Chapter 5 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 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. Diagrams are often confused with models. A diagram is a graphical view of a part of a model for a particular purpose.

The best way to understand what we mean by a diagram 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. Activity diagrams are useful for modelling sequences of actions from business processes within an organization (or between organizations) down to the detail of how an operation works. Activity diagrams are one of the techniques that can be used to model the behavioural view of a system, and their use in systems analysis and design is explained in Chapter 10, where they are used as one way of specifying operations. We introduce them here as an example of a UML diagram and because, as in the Unified Process, we use them to model the development process that we use in the book.

A systems analysis and design project needs to follow some kind of process. We have adopted a relatively lightweight process based on the Unified Process.

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?

A model is an abstract representation of something real or imaginary. Like maps, 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 reinvent 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 analysts, 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 analysts, designers and programmers may later find their freedom of action too restricted. Most systems development models today are held as data in modelling tools, and much of that data is represented visually in the form of diagrams, with supporting textual descriptions and logical or mathematical specifications of processes and data.[1]

5.2.2 What is a diagram?

A diagram is a visual representation of some part of a model. Analysts and designers use diagrams to illustrate 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.

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.

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.

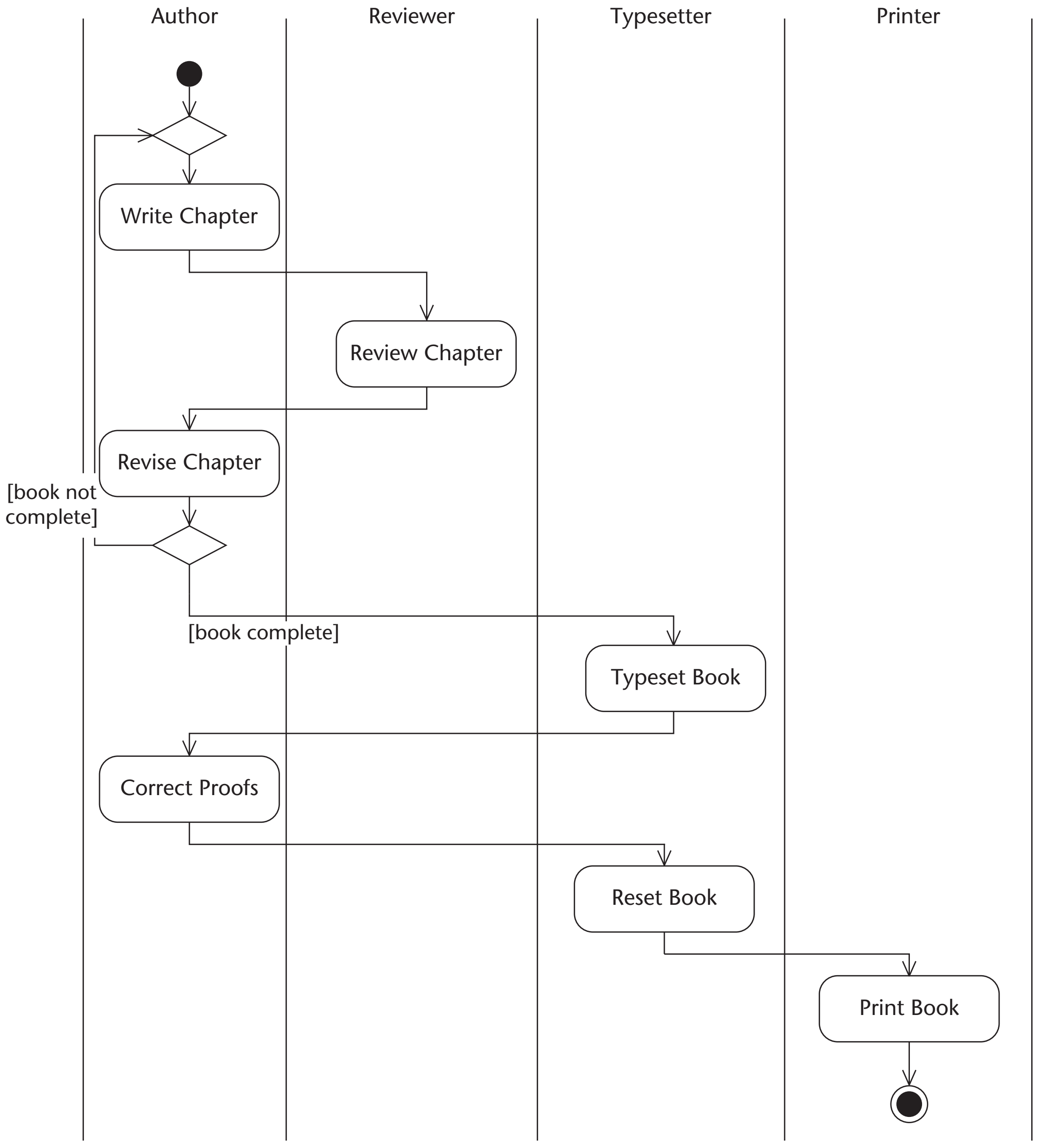

Figure 5.1 Activity diagram for producing a book.

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 2.2* (OMG, 2009). 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

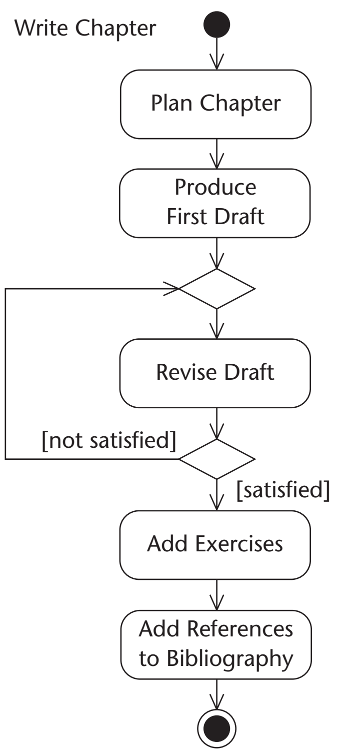

Figure 5.2 Activity diagram for the activity `Write Chapter`.

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 least if they

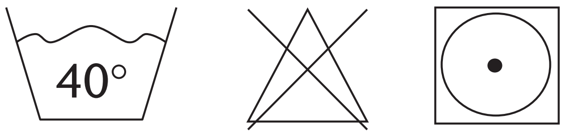

Figure 5.3 Example from a diagram standard.

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 classifies diagrams as either *structural diagrams* or *behavioural diagrams*. Structural diagrams show the static relationship between elements, and are package, class, object, composite structure, component and deployment diagrams. Behavioural diagrams show an aspect of the dynamic behaviour of the system being modelled, and are use case, activity, state machine, communication, sequence, timing and interaction overview diagrams.

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 UML 1.X specifications, and are no longer used in UML 2.2. 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, 2009) 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).

There is an example on the book's website of 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.

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.

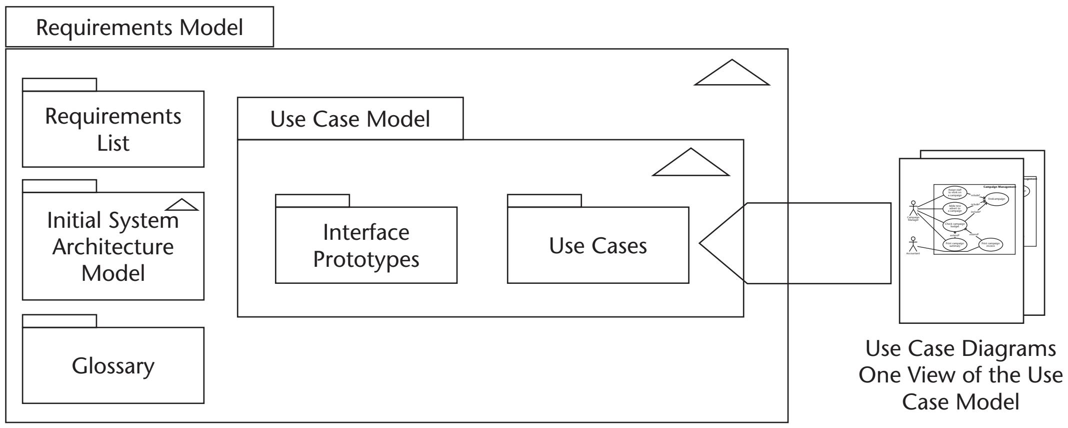

Figure 5.4 Illustration of a UML model and its relationship with one type of diagram.

