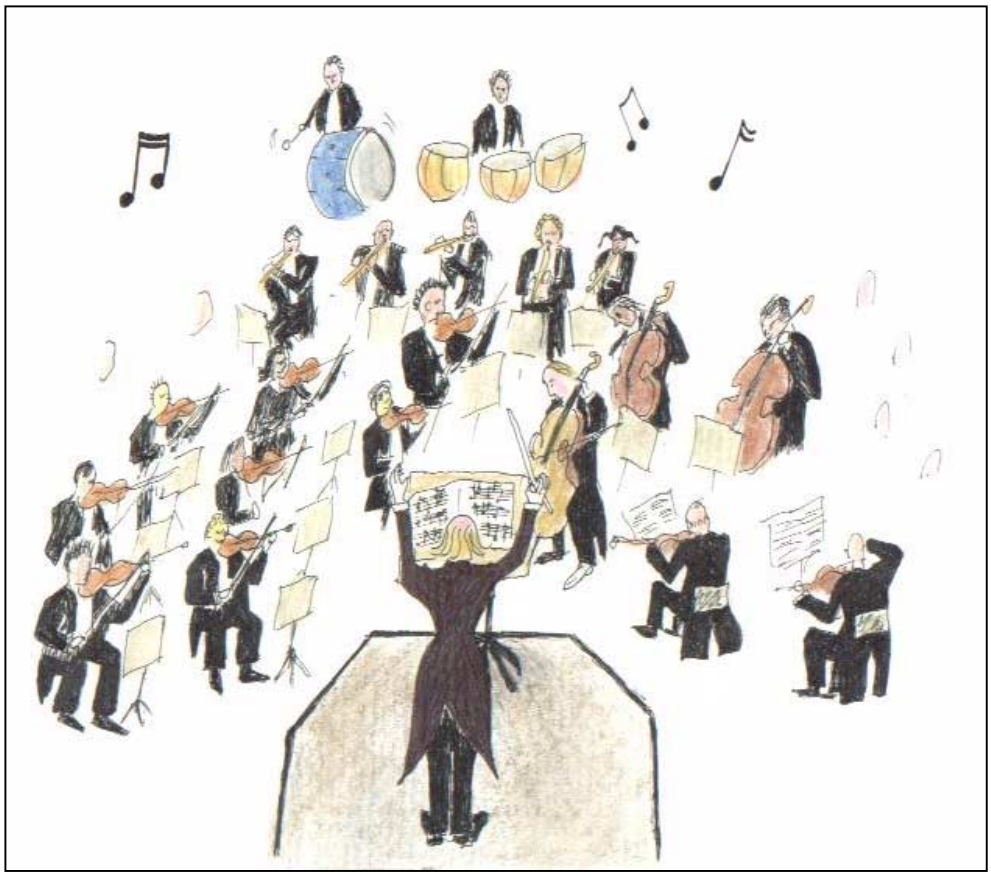
Ariadne
training

UML Applied

Object Oriented Analysis and Design Using the UML

A Course Companion



Authors and Contacts

Please contact info@ariadnetraining.co.uk, or see the website at www.ariadnetraining.co.uk for further details about Ariadne's supporting training courses. Comments and feedback are welcome.

Contents

Chapter 1
An Introduction to the UML

What is the UML?

The *Unified Modelling Language*, or the *UML*, is a graphical modelling language that provides us with a syntax for describing the major elements (called *artifacts* in the UML) of software systems. In this course, we will explore the main aspects of the UML, and describe how the UML can be applied to software development projects.

Through to its core, UML leans towards object oriented software development, so in this course, we will also explore some of the important principles of object orientation.

In this short chapter, we'll look at the origins of the UML, and we'll discuss the need for a common language in the software industry. Then we will start to look at how to exploit the UML on a software project.

A Common Language

Other industries have languages and notations, which are understood by every member of that particular field.

$$\int_0^\infty 1/x^2 \, dx$$

Figure 1 - A Mathematical Integral

Although the picture above is a fairly simple drawing (a stylised "S" figure), mathematicians the world over recognise instantly that I am representing an integral. Although this notation is simple, it masks a very deep and complicated topic (though perhaps not as deep as the concept represented by the figure of eight on its side!) So the notation is simple, but the payoff is that mathematicians all around the world can clearly and unambiguously communicate their ideas using this, and a small collection

of other symbols. Mathematicians have a *common language*. So do musicians, electronic engineers, and many other disciplines and professions.

To date, Software Engineering has lacked such a notation. Between 1989 and 1994, a period referred to as the "method wars", more than 50 software modelling languages were in common use – each of them carrying their own notations! Each language contained syntax peculiar to itself, whilst at the same time, each language had elements which bore striking similarities to the other languages.

To add to the confusion, no one language was complete, in the sense that very few software practitioners found complete satisfaction from a single language!

In the mid 1990's, three methods emerged as the strongest. These three methods had begun to converge, with each containing elements of the other two. Each method had its own particular strengths:

- **Booch** was excellent for design and implementation. Grady Booch had worked extensively with the Ada language, and had been a major player in the development of Object Oriented techniques for the language. Although the Booch method was strong, the notation was less well received (lots of cloud shapes dominated his models - not very pretty!)
- **OMT** (Object Modelling Technique) was best for analysis and data-intensive information systems.
- **OOSE** (Object Oriented Software Engineering) featured a model known as **Use Cases**. Use Cases are a powerful technique for understanding the behaviour of an entire system (an area where OO has traditionally been weak).

In 1994, Jim Rumbaugh, the creator of OMT, stunned the software world when he left General Electric and joined Grady Booch at Rational Corp. The aim of the partnership was to merge their ideas into a single, unified method (the working title for the method was indeed the "Unified Method").

By 1995, the creator of OOSE, Ivar Jacobson, had also joined Rational, and his ideas (particularly the concept of "Use Cases") were fed into the new Unified Method - now called the Unified Modelling Language[1]. The team of Rumbaugh, Booch and Jacobson are affectionately known as the "Three Amigos".

Despite some initial wars and arguments, the new method began to find favour amongst the software industry, and a UML consortium was formed. Heavyweight corporations were part of the consortium, including Hewlett-Packard, Microsoft and Oracle.

The UML was adopted by the OMG[2] in 1997, and since then the OMG have owned and maintained the language. Therefore, the UML is effectively a public, non-proprietary language.

---

[1] Officially, the spelling is "modeling", but I favour the English spelling

[2] The OMG are the Object Management Group, an industry wide, non profit making standards body. See www.omg.org for full details.

Summary

The UML is a graphical language for capturing the artifacts of software developments.

The language provides us with the notations to produce models.

The UML is gaining adoption as a single, industry wide language.

The UML was originally designed by the Three Amigos at Rational Corp.

The language is very rich, and carries with it many aspects of Software Engineering best practice.

Chapter 2

The UML within a Development Process

The UML as a Notation

The Three Amigos, when developing the UML, made a very clear decision to remove any process based issues from the language. This was because processes are very contentious - what works for company A might be a disaster for company B. A defence company requires much more documentation, quality and testing than (say) an e-commerce company. So the UML is a generic, broad language enabling the key aspects of a software development to be captured on "paper".

In other words, the UML is simply a language, a notation, a syntax, whatever you want to call it. Crucially, it does not tell you *how* to develop software.

To learn how to use the UML effectively, however, we will follow a simple process on this course, and try to understand how the UML helps at each stage. To start with, let's have a look at some common software processes.

The Waterfall Model

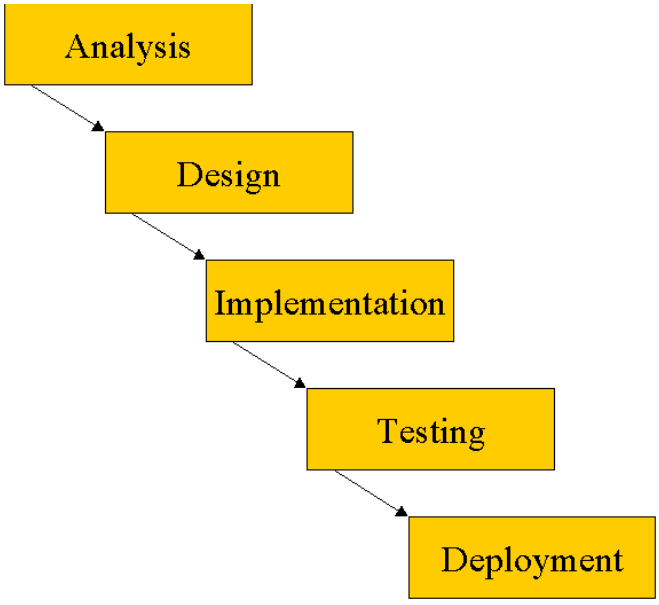


Figure 2 - The traditional "Waterfall" model

The waterfall model prescribes that each stage must be complete before the next stage can commence.

This simplistic (and easy to manage) process begins to break down as the complexity and size of the project increases. The main problems are:

- Even large systems must be fully understood and analysed before progress can be made to the design stage. The complexity increases, and becomes overwhelming for the developers.
- Risk is pushed forward. Major problems often emerge at the latter stages of the process – especially during system integration. Ironically, the cost to rectify errors increase exponentially as time progresses.
- On large projects, each stage will run for extremely long periods. A two-year long testing stage is not necessarily a good recipe for staff retention!

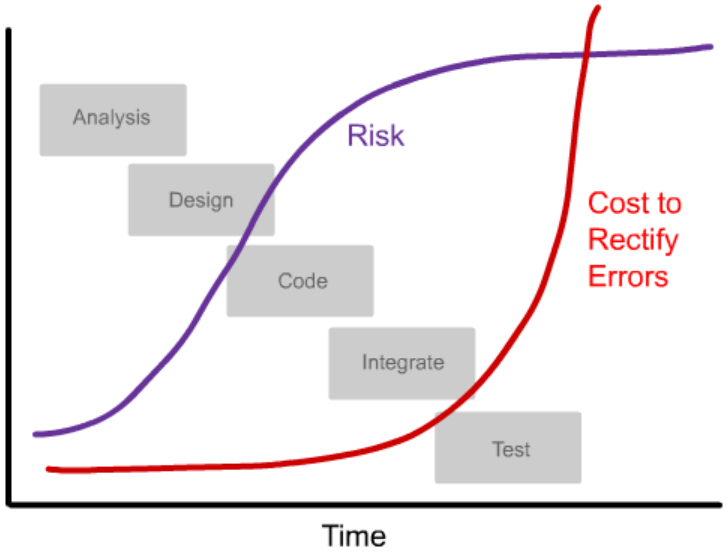


Figure 3 –Over time on the waterfall, both the risks and the cost to rectify errors increase

Also, as the analysis phase is performed in a short burst at the outset of the project, we run a serious risk of failing to understand the customer's requirements. Even if we follow a rigid requirements management procedure and sign off requirements with the customer, the chances are that by the end of Design, Coding, Integration and Testing, the final product will not necessarily be what the customer wanted.

Having said all the above, there is *nothing wrong* with a waterfall model, providing the project is small enough. The definition of "small enough" is subjective, but essentially, if the project can be tackled by a small team of people, with each person able to understand every aspect of the system, and if the lifecycle is short (a few

months), then the waterfall is a valuable process. It is much better than chaotic hacking!

In summary, the waterfall model is easy to understand and simple to manage. But the advantages of the model begin to break down once the complexity of the project increases.

The Spiral Model

An alternative approach is the **spiral model**. In this approach, we attack the project in a series of short lifecycles, each one ending with a release of executable software:

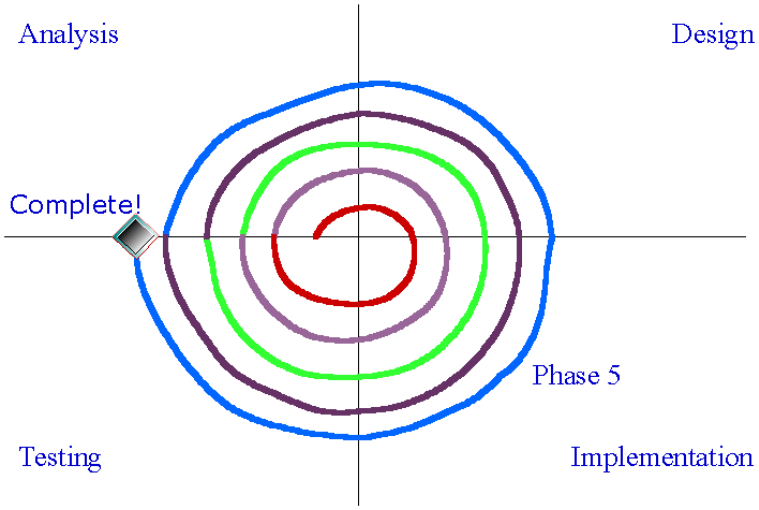


Figure 4 - a spiral process. Here, the project has been divided into five phases, each phase building on the previous one and with a running release of software produced at the end of each phase

With this approach:

- The team are able to work on the entire lifecycle (Analysis, Design, Code, Test) rather than spending years on a single activity
- We can receive early and regular feedback from the customer, and spot potential problems before going too far with development
- We can attack risks up-front. Particularly risky iterations (for example, an iteration requiring the implementation of new and untested technology) can be developed first
- The scale and complexity of work can be discovered earlier
- Changes in technology can be incorporated more easily
- A regular release of software improves morale
- The status of the project (eg – "how much of the system is complete") can be assessed more accurately

The drawbacks of a spiral process are

- The process is commonly associated with Rapid Application Development, which is considered by many to be a hacker's charter.

- The process is much more difficult to manage. The Waterfall Model fits in closely with classic project management techniques such as Gantt charts, but spiral processes require a different approach.

To counteract the drawbacks of the spiral technical, let's look at a similar, but more formal approach called an **Iterative, Incremental Framework**.

Philippe Kruchten's Whitepaper (reference [5], available from Rational Software's website) explores the traps many managers are likely to face on their first iterative development.

Iterative, Incremental Frameworks

The Iterative, Incremental Framework is a logical extension to the spiral model, but is more formal and rigorous. We will be following an Iterative, Incremental Framework through the rest of this course.

The framework is divided into four major phases: **Inception**; **Elaboration**; **Construction** and **Transition**. These phases are performed in sequence, but the phases must not be confused with the stages in the waterfall lifecycle. This section describes the phases and outlines the activities performed during each one.

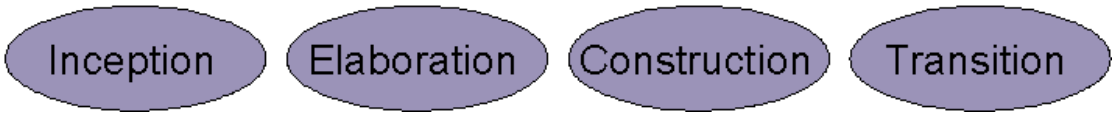


Figure 5 - the four phases of an Iterative, Incremental Framework

Inception

The inception phase is concerned with establishing the scope of the project and generally defining a vision for the project. For a small project, this phase could be a simple chat over coffee and an agreement to proceed; on larger projects, a more thorough inception is necessary. Possible deliverables from this phase are:

- A Vision Document
- An initial exploration of the customer's requirements
- A first-cut project glossary (more on this later)
- A Business Case (including success criteria and a financial forecast, estimates of the Return on Investment, etc)
- An initial risk assessment

- A project plan

We'll explore the inception phase in a little detail when we meet the case study in Chapter 4.

Elaboration

The purpose of elaboration is to analyse the problem, develop the project plan further, and eliminate the riskier areas of the project. By the end of the elaboration phase, we aim to have a general understanding of the entire project, even if it is not necessarily a *deep* understanding (that comes later, and in small, manageable chunks).

Two of the UML models are often invaluable at this stage. The *Use Case Model* helps us to understand the customer's requirements, and we can also use the *Class Diagram* to explore the major concepts our customer understands. More on this shortly.

Construction

At the construction phase, we build the product. This phase of the project is not carried our in a linear fashion – rather, the product is built in the same fashion as the spiral model, by following a series of iterations. Each iteration is our old friend, the simple *waterfall*.[3] By keeping each iteration as short as possible, we aim to avoid the nasty problems associated with waterfalls.

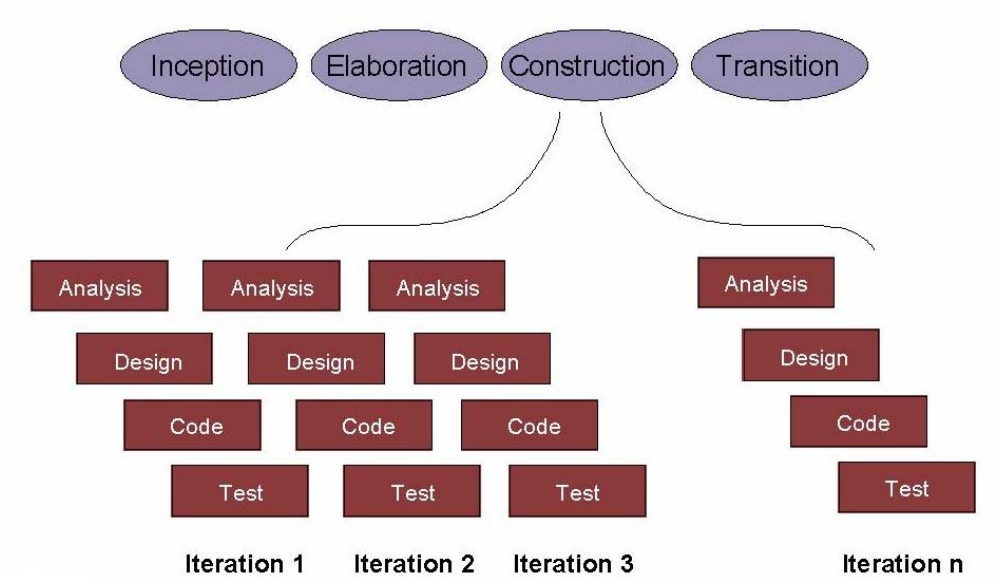


Figure 6 - The Construction Phase consists of a series of "mini waterfalls"

[3] Note that at the inception and elaboration phases, prototypes can be built. These prototypes can be developed in exactly the same way – as a series of mini waterfall iterations. However, for this course, we will keep the inception and elaboration phases simple and use the waterfalls for construction only.

At the end of as many iterations as possible, we will aim to have a running system (albeit, of course, a very limited system in the early stages). These iterations are called *Increments*, hence the name of the framework!

Transition

The final phase is concerned with moving the final product across to the customers. Typical activities in this phase include:

- Beta-releases for testing by the user community
- Factory testing, or running the product in parallel with the legacy system that the product is replacing
- Data takeon (ie converting existing databases across to new formats, importing data, etc)
- Training the new users
- Marketing, Distribution and Sales

The Transition phase should not be confused with the traditional test phase at the end of the waterfall model. At the start of Transition, a full, tested and running product should be available for the users. As listed above, some projects may require a beta-test stage, but the product should be pretty much complete before this phase happens.

How Many Iterations? How Long Should They Be?

A single iteration should typically last between *2 weeks* and *2 months*. Any more than two months leads to an increase in complexity and the inevitable "big bang" integration stage, where many software components have to be integrated for the first time.

A bigger and more complex project should **not** automatically imply the need for longer iterations – this will increase the level of complexity the developers need to handle at any one time. Rather, a bigger project should require **more** iterations.

Some factors that should influence the iteration length include: (see Larman [2], pp447-448).

- Early development cycles may need to be longer. This gives developers a chance to perform exploratory work on untested or new technology, or to define the infrastructure for the project.
- Novice staff
- Parallel developments teams
- Distributed (eg cross site) teams [note that Larman even includes in this category any team where the members are not all located on the same floor, even if they are in the same building!]

To this list, I would also add that a *high ceremony* project will generally need longer iterations. A high ceremony project is one which might have to deliver a lot of project documentation to the customer, or perhaps a project which must meet a lot of legal requirements. A very good example would be any defence related project. In this case, the documentary work will extend the length of the iteration – but the amount of

software development tackled in the iteration should still be kept to a minimum to avoid our chief enemy, complexity overload.

Time Boxing

A radical approach to managing an iterative, incremental process is *Time Boxing*. This is a rigid approach which sets a fixed time period in which a particular iteration must be completed by.

If an iteration is not complete by the end of the timebox, the iteration ends anyway. The crucial activity associated with timeboxing is the review at the end of iteration. The review must explore the reasons for any delays, and must reschedule any unfinished work into future iterations.

Larman (ref [2]) gives details on how to implement timeboxing. One of his recommendations is that the developers be responsible for (or at least, have a large say in) setting which requirements are covered in each iteration, as they are the ones who will have to meet the deadlines.

