Swing classes 375

initial 190–1
 object interaction
 and 250–2
 refinement in identification
 of classes 191–3
 further refinement 193
 template collaboration 657
 collection classes 650
 detailed design 408
 in security subsystem as
 reusable
 component 596–7
 combined fragments 650
 in interaction sequence
 diagrams 254–5
 commercial pressures 31
 communication
 diagrams 650
 class diagrams and 194–6
 classes, identification
 of 190–1, 192, 193, 194
 diagram frame 193
 and interaction sequence
 diagrams 271–5
 object interaction 271–5
 basic concepts and
 notation 271–3
 message labels in 273–5
 sequence diagrams
 and 275
 purpose of 176
 requirements
 analysis 174–6
 requirements capture 146
 communication path 650
 compilers 553
 completeness in analysis 377,
 378
 completeness in CASE 63
 complex structures in
 Structured
 English 293–4
 complexity in USDP 610–11
 component diagrams 552,
 557–8, 650
 component managers 554
 component standards 579,
 584–6
 components 226–7, 650
 CBD (Component-Based
 Development) in
 UML 240–1
 componentware 577, 589–91
 composite design
 pattern 483

composite states in state machines 312–13
composite structural pattern 426–9
composite structure diagram 650
composition (composite aggregation) 650
finding and modelling 235–6
and generalization, combination of 236–7
Computer Misuse Act (UK, 1990) 42, 380
Computer Weekly 30
computerized support 48–9
conceptual aspect of software architecture 341
concrete classes 231–2, 650
concurrency
concurrent execution 269
concurrent states 313–15, 650
in DBMS 507
in system architecture 363–5
configuration management 553–4
conflicts of interest among users 32
consistency
in analysis 377, 378
in CASE 63
checking in state machines 325
in dialogues 446
constraints on systems design 379–82
construction
in CASE 64–5
constructor operation 650
guidelines in systems design 389
phase in the USDP 57, 609
in prototyping 55
in system development process 122, 125
in TLC 51
content, accuracy in determination of 377, 378
context
neglect of 36
of a pattern 650
resulting from use of patterns 419

continuations in interaction sequence diagrams 266–7
contracts 650
in operations, specification of 284–5
control, specification of call triggers 305
case study work, exercises and projects 327
change triggers 305
event types 305
events and states 304–6
further reading 327
learning objectives 303
relative time triggers 306
review questions 326–7
signal triggers 306
state-dependent variations in behaviour 303–4
state machines
activity-expressions 308–9
basic notation for 306–11
behavioural approach to preparation 319–22
behavioural state machines 324–5
choice pseudostates 315–17
pseudostates 315–17
composite states 312–13
concurrent states 313–15
consistency checking 325
decomposition compartments 312–13
deep history pseudostate 317, 318
deferred events 308
entry activities compartments 309
entry pseudostates 315
exit activities compartments 309
exit pseudostates 315
fork pseudostates 315, 317
further notation for 311–18
guard condition 307
history pseudostates 317–18

initial pseudostate 306–7
and interaction diagrams 304
internal activities compartments 308–9
join pseudostates 315
junction pseudostates 315–17
lifecycle approach to preparation 322–4
merge 316
name compartments 308
preparation of 318–24
protocol state machines 324–5
quality guidelines 325–6
reflexive transition 311
'run-to-completion' semantics 308
shallow history pseudostates 317–18
specialization of 318
state activities compartments 308–9
state variables 305
states and events 304–6
transitions (firing of) 305
triggers and events 305–6
control classes 176, 178–9, 473, 476, 481, 650
control flow 116, 651
control structures in algorithms 282, 290
conversion tools 555
cost benefit analysis 571
costs budget overruns 30–1 of failure 43–4
coupling concept of 382–5, 651 levels between different classes 579
CPA (Critical Path Analysis) 651
CPL (early C) 84
CRC (Class Responsibility Collaboration) 250, 252, 280, 649
cards 204–7, 208

in requirements analysis 172, 204–7
creational patterns 419, 420–5
CSE (Component System Engineering) 589
data 651
data conversion 565–7
data information and IS 16
data management classes, design of 523–33
data management design access to files 500–3 application servers 532–3
architecture for persistence 496–8
backup files 504
case study work, exercises and projects 534
concurrency in DBMS 507
data management classes, design of 523–33
data management product/framework, use of 531–3
data management tools 507
data modelling and normalization 511–16
data storage and persistence 495–6
database broker framework 526–31
databases and files 505–8
DBMS (Database Management Systems) 505–10
DDL (Data Definition Language) 507
direct access to files 502
distributed databases 521–3
DML (Data Manipulation Language) 507
fifth normal form 511
file storage, problems with 506
file systems 498–505
file use, example of 504–5
files and databases 505–8
first normal form (1NF) 512, 513
secondary indexing of files 502–3
review questions 533–4
second normal form (2NF) 513
serial access to files 500
serial organization (serialization) of foreign key 517

functional dependency 511
further reading 534–5
hashed addressing of files 502
header and detail records 499
improving access to files 502
index files 504
index-sequential access to files 500–2
inheritance hierarchy mapping 517
integrity constraints in DBMS 507
inverted files 503
layered architecture 523–5
learning objectives 494
mapping classes to tables 516–19
master files 503
normalization (normal forms) 511–16
object databases 509–10, 519–21
object-relational databases 510
object-relational mappings 531–2
organization of files 500
parameter files 504
persistence framework 524–5
persistent data 494, 495–8
PersistentObject superclass 525–6
primary key 512–13
random organization of files 500
record structures and file systems 499–500
relational databases 508–9, 511–19
review questions 533–4
second normal form (2NF) 513
secondary indexing of files 502–3
security in DBMS 507
sequential organization of files 500
serial access to files 500
serial organization (serialization) of foreign key 517

files 496, 500
storage mechanisms, overview of 496
system development process 122, 125
tables, mapping classes to 516–19
tagged data records 499
temporary files 504
third normal form (3NF) 512, 514
three-schema architecture in DBMS 506–8
transaction files 503
transaction management in DBMS 507
transient data 495–6
tuning of storage in DBMS 507
types of DBMS 508–10
types of files 503–4
variable length records 499
work files 504
data management issues 372, 387–8
Data Protection Acts (1984, 1998, UK) 42, 380
data types 397
DB2 509
DBA (Database Administrator) 503
DBMS (Database Management Systems) 365, 387–8, 494, 497–8
data management design 505–10
implementation 554
DCOM (Distributed Component Object Model) 498, 580
DDL (Data Definition Language) 507
debuggers 553
decision nodes 115–16, 651
decision tables 282, 287–8
declarative (non-algorithmic) approach to operation logic 282, 286–9
decomposition compartments 312–13
decoupled subsystems 227
deep history indicator 487–8
deep history pseudostate 317, 318, 651

deferred events 308
 definitions
 abstraction 69
 architecture 339–42
 class 71–4
 component 583
 diagrams 106–8, 109–10
 diagrams in UML 110
 generalization 74–9
 information 15–16
 information systems 16–19
 key terms in system
 architecture 340
 models 104–5, 109–10
 models in UML 110–11
 object state 82–3
 objects 69–71
 subsystems in UML 110
 systems 5–14
 systems in UML 110
 user interface 441
 user interface rules 485
 views in UML 110
 degree (multiplicity) 651
 delivery delays 31
 Delphi 92, 375, 466
 dependency 651
 and collaboration 173–6
 constraints 410–12
 dependencies between
 classes and packages,
 documentation of 237–9
 deployment diagrams 552,
 558–61, 651
 deployment specification 651
 deployment view of software
 architecture 341, 342,
 343
 design
 accuracy in algorithm
 design 412
 alternatives for security
 subsystem as reusable
 component 592–3
 and analysis, qualities and
 objectives of 377–86
 application of design
 patterns 436
 associations, design
 of 404–9
 boundary classes, design
 of 483–91
 and build iteration phase in
 DSDM 615, 616
 buildable design 381

constraint 651
 constraints on systems
 design 379–82
 critical nature of 440
 data management classes,
 design of 523–33
 data management
 design 505–10
 design patterns 416–18
 economical design 380
 efficiency in design 379
 ethnographic approaches
 to design of user
 interface 453–5
 extensibility in design
 patterns 420
 framework for systems
 design 371–2
 frameworks in design
 patterns 416
 general-purpose
 design 381
 informal approaches to
 design of user
 interface 450–7
 issues in identification of
 classes 191–2
 in the iterative
 lifecycle 373–4
 logical and physical
 design 374–6
 manageable design 381
 measurable objectives in
 design 386–7
 modifiability in
 design 380–1
 observer design
 pattern 483
 of operations 412–13
 of packages and boundary
 classes 463–4
 report design 480–1
 scenario-based approaches
 to design of user
 interface 455–7
 security in systems
 design 380
 strategy design pattern
 483
 structural approaches to
 design of user
 interface 451–3
 in TLC 51
 trade-offs, prioritization
 of 389–91