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 types 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 cases and prototypes of some use cases (see Chapter 6) and an initial system architecture model which defines initial subsystems (see Section 5.2.4). Note that models can contain diagrams, data and textual information. Figure 5.4 shows this: on the left-hand side of the diagram is a UML diagram showing the contents of models and packages (see Section 5.2.4), while the right-hand side of the diagram illustrates schematically the fact that use case diagrams are one possible view of the contents of the use case model.

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 2.2 Superstructure Specification (OMG, 2009b) 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:

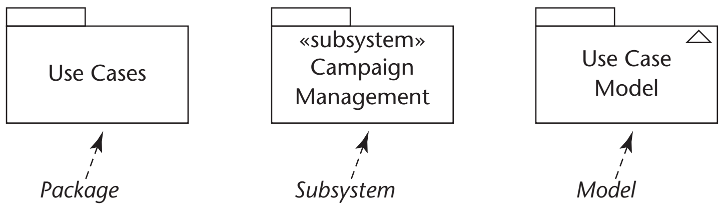

Figure 5.5 UML notation for packages, subsystems and models.

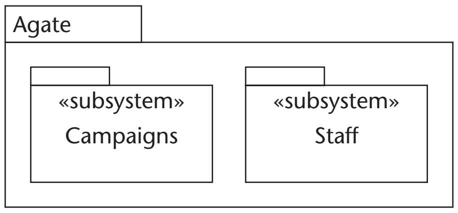

Figure 5.6 UML notation for a system containing subsystems, shown by containment.

- 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 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 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).

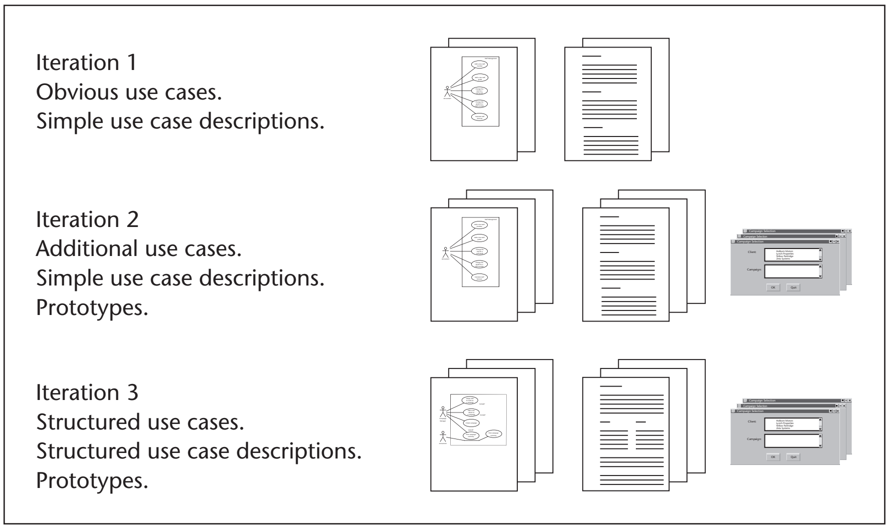

Figure 5.7 Development of the use case model through successive iterations.

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 activity diagrams, Object Constraint Language, structured English or pseudo-Code (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. We are introducing activity diagrams at this point, first to provide an illustration of a UML diagram type, and second, so that we can use them to illustrate the development process that we use in the book.

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 processes 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 each 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 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 object-oriented analysis and design replace older 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 flowchart.[2] Activity diagrams are essentially flowcharts in an object-oriented context.

UML 2.0 changed the underlying model for activity diagrams. In UML 1.X they were based on state machines (see Chapter 11), but are now distinct from state machines and based on Petri nets.

5.3.2 Notation of activity diagrams

Activity diagrams at their simplest consist of a set of *actions* linked together by flows from one action 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 *control flow*.