Implementing timeboxing is difficult. It requires a culture of extreme discipline through the entire project. It is extremely tempting to forgo the review and overstep the timebox if the iteration is "99%" complete when the deadline arrives. Once a project succumbs to temptation and one review is missed, the whole concept begins to fall apart. Many reviews are missed, future iterations planning becomes sloppy and chaos begins to set in.

Some managers assume that timeboxing prevents slippage. It does not. If an iteration is not complete once the timebox has expired, then the unfinished work must be reallocated to later iterations, and the iteration plans are reworked – this could include slipping the delivery date or adding more iterations. However, the benefits of timeboxing are:

- The rigid structure enforces planning and replanning. Plans are not discarded once the project begins to slip
- If timeboxes are enforced, there is less of a tendency for the project to descend into chaos once problems emerge, as there is always a formal timebox review not too far away
- If panic sets in and developers start to furiously hack, the hacking is stemmed once the review is held

Essentially, timeboxing allows the entire project to regularly "stand back" and take stock. It does not prevent slippage, and requires strong project management to work.

Typical Project Timings

How long should each of the four phases last? This is entirely up to individual projects, but a loose guideline is 10% inception, 30% elaboration, 50% construction and 10% transition.

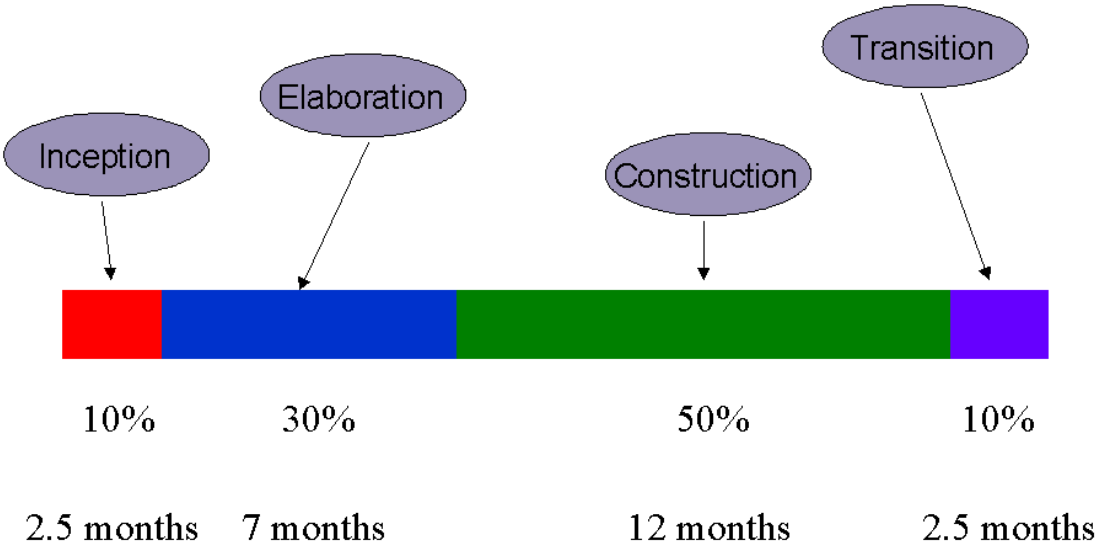


Figure 7 - Possible timings for each phase. This example shows the length of each phase for a two year project.

The Rational Unified Process

The Rational Unified Process (the RUP) is the most famous example of an Iterative, Incremental Lifecycle in use at the moment. The RUP was developed by the same "Three Amigos" that developed the UML, so the RUP is very complementary to the UML.

Essentially, Rational appreciate that every project is different, with different needs. For example, for some projects, a tiny Inception Phase is appropriate, whereas for defence projects, the Inception phase could last years.

To this end, the RUP is tailorable, and enables each phase of the process to be customised. The RUP also defines the roles of everyone on the project very carefully (in the shape of so-called *Workers* - again, these are tailorable to the project's needs).

Rational Corp produce a product to help projects work with the RUP. Full details can be found at www.rational.com. Essentially, the RUP project is an on-line, hypertext guide to every aspect of the RUP. Rational provide 30 day trials of the product.

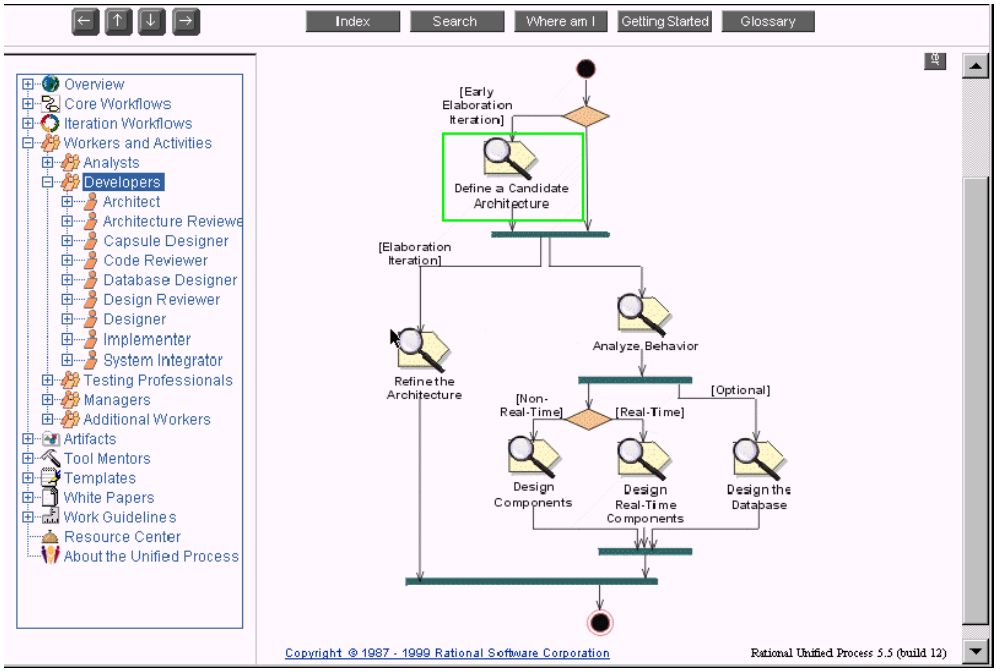


Figure 8 - Screenshot from RUP 2000 (© Rational Corp)

The precise advantages and disadvantages of the RUP are beyond the scope of this course. However, the core of the RUP, the Iterative, Incremental Lifecycle will be followed throughout this course to illustrate the key aspects of the UML models.

For more details on the RUP, Philippe Kruchten's book The Rational Unified Process– An Introduction (ref 1) covers the subject in detail.

Summary

An Iterative, Incremental Framework offers many benefits over traditional processes.

The Framework is divided into four phases - Inception, Elaboration, Construction, Transition.

Incremental development means to aim for running code at the end of as many iterations as possible.

Iterations can be timeboxed - a radical way of scheduling and reviewing iterations.

The rest of this course will focus on the Framework, and how the UML supports the deliverables of each phase in the Framework.

Chapter 3

Object Orientation

In this chapter we will look at the concept of Object Orientation[4] (OO). The Unified Modelling Language has been designed to support Object Orientation, and we'll be introducing Object Oriented concepts throughout this course. Before we begin looking at the UML in depth, however, it is worth introducing OO and looking at the advantages that OO can offer to Software Development.

Structured Programming

First of all, let's examine (in very rough terms) how software systems are designed using the Structured (sometimes called Functional) approach.

In Structured Programming, the general method was to look at the problem, and then design a collection of ***functions*** that can carry out the required tasks. If these functions are too large, then the functions are broken down until they are small enough to handle and understand. This is a process known as **functional decomposition**.

Most functions will require data of some kind to work on. The data in a functional system was usually held in some kind of database (or possibly held in memory as global variables).

As a simple example, consider a college management system. This system holds the details of every student and tutor in the college. In addition, the system also stores information about the courses available at the college, and tracks which student is following which courses.

A possible functional design would be to write the following functions:

```
add_student[5]
enter_for_exam
check_exam_marks
issue_certificate
expel_student
```

[4] I'll use the phrase "Object Orientation" to denote Object Oriented Design and/or Object Oriented Programming

[5] I'm using underscores to highlight the fact that these functions are written in code.

We would also need a data model to support these functions. We need to hold information about Students, Tutors, Exams and Courses, so we would design a database schema to hold this data.[6]

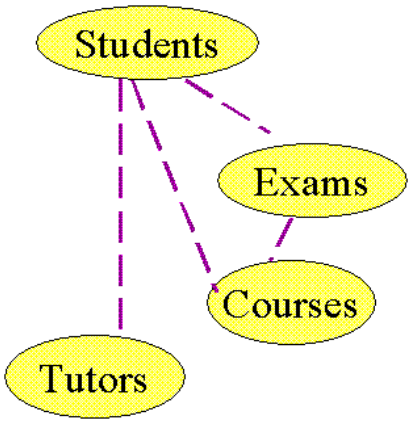


Figure 9 - Simple Database Schema. The dotted lines indicate where one set of data is dependent on another. For example, each student is taught by several tutors.

Now, the functions we defined earlier are clearly going to be dependent on this set of data. For example, the "add_student" function will need to modify the contents of "Students". The "issue_certificate" function will need to access the Student data (to get details of the student requiring the certificate), and the function will also need to access the Exam data.

[6] Note that throughout this chapter, I am not using a formal notation to describe the concepts

The following diagram is a sketch of all the functions, together with the data, and lines have been drawn where a dependency exists:

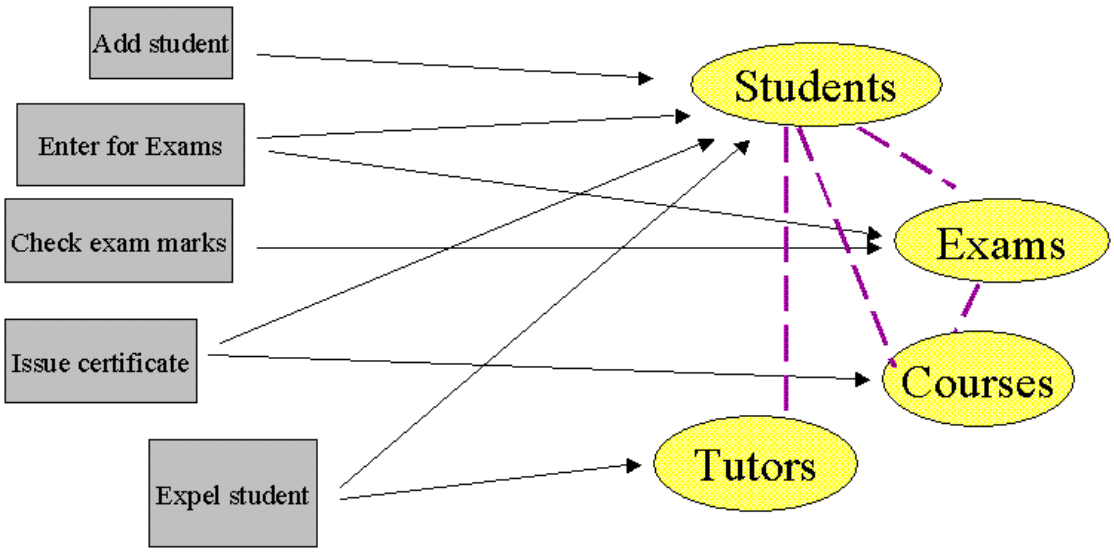


Figure 10 - Plan of the functions, the data, and the dependencies

The problem with this approach is that if the problem we are tackling becomes too complex, the system becomes harder and harder to maintain. Taking the example above, what would happen if a requirement changes that leads to an alteration in the way in which Student data is handled?

As an example, imagine our system is running perfectly well, but we realise that storing the Student's date of birth with a two digit year was a bad idea. The obvious solution is to change the "Date of Birth" field in the Student table, from a two-digit year to a four-digit year.

The serious problem with this change is that we might have caused unexpected side effects to occur. The Exam data, the Course data and the Tutors data all depend (in some way) on the Student data, so we might have broken some functionality with our simple change. In addition, we might well have broken the add_student, enter_for_exams, issue_certificate and expel_student functions. For example, add_student will certainly not work anymore, as it will be expecting a two digit year for "date of birth" rather than four.

So we have a large degree of potential knock-on problems. What is far, far worse is that in our program code, we cannot easily see what these dependencies actually are.

How many times have you changed a line of code in all innocence, without realising that you've inadvertently broken apparently unrelated functionality?

The costly Year 2000 problem (The Millennium Bug) was caused by exactly this problem. Even though the fix should be simple (make every year occupy four digits

instead of two), the potential impacts of these minor changes had to be investigated in detail.[7]

The Object Orientated Approach

OO tries to lessen the impact of this problem by simply combining related data and functions into the same module.

Looking at Figure 10 above, it is apparent that the data and functions are related. For example, the add_student and expel_student functions are clearly very closely related to the Student data.

The following figure shows the full grouping of the related data and functions, in the form of modules:

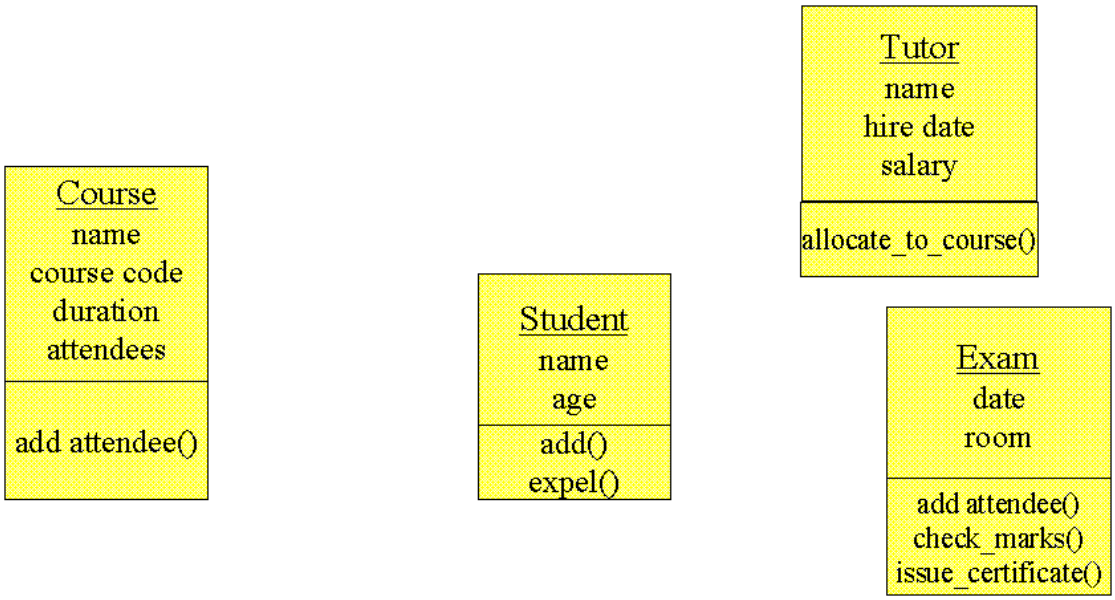


Figure 11 - Related data and functions placed in modules

A couple of points to note about this new modular system of programming:

- More than one instance of a single module can exist when the program is running. In the college system, there would be an instance of "Student" for every student that belongs to the college. Each instance would have its own values for the data (certainly each would have a different name).

- Modules can "talk" to other modules by calling each other's functions. For example, when the "add" function is called in Student, a new instance of the Student module would be created, and then the "add_attendee" function would be called from the appropriate instances of the "Course" module.

---

[7] This doesn't mean that I am implying that all non-OO Cobol systems are a load of rubbish, by the way. There is *nothing wrong* with structured programming. My suggestion in this chapter is that OO provides a method of building more robust software as our systems get larger and more complex.

Encapsulation

Crucially, only the instance that owns an item of data is allowed to modify or read it. So for example, an instance of the Tutor module cannot update or read the "age" data inside the Student module.

This concept is called **Encapsulation**, and enables the structure of the system to be far more robust, and avoid the situation as described previously, where a small change to a data member can lead to rippling changes.

With Encapsulation, the programmer of (say) the Student module can safely make changes to the data in the module, and rest assured that no other modules are dependent upon that data. The programmer might well need to update the functions inside the module, but at least the impact is isolated to the single module.

Objects

Throughout this chapter, I have referred to these collections of related data and functions as being "modules". However, if we look at the characteristics of these modules, we can see some real world parallels.

Objects in the real world can be characterised by two things: each real world object has **data** and **behaviour**. For example, a television is an object and posses data in the sense that it is tuned to a particular channel, the scan rate is set to a certain value, the contrast and brightness is a particular value and so on. The television object can also "do" things. The television can switch on and off, the channel can be changed, and so on.

We can represent this information in the same way as our previous software "modules":

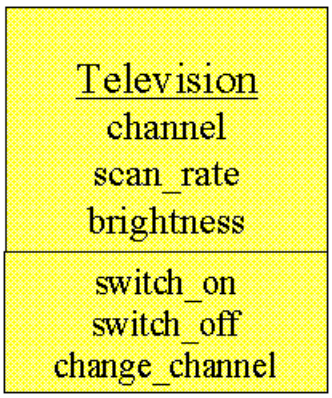


Figure 12 - The data and behaviour of a television

In some sense, then, real world "objects" can be modelled in a similar way to the software modules we discussed earlier.

For this reason, we call the modules **Objects**, and hence we have the term **Object Oriented Design/Programming**.

Since our software systems are solving real world problems (whether you are working on a College booking system, a Warehouse Management System or a Weapons Guidance System), we can identify the objects that exist in the real world problem, and easily convert them into software objects.

In other words, *Object Orientation is a better abstraction of the Real World*. In theory, this means that if the problem changes (ie the requirements change, as they always do), the solution should be easier to modify, as the mapping between the problem and solution is easier.

Terminology

The data for an object are generally called the **Attributes** of the object. The different behaviours of an object are called the **Methods** of the object. Methods are directly analogous to functions or procedures in programming languages.

The other big jargon term is **Class**. A class is simply a template for an object. A class describes what attributes and methods will exist for all instances of the class. In the college system we described in this chapter, we had a class called **Student**.

The attributes of the **Student Class** were name, age, etc. The methods were add() and expel(). In our code, we would only need to define this class once. Once the code is running, we can create instances of the class - ie, we can create objects of the class.

Each of these objects will represent a student, and each will have its own set of values of data.

The Object Oriented Strategy

Although this chapter has briefly touched on the benefits of Object Orientation (ie more robust systems, a better abstraction of the real world), we have left many questions unanswered. How do we identify the objects we need when we're designing a system? What should the methods and attributes be? How big should a class be? I could go on! This course will take you through a software development using Object Orientation (and the UML), and will answer all these questions in full.

One significant weakness of Object Orientation in the past has been that while OO is strong at working at the class/object level, OO is poor at expressing the behaviour of an entire system. Looking at classes is all very well, but classes are very "low-level" entities and don't really describe what the system as a **whole** can do. Using classes alone would be rather like trying to understand how a computer works by examining the transistors on a motherboard!

The modern approach, strongly supported by the UML is to **forget** all about objects and classes at the early stages of a project, and instead concentrate on what the system must be able to do. Then, as the project progresses, classes are **gradually** built to realise the required system functionality. Through this course, we will follow these steps from the initial analysis, all the way through to class design.

Summary

- Object Orientation is a slightly different way of thinking from the structured approach
- We combine related data and behaviour into classes
- Our program then creates instances of the class, in the form of an object
- Objects can collaborate with each other, by calling each other's methods
- The data in an object is encapsulated - only the object itself can modify the data

Chapter 4
An Overview of the UML

Before we begin to look at the theory of the UML, we are going to take a very brief run through some of the major concepts of the UML.

The first thing to notice about the UML is that there are a lot of different diagrams (models) to get used to. The reason for this is that it is possible to look at a system from many different viewpoints. A software development will have many stakeholders playing a part – for example:

- Analysts
- Designers
- Coders
- Testers
- QA
- The Customer
- Technical Authors

All of these people are interested in different aspects of the system, and each of them require a different level of detail. For example, a coder needs to understand the design of the system and be able to convert the design to a low level code. By contrast, a technical writer is interested in the behaviour of the system as a whole, and needs to understand how the product functions. The UML attempts to provide a language so expressive that all stakeholders can benefit from at least one UML diagram.