types of design
 patterns 419–20
 use of design
 patterns 435–6
 of user interface, main
 steps in 451
see also detailed design;
 systems design
 design patterns
 aliases of patterns 419
 application of 436
 behavioural patterns 419,
 429–35
 benefits of using 436–7
 capturing knowledge of
 successful
 solutions 415–16
 case study work, exercises
 and projects 438
 changeability in 419–20
 class (static)
 operations 421
 Composite structural
 pattern 426–9
 context resulting from use
 of patterns 419
 creational patterns 419,
 420–5
 dangers of using 436–7
 documenting patterns
 pattern templates 418–19
 extensibility in 420
 Factory Method
 pattern 423
 frameworks 416
 further reading 438–9
 implementation details and
 code 419
 learning objectives 415
 maintainability in 419
 pattern catalogues and
 languages 416–17
 pattern serving as guide to
 its application 418
 patterns, ‘rule of three’
 for 419
 patterns and non-
 functional
 requirements 417–18
 portability in 420
 rationale in justification of
 chosen solution 419
 related patterns 419
 restructuring for
 flexibility 420
 review questions 438
 Singleton creational
 pattern 420–5, 435
 software development
 patterns 416–18
 software development
 principles and
 patterns 417
 State behavioural
 pattern 430–5
 static operations 421
 structural patterns 419,
 425–9
 template aspects (other
 than contents) 418–19
 template contents 418
 types of 419–20
 use of 435–6
 validation of patterns 419
see also detailed design;
 system architecture;
 systems design
Design Patterns: Elements of Reusable Object-Oriented Software
 (Gamma, E. et al.) 229
 destructor operation 651
 detailed design
 accuracy requirements in
 algorithm design 412
 additional aspects in
 object-oriented detailed
 design 395–6
 algorithm design,
 constraints on 412
 algorithm design, factors
 for consideration
 in 412–13
 associations, design
 of 404–9
 attribute
 declarations 397–8
 case study work, exercises
 and projects 414
 class
 specification 397–402
 classes, keeping them to a
 minimum 409
 collection classes 408
 computational complexity
 in algorithm design 412
 concerns of 394–5
 data types 397
 dependency
 constraints 410–12
 domain integrity 412
 flexibility in algorithm
 design 412
 further reading 414
 implementation costs in
 algorithm design 412
 implementation facility in
 algorithm design 412
 integrity
 constraints 409–12
 interfaces 402–3
 learning objectives 394
 many-to-many
 associations 407–8
 object model, fine tuning in
 algorithm design 413
 object visibility 400–2
 one-to-many
 associations 405–7
 one-to-one
 associations 404–5
 operation signatures 397,
 398–400
 operations, design
 of 412–13
 performance constraints in
 algorithm design 412
 platform capabilities in
 algorithm design 412
 primary operations 398–9
 realize relationship 403
 referential
 integrity 409–10
 review questions 414
 and systems design 376
see also design patterns;
 system architecture;
 systems design
 developer's perspective of
 IS 33–5
 developing models 111–13
 development problems in
 information systems
 access to information,
 implications of 42–3
 aims and objectives, clarity
 and compatibility in
 purpose (wrong
 problem) 36, 37–8
 avoidance of
 advantages of
 prototyping 55
 agile approaches 61–2
 benefits of using
 CASE 65
 business modelling 49,
 50
 CASE (Computer Aided
 Software
 Engineering) 62–5
 case study work,
 exercises and
 projects 66
 change recommendation
 in prototyping 55
 code generators in
 CASE 64–5
 computerized
 support 48–9
 consistency/
 completeness in
 CASE 63
 construction in
 CASE 64–5
 construction in the
 USDP 57
 construction in TLC 51
 criticisms of TLC 52–3
 design in TLC 51
 development process
 subdivision 48–9
 difficulties of using
 CASE 65
 elaboration in the
 USDP 57
 encyclopaedias in
 CASE 63
 evaluation in
 prototyping 55
 feedback loops within
 TLC 53–4
 further reading 67
 inception in the
 USDP 57
 initial analysis in
 prototyping 55
 installation in TLC 51
 iterative and incremental
 development 56–7
 layering in CASE 63–4
 learning objectives 47
 lifecycle deliverables
 52
 lifecycle for
 prototyping 54–6
 lifecycle of the
 USDP 57–8
 maintenance in TLC 51
 development problems in
 information systems

development problems in information systems
avoidance of (*continued*)
maintenance tools in CASE 65
managing information systems development 58–9
methodological approaches 60
model and technique support in CASE 62–4
navigation to linked diagrams in CASE 63
object-orientation and 68, 86
objectives definition in prototyping 55
performance analysis in CASE 64
problem solving, phases of 48
problems identified, major categories of 47
problems in prototyping 56
project lifecycles 49–54
prototyping 54–6
report generation in CASE 64
repositories in CASE 63
requirements analysis in TLC 51
review questions 66
specification in prototyping 55
spiral model for incremental delivery 56–7
strategic information systems planning 49–50
syntactic correctness in CASE 63
system engineering in TLC 50–1
system simulation in CASE 64
testing in TLC 51
traceability in CASE 64
traditional lifecycle (TLC) 50–4, 61
transition in the USDP 57

USDP (Unified Software Development Process) 57–8, 343, 373
user involvement in systems development 59–60
waterfall lifecycle model 50–4
blaming users for problems 35
budget constraints 33–4
case study work, exercises and projects 45–6
change, developers' perspective on 33
change, user requirements and timescales 32–3
client's perspective 30–3
commercial pressures 31
conflicts of interest among users 32
context, neglect of 36
costs, budget overruns 30–1
costs of failure 43–4
delivery delays 31
developer's perspective 33–5
end-user's perspective 28–30
ethical dimension 40–3
ethical issues within a project 40–2
external events 33, 38
failure, complete 36
failure, costs of 43–4
further reading 46
implementation, infeasibility in 39–40
learning objectives 26
legislation, ethical dimension of 42
organizational politics 32, 34–5
personal information, ethical dimension and 42
process of development 26–7, 48–9
productivity problems 35, 38–40, 47
project failure 27, 35–40
project management, shortcomings in 39, 49
purpose, reason and understanding in 37–8
quality problems 35, 36–8, 47
reliability 29–30
requirements analysis, mistakes in 37
requirements drift 38
review questions 45
skills, compatibility in developers' 34
software, working knowledge of 34
software errors, hazardous to life 30
SSADM (Structured Systems Analysis and Design Method) 34
stakeholders, and effects on 40–1
time constraints 33–4
trust of users 32
unemployment effect 42
usability 28–9
USDP (Unified Software Development Process) 34
utility 29–30
vapourware 28
what are the problems? 27–35
why problems arise 35–40
development process 26–7, 48–9
subdivision of 48–9
development standards 388–9
device 651
diagrams activity diagrams 113–19, 169–70, 648
operations, specification of 294–5
analysis class diagram 171–2
assembly of 207–8, 218–20
basic concepts and notation in interaction sequence diagrams 253–62
business modelling with use case diagrams 153–4
class diagrams 179–203, 538–44, 649

adding boundary classes 481
communication diagrams and 194–6
derived from communication diagrams 191, 194–6
further refinement of 335–6
in requirements analysis 179–203
showing collection classes 597
showing control classes 596
collaboration diagrams 649
combined fragments in interaction sequence diagrams 254–5
communication diagrams 146, 174–6, 176, 190–1, 192, 193, 194, 271–5, 650
basic concepts and notation 271–3
class diagrams and 194–6
classes, identification of 190–1, 192, 193, 194
diagram frame 193
and interaction sequence diagrams 271–5
object interaction in 271–5
basic concepts and notation 271–3
message labels in 273–5
message labels in communication diagrams 273–5
sequence diagrams and 275
purpose of 176
requirements analysis 174–6
requirements capture 146
sequence diagrams and 275
component diagrams 552, 557–8, 650
composite structure diagram 650
continuations in interaction sequence diagrams 266–7
definitions 106–8, 109–10
deployment diagrams 552, 558–61, 651
designing interaction with sequence diagrams 473–81
diagrams defined 106–8
gates in interaction sequence diagrams 264–6
implementation diagram 652
instance diagram 653
interaction, designing with sequence diagrams 473–81
interaction diagrams 304, 653
interaction overview diagrams 275–8, 653
interaction sequence diagrams 146, 252–71, 329–30, 544–8, 656
active objects 269
asynchronous messages 267, 268
basic concepts and notation 253–62
branching 262–3
callback 267
combined fragment 254–5
communication diagrams and 271–5
concurrent execution 269
continuations 266–7
execution occurrence 255, 256
focus of control 259
frame 254–5
gates 264–6
guidelines for preparation of 269–71
interaction fragments 264–6
interaction occurrences 263
interaction operands 263
interaction operators 255, 270
lifeline 253–4, 255–8, 260–2
management of 263–7
message events 255, 256
object destruction 256, 258
object interaction 252–71
passive objects 269
procedural calls 262
real-time systems, modelling of 269
reflexive messages 257
reply (return of control) 259–60
return-values 260
send message event 255
signal (asynchronous message) 267
synchronous messages 262
threads of control 269
time constraints 267–8
management of interaction sequence diagrams 263–7
models and diagrams 104–13
difference between 109–10
navigation to linked diagrams in CASE 63
notation for component structure diagram in UML 239–40
notation of activity diagrams 114–19
notation summaries of activity diagram 629
behaviour diagrams 628–9
communication diagram 632
component diagram 634
deployment diagram 634
implementation diagrams 634
interaction diagrams 630–3
interaction overview diagram 633
sequence diagram 630–1
state machine diagram 628