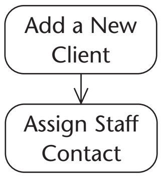

Figure 5.8 Example of two activities joined by a control flow.

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

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 2.2 activity diagrams have the semantics of Petri Nets.

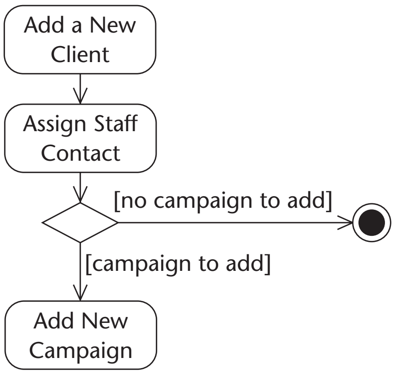

Figure 5.9 Activities with a decision node.

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:

> 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 Figure 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 Figure 5.10.

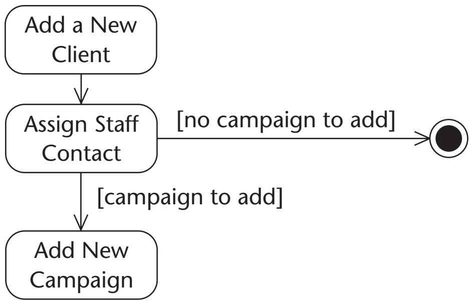

Figure 5.10 UML 1.X choice represented without an explicit decision point.

However, this is no longer possible since 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 2.2, 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. The flow of control will follow along the first control flow with a guard condition that evaluates to true. 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 a 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.

Figure 5.11 also shows a feature of UML diagrams from version 2.0 onwards: every diagram can be drawn in a *frame*, a rectangle with the heading of the diagram in the top left hand corner. The heading consists of the kind of diagram (this is optional), in this

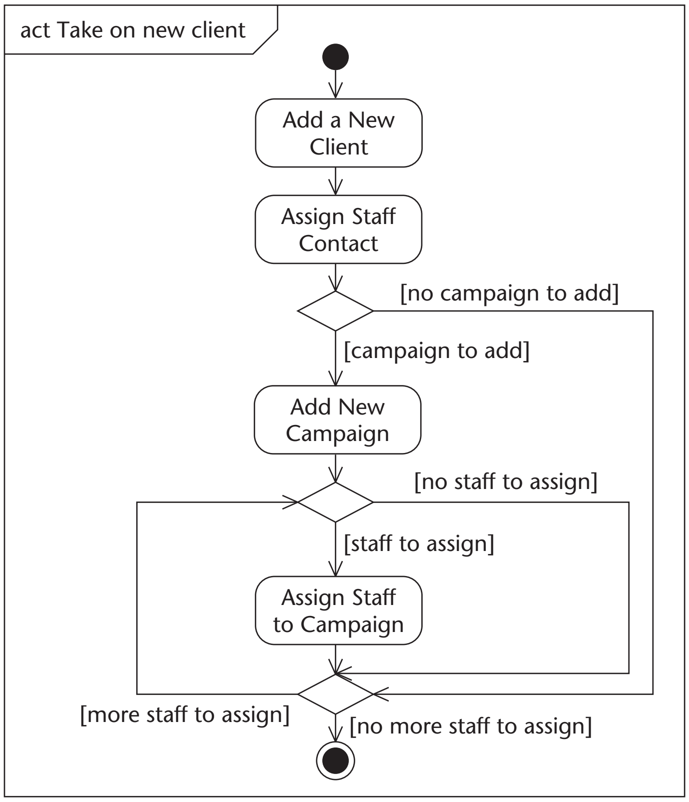

Figure 5.11 Activity diagram in frame with initial node.

case `activity` abbreviated as `act`, the name of the diagram and optional parameters. Frames are really only required for diagrams such as sequence diagrams where messages can enter the diagram from the boundary represented by the frame, as in Fig. 9.6.

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. UML 2.0 added a large number of types of actions to the metamodel for activity diagrams. These actions are the kind of actions that take place in program code. These include actions such as *AddVariableValueAction* and *CreateObjectAction*. They are intended to make it easier to create activity diagrams that can model the implementation of operations and can be compiled into a programming language: *Executable UML*.