Here's a quick look at some of the most important diagrams. Of course, we will look in detail at each of them as the course progresses:

The Use Case Diagram

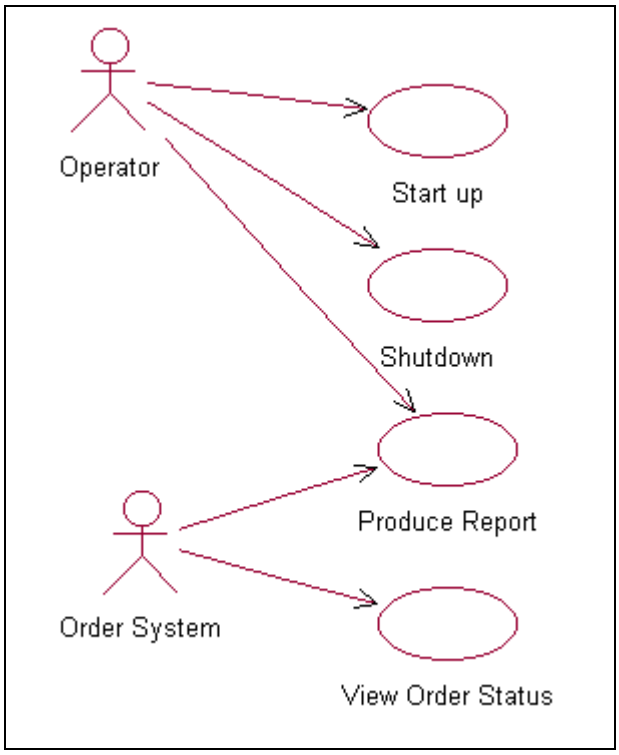


Figure 13 - The Use Case Diagram

A Use Case is a description of the system's behaviour from a user's viewpoint. This diagram is a valuable aid during analysis – developing Use Cases helps us to understand requirements.

The diagram is deliberately simple to understand. This enables both developers (analysts, designers, coders, tests) *and* the customer to work with the diagram.

However, do not be tempted to overlook Use Cases as being "too simple to bother with". We shall see that Use Cases can drive an entire development process, from inception through to delivery.

The Class Diagram

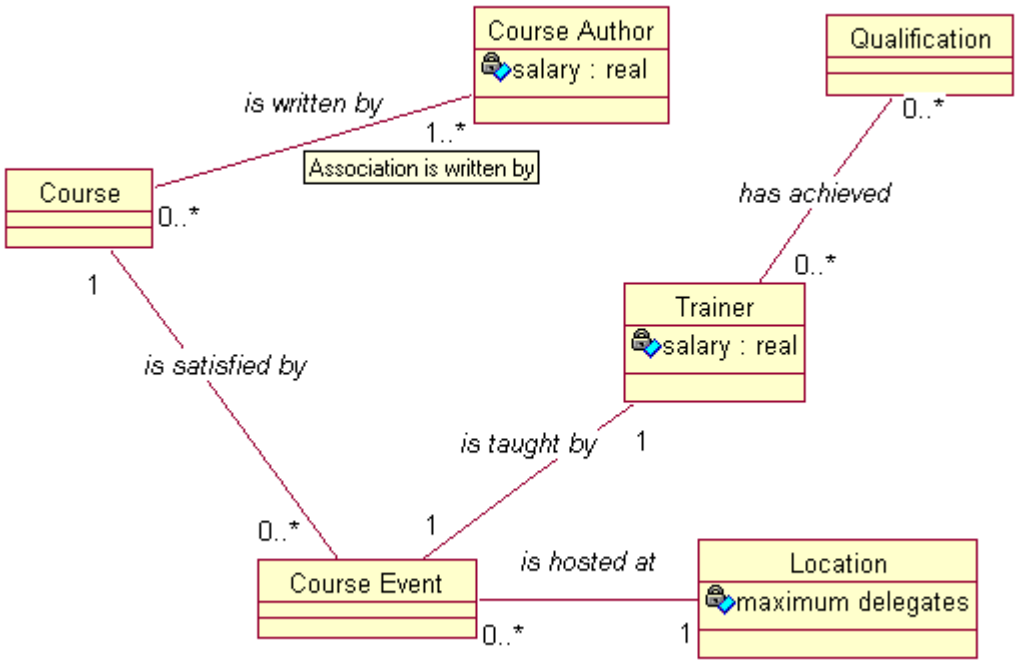


Figure 14 - The UML Class Diagram

Drawing Class Diagrams is an essential aspect of any Object Oriented Design method, so it isn't surprising that the UML provides us with the appropriate syntax. We'll see that we can use the Class Diagram at the analysis stage as well as design – we'll use the Class Diagram syntax to draw a plan of the major concepts our customer understands (and we'll call this the *Conceptual Model*). Together with Use Cases, a Conceptual Model is a powerful technique in requirements analysis

Collaboration Diagrams

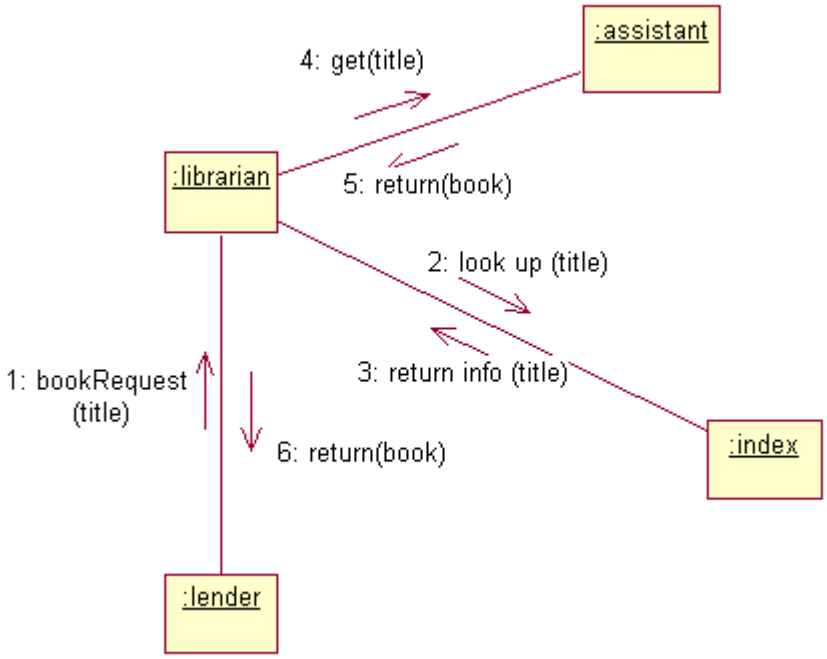


Figure 15 - The UML Collaboration Diagram

As we are developing object-oriented software, anything our software needs to do is going to be achieved by objects **collaborating**. We can draw a **collaboration diagram** to describe how we want the objects we build to collaborate.

Here is a good example of why the UML is "just" a syntax rather a true software development process. We will see that the UML notation for the diagram is simple enough, but designing effective collaborations, (that is to say "designing software which is easy to maintain and robust"), is very difficult indeed. We shall be devoting an entire chapter for guidelines on good design principles, but much of the skill in design comes from experience.

Sequence Diagram

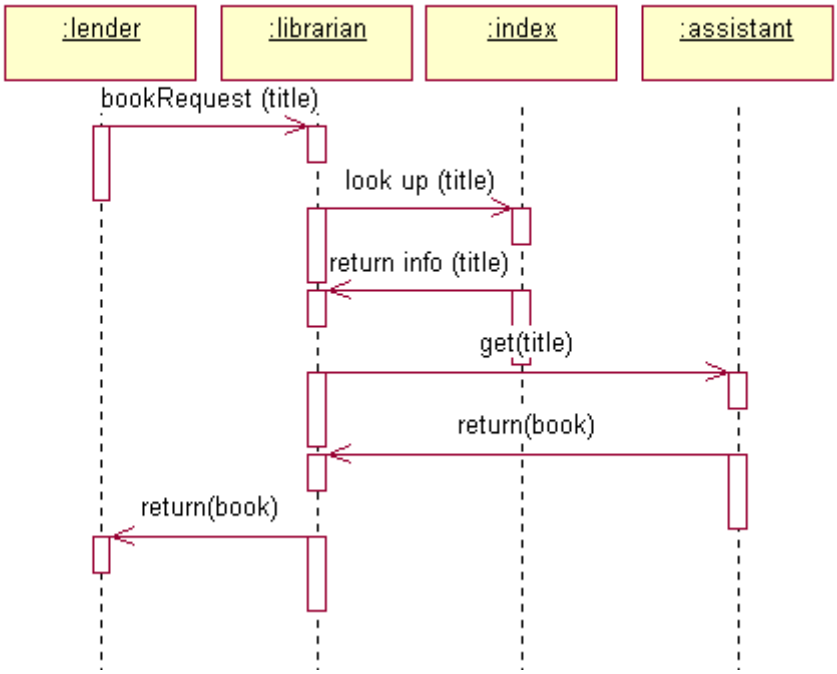


Figure 16 - A UML Sequence Diagram

The sequence diagram is, in fact, directly related to the collaboration diagram and displays the same information, but in a slightly different form. The dotted lines down the diagram indicate *time*, so what we can see here is a description of how the objects in our system interact over time.

Some of the UML modelling tools, such as Rational Rose, can generate the sequence diagram automatically from the collaboration diagram, and in fact that is exactly how the diagram above was drawn – directly from the diagram in Figure 15.

State Diagrams

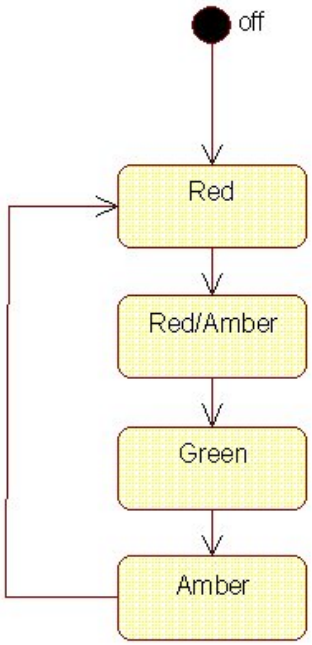


Figure 17 - A State Transition Diagram

Some objects can, at any particular time, be in a certain *state*. For example, a traffic light can be in any one of the following states:

Off, Red, Amber, Green

Sometimes, the sequence of transitions between states can be quite complex – in the above example, we would not want to be able to go from the "Green" state to the "Red" state (we'd cause accidents!).

Although the traffic light may seem like a trivial example, being sloppy with states can cause serious and embarrassing faults to occur in our software.

Take as a case in point – a gas bill is sent out to a customer who died four years ago – it really happens and it is because a programmer somewhere has not taken care with their state transitions.

As state transitions can be quite complex, the UML provides a syntax to allow us model them.

Package Diagrams

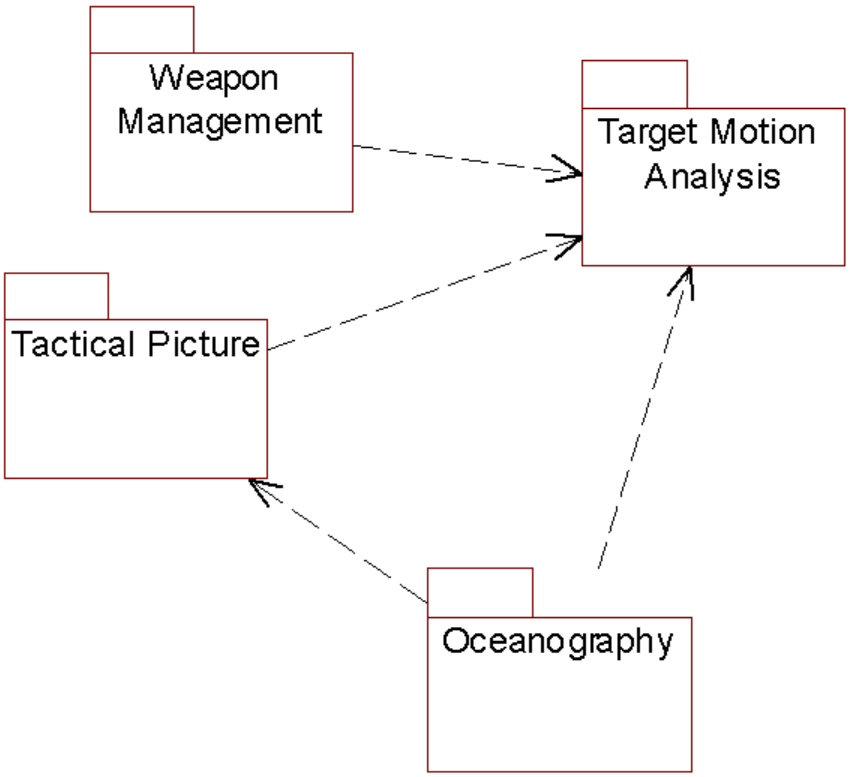


Figure 18 - The UML Package Diagram

Any non-trivial system needs to be divided up in smaller, easier to understand "chunks", and the UML Package Diagram enables us to model this in a simple and effective way. We'll be looking in detail at this model when we explore large systems in the "System Architecture" chapter.

Component Diagrams

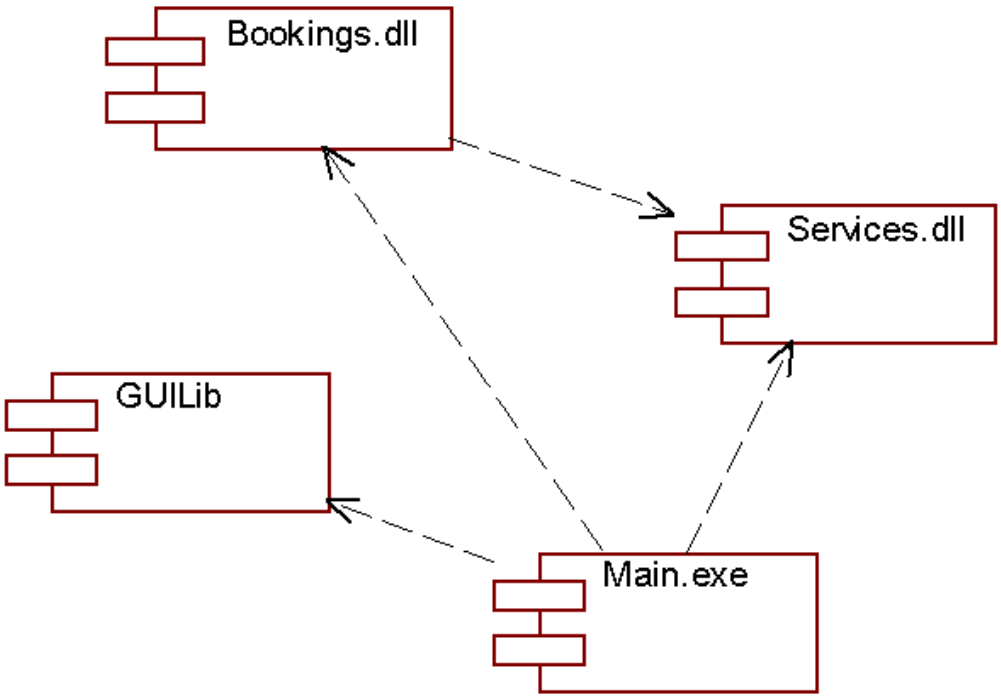


Figure 19 - The UML Component Diagram

The Component Diagram is similar to the package diagram - it allows us to notate how our system is split up, and what the dependencies between each module is. However, the Component Diagram emphasises the physical software components (files, headers, link libraries, executables, packages) rather than the logical partitioning of the Package Diagram. Again, we'll look at this diagram in more detail in the System Architecture chapter.

Deployment Diagrams

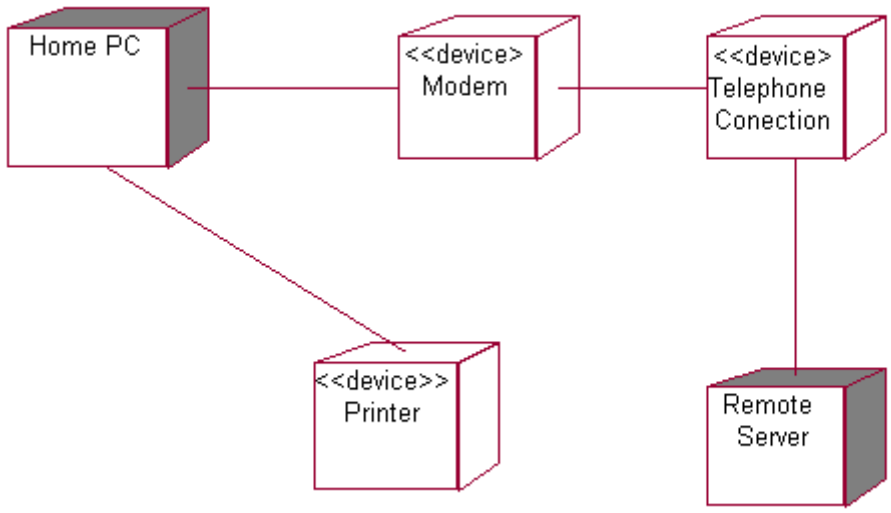


Figure 20 - A UML Deployment Diagram

The UML provides a model to allow us to plan how our software is going to be deployed. The diagram above shows a simple PC configuration, for example.

Summary

The UML provides many different models of a system. The following is a list of them, with a one sentence summary of the purpose of the model:

- **Use Cases** - "How will our system interact with the outside world?"
- **Class Diagram** - "What objects do we need? How will they be related?"
- **Collaboration Diagram** - "How will the objects interact?"
- **Sequence Diagram** - "How will the objects interact?"
- **State Diagram** - "What states should our objects be in?"
- **Package Diagram** - "How are we going to modularise our development?"
- **Component Diagram** - "How will our software components be related?"
- **Deployment Diagram** - "How will the software be deployed?"

Chapter 5
The Inception Phase

For the rest of this course, we are going to concentrate on a case study to describe how the UML is applied on real projects. We shall use the process outlined in Chapter 1, as in the diagram below:

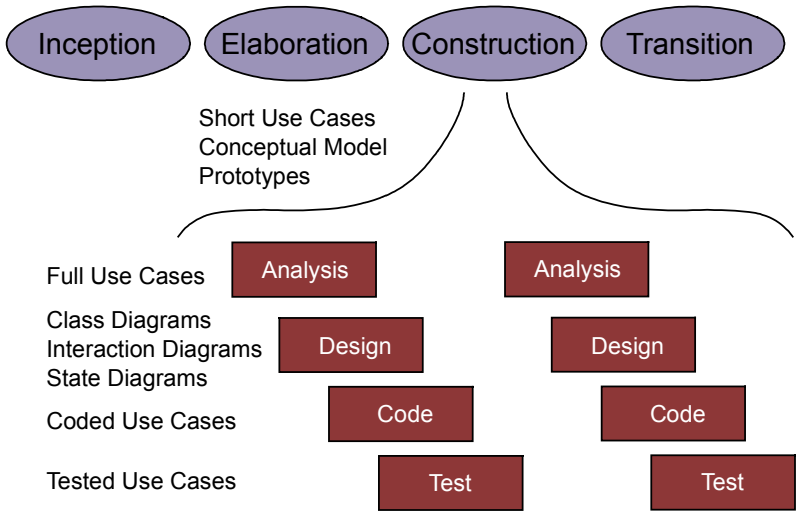


Figure 21 - The process for our case study

In the diagram, I have included the name of each model we will produce at each stage. For example, at the design stage we will produce Class Diagrams, Interaction Diagrams and State Diagrams. Of course, we'll explore these diagrams throughout this course.

To recap the Inception Phase, the key activities in the phase are:

- Specify the vision for the product
- Produce a business case
- Define the scope of the project
- Estimate the overall cost of the project

The size of the phase depends upon the project. An ecommerce project may well need to hit the market as quickly as possible, and the only activities in Inception might be to define the vision and get finance from a bank via the business plan.

By contrast, a defence project could well require requirements analysis, project definition, previous studies, invites to tender, etc, etc. It all depends on the project.

On this course, we assume the inception phase is already complete. A business study has been produced (see separate document) that details our customer's initial requirements and a description of their business model.

Chapter 6
The Elaboration Phase

In the Elaboration Phase, we are concerned with exploring the problem in detail, understanding the customer's requirements and their business, and to develop the plan further.

We must get in to the correct frame of mind to attack this phase correctly. We must try not to get bogged down with too much detail – especially implementation details.

We need to have a very broad view of the system and understand system-wide issues. Kruchten (ref [1]) calls this a *mile wide and inch deep view*.

Prototyping

A key activity in the Elaboration Phase is the mitigation of risks. The sooner risks are identified and shot down, the lesser their impact will be on the project.

Prototyping difficult or problematic areas of the project are a tremendous help in the mitigation of risks. Given that we don't want to get bogged down in implementation and design at this phase, the prototypes should be very focussed, and explore just the area of concern.

Prototypes can be thrown away at the end of the exercise, or they can be reused during the construction phase.

Deliverables

Apart from prototypes, we are going to develop two UML models to help us towards our goal of understanding the problem as a whole.

The first model is the **Use Case Model**. This will help us to understand what the system needs to do, and what it should look like to the "outside world" (ie the users, or perhaps the systems it must interface to).

The second model is the **Conceptual Model**. This model allows us to capture, using UML, a graphical statement of the customer's problem. It will describe all of the major "concepts" in the customer's problem, and how they are related. To build this, we'll use the UML Class Diagram. We will use this Conceptual Model in the Construction Phase to build our software classes and objects.