diagrams
 notation summaries of
 (continued)
 static structure
 diagrams 626–8
 timing diagram 632
 use case diagram 626
object diagram (instance diagram) 655
 purpose of activity
 diagrams 113–14
state machines and
 interaction diagrams 304
stereotyped diagrams 119
time constraints in
 sequence
 diagrams 267–8
timing diagrams 278–9,
 657
 in object
 interaction 278–9
UML (Unified Modelling Language)
 communication diagrams
 in 271–5
 definition of diagrams
 in 110
 diagrams, elements
 of 108
 interaction overview
 diagrams in 275–8
 interaction sequence
 diagrams in 252–71
 timing diagrams
 in 278–9
 use cases and use case
 diagrams 144, 145–54,
 171–2, 658
dialogue metaphor for user interface 441–4
Digital Equipment Corporation 455
direct access to files 502
direct changeover 568–9
direct manipulation
 metaphor for user interface 445–6
Disability Discrimination Act (1995), UK 460
distributed databases 521–3
distributed systems,
 architectures for 360–2
DML (Data Manipulation Language) 507
document sampling 136–8

documentation
 generators 555
 documentation standards 555–7
 documenting patterns
 pattern templates 418–19
 documenting requirements 143–5
 domain integrity 412
 domain model 651
 domain modelling 196
 Dot.com deathwatch 37
 DSDM (Dynamic Systems Development Method) 59, 61, 134, 229, 606, 612, 613–18, 623, 624
 business study phase 615, 616
 design and build iteration phase 615, 616
 feasibility study phase 615, 616
 functional model iteration phase 615, 616
 implementation phase 615, 616–17
 lifecycle 615–17
 principles
 underlying 614–5
 project selection in 617–18
 timeboxing 617
 duplication of classes 198
 dynamic interaction among classes 189
EAI (Enterprise Application Integration) tools 345
economical design 380
efficiency in design 379
e-GIF (e-Government Interoperability Framework), UK 347
Eiffel 84, 87
EJBs (Enterprise Java Beans) 495, 498, 532, 553, 590
elaboration in the USDP 57, 608–9
encapsulation 80–1, 226, 651
encyclopaedias in CASE 63
end-user's perspective of IS 28–30
enhancement requests 573

enterprise architectures 344, 346–7
enterprise (or business) rules 184–6, 651
entire system representation 198
entity classes 176, 177–8, 651
entry activities compartments 309
entry pseudostates 315, 651
EPOC 376
Ericsson 580, 586
essential use cases 147
ethical issues ethical dimension in IS 40–3
 legislation, ethics and 42
 personal information, ethics and 42
 within a project 40–2
ethnographic approaches to design of user interface 453–5
EU Council (29 May 1990) directive on HCI 459
EU (European Union) 347, 380
EU Telematics Applications Programme 452
evaluation in prototyping 55
evaluation report 570–2
EventObject subclasses 482
events 651
 event-action table, preparation of 489–91
 event-driven programming 84–5
 event types 305
 external 33, 38
 and states 304–6
examples
 boardroom meeting, problems (and possible solution, with IS implications) 97
 class and object, object-orientation 74
 design, hidden costs of deficiencies in 43
 fighter command's 1940 information system 4
 of file use 504–5
 identification of correct class for an attribute 200–1
information systems team meeting 102
interviewing, guidelines on 135
methodology and study projects 620
online retail system 3
operations as responsibilities 188
poor system usability 29
questionnaires, guidelines on 140–1
railway signalling system 5
security subsystem as reusable component, case study example 591–9
study projects and methodology 620
UML metamodel 108–9
 see also guidelines
exception mechanism 651
execution aspect of software architecture 341
 environment 652
 occurrence 255, 256
exit activities compartments 309
exit pseudostates 315, 652
experience, learning from 570
expert systems 139–40
extend relationship in use cases 148–9, 652
extensibility in design patterns 420
extention point 652
external events 33, 38
eXtreme Programming *see* XP
Façade pattern 594–5, 600
fact-finding techniques 132–41
Factory Method pattern 423
failure complete 36
 costs of 43–4
FEAF (Federal Enterprise Architecture Framework) 347
feasibility study phase in DSDM 615, 616
feedback loops within TLC 53–4
 to users 573–4
fifth normal form 511
fighter command's 1940 system 4
file storage, problems with 506
file systems 498–505
file use, example of 504–5
files access to 500–3
 backup files 504
 and databases 505–8
 direct access to 502
DLL (Dynamic Link Library) files in Windows 223
hashed addressing of 502
improving access to 502
index files 504
index-sequential access to 500–2
inverted files 503
master files 503
organization of 500
parameter files 504
random organization of 500
record structures and file systems 499–500
secondary indexing of 502–3
sequential organization of 500
serial access to 500
serial organization of 500
temporary files 504
transaction files 503
types of 503–4
work files 504
final node 116
final pseudostate 652
finding and modelling 230–5
first normal form (1NF) 512, 513
fixed length records 499
flexibility in algorithm design 412
 in systems design 380–1
 of the user interface 458
flow final node 652
forces (of a pattern) 652
foreign key in data management design 517
fork 652
Gamma patterns 419

Gantt charts 589
 gates in interaction sequence diagrams 264–6
Gemstone/S 510
 general-purpose design 381
 generality in class 75, 76
 generalization 74–9, 225, 652
 bottom-up approach to finding 234
 and composition (composite aggregation), combination of 236–7
 finding and modelling 230–5
 hierarchies of, disjoint nature of 78, 79
 overuse of 234–5
 practical uses of 77–8
 top-down approach to finding 233–4
 in use cases 150–1
 usefulness of 232–3
 when not to use 234–5
GIS (Geographic Information System) 88
 goals, IS and achievement of 20–1
GOF (Gang of Four) patterns 419
 guard condition 307, 652
Guardian 37
GUI (Graphical User Interface) 445, 484, 495, 497, 590
 spread of 85
 guidelines analysis, criteria for good analysis 377–9
 behavioural approach to state machine preparation, steps to 320
 construction in systems design 389
CRC (Class-Responsibility-Collaboration) cards, process of using 205
 design, criteria for good design 379
 evaluation reporting 575
HCI and systems design 388
 input/output device guidelines 388–9

interaction sequence diagrams, preparation of 269–71
 on interviewing 135
 layered architecture development 353
 lifecycle approach to state modelling, steps to 323–4
 mapping classes to tables 516–17
 mapping inheritance 517
 modelling techniques, general rules for 107
 multiple processors on different platforms, allocation of 365–6
OCL, structural elements of 296
 operation specification, contracts and 285
 operation specification, tests for 289
 pattern application, procedure for 436
 patterns, critical aspects of 228
 patterns, key principles underlying 417
 for preparation of interaction sequence diagrams 269–71
 quality guidelines for state machines 325–6
 on questionnaires 140–1
 requirements models, objectives of 172–3
 scenarios, roles for 456
 state machine quality 325–6
 state machines, consistency checking 325
 system already in use, case for investigating 130
 system architecture, key features of 339–40
 systems design guidelines for HCI 388
 use cases, documentation of 148
see also examples
 hard vs soft methodologies 621–4

hashed addressing of files 502
HCI (Human-Computer Interaction) 349, 376, 381
 Americans With Disabilities Act (1990) 460
 case study work, exercises and projects 461–2
 Code of Practice, Disability Rights Commission, UK 460
 Disability Discrimination Act (1995), UK 460
 EU Council (29 May 1990) directive 459
 further reading 462
Health & Safety (Display Screen Equipment) Regulations 459–60
ISO 14915–2002 on interactive user interfaces 459
ISO 9241 on ergonomic requirements 459
 learning objectives 440
OSHA (Occupational Safety and Health Administration), USA 460
 review questions 461
 standards and legal requirements 458–60
 systems design guidelines 388
 user interface attitude of users of the 458
 characteristics of good dialogues 446–9
 consistency in dialogues 446
 design, critical nature of 440
 design, main steps in 451
 dialogue metaphor for 441–4
 direct manipulation metaphor for 445–6
 ethnographic approaches to design of 453–5
 flexibility of the 458
 formal approaches to design of 450–7
 informal approaches to design of 450–7
 learnability in the 458
 metaphors representing 441–6
 scenario-based approaches to design of 455–7
 structured approaches to design of 451–3
 style guides 449
 system feedback, adequacy of 447–8
 throughput of the 458
 usability, achievement of 457–8
 usability, ISO definition of 458
 user input, minimization of 448–9
 user support, appropriateness in 447
 what is it? 441
 header and detail records 499
Health and Safety (Display Screen) Regulations (UK, 1992) 42, 459–60
 heritage systems, influence on system architecture 345
 Hewlett-Packard 580, 586
 Hibernate framework 532
 hierachic classification 75
 hierarchies of generalization, disjoint nature of 78, 79
 high-level requirements, description of 484–5
 history pseudostates 317–18
 horizontal prototyping 466
HTML (Hypertext Markup Language) 37, 152, 366, 499, 554, 555, 556
 human affairs and IS 1–3, 14–15
I-CASE (Integrated-CASE)
see CASE
IBM 578, 586, 600
IBM-Rational Unified Process 61, 119, 341, 343, 373, 552, 603, 604, 605
ICONIX method 196
 identification of classes 189–94, 196–200
 identification of objects in collaboration 189–90
IDEs (Integrated Development Environments) 553–4
IDL (Internet Definition Language) 522, 554
IDL2JAVA tool 399, 522
IEEE (Institute of Electrical and Electronics Engineers) 340–1, 368
IIOP (Internet IOP) 522
IKIWISI (I'll Know It When I See It) 373
Immigration and Nationality Directorate, UK 27
 implementation actions list 572
 application containers 554
 black box testing 562
 bug reporting database 573
 candidate components for reuse 572
 maintenance 574
 parallel running 569
 phased changeover 569
 pilot projects 569–70
 plans for 574
 positive experiences 571
 problems and issues 571
 quantitative data for future planning 571
 regression testing 564
 review and maintenance 570–4
 review process 570–2
 review questions 575–6
 run-time support 553
 software testing 561–5
 software tools 552–7
 strategies for 568–70
 system development process 122, 126
 technical and user documentation 574
 test documentation 564–5
 testing, aim of 563–4
 testing, of classes 563–4
 testing, what is tested? 562–4
 testing, who carries it out? 561–2