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:

- 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

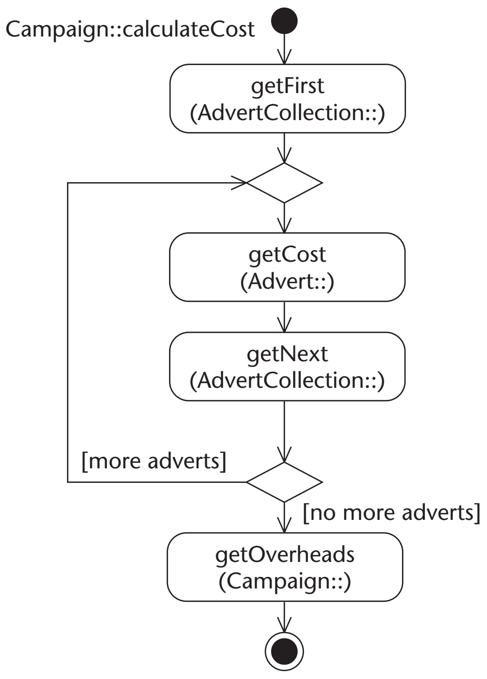
Campaign::calculateCost
getFirst (AdvertCollection::)
getCost (Advert::)
getNext (AdvertCollection::)
[more adverts]
[no more adverts]
getOverheads (Campaign::)

Figure 5.12 Activity diagram with operations of classes as actions.

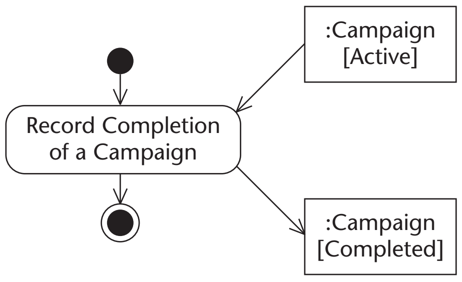

Figure 5.13 Activity diagram with object flows.

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. This is one of those specialized actions to support Executable UML: a *CallOperationAction*.

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 may result 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 Fig. A2.2.)

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.

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

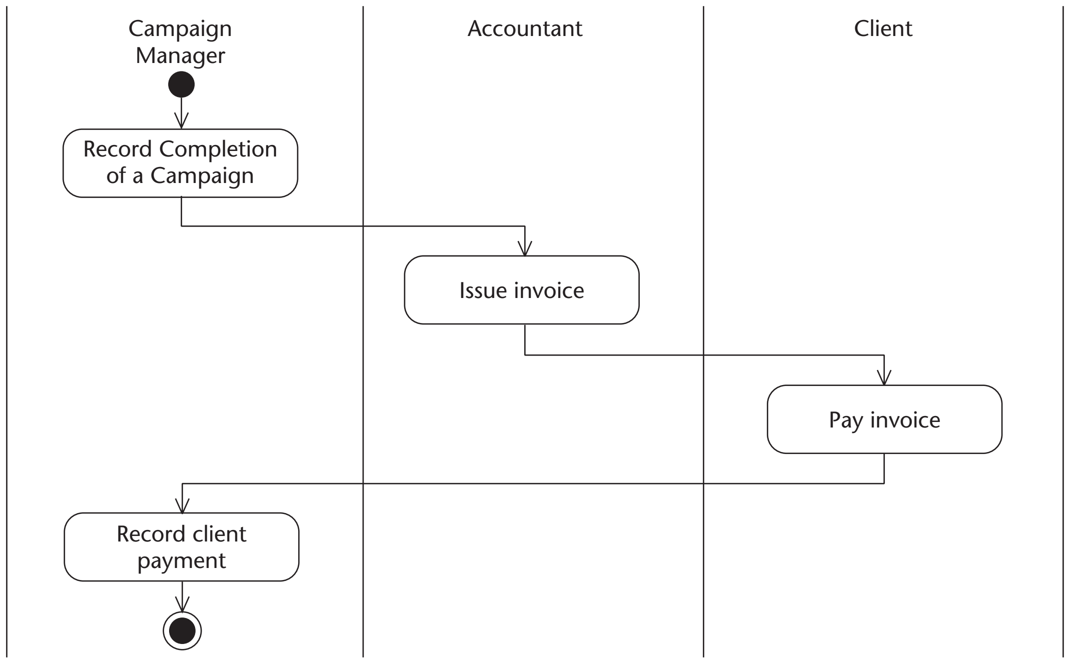

Figure 5.14 Activity diagram with activity partitions.