We'll cover these two models, in depth, in the next two chapters.

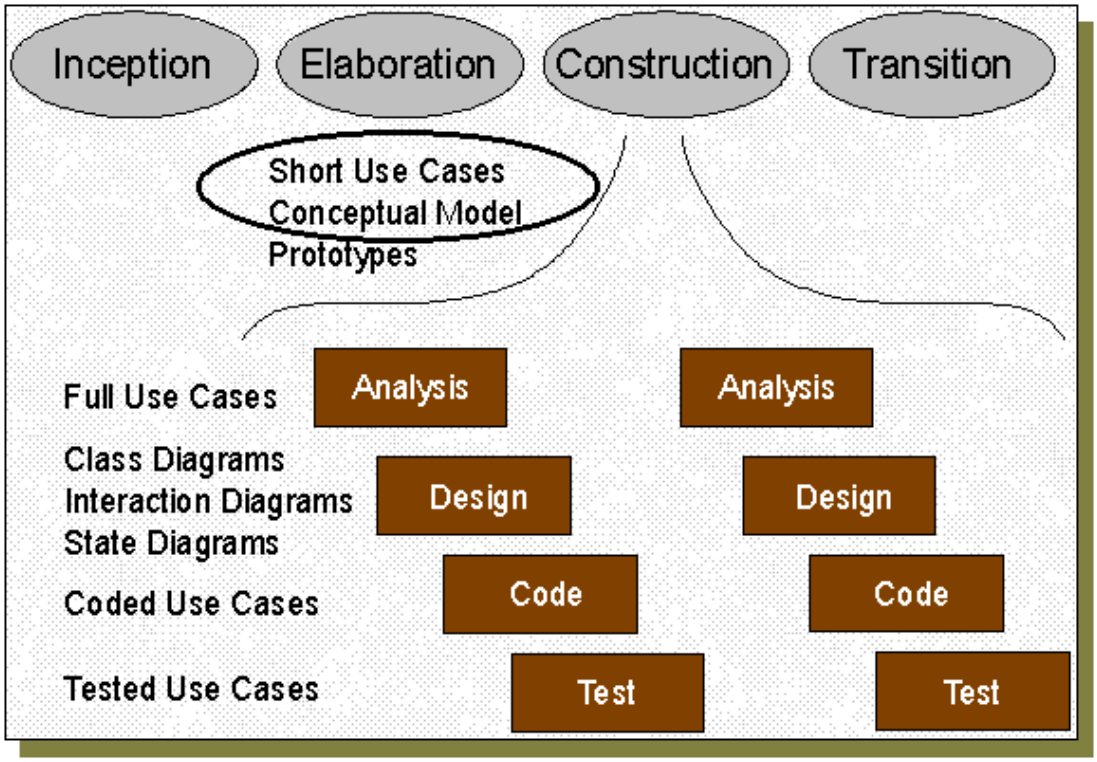


Figure 22 - Two UML models built during Elaboration

Summary

The Elaboration Phase is concerned with developing an understanding of the problem, without worrying about deep design details (except where risks are identified and prototypes are required).

Two models will help us with this phase: The Use Case Model and the Conceptual Model.

Chapter 7
Use Case Modelling

A very powerful UML tool is the Use Case. A Use Case is simply a description of a set of interactions between a user and the system. By building up a collection of Use Cases, we can describe the entire system we are planning to create, in a very clear and concise manner.

Use cases are usually described using verb/noun combinations – for example, "Pay Bills", "Update Payroll", or "Create Account".

For example, if we were writing a missile control system, typical Use Cases for the system might be "Fire Missiles", or "Issue Countermeasures".

Along with the name of the use case, we will provide a full textual description of the interactions that will occur between the user and the system. These textual descriptions will generally become quite complicated, but the UML provides an astoundingly simple notation to represent a Use Case, as follows:

Figure 23 - Use Case Notation

Actors

A Use Case cannot initiate actions on its own. An **actor** is someone who **can** initiate a Use Case. For example, if we were developing a banking system, and we have a Use Case called "withdraw money", then we would identify that we require **customers** to be able to withdraw money, and hence a **customer** would become one of our actors. Again, the notation for an actor is simple:

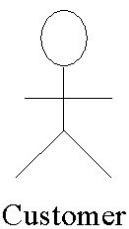


Figure 24 - UML Notation for an Actor

Going further, actors can be more than just people. An actor can be anything external to the system that initiates a Use Case, such as another computer system. An actor could also possibly be a more abstract concept such as **time**, or a specific **date**. For example, we may have a use case called "Purge Old Orders" in an order processing system, and the initiating actor could be "Last Working Day".

As we have noted, actors are related to Use Cases, in the sense that it is an actor that will initiate a particular use case. We can represent this on a Use Case diagram by connecting the actor to the use case:

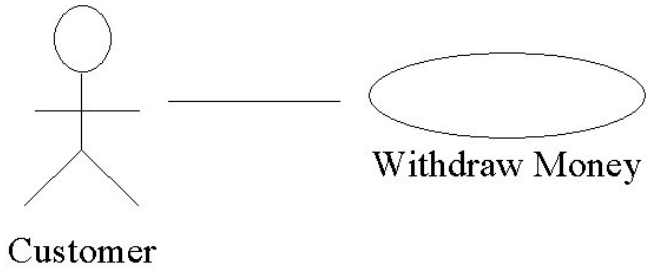


Figure 25 - an Actor's relationship to a Use Case

Clearly, for most systems, a single actor can interact with many use cases, and a single use case can be initiated by many different actors. This leads to the full use case diagram, an example of which follows:

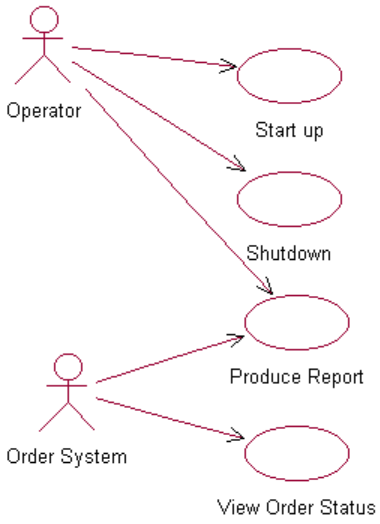


Figure 26 - A complete system described using actors and use cases

The Purpose of Use Cases

Given the simple definition of "Use Case" and "Actor", together with the simple visualisation of Use Cases through the UML model, we could be forgiven for thinking

that Use Cases are simple – almost too simple to worry about. Wrong. Use Cases are immensely powerful.

- Use Cases define the scope of the System. They enable us to visualise size and scope of the entire development.
- Use Cases are very similar to requirements, but whilst requirements tend to be vague, confusing, ambiguous and poorly written, the tighter structure of Use Cases tend to make them far more focused
- The "sum" of the use cases is the whole system. That means that anything not covered by a use case is outside the boundary of the system we are developing. So the Use Case diagram is complete, with no holes.
- They allow for communication between the customer and developers (since the diagram is so simple, anyone can understand it)
- Use Cases guide the development teams through the development process – we shall see that Use Cases are the backbone of our development, and we refer to them in everything we do
- We'll see that Use Cases provide a method for planning our development work, and allow us to estimate how long the development will take
- Use Cases provide the basis for creating system tests
- Finally, Use Cases help with the creation of user guides!

It is often claimed that Use Cases are simply an expression of the system requirements. Anyone making this claim are clearly missing the point of Use Cases![8]

Use Case Granularity

It can be difficult to decide upon the granularity of use cases – in a particular scenario, should each user-system interaction be a use case, or should the use case encapsulate all of the interactions? For example, let us consider the example of the ATM machine. We need to build the ATM system to allow a user to withdraw money. We might have the following series of common interactions in this scenario:

- enter card
- enter pin number
- select amount required
- confirm amount required
- remove card
- take receipt

Should each of these steps – for example, "enter pin number" be a use case?

[8] Though, of course, Use Cases are closely related to requirements. See reference [9] for an excellent treatment on requirements through Use Cases

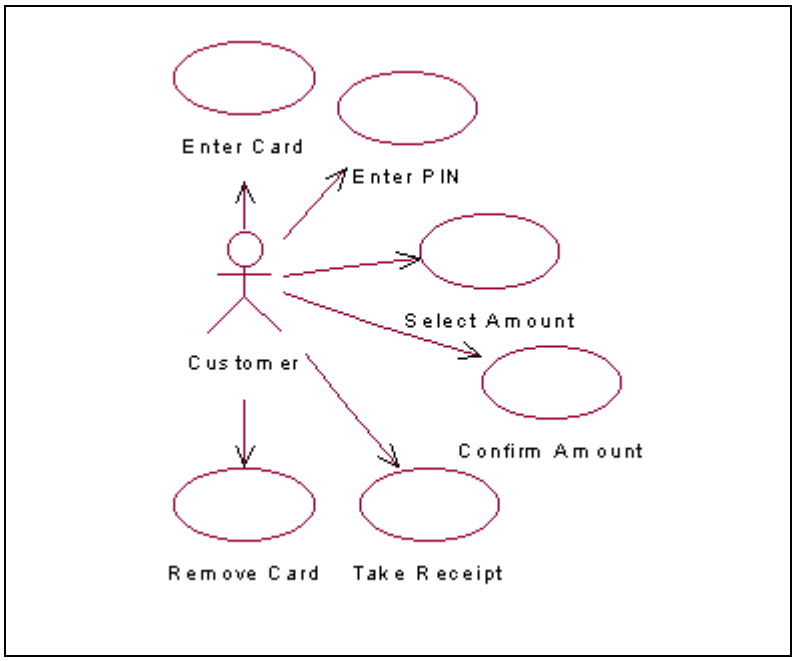


Figure 27 - A Useful Use Case Diagram?

This is a classic mistake in the construction of Use Cases. Here, we have generated a large number of small, almost inconsequential use cases. In any non-trivial system, we would end up with a huge number of Use Cases, and the complexity would become overwhelming.

To handle the complexity of even very large systems, we need to keep the Use Cases at a fairly "high level". The best way to approach a Use Case is to keep the following rule-of-thumb in mind:

A Use Case should satisfy a goal for the actor

Applying this simple rule to our example above, we can ask the question "Is **take receipt**", for example, the goal for our customer? Well, not really. It wouldn't be the end of the world if the receipt wasn't dispensed.

Apply the rule to the other Use Cases, and you'll find that really, none of them describe the goal of the user. The goal of the user is to **withdraw money**, and that should be the use case!

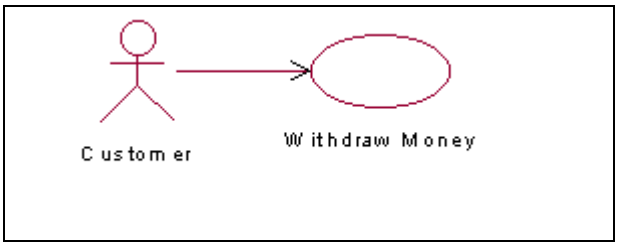


Figure 28 - A more focused Use Case

This approach can feel painful at first, as we are used to performing "functional decomposition", where complex tasks are broken down into smaller and smaller tasks.

We will see later that Use Cases can be decomposed, but we must leave that step until we begin construction.

Use Case Descriptions

Each Use Case contains a full set of textual details about the interactions and scenarios contained within it.

The UML does not specify what the structure and contents of this document should be – this is up to individual projects/companies to specify[9]. We shall use the following template:

Use Case:	Use Case Name
Short Description:	A Brief Description of the Use Case
Pre-Conditions:	A description of the conditions that must be satisfied before the use case is invoked
Post-Conditions :	A description of what has happened at the end of the use case
Main Flow:	A list of the system interactions that take place under the most common scenario. For example, for "withdraw money", this would be "enter card, enter pin, etc..."
Alternate Flow(s):	A description of possible alternative interactions.
Exception Flow(s):	A description of possible scenarios where unexpected or unpredicted events have taken place

Figure 29 - Template for a Use Case Description

Use Cases at the Elaboration Phase

Our main job at the elaboration phase is to identify as many of the potential Use Cases as possible. Bearing in mind the "mile wide and inch deep" principle, our aim is to provide sketchy details of as many Use Cases as possible – but without the need to provide the full detail of each Use Case. This will help us to avoid complexity overload.

At this stage, a Use Case diagram (with actors and Use Cases), plus a brief description of each Use Case, will suffice. We can revisit the full details of the Use Cases during the construction phase. Once we have identified the use cases, we can cross reference Use Cases to requirements and ensure that we have caught all of the requirements.

If we identify some very risky Use Cases at this phase, however, it will be necessary to explore the details of the risky Use Cases. The production of prototypes at this stage will help mitigate the risks.

[9] An excellent example of the UML providing the syntax, but deliberately not specifying how to use the syntax

Finding Use Cases

One approach to finding Use Cases is via interviews with the potential users of the system. This is a difficult task, given that two people are likely to give two completely different views on what the system should do (even if they work for the same company)!

Certainly, most developments will involve some degree of direct one-to-one user communication. However, given the difficulty of gaining a consistent view of what the system will need to do, another approach is becoming more popular – the *workshop*.

Joint Requirements Planning Workshops (JRP)

The workshop approach pulls together a group of people interested in the system being developed (the *stakeholders*). Everyone in the group is invited to give their view of what the system needs to do.

Key to the success of these workshops is the *facilitator*. They lead the group by ensuring that the discussion sticks to the point, and that all the stakeholders are encouraged to put their views across, and that those views are captured. Good facilitators are priceless!

A scribe will also be present, who will ensure that everything is documented. The scribe might work from paper, but a better method is to connect a CASE tool or drawing tool to a projector and capture the diagrams "live".

The simplicity of the use case diagram is critical here – all stakeholders, even non-computer literate stakeholders, should be able to grasp the concept of the diagram with ease.

A simple method of attacking the workshop is:

1) Brainstorm all of the possible actors first

2) Next, brainstorm all of the possible Use Cases

3) Once brainstorming is complete, as a group, justify each Use Case through by producing a simple, one line/paragraph description

4) Capture them on the model

Steps 1) and 2) can be reversed if desired.

Some good advice on the workshop:

- *don't work too hard trying to find every single Use Case and Actor!* It is natural that some more use cases will emerge later on in the process.
- *If you can't justify the Use Case at step 3), it might not be a use case*. Feel free to remove any Use Cases you feel are incorrect or redundant (they'll come back later if they're needed!)

The above advice is not a license to be sloppy, but remember the benefit of iterative processes is that everything doesn't have to be 100% correct at every step!

Brainstorming Advice

Brainstorming is not as easy as it sounds; in fact I have rarely seen a well-executed brainstorm. The key things to remember when participating in a brainstorming session are:

- Document ALL ideas, no matter how outrageous or nonsensical they seem. Stupid ideas *might* turn out to be very sensible ideas after all
- Also, silly ideas may trigger off more sensible ideas in other people's mind – this is called *leapfrogging ideas*
- Never evaluate or criticise ideas. This is a very hard rule to observe – we are trying to break human nature here!

  "mmm. No, that won't work. We won't bother to document that!"

The facilitator should keep on their toes and ensure that all ideas are captured, and that all of the group participate.

On the course, a Use Case Workshop will be carried out alongside our client.

Summary

Use Cases are a powerful way of modelling what the system needs to do.

They are an excellent way of expressing the system's scope (What's in = Sum of the Use Cases; what's out = The Actors).

We need to keep an eye on the granularity of the Use Cases to contain complexity.

The best way of building Use Cases is with the customer, in a workshop.

Chapter 8
Conceptual Modelling

Conceptual Modelling (sometimes called *Domain Modelling*) is the activity of finding out which concepts are important to our system. This process helps us to understand the problem further, and develop a better awareness of our customer's business.

Once again, the UML does not tell us how or when to do Domain Modelling, but it does provide us with the syntax to express the model. The model we are going to use is the **class diagram**.

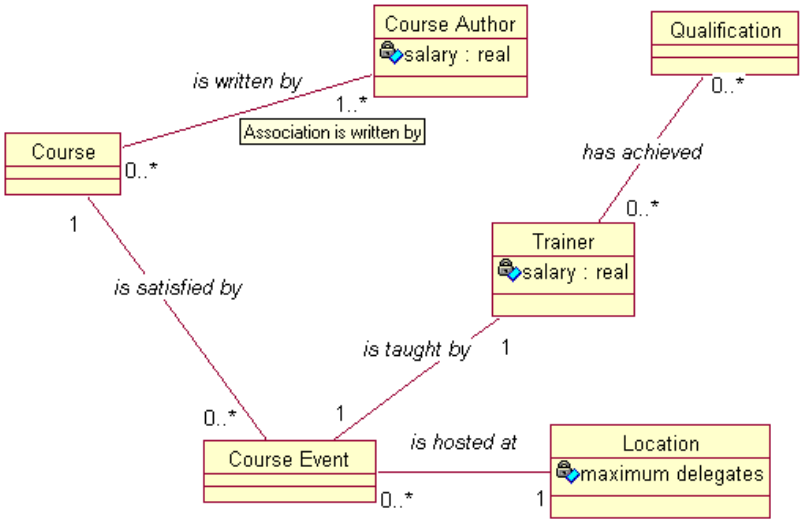


Figure 30 - A UML Class Diagram

Developing a Class Diagram is key to any object oriented design process. The Class Diagram will essentially provide the structure of the eventual code we produce.

At this stage, however, we are not yet interested in the system design (we are still analysing), so the class diagram we produce at this stage will be quite sketchy, and will not contain any design decisions.

For example, we would not want to add a "LinkList" class at this stage, as that is tying us down to a particular solution far too early.

We are essentially producing an **Analysis Class Diagram**. Many practitioners prefer to completely distinguish their Analysis Class Diagram from the Design Class Diagram by calling the Analysis Diagram the **Conceptual Model** – a term I shall use for the rest of this course.

On the conceptual model, we aim to capture all of the concepts or ideas that the customer recognises. For example, some good examples of concepts would be:

- **Lift** in a lift control system
- **Order** in a home shopping system
- **Footballer** in a football transfers system (or a PlayStation football game!)
- **Trainer** in a stock management system for a shoe shop
- **Room** in a room booking system

Some very bad examples of concepts are:

- **OrderPurgeDaemon** the process that regularly deletes old orders from the system
- **EventTrigger** – the special process that waits for 5 minutes and then tells the system to wake up and do something
- **CustomerDetailsForm** – the window that asks for details of the new customer in a shopping system
- **DbArchiveTable** – the database table holding a list of all old orders

These are bad concepts, because they are focussing on design – the solution, and not the problem. In the DbArchiveTable example, we are already tying ourselves down to a relational database solution. What if it turns out later that it is more efficient, cheaper, and perfectly acceptable to use a simple text file?

The best rule of thumb here is:

If the customer doesn't understand the concept, it probably isn't a concept!

Designers hate the conceptual step – they cannot wait to crack on with design. We shall see, however, that the conceptual model will slowly transform into a full design class diagram as we move through the construction phase.

Finding Concepts

I recommend a similar approach to finding Use Cases – a workshop is best – once again, with as many interested *stakeholders* as possible.

Brainstorm suggestions for concepts, capture **all** the suggestions. Once brainstorming is complete, work as a group to discuss and justify each suggestion. Apply the rule of thumb that the customer must understand the concept, and discard any that don't apply to the problem, and discard any that are touching on design.

Extracting Concepts From Requirements

The requirements document is a good source of concepts. Craig Larman (ref [2]) suggests the following candidate concepts from the requirements:

- Physical or tangible objects
- Places
- Transactions
- Roles of People (eg Customer, Sales Clerk)
- Containers for other Concepts
- Other Systems external to the system (eg Remote Database)

- Abstract Nouns (eg Thirst)
- Organisations
- Event (eg Emergency)
- Rules/Policies
- Records/Logs

A couple of points here. First of all, gathering concepts in a mechanical manner is poor practise. The above list are good suggestions, but it would be wrong to think that it is enough to run through the requirements document with a highlighter pen, pulling out some phrases and setting them as concepts. You **must** have input from the customer.

Secondly, many practitioners suggest extracting **noun phrases** from documents. This approach has been is common usage for almost 20 years, and although there is nothing inherently wrong with it, I hate the implication that mechanically searching for nouns will result in a good list of concepts/classes. Sadly, the English language is far too ambiguous to allow for such a mechanical approach. I'll say it again – **input from the customer is essential!**

The Conceptual Model in the UML

Now that we've seen how to discover the concepts, we need to look how to capture the concepts in the UML. We'll use the core aspects of the **class diagram**.

We represent our concept in a simple box, with the title of the concept (by convention, capitalised) at the top of the box.