implementation (*continued*)
 testing tools 555
 user documentation 574
 user documentation (manuals) 567
 user satisfaction 571
 user training 567–8
 visual editors 553
 white box testing 563
 implementation diagram 652
 implementation facility in algorithm design 412
 implementation phase in DSDM 615, 616–17
 implementation view of software architecture 341, 342, 343
In The Age of The Smart Machine (Zuboff, S.) 1
 inception in the USDP 57, 608
 include relationship in use cases 148–9, 652
 incremental development 56–7, 652
 index files 504
 index-sequential access to files 500–2
 influences on system architecture 344–8
 informal approaches to design of user interface 450–7
 information 653
 implications of access to 42–3
 and information systems 15–19
 information flow 16–17
 information systems
 academic study of 2
 B2B (business-to-business) 23
 B2C (business-to-customer) 23
 business strategies, identification of 19–20
 case study work, exercises and projects 24–5
 classification of 16
 contribution to achievement of goals 20–1
 data, information and 16

fighter command's 1940 system 4
 further reading 25
 human affairs and 1–3, 14–15
 information and 15–19
 information technology (IT) 18–19, 21–3
 and IT strategies 21–3
 knowledge, information and 16
 management support systems 17
 MIS (Management Information System) 17
 office systems 18
 online retail system 3
 operational systems 16–17
 practical applications 4–5
 railway signalling 5
 real-time control systems 18
 review questions 24
 success with, strategies for 19–23
 SWOT (Strengths, Weaknesses, Opportunities, Threats) approach 20
 technological progress in IT 19
 transactions and information flow 16–17
 Value Chain Model 20–2
 what are they? 2–24
see also development problems
 information technology (IT) 1–2
 Informix 509
 Ingres 509
 inheritance 78–9, 227, 653
 multiple inheritance 79, 235
 object-orientation and 78–9
 in requirements model, refinement of 227
 transitive operation of 78–9
 inheritance coupling 383–4
 inheritance hierarchy of boundary classes 464, 481
 mapping of 517
 initial analysis in prototyping 55
 initial node 116, 653
 initial pseudostate 306–7, 653
 input/output device guidelines 388–9
 installation in TLC 51
 installation tools 555
 instance diagram 653
 instance (single object) 71, 653
 instance value (of an attribute) 653
 instances and classes, relative stability of 179–80
 integrity constraints 409–12, 653
 in DBMS 507
 interaction 653
 designing with sequence diagrams 473–81
 interaction constraint 653
 interaction coupling 383
 interaction diagrams 304, 653
 interaction fragments 264–6
 interaction occurrences 263, 653
 interaction operands 263, 653
 interaction operators 255, 270, 653
 interaction overview diagrams 275–8, 653
 interaction sequence diagrams 146, 656
 active objects 269
 asynchronous messages 267, 268
 basic concepts and notation 253–62
 branching 262–3
 callback 267
 combined fragment 254–5
 communication diagrams and 271–5
 concurrent execution 269
 continuations 266–7
 execution occurrence 255, 256
 focus of control 259
 frame 254–5
 gates 264–6

guidelines for preparation of 269–71
 interaction fragments 264–6
 interaction occurrences 263
 interaction operands 263
 interaction operators 255, 270
 lifeline 253–4, 255–8, 260–2
 management of 263–7
 message events 255, 256
 object destruction 256, 258
 object interaction 252–71
 passive objects 269
 procedural calls 262
 real-time systems, modelling of 269
 reflexive messages 257
 reply (return of control) 259–60
 return-values 260
 send message event 255
 signal (asynchronous message) 267
 synchronous messages 262
 threads of control 269
 time constraints 267–8
 interface 402–3, 593–4, 599, 653
 of an object 80
 modelling using state machines
 deep history indicator 487–8
 event-action table, preparation of 489–91
 high-level requirements, description of 484–5
 interface behaviour, description of 485
 interface rules, definition of 485
 main user tasks, description of 484–5
 state machine, drawing up 485–9
 tasks involved 484
see also user interface
 interface class 654
 interface design patterns, use of 482–3
 internal activities compartments 308–9
 interpreters 553
 interviewing 133–4, 135
 invariants 654
 inverted files 503
 IOP (Inter-ORB Protocol) 522
 ISO 14915–2002 on interactive user interfaces 459
 ISO 9241 on ergonomic requirements 459
 ISO (International Standards Organization) 131, 458–9
 seven layer model 351, 352
 IT (Information Technology) 18–19, 21–3
 strategies and information systems 21–3
 technological progress in 19
 iteration in Structured English 292–3
 iterative development 56–7, 112–13
 Jasmine ODBMS 510
 Java 34, 84, 87, 92, 367, 375, 381, 397, 403, 419, 421, 466, 472, 475, 482–3, 497, 508, 519, 521–4, 527, 553, 556, 579, 584–6, 590
 Java applets 481
 Java AWT (Abstract Windowing Toolkit) 375, 382, 470, 471, 472, 479, 481, 590, 593
 Java Swing classes 375
 JCP (Java Community Process) 519
 JDBC (Java Database Connectivity) 375, 495, 497, 519, 527, 528, 554
 JDO (Java Data Objects) 519, 531–2, 554
 JDOQL (JDO Query Language) 532
 J2EE (Java 2 Enterprise Edition) 344, 346, 355, 531, 532, 559, 590
 join node 654
 join pseudostates 315
 JRMP (Java Remote Method Protocol) 522
 JSP (Java Server Pages) 498
 JSR (Java Specific Request) 519
 junction pseudostates 315–17, 316, 654
 knowledge 654
 acquisition of 140
 information and 16
 KPMG Management Consulting 452
 languages
 ADL (Architecture Description Language) 342
 assembly language 84
 characteristics of object-oriented languages 87–8
 currently available 87–8
 OCL (Object Constraint language) 108–9
 pattern catalogues and languages 416–17
 layered architecture 523–5
 layered subsystems 350–5
 layering in CASE 63–4
 learnability in the user interface 458
 learning objectives
 boundary classes, design of 463
 control, specification of 303
 data management design 494
 design patterns 415
 detailed design 394
 development problems, avoidance of 47
 development problems, information systems 26
 HCI (Human-Computer Interaction) 440
 implementation 551
 information systems, what are they? 1
 modelling concepts 103
 object interaction 249
 object-orientation 68

learning objectives
(continued)
operations, specification of 282
requirements analysis 171
requirements capture 128
requirements model, refinement of 222
reusable components 577
software development processes 602
system architecture 338
systems design 371
legacy systems 129, 654
legislation, ethics and 42
lifecycles
DSDM lifecycle 615–17
lifecycle approach to preparation of state machines 322–4
lifecycle deliverables 52
lifecycle (of a project) 654
problems in object-orientation 86
project lifecycles 49–54
for prototyping 54–6
USDP lifecycle 57–8
waterfall lifecycle model 50–4
see also TLC
lifelines 253–4, 255–8, 260–2, 654
limitations of object-orientation 88
link 654
and association 182–3
Linux 376, 509, 555, 585
logical and physical design 374–6
logical cohesion 383
logical concurrency 363–5
logical view of software architecture 341, 342, 343
Logidex
(LogicLibrary) 344–5, 578, 586
London Ambulance Service 30, 43, 44
London Stock Exchange 27, 31, 32, 43, 44
Lotus Notes 18
LSP (Liskov Substitution Principle) 385–6
machine code 84
Macintosh Human Interface Guidelines (Apple, 1996) 449
MacOS 376
maintainability in design patterns 419
in systems design 381
maintenance activities in implementation 572–4
outsourcing maintenance 574
review and maintenance 570–4
in TLC 51
tools in CASE 65
manageable design 381
management of information systems development 58–9
of interaction sequence diagrams 263–7
management support systems 17
of reuse, difficulties in 224
many-to-many associations 407–8
mapping classes to tables 516–19
master files 503
maturity in the USDP 610
MDA (Model Driven Architecture) 368, 605, 654
influence on system architecture 345–6
measurable objectives in design 386–7
merge 316
merge node 654
message events 255, 256
message passing 79–81, 654
messages 654
asynchronous messages 267, 268, 649
message labels in communication diagrams 273–5
message labels in object interaction 273–5
reflexive messages 257
send message event 255
signal (asynchronous message) 267
signatures (message protocols) 80
synchronous messages 262, 657
methodological approaches 60
methodology 654
software development processes 604–6
aspects of development process 604–5
hard vs soft methodologies 621–4
issues in choosing a 619–21
rationale for use of a 606
variety in approaches to 605–6
methods 654
implementing operations 187–8
Microsoft 580
Microsoft COM architecture 589
Microsoft Outlook 18
Microsoft PowerPoint® 152
Microsoft Windows 92, 349, 376
DLL (Dynamic Link Library) files in 223
Microsoft.NET 344, 346, 586
middleware 375
MIS (Management Information System) 17
model and technique support in CASE 62–4
model consistency in object interaction 279–80
model transitions 86
modelling components in UML 241–4
modelling concepts 119–26
activity diagrams 113–19
activity edge 116
activity partitions 117, 118
case study work, exercises and projects 127
control flow to final node 116
decision nodes 115–16
developing models 111–13
diagrams defined 106–8
final node 116
further reading 127
initial node 116