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 Chapter 22 on 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). USDP was originally developed by the team that created UML. It is claimed that 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 centrism
- visual modelling techniques.

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

USDP does not follow the traditional Waterfall Lifecycle shown in Fig. 3.3 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 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

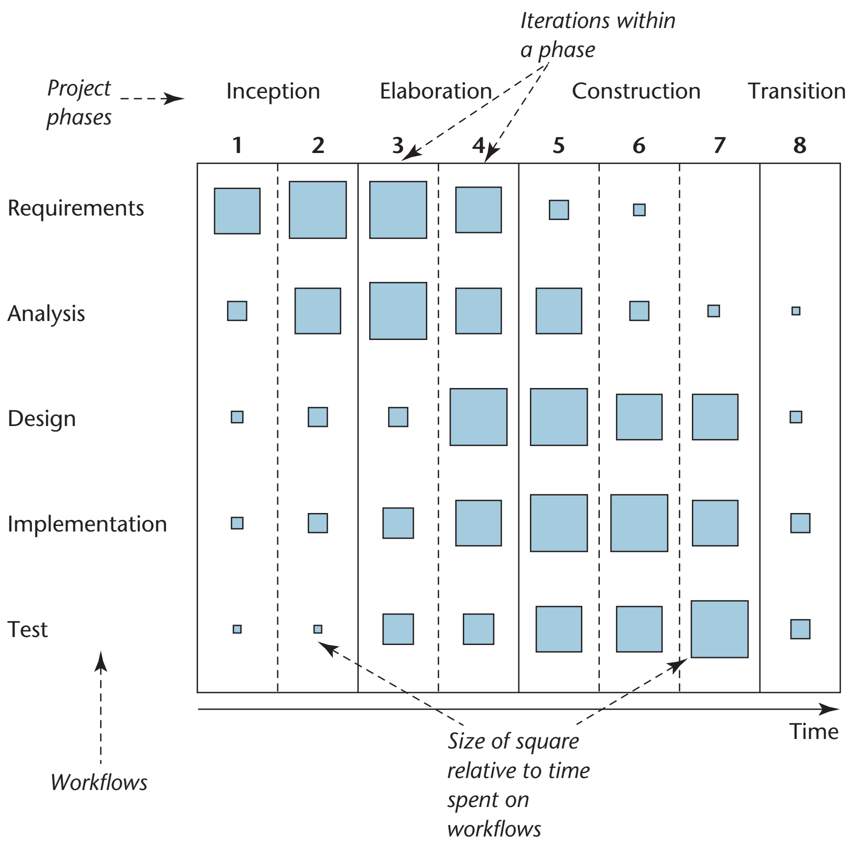

Figure 5.15 Phases and workflows in the Unified Software Development Process.