Figure 31 - The "Race" concept captured in UML (for a Horse Racing System)

Notice that inside the large box are two smaller, empty boxes. The box in the middle will be used shortly, to capture the **attributes** of the concept – more on this in a moment. The bottom box is used to capture the **behaviour** of the concept – in other words, what (if anything) the concept can actually do. Deciding on the behaviour of the concept is a complicated step, and we defer this stage until we are in the design stage of construction. So we needn't worry about behaviour for now.

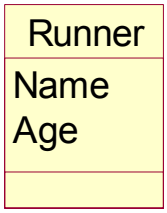


Figure 32 - The UML captures the attributes and behaviour of a concept

In the example above, we have decided that every runner will have two attributes – "Name" and "Age". We leave the bottom area blank until later, when we decide what a "Runner" is able to do.

Finding Attributes

We need to determine what the attributes of each concept are – and again, a brainstorming session with the stakeholders is probably the best way to achieve this.

Often, arguments arise over whether or not an attribute should become a concept on its own. For example, lets say we are working on a Staff Management system. We have identified that one concept would be "Manager". A suggestion for an attribute might be "salary", as follows:

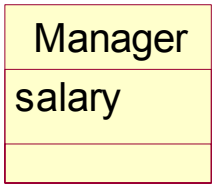


Figure 33 - Manager concept, with the attribute "Salary"

That looks fine, but someone might argue that "Salary" is also a concept. So, should we make promote it from an attribute to a concept?

I have seen many modelling sessions descend into chaotic arguments over issues like this one, and my advice is to simply not worry about it*: if in doubt, make it another concept.* These kind of problems usually resolve themselves later on anyway, and it really isn't worth wasting valuable modelling time on arguments!

Figure 34 - Manager and Salary, two separate concepts

Guidelines for Finding Attributes

The following rules of thumb may be helpful when deciding between attributes and concepts – but heed the advice above and don't worry too much about the distinction. *If in doubt, make it a concept!*

- Single valued strings or numbers are usually attributes[10]
- If a property of a concept cannot *do* anything, it might be an attribute - eg for the manager concept, "Name" clearly sounds like an attribute. "Company Car" sounds like a concept, because we need to store information about each car such as the registration number and colour.

Associations

The next step is to decide how our concepts are related. In any non-trivial system, at least some of the concepts are going to have some kind of conceptual relationship with other concepts. For example, back to our Staff Management system, given the following two concepts:

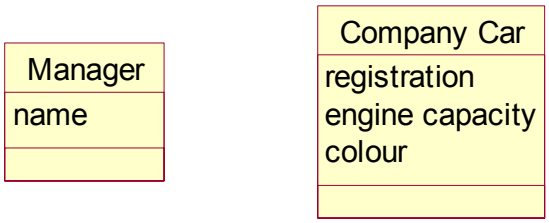


Figure 35 - Manager and Company Car Concepts

These concepts are related, because in the company we are developing a system for, each Manager drives a Company Car.

We can express this relationship in the UML by connecting the two concepts together with a single line (called an **association**), as follows:

Figure 36 - "Manager" and "Company Car" related by association

Two important things to note about this association. First of all, the association has a descriptive name – in this case, "drives". Secondly, there are numbers at each end of the association. These numbers describe the **cardinality** of the association, and tell us how many instances of each concept are allowed.

[10] But not always – this is a rule of thumb and shouldn't be followed slavishly.

In this example, we are saying that “Each Manager Drives 1 Company Car”, and (going from right to left) “Each Company Car is driven by 1 Manager”.

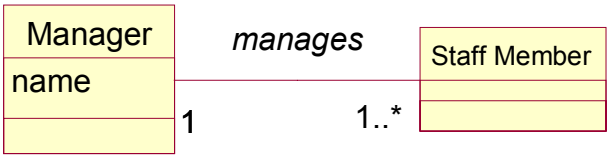


Figure 37 - Another Association Example

In the above example, we see that “Each Manager manages 1 or more staff members”; and (going the other way), “Each Staff Member is managed by 1 Manager”.

Every association should work like this – when reading the association back in English, the English sentence should make perfect sense (especially to the customer). When deciding upon association names, avoid weak names like “has” or “is associated to” – using such weak language could easily hide problems or errors that would otherwise have been uncovered if the association name was more meaningful.

Possible Cardinalities

Basically, there are no restrictions on the cardinalities you are able to specify. The following diagram lists some examples, although this list is by no means exhaustive. The * notation denotes "many". Note the slight difference between "*" and "1..*". The former is a vague "many", meaning that perhaps any number of concepts are allowed, or maybe we haven't made the decision yet. The latter is more concrete, implying that one or more are allowed.

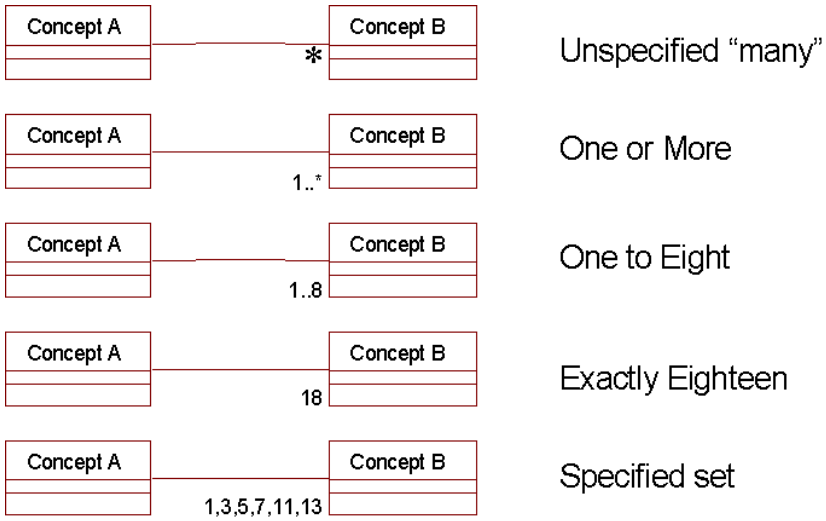


Figure 38 - Example Cardinalities

Building the Complete Model

Finally, let’s look at a methodical system for determining the associations between the concepts. Assume we have completed the brainstorm session and uncovered several concepts for the Staff Management system. The set of concepts are in the figure below (I’ve omitted the attributes for clarity).

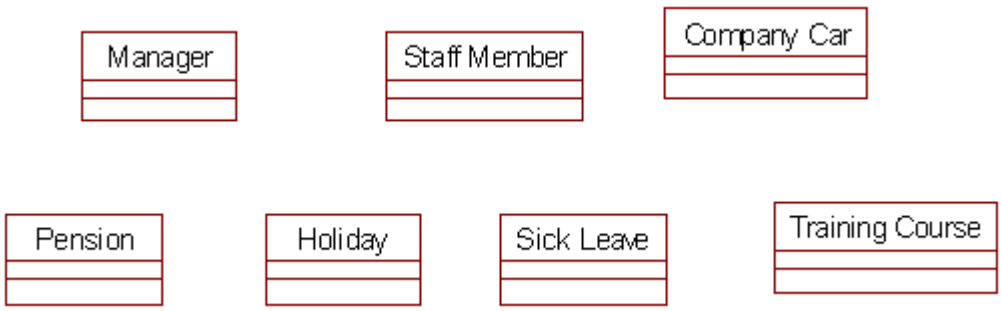


Figure 39 - Set of Concepts for Staff Management

The best way to proceed is to "fix" one concept, say "Manager" and consider every other concept in turn. Ask yourself "are these two concepts related?", and if so, immediately decide on the name of the association, and the cardinality…

> "Are **Manager** and **Staff Member** related? Yes, *Each Manager manages 1 or more staff members*.
> **Manager** and **Company Car**? Yes, *Each Manager drives 1 company car.*
> **Manager** and **Pension**? Yes, *Each Manager contributes to 1 pension"*

And so on until the model is complete. A common mistake at this stage is to decide two concepts are related, draw a line on the diagram and leave off the association name until later. This is making extra work for yourself – you'll find that once you've finished adding lines, you'll have no idea what any of them mean (and they'll usually look like spaghetti), and you have to start all over again!

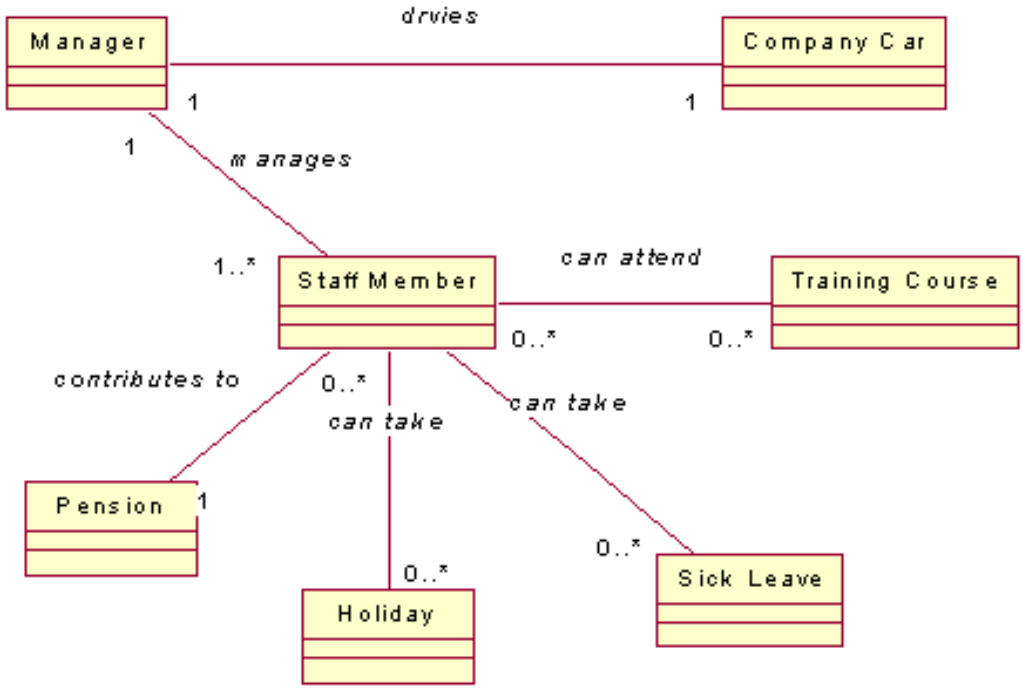


Figure 40 - The simple conceptual model, completed

When building the model, it is important to remember that associations are less important than attributes. Any missing associations will be easily picked up during design, but it is harder to spot missing attributes.

Furthermore, it can be tempting to overspecify the map of associations "just in case", and end up with quite a confusing and complex diagram. So a good rule of thumb is to concentrate on concepts and attributes, and try to fix the most obvious associations.

At the end of the modelling, the diagram should make sense to the customer when you "read back" the diagram in English.

Summary

Conceptual Models provide a powerful way of investigating the problem further.

Later on, we'll expand our model into the design aspects.

This model will eventually be one of the key inputs when we build the code.

To build the model, use a workshop technique as with the Use Cases.

Chapter 9
Ranking Use Cases

We have a lot of work ahead – how can we divide up the work into the simple, manageable iterations that we described in the early stages of the course?

The answer is by focusing on our Use Cases. In each iteration, we design, code and test just *a few* Use Cases. So effectively, we have already decided how we are going to divide up the work into iterations – the only thing we haven't done yet is to decide upon the order in which we will attack them.

To plan the order, we allocate each Use Case a *rank*. The rank is simply a number that indicates which iteration the Use Case will be developed in. Any good Case Tool should allow the rank to be captured as part of the model.

There are no hard and fast rules on how to allocate ranks. Experience and knowledge of software development plays a large part in setting the rank. Here are some guidelines on which Use Cases should be allocated a higher rank (ie to be developed earlier rather than later):

- Risky Use Cases
- Major Architectural Use Cases
- Use Cases exercising large areas of system functionality
- Use Cases requiring extensive research, or new technologies
- "Quick Wins"
- Large payoffs for the customer

Some Use Cases will need to be developed over several iterations. This could be because the Use Case is simply too big to design, code and test in one iteration, or it could be because the Use Case depends upon many other Use Cases being complete ("Start Up" is a classic example of this).

This shouldn't cause too many problems – simply break the use case down into several versions. For example, here is a large Use Case, which is to be developed over three iterations. At the end of each iteration, the Use Case can still perform a useful task, but to a limited degree.

"Fire Torpedoes" Use Case:

Version 1a allows the user to set a target (Rank : 2)
Version 1b allows the user to prime the weapons (Rank : 3)
Version 1c allows the user to discharge the weapon (Rank : 5)

Summary

Use Cases allow us to schedule the work across our multiple iterations

We rank the Use Cases to determine the order in which they are attacked

Ranking Use Cases is based on your own knowledge and experience.

Some rules of thumb will help in your early days.

Some Use Cases will span several iterations.

Chapter 10
The Construction Phase

In this short chapter, we'll take stock of what we've done, and what needs to be done next.

In the Elaboration Phase, we needed to understand the problem as fully as possible, without going into too much detail. We built a **Use Case Model**, and created as many Use Cases as possible. We did not fill in the complete details of the Use Cases, instead we supplied a very brief description of each one.

The next step was to build a conceptual model, where we captured the concepts driving our development. This conceptual model will provide us with the foundations of the design.

We then ranked each of our Use Cases, and in doing so, we have planned the order of the Use Case development.

This completes our Elaboration Phase. A complete review of the phase would be held, and a Go/No Go decision needs to be made. After it, we may have discovered during Elaboration that we really cannot provide a solution for our customer – better to find out now than at the end of coding!

Construction

Now that we are in the Construction Phase, we need to build the product, and take the system to the state where it can be delivered to the user community.

Recall that our general plan of attack is to follow a series of short waterfalls, with a small number of Use Cases developed in each iteration. At the end of each iteration, we will review progress, and preferably timebox the iteration.

Ideally, we will aim to achieve a running (albeit, of course, limited) system at the end of each iteration.

Each stage of the waterfall will produce a set of documents or UML models.

- In Analysis, we will produce some Expanded (or Full) Use Cases
- In Design, we will produce Class Diagrams, Interaction Models and State Diagrams
- In Code, we will produce running and unit tested code

Then the iteration is tested (ie all of the use cases need to be demonstrably working), and then we reach the review.

Summary

We have completed Elaboration, and we are now ready to begin construction. We will look at each model in turn and see how it benefits the construction exercise.

Chapter 11

The Construction Phase : Analysis

The first stage of the construction phase is Analysis. We need to revisit the Use Cases we are building in this iteration, and enhance and expand those Use Cases. By concentrating on the full detail of only a few Use Cases per iteration, we are reducing the amount of complexity we have to manage at any one time.

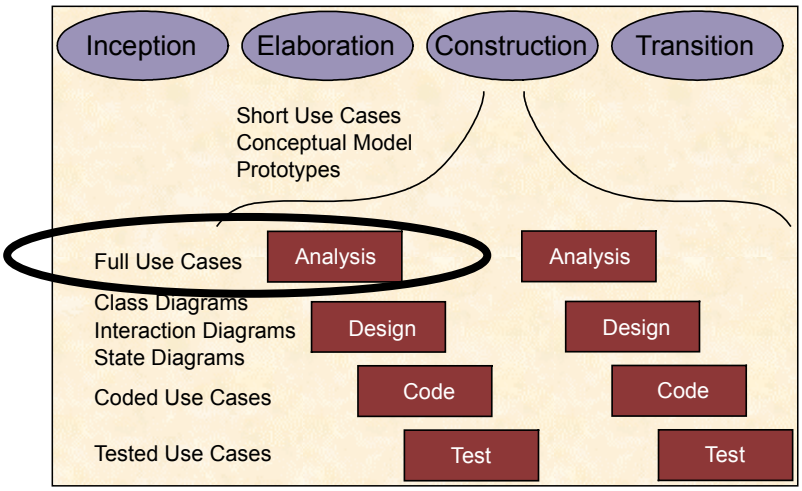


Figure 41 - Analysis Phase of Construction

Remember that even though we are now in construction, we are still at the analysis stage – albeit a more detailed analysis than the analysis we did in elaboration. Therefore, we must bear in mind that we are only concerned with the **problem**, and not the **solution**. So we are looking at what the system has to do, without worrying about how it will do it.

Back to the Use Cases

During Elaboration, we produced Short Use Cases and decided to defer the full details (ie Main Flow, Alternate Flow, Pre and Post Conditions) until construction phase. The time has come to complete the full details (but only for the Use Cases we are dealing with in this iteration).

Use Case	Place Bet
Short Description:	
The user places a bet on a particular horse after choosing a race	
Actors:	Gambler

Requirements	R2.3; R7.1
Pre-Conditions:	
Post-Conditions	
Main Flow:	
Alternate Flow(s):	
Exception Flow(s):	

Figure 42 - The Short Use Case, Place Bet

The diagram above shows an example Short Use Case. Each of the headings needs to be filled in. The best way to explain how to fill in the headings is with a specific example, so lets have a look at the **Place Bet** Use Case:

1. Pre-Conditions

This section describes the system conditions which must be satisfied before the Use Case can actually take place. In the Place Bet, example, a good pre-condition could be:

"*The User Has Successfully Logged In*".

Clearly, the betting system needs to validate customers before they can start gambling. However, the validation of the user is not part of this Use Case, so we must ensure that this condition has been satisfied before the betting takes place.

2. Post Conditions

The post conditions describe the state the system will be in at the end of the Use Case. The post-condition is conventionally written in past tense language. So in our Place Bet example, the post condition would be:

"*The User placed a bet and the bet was recorded by the system*"

There can be more than one post condition, depending on the outcome of the Use Case. These different post conditions are described using "if then" language". Eg "*If a new customer, then a customer account was created. If an existing customer, then the customer details were updated*".

3. Main Flow

The main flow section describes the most likely, or the most conventional, flow of events through the Use Case. Clearly, in the Place Bet Use Case, many things can go wrong. Perhaps the user cancels the transaction. Maybe the user has insufficient funds to place a bet. These are all events we have to consider, but really, the most conventional flow through this use case is going to be a user successfully placing a bet.

In the main flow, we need to detail the interactions between the actor and the system. Here is the main flow for "Place Bet":

(*(1) On initiation of Place Bet by the gambler, a list of the day's races are requested from the system, and (2) the list of races are displayed*

(3) The Gambler chooses the race to bet on [A1] and (4) the system presents a list of the runners for that race

(5) The Gambler chooses the horse to bet on [A1] and enters the required stake [E1]

(6) The User Confirms the transaction and (7) the system displays a confirmation message

Notice that every actor/system interaction is broken down into steps. In this case, there are seven steps in the main flow of the Use Case. The [A1] and [E1] notation will be explained in a moment, when we look at Alternate Flows and Exception Flows.

Alternate Flows

Alternate flows are simply less common (but legitimate) flows through the Use Case. The alternate flow will typically share many steps with the main flow, so we can notate the point in the main flow where the alternate flow takes over. We have done this in step (3) of the main flow above, through the [A1] notation. This is because when the user chooses the race to bet on, they can cancel the transaction. They can also cancel the transaction at step 5, when they are required to enter the stake.

"*(A1) The User Cancels the Transaction*
Post Condition -> No bets were placed"

In this case, the Alternate flow has resulted in a change to the post condition – no bets were placed.[11]

Exception Flows

Finally, the exception flow describes exceptional situations. In other words, a flow where an error has occurred, or an event that couldn't have otherwise been predicted.

In our place bet example, we could have the following exception:

"*(E1) The users credit is not sufficient to fund the bet. The User is informed and the Use Case terminates*"

When we move to program code. the items under Exception Flow should map to exceptions in the program - if your target language supports exceptions. Many modern languages do support them - Java, C++, Delphi and Ada to name but four.

---

[11] Some UML practitioners prefer to say that an alternate flow will always result in the same post conditions as the main flow. Another example where the UML can be applied in many different ways. I prefer to allow the Alternate flow to be any legitimate but less common flow, resulting in any post condition you want.

The Complete Use Case

Use Case	Place Bet
Short Description:	The user places a bet on a particular horse after choosing a race
Actors:	Gambler
Requirements	R2.3; R7.1
Pre-Conditions:	The User has successfully logged in
Post-Conditions:	A bet was placed and the bet was recorded by the system
Main Flow:	

(1) On initiation of Place Bet by the gambler, a list of the day's races are requested from the system, and (2) the list of races are displayed

(3) The Gambler chooses the race to bet on [A1] and (4) the system presents a list of the runners for that race

(5) The Gambler chooses the horse to bet on [A1] and enters the required stake [E1]

(6) The User Confirms the transaction and (7) the system displays a confirmation message

Alternate Flow(s):

(A1) The gambler cancels the transaction.

Post Condition -> No bets were placed

Exception Flow(s):

(E1) The user's credit is not sufficient to fund the bet. The user is informed and the Use Case Terminates

Figure 43 - Full Use Case Description

The UML Sequence Diagram

Producing the Use Case Descriptions is difficult. Many people find the distinction between analysis and design especially difficult – often the Use Case descriptions become littered with design decisions.