iterative development 112–13
learning objectives 103
models, UML notation for 111
models and diagrams 104–13
difference between 109–10
UML notation for 111
modifiability in design 380–1
modular construction 654
modular software 85–6
module aspect of software architecture 341
MOF (Meta Object Facility) 605
MONO project 585
MoSCoW (Must Should Could Won't) rules for prioritization 145, 617
MSIL (Microsoft Intermediate Language) 553, 585
multiple inheritance 79, 235
multiple outcomes in Structured English 292
multiplicity 184–6, 655 determination of 203
Mushroom Management 229
MVC (Model–View–Controller) 464, 481, 482, 483, 492, 590
structure 417
system architecture 356–60
MySQL 509
name compartments 308
naming standards 555–6
National Air Traffic Control Service, UK 27
National Physical Laboratory, UK 453
National Probation Service, UK 27
navigation to linked diagrams in CASE 63
NIH (Not Invented Here) syndrome 224
node 655
non-algorithmic (declarative) approach to operation logic 282, 286–9
non-functional requirements 655
implementation 571
modelling requirements capture 154–5
modelling tools 552–3
models defined 104–5
and diagrams 104–13
difference between 109–10
UML notation for 111
normalization 494, 655
data modelling and 511–16
technique of 202
notation of activity diagrams 114–19
for component structure diagram in UML 239–40
summaries of activity diagram 629
associations 628
behaviour diagrams 628–9
class notation 627
communication diagram 632
component diagram 634
deployment diagram 634
implementation diagrams 634
interaction diagrams 630–3
interaction overview diagram 633
object instance notation 626
sequence diagram 630–1
state machine diagram 628
static structure diagrams 626–8
timing diagram 632
use case diagram 626
of use cases 146–51
O₂ 519
object 655
object databases 509–10, 519–21
object design *see* detailed design
object destruction 256, 258
object diagram (instance diagram) 655
object flows 117, 118, 655
object interaction case study work, exercises and projects 281
and collaboration 250–2
communication diagrams 271–5
object interaction

communication diagrams
(continued)
basic concepts and
notation 271–3
message labels in 273–5
message labels in
communication
diagrams 273–5
sequence diagrams
and 275
fundamental to object-
orientation 249–50
further reading 281
interaction overview
diagrams 275–8
interaction sequence
diagrams 252–71
active objects 269
asynchronous
messages 267, 268
basic concepts and
notation 253–62
branching 262–3
callback 267
combined
fragment 254–5
communication diagrams
and 271–5
concurrent
execution 269
continuations 266–7
execution
occurrence 255, 256
focus of control 259
frame 254–5
gates 264–6
guidelines for
preparation of 269–71
interaction
fragments 264–6
interaction
occurrences 263
interaction operands 263
interaction
operators 255, 270
lifeline 253–4, 255–8,
260–2
management of 263–7
message events 255,
256
object destruction 256,
258
passive objects 269
procedural calls 262
real-time systems,

modelling of 269
reflexive messages 257
reply (return of
control) 259–60
return-values 260
send message event 255
signal (asynchronous
message) 267
synchronous
messages 262
threads of control 269
time constraints 267–8
learning objectives 249
model consistency 279–80
review questions 280–1
timing diagrams 278–9
object model, fine tuning in
algorithm design 413
object-orientation
abstraction, increase in 84
basic concepts 68, 69–83
case study work, exercises
and projects 89–90
characteristics of object-
oriented languages 87–8
class and object 71–2, 74
class membership 72–3
contribution to
reuse 225–30
detailed design, additional
aspects of 395–6
development problems,
avoidance and 68, 86
encapsulation 80–1
event-driven
programming 84–5
further reading 90
generality in class 75, 76
generalization 74–9
GUIs (Graphical User
Interfaces), spread of 85
hierarchic classification 75
hierarchies of
generalization, disjoint
nature of 78, 79
inheritance 78–9
instance (single object) 71
interface of an object 80
languages available
today 87–8
learning objectives 68
lifecycle problems 86
limitations of 88
message passing 79–81
model transitions 86
modular software 85–6
multiple inheritance 79
object interaction
fundamental to 249–50
object-oriented analysis
and design 2
object state 82–3
objects 69–71
operations, invoking
(calling upon) 80
origins of 83–7
polymorphism 81–2
practical uses of
generalization 77–8
requirements reuse,
difficulties in 224–5
reusable software
components 86–7
review questions 89
semantics of a model 77
signatures (message
protocols) 80
specialization 76–7
specificity in class 75–6
taxonomic relationship 75
transitive operation of
inheritance 78–9
*Object-Oriented Software
Engineering* (Jacobsen, I.
et al.) 586, 588
object-relational
databases 510
object-relational
mappings 531–2
object state 82–3
and associations 183–4
and attributes 181–2
and operations 188
object visibility 400–2
Objective-C 84
objectives
and constraints in systems
design 379–82, 386–7,
390
definition in
prototyping 55
objects 69–71
categories of 197–8
ObjectStore 510, 519, 520,
521, 596, 597
observation 134–6
observer design pattern 483
OCL (Object Constraint
Language) 108–9, 113,
126, 407
and operations,
specification of 283,
295–9, 301
ODBC (Open Database
Connectivity) 375, 380,
495, 497, 528, 554
ODBMS (Object Database
Management
Systems) 509–10, 533
ODL (Object Definition
Language) 519, 521
ODMG (Object Data
Management
Group) 519, 522
office systems 18
OLE (Object Linking and
Embedding) 589
OMG (Object Management
Group) 579, 585, 605
OML (Object Manipulation
Language) 519, 520
online retail system 3
one-to-many
associations 405–7
one-to-one
associations 404–5
Open Group 348
Open Source software 554
operational systems 16–17
operations 186–8, 655
allocation of 203
design of 412–13
finding 203
invoking (calling upon) 80
operation cohesion 384
operation logic, description
of 286–95
operation signature 397,
398–400, 655
operation specifications
complexity of 282, 301
creation of 299–300
role of 283–4
reality in 199
redefined in refinement of
requirements
model 230–1
specification of 294–5
algorithmic (procedural)
approach to operation
logic 282, 289–95
structures for architecture
and development 362–3
organizational politics 32,
34–5
origins of object-
orientation 83–7
contracts in 284–5

control structures in
algorithms 282, 290
decision tables 282,
287–8
declarative (non-
algorithmic) approach
to operation logic 282,
286–9
further reading 302
learning objectives 282
non-algorithmic
(declarative) approach
to operation logic 282,
286–9
OCL (Object Constraint
Language) and 283,
295–9, 301
operation logic,
description of 286–95
operation specifications
complexity of 282, 301
creation of 299–300
role of 283–4
pre- and post-
conditions 288–9
procedural (algorithmic)
approach to operation
logic 282, 289–95
pseudo-code 294
review questions 301
Structured English 282,
291–4
complex structures
in 293–4
iteration in 292–3
main principles of 291
multiple outcomes
in 292
Oracle 39, 509, 528
ORBs (Object Request
Brokers) 522, 554, 585
organization
of the analysis model,
packages and
dependencies 237–9
of files 500
organizational structure for
reusability 580–2
structures for architecture
and development 362–3
organizational politics 32,
34–5
origins of object-
orientation 83–7
OSHA (Occupational Safety
and Health
Administration),
USA 460
OSI (Open Systems
Interconnection), seven
layer model 351
outside-in development of
applications 467
outsourcing
maintenance 574
overuse of
generalization 234–5
packages 655
and boundary classes,
design of 463–4
partitioning into 237–9
UML notation for 111
PalmOS 376
parallel running 569
parameter files 504
partitioned subsystems 350,
355–6
passive objects 269
patterns 655
distinguished from
strategies 244
movement of 223
and non-functional
requirements 417–18
pattern catalogues and
languages 416–17
pattern templates,
documenting
patterns 418–19
requirements model,
refinement of 227–30
'rule of three' for 419
serving as guides to
application 418
see also design patterns
PD (Participatory
Design) 611–13
PE (Portable Executable)
format, Microsoft 585
peer-to-peer
communication 350
performance analysis in
CASE 64
performance constraints
in algorithm design
412
persistence
framework 524–5
persistence mechanism 596

persistent data 494, 495–8
PersistentObject
 superclass 525–6
 personal information, ethics and 42
 phased changeover 569
 phases and workflows in the USDP 607–9
 philosophy and principles of the USDP 607
 physical and logical design 374–6
 physical concurrency 363–5
 physical inputs/outputs, classes and 198
 pilot projects 569–70
PIM (Platform-Independent Model) 345–6, 655
PL/X 578
 planning for reuse lateness of 579 strategic considerations 586–9
see also reusable components
 plans for implementation 574
 platform capabilities in algorithm design 412
 platform dependence of components 579
 polymorphism 81–2, 655
 portability in design patterns 420 in systems design 381 positive experiences in implementation 571 post-condition 655 PostgreSQL 510 practical applications of IS 4–5 practical uses of generalization 77–8 pre- and post-conditions 288–9 pre-condition 655 Premerlani 519 preparation of state machines 318–24 presentation layer architecture 464–6 primary key 512–13 primary operations 398–9, 656