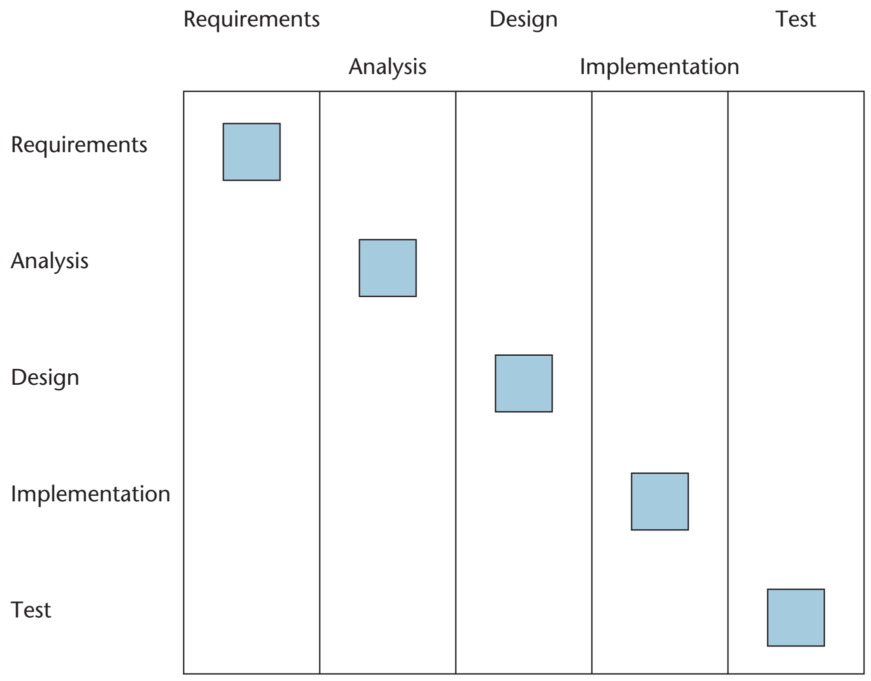

Figure 5.16 Phases and activities in a simplified waterfall process.

or another in most system development methodologies. The 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 architecture and design
- 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.

Activity	Techniques	Key Deliverables	Diagrams Used
Requirements capture and modelling	Requirements elicitation Use case modelling Architectural modelling Prototyping	Use case model Requirements list Initial architecture Prototypes	Use case diagrams Package diagrams
Requirements analysis	Communication diagrams Class and object modelling Analysis modelling	Analysis models	Class diagrams Object diagrams Communication diagrams
System architecture and design	Deployment modelling Component modelling Package modelling Architectural modelling Design patterns	Overview design and implementation architecture	Package diagrams Component diagrams Deployment diagrams Class diagrams
Class design	Class and object modelling Interaction modelling State modelling Design patterns	Design models	Class diagrams Object diagrams Sequence diagrams State machine diagrams
Interface design	Class and object modelling Interaction modelling State modelling Package modelling Prototyping Design patterns	Design models with interface specification	Class diagrams Object diagrams Sequence diagrams State machine diagrams Package diagrams
Data management design	Class and object modelling Interaction modelling State modelling Package modelling Design patterns	Design models with database specification	Class diagrams Object diagrams Sequence diagrams State machine diagrams Package diagrams
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.

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.

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 and a requirements list. A use case captures an element of functionality and the requirements model may include 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 in terms of an outline package structure (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.

Requirements analysis

This activity analyses the requirements. 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

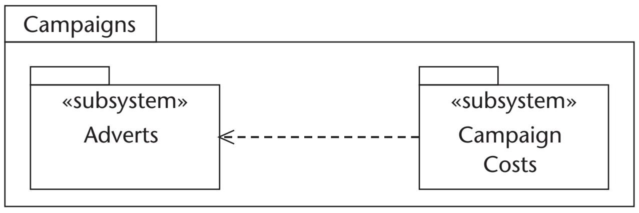

Figure 5.18 Part of the initial system architecture for the Agate system.

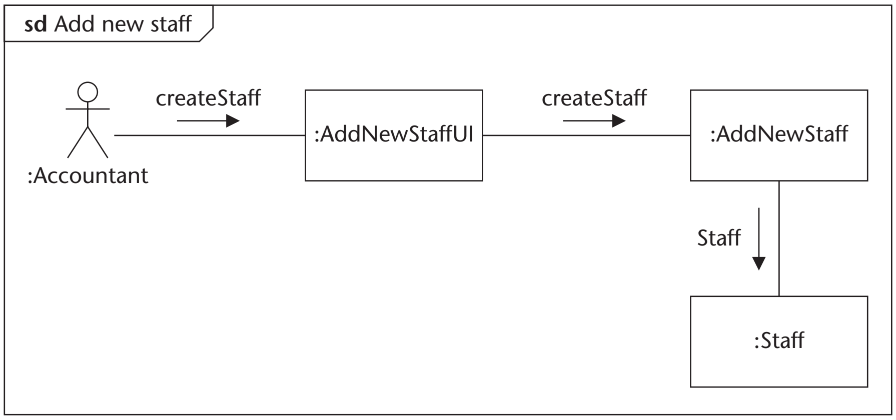

Figure 5.19 Part of a communication diagram for the use case `Add New Staff`.