Here's an example from the Place Bet Use Case:

"*The User selects the race to bet on. The system interrogates the race database and compiles an array of runners for the race.*"

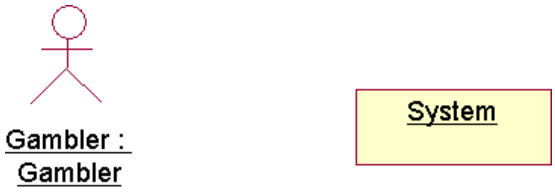
This is a poor Use Case description. By talking about the race database and introducing arrays, we are tying ourselves down to specific design decisions.

When building the Use Cases, we need to treat the system as a "black box", which can accept requests from actors and return results to the actor. We are **not** concerned (yet) with how the black box fulfils that request.

We recommend the use of a UML **Sequence Diagram**. A sequence diagram is useful in a variety of different situations, especially at the design stage. However, the

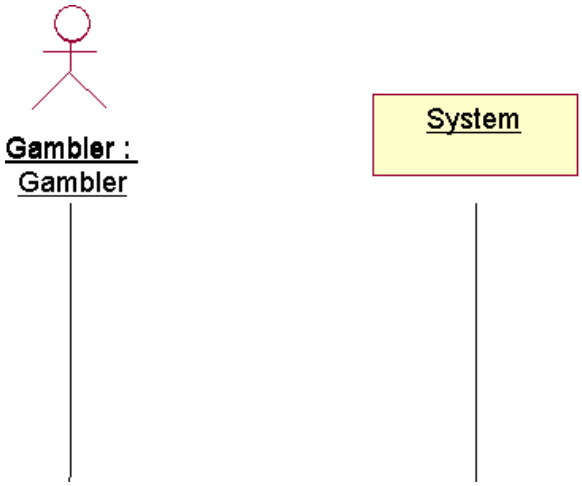
diagram can be used in analysis to help us with this “black box” analysis of the system. Here’s how the diagram works:

1



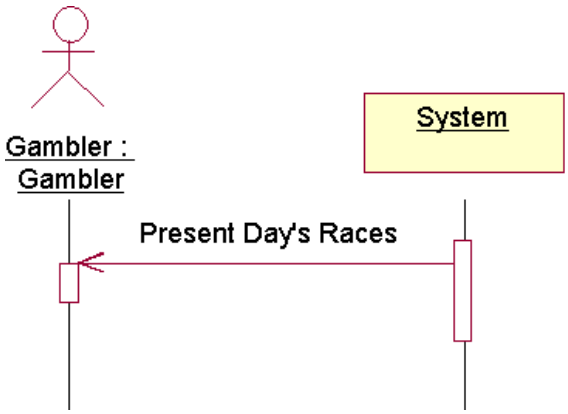
We place the actor on the left of the diagram, and on the right of the diagram, we represent the entire system as a single box.

2



Next, we add vertical “timelines”. The lines represent the passage of time in the downward direction

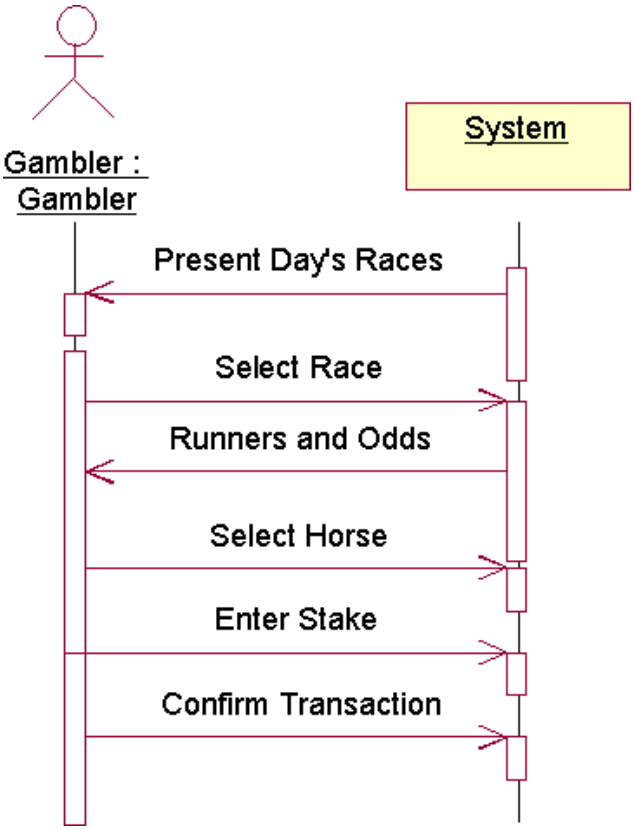
3



Interactions between the user and the system are represented as a line with an arrow between the system and the actor.

A description of the interaction is written alongside the arrow.

4



Continue adding the interactions down the timeline.

The long boxes down the timeline indicate when the system of the actor is "active". This notation is more important when we draw sequence diagrams in design – for now, it doesn't really matter(these boxes were added by our CASE tool).

Once the System Sequence Diagram is complete, it is a fairly simple and mechanical task to write the description of the main flow for the use case. There is no need to laboriously draw these diagrams for every single alternate and exception flow, although it would be worthwhile for very complicated or interesting alternatives.

Summary

In this chapter, we moved into the construction phase. We focussed on a handful of Use Cases in the iteration, and we explored the detail we need to develop for the full Use Case.

We learnt the basics of a new UML diagram, the **System Sequence Diagram**, and saw that this diagram can be helpful when producing the detailed use case.

Now we have the details behind the use cases, the next stage is to produce a detailed design. We've looked at the **what** – we now look at the **how**.

Chapter 12

The Construction Phase : Design

Design - Introduction

By now, we have a full grasp of the problem we are trying to solve (for this iteration). We have developed the Use Cases for the first iteration to a deep level of detail, and we are now ready to design the solution to the problem.

Use Cases are satisfied by objects interacting. So in this stage, we need to decide on what objects we need, what the objects are responsible for doing, and when the objects need to interact.

The UML provides two diagrams to allow us to express the interaction of objects, namely the **Sequence Diagram** and the **Collaboration Diagram**. These two diagrams are very closely related (some tools can generate one diagram from the other one!) Collectively, the Sequence and Collaboration Diagrams are called the **Interaction Diagrams**.

When we decide on the objects we need, we must document the classes of objects we have, and how the classes are related. The UML **Class Diagram** allows us to capture this information. In fact, much of the work of producing the Class Diagram is already done – we'll use the Conceptual Model we produced earlier as a starting point.

Finally, a useful model to build at the design stage is the **State Model**. More details on this later.

So, in design, we are going to produce three types of model – the **Interaction Diagram**, the **Class Diagram** and the **State Diagram**.

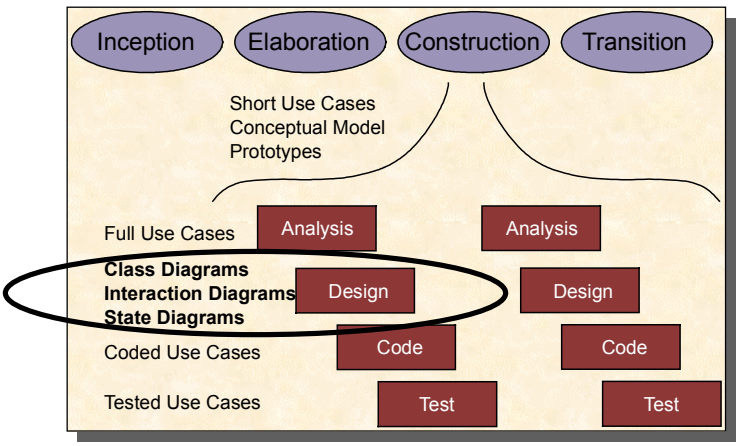


Figure 44 - The deliverables from the design stage

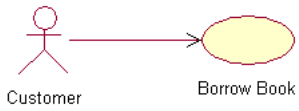
Collaboration of Objects in Real Life

So, our Use Cases are going to be satisfied by the collaboration of different objects. This is actually what happens in real life. Consider a library. The library is run by a librarian who runs the front desk. The librarian is responsible for handling queries from customers, and is responsible for managing the library index. The librarian is also in charge of several library assistants. The assistants are responsible for managing the library shelves (the librarian couldn't do this – or she wouldn't be able run the front desk efficiently).

The objects in this library system are:

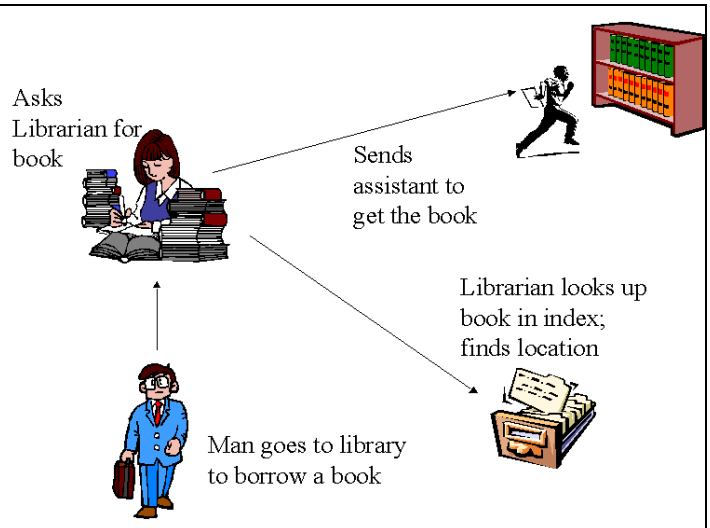
Customer
Librarian
Library Assistant
Library Index
Bookshelf

Let's consider the obvious Use Case – Borrow Book



How would this Use Case be satisfied? Let's assume that the customer does not know where the book is located and needs help. This could be the chain of events:

1. The Customer goes to the librarian and asks for "Applied UML" by Ariadne Training.
2. The Librarian looks in the index for the name of the book. She finds the index card for the book, which says that the book is located at shelf 4F.
3. The Librarian asks the assistant to retrieve the book from shelf 4F.



4. The Librarian gets the requested book and returns it to the librarian.
5. The librarian checks out the book and hands it to the customer.

Figure 45 - Chain of events for "Borrow Book"

Even though this example is very simple, it was still not particularly easy to consider the responsibilities of each object. This is one of the key activities of Object Oriented Design – getting the responsibilities of each object correct. For example, if I had decided to let the librarian physically retrieve the book, I'd have designed a very

inefficient system. Later on in this chapter, we'll present some guidelines for allocating responsibilities to objects.

Collaboration Diagrams

In this section, we will look at the syntax of the UML Collaboration diagram. We will look at how to use the diagram in the next section.

A collaboration diagram allows us to show the interactions between objects over time. Here is an example of a completed collaboration diagram:

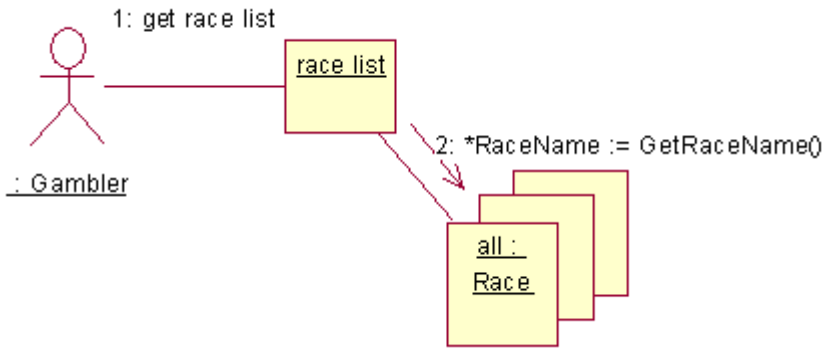
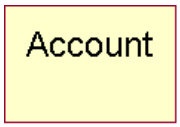


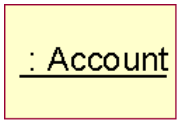
Figure 46 - Collaboration Diagram

Collaboration Syntax : The Basics

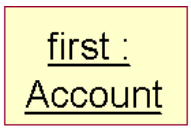
A class on the collaboration diagram is notated as follows:



An instance of a class (in other words, an object) is notated as follows:

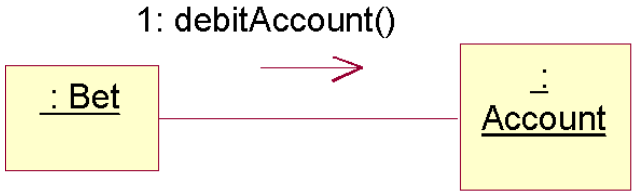


Sometimes, we will find it useful to name an instance of a class. In the following example, I want an object from the Account class, and I want to call it "first":

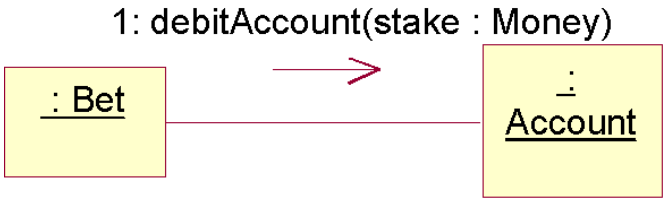


If we want one object to communicate with another object, we notate this by connecting the two objects together, with a line. In the following example, I want a "bet" object to communicate with an "account" object:

Once we have notated that one object can communicate with another, we can send a named message from one object to the other. Here, the "bet" object sends a message to the "account" object, telling it to debit itself:



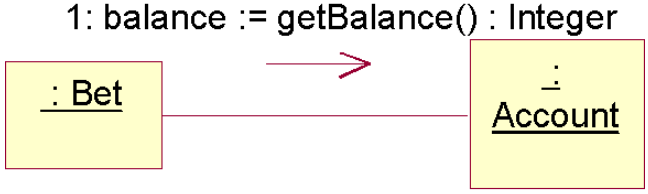
If we want to pass parameters with the message, then we can include the parameter in the brackets, as follows. The datatype of the parameter (in this case, a class called "Money") can be shown, optionally.



A message can return a value (analogous to a function call at the programming stage). The following syntax is recommended in the UML standard if you are aiming for a language neutral design. However, if you have a target language in mind, you can tailor this syntax to match your preferred language.

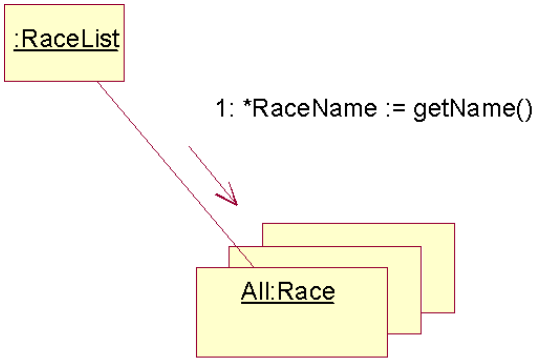
```
return := message (parameter : parameterType) : returnType
```

In the following example, the bet object needs to know the balance of a particular account. The message "getBalance" is sent, and the account object returns an integer:



Collaboration Diagrams : Looping

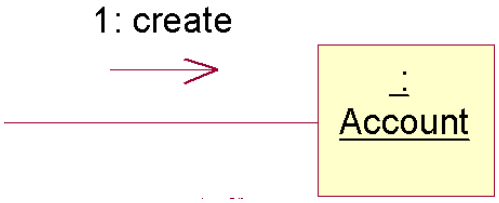
If we need to introduce a loop into a collaboration diagram, we use the following syntax. In this example, an object from the "Race List" class needs to assemble itself. To do this, it asks every member of the "Race" class to return its name.



The asterisk denotes that the message is to be repeated. Rather than specifying an individual object name, we have used the name "All" to denote that we are going to iterate across all of the objects. Finally, we have used the UML notation for a collection of objects with the "stacking" of the object boxes.

Collaboration Diagrams : Creating new objects

Sometimes, one object will want to create a new instance of another object. The method for doing this varies between languages, so the UML standardises creation through the following syntax:



The syntax is rather strange, really – you are sending a message called "Create" to an object that doesn't yet exist![12]

Message Numbering

Notice that all of the messages we have included so far have a mysterious "1" alongside them? This indicates the order in which the message is executed in order to satisfy the Use Case. As we add more messages (see the full example on page 69), we increment the message number sequentially.

[12] In reality, you are sending a message to the class, and in most languages you are also calling the constructor.

Collaboration Diagrams : Worked Example

Let's pull all of that theory together and see how the notation works in practise. Let's build the "place bet" Use Case using a collaboration diagram.

This example is far from perfect, and leaves a lot of questions unanswered (we'll list the unresolved issues at the end of this chapter). However, as a first cut, the example should illustrate how collaboration diagrams are built. We'll revisit these design issues in later chapters.

See page 61 for the full Use Case Description for "Place Bet".

To build this diagram, we need some objects. Where do we get the objects from? Well, we will certainly have to invent some new objects as we go along, but many of the candidate objects should come directly from our old friend, the conceptual model we built at the elaboration phase. Here is the conceptual model for the betting system:

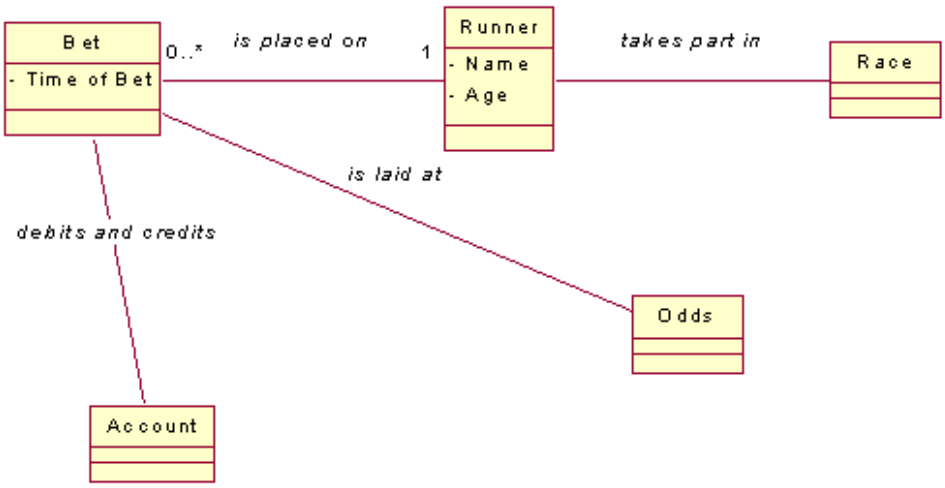
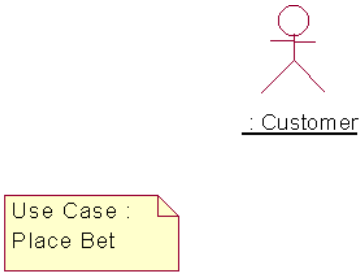


Figure 47 - Betting System Conceptual Model

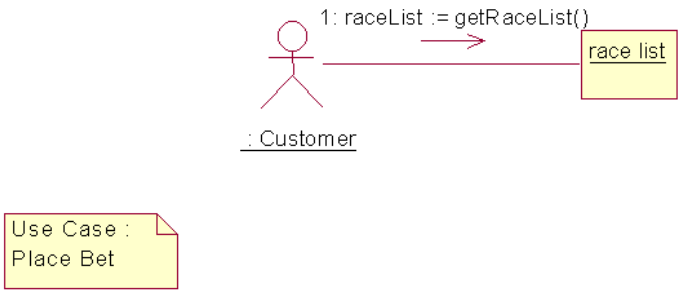
Where there are associations, such as "is placed on", we will probably use these associations to pass messages on the collaboration diagram. We could possibly decide that we need to (for example) pass a message between "account" and "race". This is perfectly valid, but as the association was not discovered at the conceptual stage, we might have possibly broke some of the customer's requirements. If this happens, *check with the customer!*

With the Use Case description and the conceptual model in mind, let's build the collaboration for "Place Bet".

1. First of all, we start with the initiating actor, the customer. The actor symbol is not strictly part of the UML collaboration diagram, but it is extremely useful to include it on the diagram anyway.