problems and issues in implementation 571 major categories of 47 problem solving, phases of 48 in prototyping 56 *see also* development problems procedural (algorithmic) approach to operation logic 282, 289–95 procedural calls 262, 656 process, method and methodology 603–4 process of development 26–7, 48–9 process view of software architecture 341, 342, 343 processing node 656 processor allocation 365–6 productivity problems 35, 38–40, 47 Progress 509 project choice for reusability 579–80 project failure 27, 35–40 project lifecycles 49–54 project management, shortcomings in 39, 49 project selection in DSDM 617–18 Prolog 84 property 656 protocol state machine *see* state machine prototype 656 prototyping 54–6 advantages of 55 change recommendation in 55 construction in 55 in design of boundary classes 466–9 evaluation in 55 horizontal prototyping 466 initial analysis in 55 lifecycle for 54–6 objectives definition in 55 problems in 56 specification in 55 supporting use cases with 151–3 throwaway prototyping 466 vertical prototyping 466 proxy components, introduction of 361–2 **PSE Pro (ObjectStore Personal Storage Edition)** 521, 554, 596 pseudo-code 294 pseudostates choice pseudostate 315–17, 649 deep history pseudostate 317, 318, 651 entry pseudostate 315, 651 exit pseudostate 315, 652 final pseudostate 652 fork pseudostate 315, 317 history pseudostate 317–18 initial pseudostate 306–7, 653 join pseudostate 315 junction pseudostate 315–17, 316, 654 shallow history pseudostate 317–18, 656 **PSM (Platform-Specific Model)** 346, 655 purpose of activity diagrams 113–14 clarity and compatibility in 36, 37–8 of communication diagrams 176 reason and understanding in 37–8 of use cases 146 Python 84 quality guidelines for state machines 325–6 quality improvement, testing deliverables 373, 374 quality problems 35, 36–8, 47 quantitative data for future planning 571 query operation 656

questionnaires 138, 139, 140–1 **QWAN (Quality Without a Name)** patterns 229 **RAD (Rapid Application Development)** 613, 614 **RAID (Redundant Arrays of Inexpensive Disks)** 380 railway signalling 5 random organization of files 500 **RAS (Reusable Asset Specification)** 344, 579, 584, 585, 586, 600 rationales in justification of chosen solution 419 for use of a methodology 606 **RDBMS (Relational Database Management System)** 88, 494, 508, 509 real-time control systems 18 real-time systems, modelling of 269 real use cases 147 realize relationship 403, 656 record structures and file systems 499–500 refactoring 656 referential integrity 409–10 reflexive messages 257 reflexive transition 311 regression testing 564 related patterns 419 relation 656 relational databases 508–9, 511–19 relative time triggers 306 reliability 29–30 in systems design 380 reply (return of control) 259–60, 656 report design 480–1 report generation in CASE 64 repositories 656 in CASE 63 requirements analysis analysis class **use case description, extended diagram** 171–2 assembly of 207–8 analysis class stereotypes 176–9 association instances (links) 182–3 associations 182–4 associations, finding and adding 202–3 associations, reality in 199–200 attributes 180–2 attributes, adding and locating 200–2 attributes, reality in 198–9 boundary classes 176, 177 business rules 184–6 cardinalities allowed (multiplicity) 184–6 case study work, exercises and projects 209–12 categories of classes 197–8 class as descriptor for collection of objects 191 class diagrams 179–203 communication diagrams and 194–6 classes, identification of 189–94, 196–200 application domain, concepts in 190 class diagrams derived from communication diagrams 191, 194–6 class names 191 collaboration, further refinement in 193 collaboration, initial 190–1 collaboration, refinement in 191–3 communication diagram frame 193 communication diagrams 190–1, 192, 193, 194 design issues 191–2 dynamic interaction among classes 189 first steps 189 identification of objects in collaboration 189–90 use case description, extended assembly of 207–8 operations, finding 203 operations, reality in 199 physical inputs/outputs, classes and 198

requirements analysis
(continued)
 relative stability of classes
 and instances 179–80
 requirements model, role
 of 172–3
 responsibilities, operations
 as 188
 review questions 209
 sale transaction, class or
 operation? 199
 significance of
 associations 183
 specificity in classes 198
 system development
 process 122, 123–4
 system scope, classes,
 objects and 197–8
 in TLC 51
 use case realization 173–9
 vagueness of classes 198
 requirements capture 139
 actors and use
 cases 145–54
 actors (system users) 146
 alternative courses (paths)
 in use cases 147–8
 background reading 132–3
 behaviour
 specifications 146
 business modelling with
 use case diagrams 153–4
 case study work, exercises
 and projects 158–9
 CASE tool support 150,
 153
 characteristics of system
 already in use 129–30
 communication
 diagrams 146
 document sampling 136–8
 documenting
 requirements 143–5
 essential use cases 147
 expert systems 139–40
 extend relationship in use
 cases 148–9
 fact-finding
 techniques 132–41
 mnemonic (SQIRO) for
 remembering 139
 functional
 requirements 131
 further reading 159
 generalization in use

cases 150–1
 include relationship in use
 cases 148–9
 interviewing 133–4, 135
 knowledge acquisition 140
 learning objectives 128
 legacy systems 129
 and modelling 121–3,
 154–5
 MoSCoW (Must Should
 Could Won't) rules for
 prioritization 145, 617
 new requirements 130–2
 non-functional
 requirements 131–2
 notation of use
 cases 146–51
 observation 134–6
 prototyping, supporting use
 cases with 151–3
 purpose of use cases 146
 questionnaires 138, 139,
 140–1
 real use cases 147
 review questions 157–8
 scenarios in use
 cases 147–8
 sequence diagrams 146
 specialization in use
 cases 150
 stereotypes 148
 UML comments 148–9
 usability requirements 132
 use case descriptions 146
 use cases (and use case
 diagrams) 144, 145–54,
 171–2
 user involvement 141–2
 user requirements 128–32
 wrappers 129
 requirements drift 38
 requirements model,
 refinement of
 abstract classes 231–2
 abstraction
 mechanisms 247
 abstraction mechanisms in
 UML 222–3
 adding further
 structure 230–9
 aggregation, concept
 of 235–6, 236–7
 analysis patterns 245–6
 antipatterns 229
 case study work, exercises

and projects 247–8
 component-based
 development (CBD) in
 UML 240–1
 components 226–7
 composition (composite
 aggregation)
 finding and
 modelling 235–6
 and generalization,
 combination of 236–7
 concrete classes 231–2
 decoupled subsystems 227
 dependencies between
 classes and packages,
 documentation of 237–9
 encapsulation 226
 further reading 248
 'future-proofing' 231
 generalization 225
 bottom-up approach to
 finding 234
 and composition
 (composite
 aggregation),
 combination of 236–7
 finding and
 modelling 230–5
 overuse of 234–5
 top-down approach to
 finding 233–4
 usefulness of 232–3
 when not to use 234–5
 inheritance 227
 learning objectives 222
 management of reuse,
 difficulties in 224
 modelling components in
 UML 241–4
 multiple inheritance 235
 NIH (Not Invented Here)
 syndrome 224
 notation for component
 structure diagram in
 UML 239–40
 object-orientation,
 contribution to
 reuse 225–30
 requirements reuse,
 difficulties in 224–5
 operations
 redefined 230–1
 organizing the analysis
 model, packages and
 dependencies 237–9

packages, partitioning
 into 237–9
 pattern, distinguished from
 strategy 244
 patterns 227–30
 patterns, movement of 223
 QWAN (Quality Without a
 Name) patterns 229
 redefinition of
 operations 230–1
 reuse, achievement
 of 223–5
 reuse, appropriateness
 of 224
 review questions 247
 software and specification,
 reuse of 223–30
 software development
 patterns 244–6
 strategy, distinguished from
 pattern 244
 structure, adding to 230–9
 subsystems, decoupled 227
 superclass 231
 UML support for
 modelling
 components 239–44
 requirements model, role
 of 172–3
 RESPECT User
 Requirements
 Framework 452, 453
 responsibilities, operations
 as 188
 responsibility 656
 restructuring for
 flexibility 420
 return-values 260
 usability in design 382
 reusable components
 AFE (Application Family
 Engineering) 589
 appropriate unit for
 reuse 582–4
 arguments for reuse
 578–9
 ASE (Application System
 Engineering) 589
 case study work, exercises
 and projects 600–1
 component standards 579,
 584–6
 componentware 577,
 589–91
 coupling levels between
 different classes 579
 CSE (Component System
 Engineering) 589
 further reading 601
 learning objectives 577
 levels of reuse, reasons for
 deficiencies in 578–9
 organizational structure for
 reusability 580–2
 planning a strategy for
 reuse 586–9
 planning for reuse too
 late 579
 platform dependence of
 components 579
 project choice for
 reusability 579–80
 review questions 600
 RSEB (Reuse-driven
 Software Engineering
 Business) 586, 588–9
 security subsystem as
 reusable component,
 case study
 example 591–9
 access control 591–2
 class diagram showing
 collection classes 597
 class diagram showing
 control classes 596
 collection classes 596–7
 design
 alternatives 592–3
 Façade pattern 594–5
 interface 593–4, 599
 persistence
 mechanism 596
 security classes, separate
 packaging for 593–4
 security package
 dependencies 594,
 597–8
 security server control
 classes 594, 596
 SELECT perspective 586,
 587–8
 standards for reusable
 components, lack of 579
 reusable software
 components 86–7
 reuse 656
 achievement in refinement
 of requirements

model 223–5
 appropriateness of 224
 reverse-engineering 345–6
 review and
 maintenance 570–4
 review process 570–2
 Review Questions
 boundary classes, design
 of 492
 control, specification
 of 326–7
 data management
 design 533–4
 design patterns 438
 detailed design 414
 development problems,
 avoidance of 66
 development problems,
 information systems 45
 HCI (Human–Computer
 Interaction) 461
 implementation 575–6
 information systems 24
 modelling concepts 126–7
 object interaction 280–1
 object-orientation 89
 operations, specification
 of 301
 requirements analysis 209
 requirements
 capture 157–8
 requirements model,
 refinement of 247
 reusable components 600
 software development
 processes 624
 system architecture 368–9
 systems design 392
 RIBA (Royal Institute of
 British Architects) 339
 risk mitigation 373
 RMI (Remote Method
 Invocation) 494–5,
 498, 522, 536, 539,
 543, 598
 RPC (Remote Procedure
 Calls) 522
 RSEB (Reuse-driven
 Software Engineering
 Business) 579–80, 581,
 588, 599–600
 reusable components 586,
 588–9
 run-time support 553