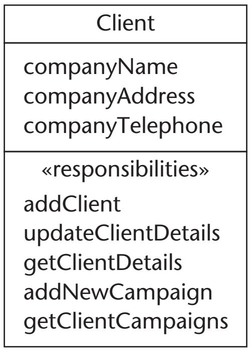

Figure 5.20 Partly completed sample analysis class.

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. Object diagrams may be used to analyse the links between objects in order to determine the associations between classes.

System architecture and design

In this activity 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.

As well as package diagrams, shown here, component diagrams are used to model logical components of the system, and deployment diagrams are used to show the physical architecture of processors and the software that will run on them.

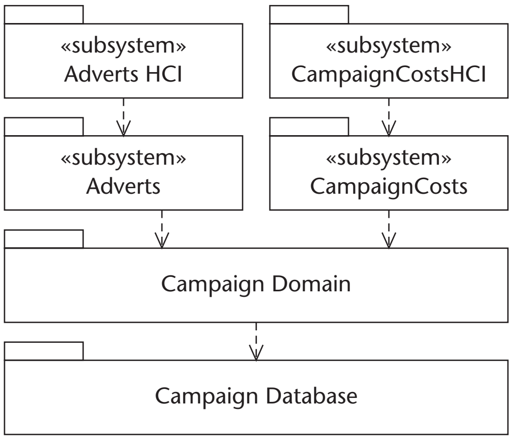

Figure 5.21 Possible architecture for part of the Agate system.

System architecture and 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 architecture and design is explained in Chapter 13.

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 machine diagrams may be prepared for objects 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.

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

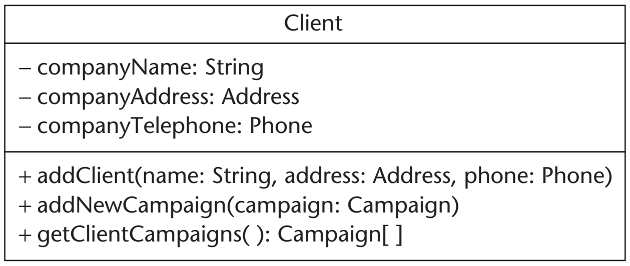
Client
– companyName: String
– companyAddress: Address
– companyTelephone: Phone
+ addClient(name: String, address: Address, phone: Phone)
+ addNewCampaign(campaign: Campaign)
+ getClientCampaigns( ): Campaign[ ]

Figure 5.22 Partly completed sample design class.

the system and the nature of online help. Interface design is explained in Chapter 17 and is very dependent on class design. Sequence diagrams are used to model the interaction between instances of classes, and state machine diagrams are used to model the way in which the user interface responds to user events, such as mouse clicks and the entry of data. The class model is updated with new classes representing the user interface, and detail is added as the interaction becomes better understood.

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. Sequence diagrams are used to model the interaction between instances of classes, and state machine diagrams are used to model the way that objects change state over time in response to real world events. The class model is updated with new classes representing the way in which data will be stored, including data management frameworks.

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. Class, sequence, state machine, component and deployment diagrams provide the specification to the developers.

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. Different kinds of tests are carried out as the construction work proceeds. Testing is not all left to the end.

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 system 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 defines two types of diagram: structural and behavioural. 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 represented visually in 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 two types of 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 action in a UML activity diagram?

5.8 What links actions in an activity diagram?

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

5.10 What is the notation for the two special nodes that start and finish 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 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

Although activity diagrams are used in UML for a variety of purposes, including modelling business processes, there are other notations that are becoming more widely used specifically for business process modelling. These have their origins in workflow modelling notations and the growth of business process automation packages that use web services to carry out steps in a business process. The Business Process Modelling Notation (BPMN) is becoming the standard in this area, and is now managed by the OMG (http://www.bpmn.org/).

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 USDP.

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

Executable UML is explained in a book by Mellor and Balcer (2002).