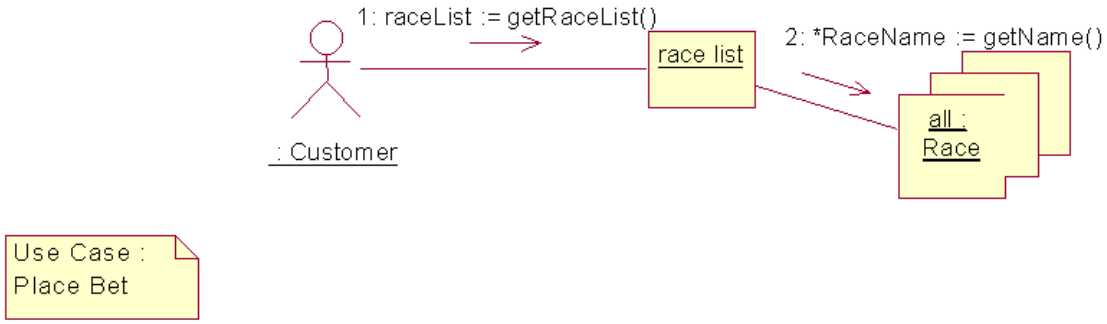


2. Now, according the Use Case description, when the customer selects the "place bet" option, a selection of races are presented. So we'll need an object that contains a full list of the races for today, so we create an object called "Race List"[13] This is an object which was not represented on the conceptual model. This is called a *design class.*



3. So, the actor sends a message to the new "Race List" object called "getRaceList". Now, the next job is for the race list to assemble itself. It does this by looping around all of the Race objects, and asking them what their name is.

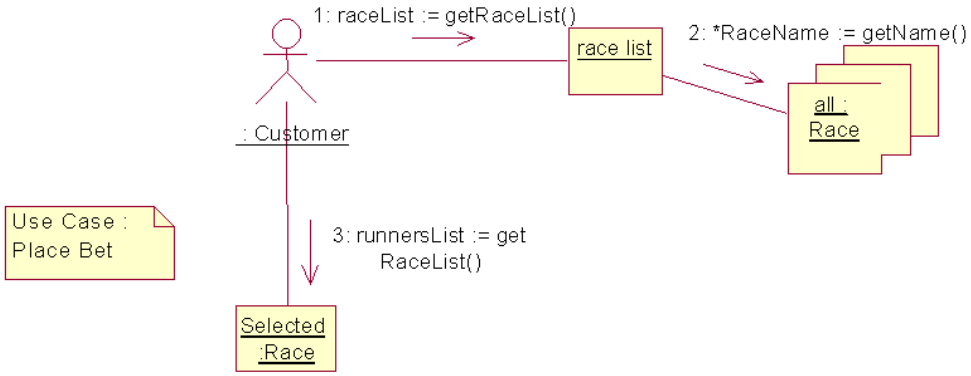
   The race object has been taken from the conceptual model.



4. Next, we assume the race list is now returned back to the customer. The ball is now in their court, and according to the Use Case description (page 61), the user now selects a race from the list.

[13] This will be a container, or array, or something similar, depending on the target language

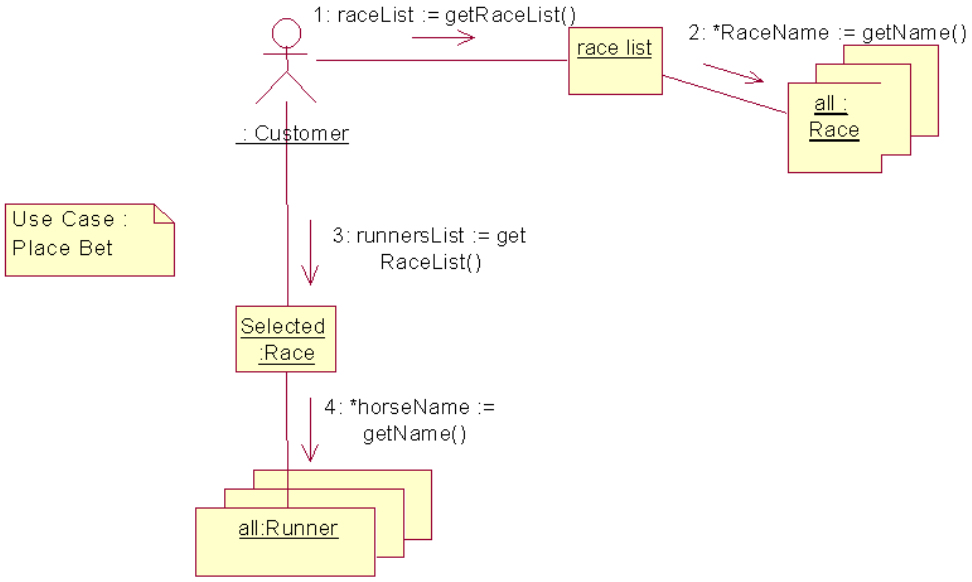
We may now assume that a race has been selected. We now need to get a list of runners for the selected race, and to find that out I have decided to make the race object responsible for keeping a list of its runners. We'll see in future chapters why and how this kind of decision is made.



So, message number 3 is sent to the selected race, and the message is asking for a list of runners for that race.

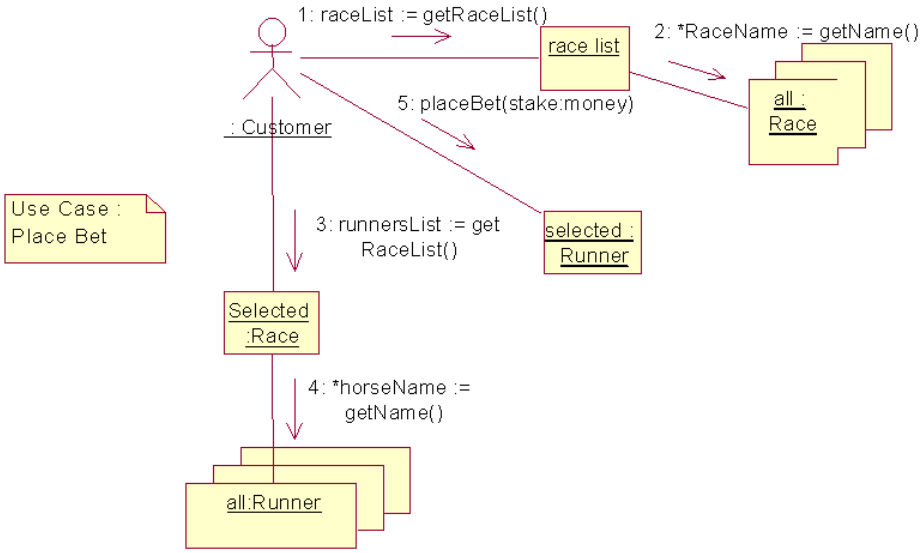
5. How does the race object know what runners are part of that race? Once again, we'll do this using a loop, and we'll get the race object to assemble a list of runners.

   How this is actually achieved in the real system is not trivial. Clearly, we are going to store these runners on a database of some kind, so part of the physical design process is going to be to construct a mechanism for retrieving the horse records from the database. For now, however, it is enough to say that the selected Race object is responsible for assembling a list of the runners for that race.



6. The list of runners is now returned back to the customer. The ball is again in their court, and according to the Use Case description, they must now select a runner, and then select their stake money.

Now that we know the runner and the stake, we can send a message to the selected runner, and tell it that a bet has been placed upon it.



By building this collaboration diagram, we have NOT mapped out exactly what the code will look like. The following issues have been left unresolved:

1. We have not mentioned anything on the diagram about how the user inputs data into the system, and how the data (such as the list of runners) is output on the screen. Somehow all of this happens as if by magic "inside" the actor.

   We'll see later that this is good design. We want to make our design as flexible as possible, and by including detail about the User Interface at this stage, we are tying ourselves down to one specific solution.

2. How does the "Race" object find out what runners are part of that race? Clearly, there is some kind of database (or even network) operation going on here. Again, we do not want to tie our design down at this stage, so we defer these details until later.

3. Why have we made the "runner" object responsible for tracking which bets have been placed on it? Why didn't we create another class, perhaps called "bet handler" or "betting system"? This issue will be explored in the following chapter.

What we **have** done is decide on the responsibilities of each class. What we are doing, in effect, is building upon the conceptual model we produced at the elaboration stage.

Some Guidelines For Collaboration Diagrams

As we progress through this course, we'll expand on how to produce good diagrams. For now, keep the following guidelines in mind:

1. **Keep the diagram simple!!!** It seems that in our industry, unless a diagram covers hundreds of pages of A4 and looks very complicated, the diagram is trivial! The best rule to apply to the collaboration diagram (and the other UML diagrams too) is to keep them as simple as possible. If the collaboration for a use case gets complicated, break it up. Perhaps produce a separate diagram for each user/system interaction.

2. **Don't try to capture every scenario.** Every use case comprises a number of different scenarios (the main flow, several alternatives and several exceptions). Usually, the alternatives are fairly trivial and not really worth the bother of including.

   A common mistake is to cram every scenario on one diagram, making the diagram complex and difficult to code.

3. **Avoid creating classes whose name contains "controller", "handler", manager" or "driver".** Or at least, be suspicious if you do come up with an object with such a name. Why? Well, these classes tend to suggest that your design is not object oriented. For example, in the "Place Bet" use case, I could have created a class called "BetHandler" that deals with all of the betting functionality. But this would be an action oriented rather than an object oriented solution. We already have the "Bet" object from the collaboration diagram, so why not use it, and give it the responsibility of handling bets?

4. **Avoid God classes.** Similarly, if you end up with a single object that does a lot of work and does not collaborate much with other objects, you have probably built an action oriented solution. Good OO solutions consist of small objects who don't do too much work themselves, but work with other objects to achieve their goal. We'll go into much more detail about this later.

Chapter Summary

In this chapter, we began to construct a software solution to our Use Cases. The collaboration diagram enabled us to allocate responsibilities to the classes we derived during elaboration.

We touched on a few issues we should be aware of when allocating responsibilities, even though we need to learn more about this topic later. An example for "Place Bet" was studied.

In the next section, we'll see how we can expand the Conceptual Model and progress it towards a true Class Diagram.

Chapter 13

Design Class Diagrams

Recall that at the elaboration stage, we built a conceptual model. The conceptual model contained details about the customer's problem, and concentrated on the customer's concepts, and the properties of those concepts. We did not allocate behaviour to any of those concepts.

Now that we have begun to create collaboration diagrams, we can progress the conceptual model, and build it into a true **Design Class Diagram**. In other words, a diagram which we can base our final program code upon.

Producing the design class diagram is a fairly mechanical process. In this chapter, we'll examine an example Use Case, and how the conceptual model is modified because of it.

Crediting and Debiting Accounts

At the end of the "place bet" Use Case, the "bet" object sends a message to the customer's "Account" object, to tell it that it must be decremented. The following conceptual model was the basis of this design:

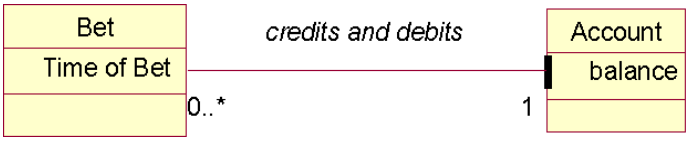


Figure 48 - Conceptual Model for this example

From the conceptual model, the following (portion of a) collaboration was developed:

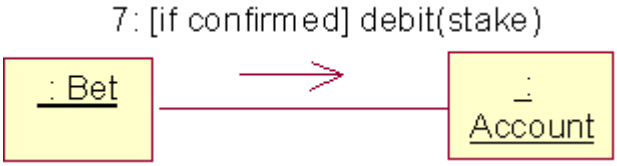


Figure 49 - Part of the Collaboration for "Place Bet"

Step 1 : Add Operations

From the collaboration diagram, we can see that the "Account" class needs to provide the behaviour "debit". So we add this operation to the lower half of the class symbol.

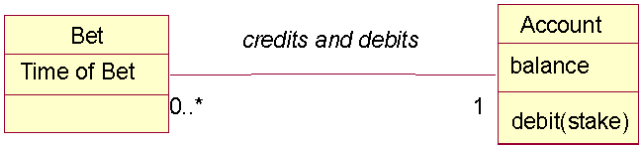


Figure 50 - Classes with Operation Added

Note that most people don't bother to add the *create* operations, as this will clutter the diagram (most classes need one anyway).

Step 2 : Add Navigability

The direction of the messages which are being passed through an association are also added. In this case, the message is being sent from the bet class to the account class, so we orient the association from the caller to the receiver:

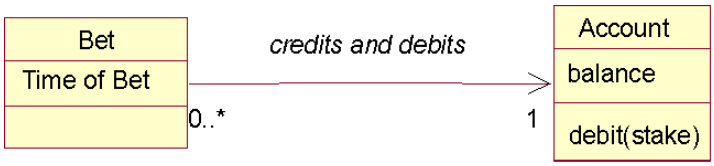


Figure 51 - Account Class with Navigability Added

Sometimes, a situation will arise where messages need to be passed both ways across an association. What to do in this case? The UML notation for this situation is to simply leave the arrow head off the association – a **bi-direction association**.

Many modellers believe that bi-directional associations are erroneous and need to be removed from the model somehow. In actual fact, there is nothing fundamentally wrong with a bi-directional relationship, but it does *suggest* a bad design. We'll explore this problem in a later chapter.

Step 3 : Enhance Attributes

We can also decide upon the datatype of the attributes at this stage. Here, we have decided to store the balance of an account as a **float**. Of course, this is dependent upon the choice of language.

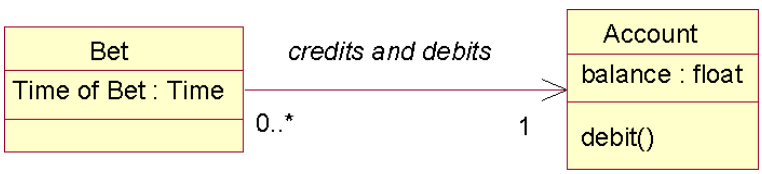


Figure 52 - Datatypes added

Step 4 : Determine Visibility

A fundamental concept of Object Orientation is encapsulation – the idea that the data held by an object is kept private from the outside world (ie from other objects).

We can signal which attributes and operations are public or private on a UML Class diagram by preceding the attribute/operation name by a plus sign (for public) and a minus sign (for private).

All attributes will be private, unless there is an extremely good reason (and there rarely is). Usually, operations will be public, unless they are helper functions, only to be used by operations contained within the class.

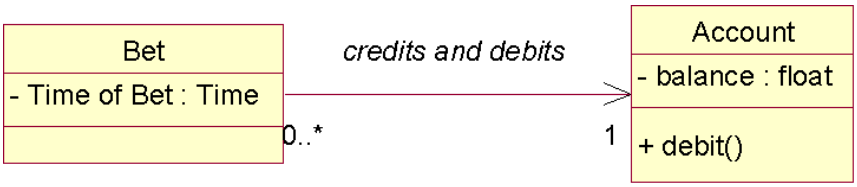


Figure 53 - The Class Diagram, complete!

Now that the class diagram is complete, we now have enough information to produce the code. We'll examine the transition to code in a later chapter.

Aggregation

An important aspect of object oriented design is the concept of **Aggregation** – the idea that one object can be built from (aggregated from) other objects.

For example, in a typical computer system, the computer is an aggregation of a CPU, a graphics card, a soundcard and so on.

We can denote aggregation in UML using the aggregation symbol – a diamond at the end of an association link.

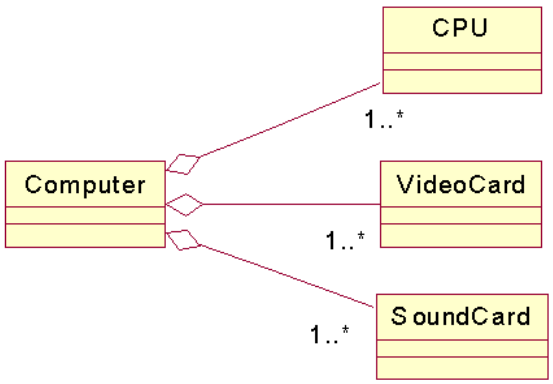


Figure 54 - Computer built from other objects

If you spot aggregation on your conceptual model, it may be clearer to explicitly notate the fact, using the aggregation symbol.

Composition

A very similar concept to aggregation is composition. Composition is stronger than aggregation, in the sense that the relationship implies that the whole cannot exist without the parts.

In the aggregation example above, for example, if we removed the soundcard, the Computer would still be a computer. However, a book isn't a book without its pages, so we say that a book is **composed of** the pages.

The notation for this is similar to aggregation, except this time the diamond is filled, as follows:

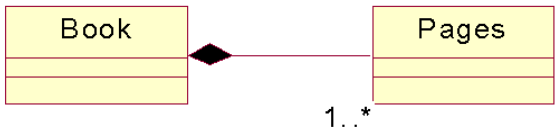


Figure 55 - A book is composed of 1 or more pages

Finding Aggregation and Composition

Finding these relationships on your class diagram is useful, but it is far from crucial to the success of your design. Some UML practitioners go further, and claim that these relationships are redundant, and should be removed (aggregation and composition can be modelled as an association with a name like "is composed from").

I believe, however, that as aggregation is one of the key concepts of Object Orientation, it is definitely worth explicitly notating its presence.

Summary

In this chapter, we looked at how to progress the class model, based on our work on collaborations. The transition from conceptual to design class model is actually fairly trivial and mechanical, and shouldn't cause too many headaches.

Chapter 14

Responsibility Assignment Patterns

In this section, we are going to take time out from the development process, and look closely at the skills involved in building good Object Oriented Designs.

Some of the advice given in this chapter may appear to be quite obvious and trivial. In fact, it is the violation of these simple guidelines that causes most of the problems in object oriented design.

The GRASP Patterns

To improve the way in which we produce our collaboration diagrams, we'll study the so-called "GRASP" patterns, as described by Larman in reference [2].

What is a pattern?

A pattern is a well used, extremely general, solution to a common occurring problem. The pattern movement began as an internet-based discussion community, but was popularised through the classic textbook "Design Patterns" (reference 6), written by the so called "Gang of Four".

To aid communication, each design pattern has an easy to remember name (such as Factory, Flywheel, Observer), and there are at least a handful of design patterns that every self respecting designer should be familiar with.

We'll be looking at some of the "Gang of Four's" patterns later, but first we'll study the GRASP patterns.

GRASP stands for "General Responsibility Assignment Software Patterns", and they help us to ensure we allocate behaviour to classes in the most elegant way possible.

The patterns are called **Expert, Creator, High Cohesion, Low Coupling** and **Controller**. Let's look at them in turn:

Grasp 1 : Expert

This is, on the face of it, a very simple pattern. It is also the one which is most commonly broken! So, this pattern should be at the forefront of your mind whenever you are building collaboration diagrams or creating design class diagrams.

Essentially, the Expert pattern says "given a behaviour, which class should the behaviour be allocated to?".

Wise allocation of behaviour leads to systems which are:

- Easier to understand
- More easily extendible
- Reusable
- More Robust

Let's look at a simple example. We have three classes, one representing a Purchase Order, one for an Order Line, and finally, one for a SKU (see the Course Case Study if these terms are not clear to you).

Here is a fragment from the conceptual model:

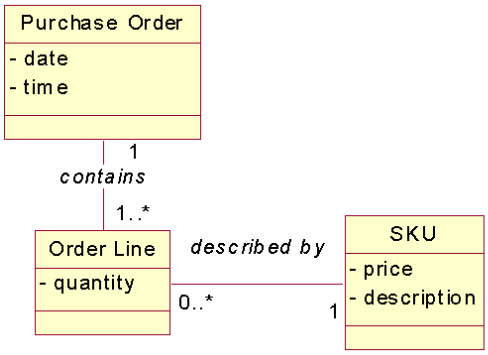


Figure 56 - Fragment from the conceptual model

Now, imagine that we are building collaboration for one of the use cases. This use case demands that the total value of a selected purchase order is presented to the user. Which class should be allocated the behaviour called "calculate_total()"?

The expert pattern tells us that the only class who should be allowed to deal with the total cost of purchase orders is the purchase order class itself – because that class should be an expert about all things to do with purchase orders.

So, we allocate the "calcuate_total()" method to the Purchase Order class.

Now, to calculate the total of a purchase order, the purchase order needs to find out the value of all of the order lines.

A poor design for this would be to let the purchase order see the contents of every order line (via accessor functions), and then sum up the total. This is breaking the expert pattern, because the only class who should be allowed to calculate the total of an order line is the order line class itself.

So we allocate a further behaviour to the order line class, called subtotal(). This method returns the total cost of the single order line. To achieve this behaviour, the order line class needs to find out the cost of a single SKU through another method (this time, an accessor) called price() in the SKU class.