'run-to-completion'
semantics 308
RUP *see* IBM-Rational Unified Process

SAGA (German Standards and Architectures for eGovernment Applications) 347

sale transaction, class or operation? 199

SanFrancisco project 590–1, 600

Sarbanes-Oxley Act (SOX, US) 347

scenario-based approaches to design of user interface 455–7

scenarios in use cases 147–8

scheduled handling of concurrency 364–5

scope, precision in determination of 377, 378

screen-scraping 345

seamlessness of O-O methods 374

second normal form (2NF) 513

secondary indexing of files 502–3

security classes, separate packaging for 593–4

security in DBMS 507

security in systems design 380

security package dependencies 594, 597–8

security server control classes 594, 596

security subsystem as reusable component, case study example 591–9

access control 591–2

class diagram showing collection classes 597

class diagram showing control classes 596

collection classes 596–7

design alternatives 592–3

Façade pattern 594–5

interface 593–4, 599

persistence mechanism 596

security classes, separate packaging for 593–4

security package dependencies 594, 597–8

security server control classes 594, 596

Select Business Solutions 345

SELECT perspective 586, 587–8, 599

semantics of a model 77

send message event 255

sequence diagrams *see* interaction sequence diagrams

sequential cohesion 383

sequential organization of files 500

Serco Usability Services 453

serial access to files 500

serial organization (serialization) of files 496, 500

services 656

influence on system architecture 345

shallow history pseudostate 317–18, 656

SIB (Standards Information Base) 348

signal *see* asynchronous message

signal triggers 306

signatures (message protocols) 80

singleton creational pattern 420–5, 435, 541

skills, compatibility in developers' 34

Smalltalk 34, 84, 87, 229, 419, 464, 481, 553

SOA (Service-Oriented Architecture) 345

SOAP (Simple Object Access Protocol) 498, 522, 579, 584

soft methodologies vs hard methodologies 621–4

software and specification, reuse of 223–30

testing in

implementation 561–5

tools in implementation 552–7

working knowledge of 34

software architecture 340–2, 657

software development patterns

design patterns 416–18

principles and 417

requirements model, refinement of 244–6

software development processes

abstraction levels in software development 603–4

case study work, exercises and projects 625

DSDM (Dynamic Systems Development Method) 613–18

business study phase 615, 616

design and build iteration phase 615, 616

feasibility study phase 615, 616

functional model iteration phase 615, 616

implementation phase 615, 616

lifecycle 615–17

principles underlying 614–5

project selection in 617–18

timeboxing 617

further reading 625

learning objectives 602

methodology 604–6

aspects of development process 604–5

hard vs soft methodologies 621–4

issues in choosing a 619–21

rationale for use of a 606

variety in approaches to 605–6

PD (Participatory Design) 611–13

process, method and methodology 603–4

review questions 624

SSADM (Structured Systems Analysis and Design Method) 611

systems development methodologies 602–6

USDP (Unified Software Development Process) 607–11

artifacts 610

complexity 610–11

construction phase 609

elaboration phase 608–9

inception phase 608

maturity 610

phases and workflows 607–9

philosophy and principles 607

transition phase 609

workers and activities 609–10

XP (eXtreme Programming) 618–19

Software Engineering Institute 621

software errors, hazardous to life 30

solutions, selected 635–47

specialization 76–7, 657

specialization cohesion 384–5

specialization in use cases 150

specialization of state machines 318

specification in prototyping 55

specificity in classes 75–6, 198

SPEM (Software Process Engineering Metamodel) 605

spiral model for incremental delivery 56–7

SQIRO (Sampling, Questionnaires, Interviewing, Reading (or Research) and Observation) for fact finding 139

SQL-Server 509

SQL (Structured Query Language) 497, 508, 510, 517, 518, 519, 520, 527, 532, 547, 549, 553, 572

SSADM (Structured Systems Analysis and Design Method) 34, 86, 130, 131, 304, 611

software development processes 611

SSM (Soft Systems Methodology) 622

stakeholders 657

effects on 40–1

Standard Life 578

standards and legal requirements in HCI 458–60

standards for reusable components 579

state 657

see also object state

state activities compartments 308–9

state behavioural pattern 430–5

state-dependent variations in behaviour 303–4

state machine 657

state machines activity-expressions 308–9

basic notation for 306–11

behavioural approach to preparation 319–22

behavioural state machines 324–5

choice pseudostates 315–17

composite states 312–13

concurrent states 313–15

consistency checking 325

decomposition compartments 312–13

deep history pseudostate 317, 318

deferred events 308

drawing up interface model 485–9

entry activities compartments 309

entry pseudostates 315

exit activities compartments 309

exit pseudostates 315

fork pseudostates 315, 317

further notation for 311–18

guard condition 307

history pseudostates 317–18

initial pseudostate 306–7

and interaction diagrams 304

internal activities compartments 308–9

join pseudostates 315

junction pseudostates 315–17

lifecycle approach to preparation 322–4

merge 316

name compartments 308

preparation of 318–24

protocol state machines 324–5

quality guidelines 325–6

reflexive transition 311

'run-to-completion'

semantics 308

shallow history pseudostates 317–18

specialization of 318

state activities compartments 308–9

static conditional branch branch 316

submachines 312–13

substates 312

transition strings 307

state variables 305

statecharts 304

states and events 304–6

static conditional branch *see* junction pseudostate

static operations 421

stereotyped diagrams 119

stereotypes 148, 657

analysis class stereotypes 176–9, 648

storage mechanisms, overview of 496

strategies business strategies, identification of 19–20

distinguished from patterns 244

for implementation 568–70

information technology (IT) and information systems (IS) 21–3

strategies (*continued*)
 planning a strategy for reuse 586–9
 strategic information systems planning 49–50 for success with IS 19–23
 strategy design pattern 483
 structural additions to requirements model, refinement of 230–9
 structural approaches to design of user interface 451–3
 structural patterns 419, 425–9
Structured English
 complex structures in 293–4
 iteration in 292–3
 main principles of 291
 multiple outcomes in 292
 operations, specification of 282, 291–4
STUDIO (STructured User-interface Design for Interface Optimization) 452, 484
style guides, user interface 449
subclass 657
subdivision of the development process 48–9
submachines 312–13
substates 312
subsystems 349–50, 657
 decoupled 227
 UML notation for 111
 Sun Microsystems Inc. 92, 532
superclass 231, 657
swimlanes in UML 117
SWOT (Strengths, Weaknesses, Opportunities, Threats) approach 20
SWT (Standard Widget Toolkit) 375
Sybase 509
synchronizing operation 657
synchronous messages 262, 657
syntactic correctness in CASE 63
system 657

system architecture
 ADL (Architecture Description Language) 342
 Agate software architecture 366–8
 architectural models, why produce them? 342–4
 architectural styles 348–63
 architecture, what is meant by? 339–42
 aspects of software architecture 341
 broadness of topic 338–9
 broker architecture for distributed systems 360–1
 case study work, exercises and projects 369
 client-server communication 350
 conceptual aspect of software architecture 341
 conceptual aspect of software architecture 341
 concurrency 363–5
 code aspect of software architecture 341
 deployment view of software architecture 341, 342, 343
 distributed systems, architectures for 360–2
 enterprise subsystems 349–50
 execution aspect of software architecture 341, 348
 existing systems, influence on 344–5
 further reading 369–70
 heritage systems, influence on 345
 implementation view of software architecture 341, 342, 343
 influences on 344–8
 key features of 339–40
 layered subsystems 350–5
 learning objectives 338
 logical concurrency 363–5

logical view of software architecture
 341, 342, 343
 MDA (Model-Driven Architecture), influence on 345–6
 module aspect of software architecture 341
 MVC (Model-View-Controller) architecture 356–60
 organization structures for architecture and development 362–3
 partitioned subsystems 350, 355–6
 peer-to-peer communication 350
 physical concurrency 363–5
 process view of software architecture 341, 342, 343
 processor allocation 365–6
 proxy components, introduction of 361–2
 reverse-engineering, influence on 345–6
 review questions 368–9
 scheduled handling of concurrency 364–5
 screen-scraping 345
 services, influence on 345
 software architecture 340–2
 subsystems 349–50
 technological reference architectures 344, 348
 terminal emulation 345
 use case view of software architecture 341, 342, 343
 views of software architecture 341, 342
see also design patterns; detailed design; systems design

system development process
 class design 122, 124–5
 construction 122, 125
 data management design 122, 125
 implementation 122, 126
 main activities 120–6

modelling
 concepts 119–26
 requirements analysis 122, 123–4
 requirements capture and modelling 121–3
 system design 122, 124
 testing 122, 125
 underlying principles 119–20
 user interface design 122, 125
system engineering in TLC 50–1
system feedback, adequacy of 447–8
system of subsystems in containment, UML notation for 111
system scope, classes, objects and 197–8
system simulation in CASE 64
systems design
 analysis, criteria for good 377–9
 analysis and design, difference between 371, 372–4
 analysis and design, qualities and objectives of 377–86
 buildable design 381
 case study work, exercises and projects 393
 change management 373–4
 class cohesion 384
 cohesion in design 382–5
 coincidental cohesion 383
 completeness in analysis 377, 378
 consistency in analysis 377, 378
 constraints on 379–82
 construction guidelines 389
 content, accuracy in determination of 377, 378
 coupling, concept of 382–5
 data management issues 372, 387–8

design, criteria for good 379
design and analysis, qualities and objectives of 377–86
design in the iterative lifecycle 373–4
design trade-offs, prioritization of 389–91
detailed design and 376
development standards 388–9
economical design 380
efficiency in design 379
flexibility in design 380–1
framework for 371–2
functional effectiveness 379
further reading 393
general-purpose design 381
guidelines for HCI 388
HCI guidelines 388
implementation, design for 391
inheritance coupling 383–4
input/output device guidelines 388–9
interaction coupling 383
learning objectives 371
logical and physical design 374–6
logical cohesion 383
LSP (Liskov Substitution Principle) 385–6
maintainability in design 381
manageable design 381
measurable objectives in design 386–7
middleware 375
modifiability in design 380–1
objectives and constraints 379–82, 386–7, 390
operation cohesion 384
physical and logical design 374–6
portability in design 381
quality improvement, testing deliverables 373, 374
reliability in design 380

reusability in design 382
review questions 392
risk mitigation 373
scope, precision in determination of 377, 378
seamlessness of O-O methods 374
security in design 380
sequential cohesion 383
specialization cohesion 384–5
system design, detailed design and 376
system development process 122, 124
team learning 373, 374
temporal cohesion 383
usability in design 381–2
see also design patterns; detailed design; systems design
systems development methodologies 602–6
systems theory basic concepts of general theory 5–15
boundary 6–9
control in systems 11
emergent properties 13–14
environment 6–9
feed-forward 13
feedback 11–13
holism in systems approach 13–14
input 9–10
interface 9–10
negative feedback 11–13
output 9–10
positive feedback 11–13
subsystems 10
systems, characteristics of 5–6
systems, relationship with reality 8
transformation 9–10

tables 657
 mapping classes to 516–19
 tagged data records 499
tasks 484, 657
Taurus system at LSE 27, 32, 43, 44
taxonomic relationship 75
TCL/TK 152

TEAF (Treasury Enterprise Architecture Framework) 347
team learning 373, 374
technical and user documentation 574
technique 657
technological progress in IT 19
technological reference architectures 344, 348
templates aspects of (other than contents) 418–19
contents of 418
documenting patterns pattern templates 418–19
template collaboration 657
temporal cohesion 383
temporary files 504
terminal emulation 345
test documentation 564–5
testing aim in implementation 563–4
black box testing 562
of classes 563–4
quality improvement, testing deliverables 373, 374
regression testing 564
software implementation 561–5
system development process 122, 125
tools for 555
in traditional lifecycle (TLC) 51
what is tested in implementation? 562–4
white box testing 563
who carries out tests in implementation? 561–2
third normal form (3NF) 512, 514
3GLs 84
threads of control 269
three-schema architecture in DBMS 506–8
Thriving on Chaos (Peters, T.) 131
throughput of the user interface 458
throwaway prototyping 466

time constraints 33–4
in sequence diagrams 267–8
timeboxing in DSDM 617
timing diagrams 657
in object interaction 278–9
TLC (Traditional Life Cycle) 50–4, 61
construction in 51
design in 51
feedback loops within 53–4
installation in 51
maintenance in 51
requirements analysis in 51
system engineering in 50–1
testing in 51
TOGAF (Open Group Architecture Framework) 348
top-down approach to finding 233–4
TQM (Total Quality Management) 621
traceability in CASE 64
transaction files 503
transaction management in DBMS 507
transactions 658
transactions flow 16–17
transient data 495–6
transitions 658
in control, specification of 305
transition phase in the USDP 609
transition strings in state machines 307
transitive operation of inheritance 78–9
triggers and events 305–6
trust of users 32
tuning of storage in DBMS 507
types 658
of DBMS 508–10
of design patterns 419–20
of files 503–4
UCM (User-interface Control Model) 484
UDDI (Universal Description, Discovery and Integration) 585

UK Passport Agency 27
UML (Unified Modelling Language) 60, 74, 126, 128, 143, 213, 222–3, 250–2, 280, 303–4, 368, 397
abstraction mechanisms in 222–3
CBD (Component-Based Development) in 240–1
comments in requirements capture 148–9
communication diagrams in 271–5
concept of a model 103, 110–11
definition of diagrams in 110
diagrams, elements of 108
diagrams in 110
frames in, flexibility in use of 305
interaction overview diagrams in 275–8
interaction sequence diagrams in 252–71
interfaces in 402–3
metamodel 108–9
model consistency in 279–80
modelling components in 241–4
models in 110–11
notation for component structure diagram in 239–40
notation for models 111
notation for packages 111
notation for subsystems 111
notation for system of subsystems in containment 111
Request for Proposals 342
Specification 107–8, 110–11, 471, 557
state machine notation 304–26, 489
state machines 452
state machines, specialization of 318
subsystems in 110
support for modelling components 239–44
swimlanes in 117

systems in 110
timing diagrams in 278–9
views in 110
unemployment effect 42
UNIX 39, 349, 364, 376
UP see USPD
usability 28–9
achievement in user interface 457–8
ISO definition of 458
in systems design 381–2
usability requirements 132, 658
USDP (Unified Software Development Process) 34, 57–8, 113–14, 119–20, 126, 189, 256, 338, 343, 373, 464, 607–11, 624
artifacts 610
complexity 610–11
construction 57
construction phase 609
elaboration 57
elaboration phase 608–9
inception 57
inception phase 608
lifecycle 57–8
maturity 610
phases and workflows 607–9
philosophy and principles 607
transition 57
transition phase 609
workers and activities 609–10
use case description extended version 189–90
in requirements capture 146
use case realization 173–9, 658
use case view of software architecture 341, 342, 343
use cases and actors (system users) 145–54
essential use cases 147
extend relationship in 148–9, 652
generalization in 150–1
include relationship in 148–9, 652
notation of 146–51
prototyping and 151–3
purpose of 146
real use cases 147
scenarios in 147–8
specialization in 150
and use case diagrams 144, 145–54, 171–2, 658
use of design patterns 435–6
usefulness of generalization 232–3
user documentation 574
manuals 567
user input, minimalization of 448–9
user interface attitude of users of the 458 behaviour, description of 485
characteristics of good dialogues 446–9
consistency in dialogues 446
design, critical nature of 440
design, main steps in 451
design, system development process 122, 125
dialogue metaphor for 441–4
direct manipulation metaphor for 445–6
ethnographic approaches to design of 453–5
flexibility of the 458
formal approaches to design of 450–7
informal approaches to design of 450–7
learnability in the 458
metaphors representing 441–6
prototyping in design of boundary classes 466–9
rules, definition of 485
scenario-based approaches to design of 455–7
structured approaches to design of 451–3
style guides 449
system feedback, adequacy of 447–8
throughput of the 458
usability, achievement of 457–8
usability, ISO definition of 458
user input, minimalization of 448–9
user support, appropriateness in 447
what is it? 441
user involvement in requirements capture 141–2
in systems development 59–60
user requirements 128–32, 658
user satisfaction 571
user story 658
user support, appropriateness in 447
users blaming for problems 35 trust of 32 utility 29–30
vagueness of classes 198
validation of patterns 419
Value Chain Model 20–2
vapourware 28
variable length records 499
VAT (Value Added Tax) 396
vertex 658
vertical prototyping 466
views of software architecture 341, 342
visibility 658
Visual Basic 34, 375, 466, 497, 553, 578, 589
Visual C++ 589
Visual Café 466
visual editors 553
WAP (Wireless Access Protocol) 450
waterfall lifecycle model 50–4
WebLogic (BEA) 554
Websphere (IBM) 554
Wessex Regional Health Authority, UK 28
white box testing 563
Windows CE 376
Windows Forms library 472

- Windows Interface Guidelines for Software Design* (Microsoft, 1997) 449
Windows NT 376
Windows XP 364
Wordperfect 39
work files 504
workers and activities in USDP 609–10
wrappers 129, 658
- WSDL (Web Services Description Language) 585
WWW (World Wide Web) 37, 92, 242, 441, 450, 584
XMI (XML Metadata Interchange) 552, 584
XML (Extensible Markup Language) 154, 498, 500
- XP (eXtreme Programming) 61, 171, 284, 562, 612, 622, 624, 652
software development processes 618–19
- Zachman framework for enterprise architecture 347