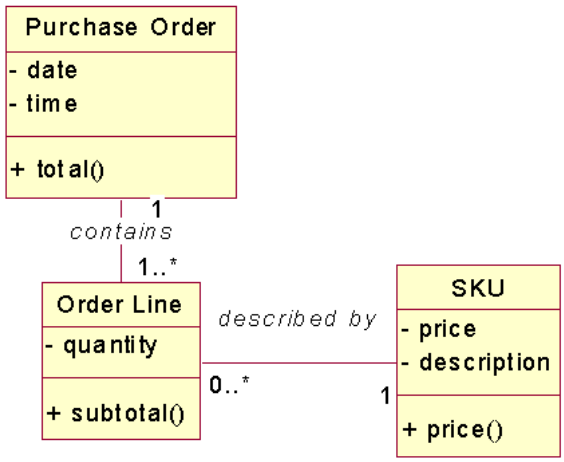


Figure 57 - Behaviours allocated by observing the expert pattern

This leads to the following collaboration diagram:

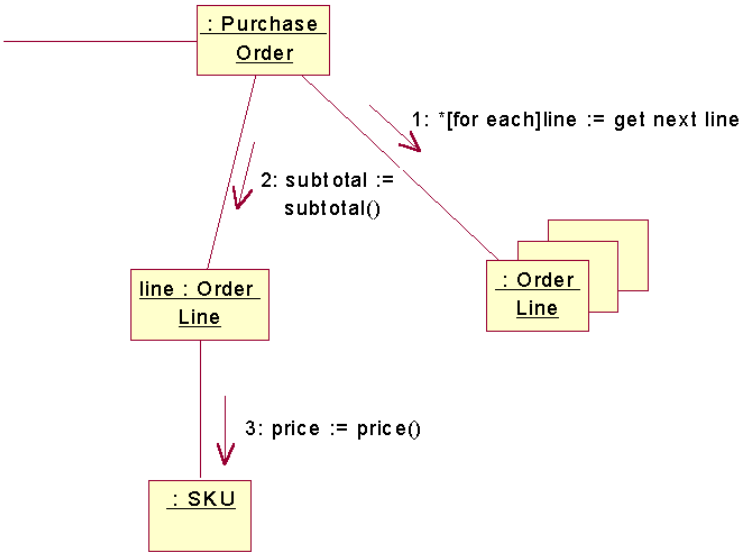


Figure 58 - three classes of objects collaborating to provide the total cost of a purchase order

Grasp 2 : Creator

The Creator pattern is a specific application of the Expert pattern. It asks the question "who should be responsible for creating instances of a particular class?"

The answer is that Class A should be responsible for creating objects from Class B if:

- A Contains B Objects
- A *closely uses* B Objects
- A *has the initialising data* that will be passed to B Objects

For example, lets return to the purchase order example. Let's say that a new purchase order has been created. Which class should be responsible for creating the corresponding purchase order lines?

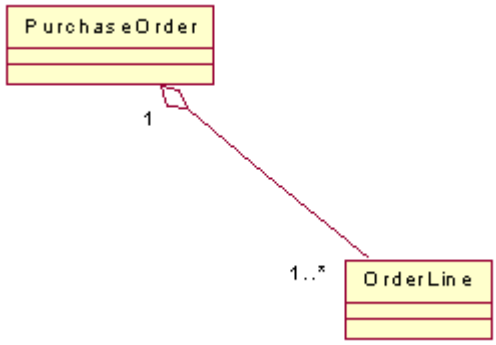


Figure 59 - Which class should create purchase orders?

The solution is that, as a purchase *contains* purchase order lines, then the purchase order class (and only that class) should be responsible for creating order lines.

Here is the collaboration diagram for this situation:

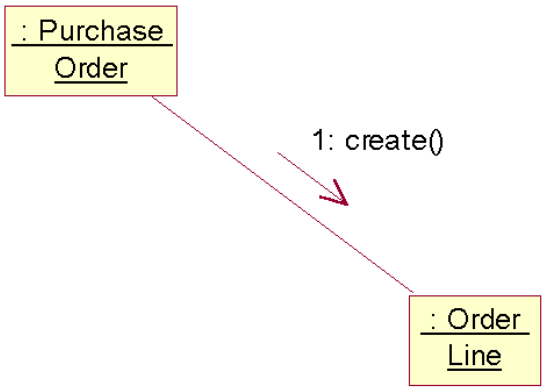


Figure 60 - The Purchase Order creating Order Lines

Grasp 3 : High Cohesion

It is extremely important to ensure that the responsibilities of each class are focussed. In a good object oriented design, each class should not do too much work. A sign of a good OO design is one where every class has only a small number of methods.

Consider the following example. In the design for a Lift Management system, the following class has been designed:

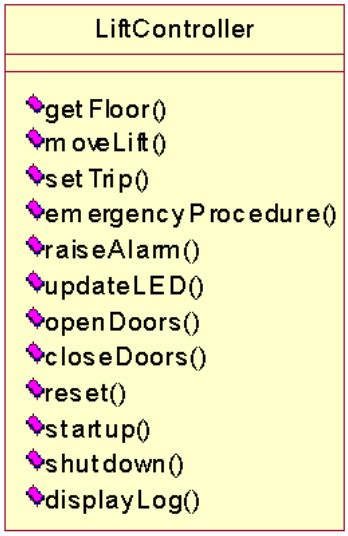


Figure 61 - A class from the Lift Management System

Is this a good design? Well, the class obviously does a lot of work – raising alarms, starting up/shutting down, moving the lift and updating the display indicator. This is a bad design because the class is not cohesive.

This class would be difficult to maintain, as it isn't obviously clear what the class is supposed to be doing.

The rule of thumb to follow when building classes is that each class should capture only one **key abstraction** – in other words, the class should represent one "thing" from the real world.

Our Lift controller is trying to model at least three separate key abstractions – an Alarm, the lift Doors and the Fault Log. So a better design is to break the Lift Controller into separate classes.

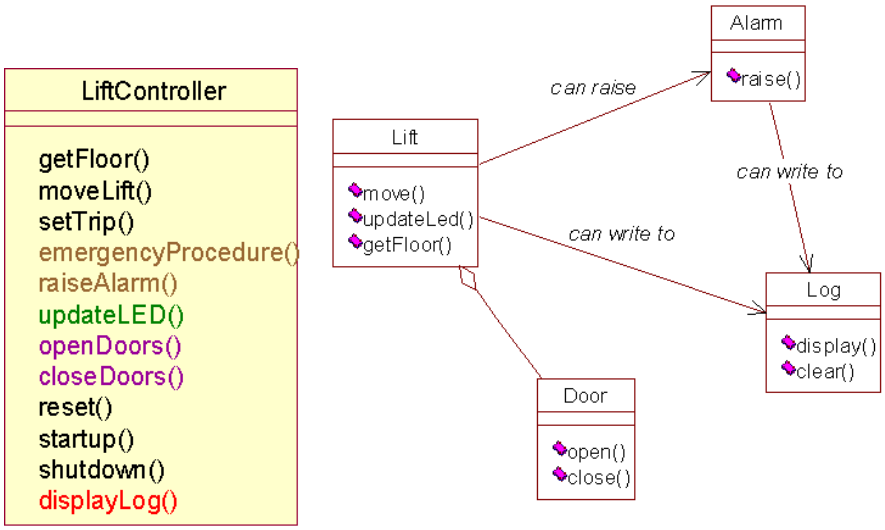


Figure 62 - The Lift Controller class modelled as four separate, more cohesive, classes

Grasp 4 : Low Coupling

Coupling is a measure of how dependent one class is on other classes. High Coupling leads to code that is difficult to change or maintain – a single to change to just one class could lead to changes "rippling" throughout the system.

The Collaboration Diagram provides an excellent means for spotting coupling, and as a result, high coupling can be also be spotted through the Class Diagram.

The following is an example extract from a Class Diagram that is showing clear signs of high coupling:

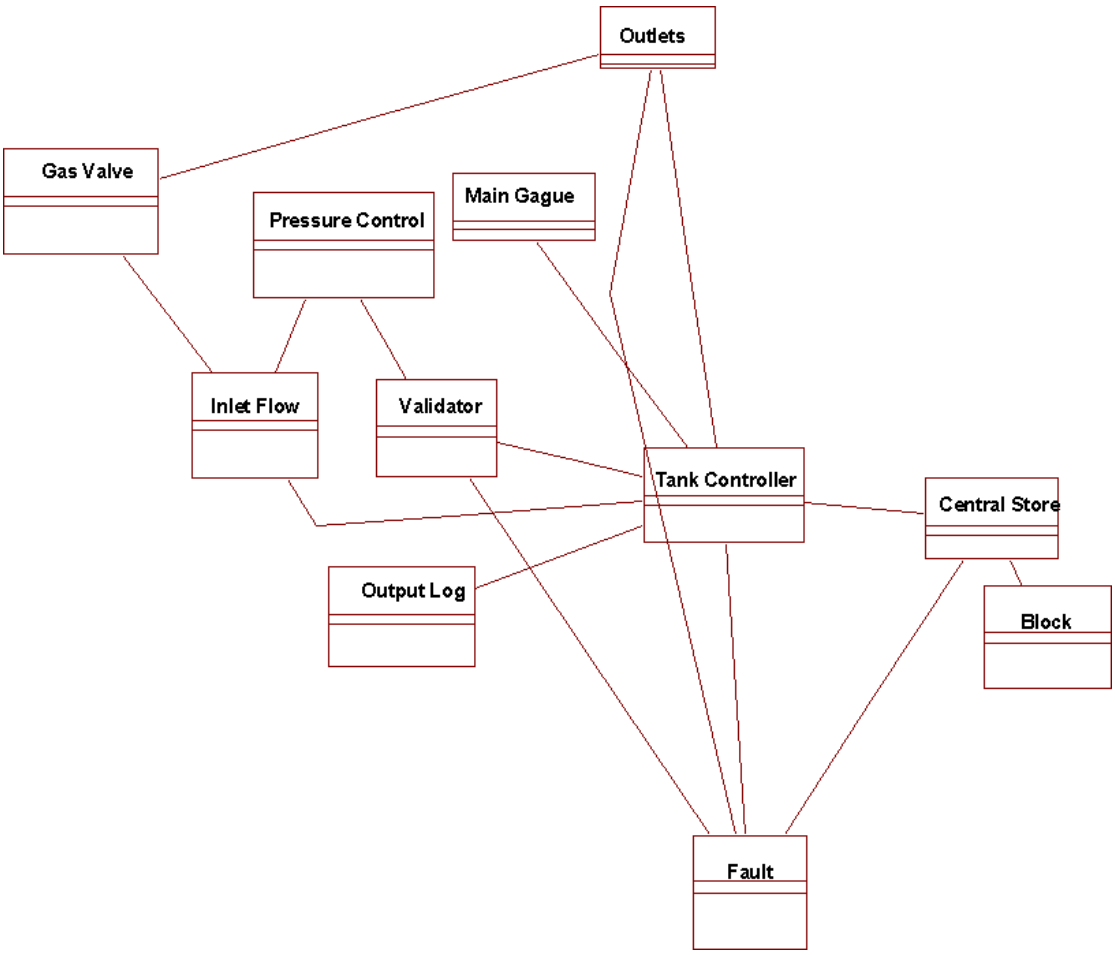


Figure 63 - High Coupling in a Class Diagram

Are all of the associations in Figure 63 really necessary? The designer of this system should ask some serious questions about the design. For example:

- Why is the Fault class associated to the Outlets class directly, when an indirect association exists through the Tank Controller Class? This link may have been put in place for performance reasons, which is fine, but more likely the link has appeared due to sloppiness in the Collaboration Modelling

- Why does Tank Controller have so many associations? This class is probably incohesive and doing too much work.

Following the conceptual model is an excellent way of reducing coupling. Only send a message from one class to another class if an association was identified at the conceptual modelling stage. This way, you are restricting yourself to introducing coupling only if the customer agreed that the concepts are coupled in real life.

If, at the Collaboration stage, you realise that you wish to send a message from one class to another and they are NOT associated on the conceptual model, then ask a very serious question about whether or not the coupling exists in the real world. Talking to the customer may help here - perhaps the association was overlooked when you built the conceptual model.

So the rule of thumb here is : **Keep coupling to a minimum - the conceptual model is an excellent source of advice on what that minimum should be. It is fine to raise the level of coupling, as long as you have thought very carefully about the consequences!**

Worked Example

Let us look at a Purhcase Ordering System. In the conceptual model, it was identified that Customers own Purchase Orders (because they raise them):

For the "Create Purchase Order" Use Case, which class should be responsible for creating new purchase orders? The Creator Pattern suggests that the Customer Class should be responsible:

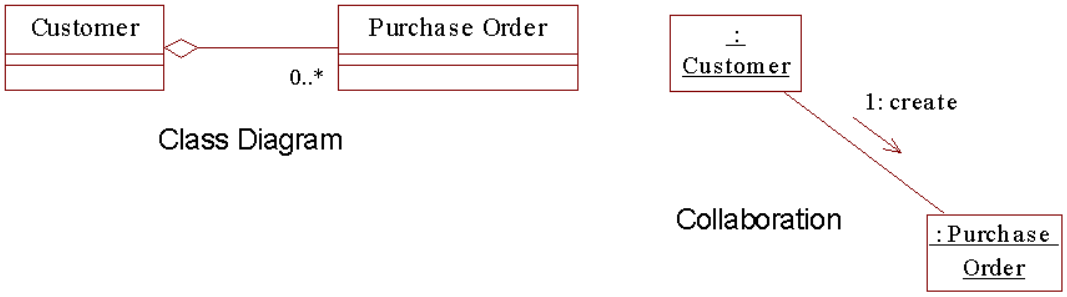


Figure 64 - Class Diagram and Collaboration for "Create Purchase Order"

So, we have now coupled Customer and Purchase Order together. That is fine, because they are coupled together in real life too.

Next, once the Purchase Order has been created, the Use Case needs to add lines to the Order. Who should be responsible for adding lines?

One approach is to let the Customer class do the work (after all, it does have the required initialising data - how many lines, what products, what quantities, etc).

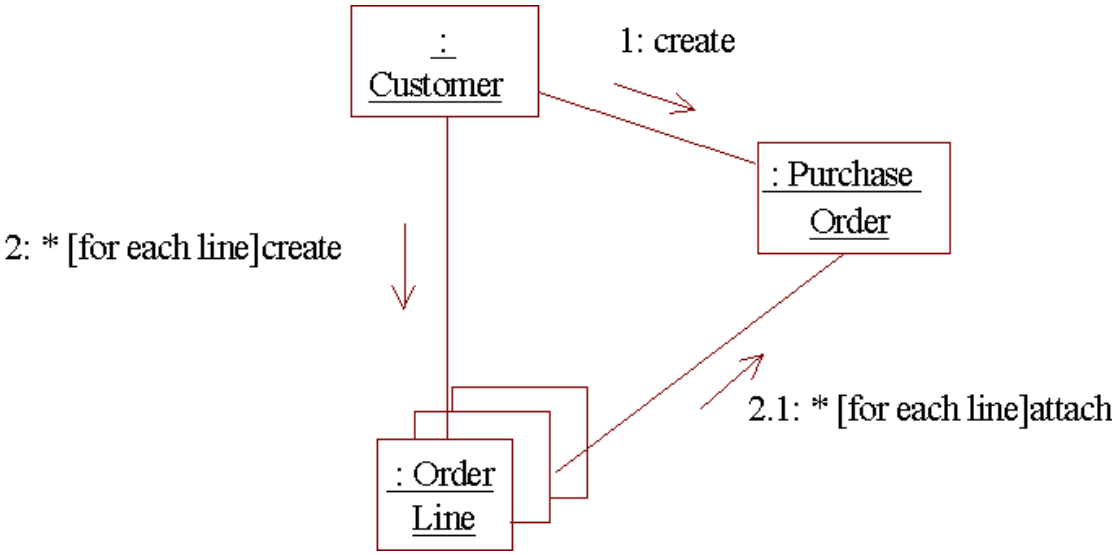


Figure 65 - First attempt. The customer objects creates the lines because it holds the initialising data

This approach has raised coupling artificially however. We now have all three class dependent upon each other, whereas if we had made the Purchase Order responsible for creating the lines, we would have the situation where Customers have no knowledge of Order Lines:

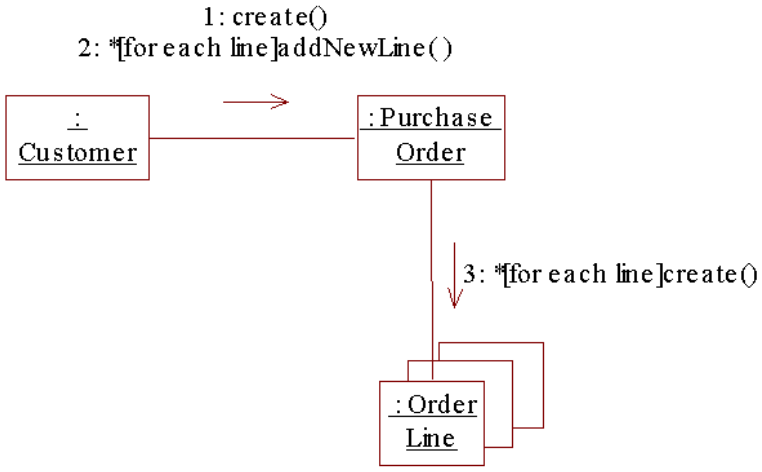


Figure 66 - Adding Lines, with reduced coupling

So now, if the implementation of the Order Line class changed for any reason, the only class affected would be the Purchase Order Class. All the coupling that exists on this design was coupling that was identified at the conceptual stage. Customers own Purchase Orders; Purchase Orders own Lines. So it makes sense that this coupling should exist!

The Law of Demeter

This Law, also known as *Don't Talk to Strangers* is an effective method of combating coupling. The Law states that any method of an object should only call methods belonging to:

- Itself
- Any parameters that were passed in to the method
- Any objects it created
- Any directly held component objects

Make your objects "shy" and coupling will be reduced!

Final Words on Coupling

Some more issues to consider:

- **Never** make an attribute of a class public - a public attribute instantly opens up the class to abuse (the exception is constants held by the class)
- Only provide get/set methods when strictly necessary
- Provide a minimum public interface (ie only make a method public if it has be accessed by the outside world)
- Don't let data flow around the system - ie minimise data passed as parameters
- Don't consider coupling in isolation - remember High Cohesion and Expert! A completely uncoupled system will have bloated classes doing too much work. This often manifests itself as a system with a few "active objects" that don't communicate.

Grasp 5 : Controller

The final GRASP pattern we will look at in this section is the controller pattern. Let's return to the bookmaking system, and return to the Place Bet Use Case. When we built the collaborations for this Use Case (see page 72), we realised that we hadn't really considered how the input would be entered by the user, and how the results are displayed to the user.

So, for example, we need to display the details of a race to the user. Which class should be responsible for satisfying this requirement?

Application of the **Expert** pattern suggests that as the details pertain to a Race, then the Race class should be the expert in displaying the relevant details.

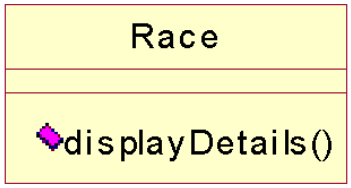


Figure 67 - Should "Race" be responsible for displaying its details?

This sounds ok at first, but actually, we have **violated** the expert pattern! The Race class should indeed be an expert at everything to do with races, but it must not be an expert on other matters – like Graphical User Interfaces!

In general, adding information about GUI's (and databases, or any other physical object) into our classes is poor design – imagine if we have five hundred classes in our system, and many of them read and write to the screen, perhaps to a text based console. What would happen if the requirement changed, and we wanted to replace the text-based screen for a windows-based GUI? We would have to trawl through all of our classes, and laboriously work out what needs to be altered.

It is much better to keep all of the classes from the conceptual model (I'll call them "Business Classes") pure, and remove all reference to GUI's, Databases and the like. But how our Race class can display the race details?

Solution – Controller Pattern

One possible solution is the use of a Controller Pattern. We can introduce a new class, and make it sit between the actor and the business classes.

The name of this controller class is usually called <UseCaseName>Handler. So in our case, we need a handler called "PlaceBetHandler".

The handler reads the commands from the user, and then decides which classes the messages should be directed to. The handler is the only class that will be allowed to read and write to the screen.

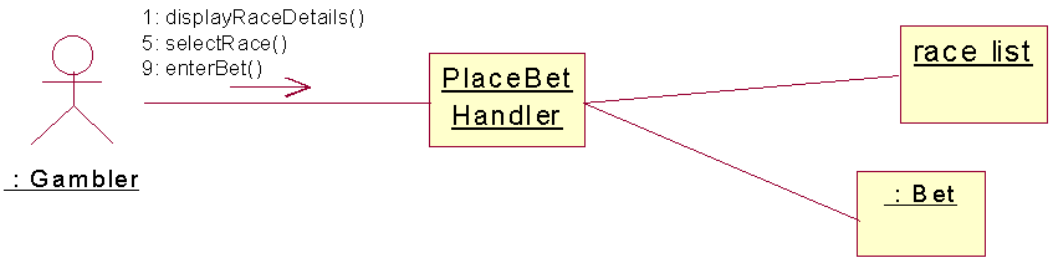


Figure 68 - Use Case Controller added to the design

In the event of us needing to replace the user interface, the only classes we have to modify are the controller classes.

Summary

In this chapter, we explored the "GRASP" Patterns. Careful application of the five GRASP patterns leads to more understandable, modifiable and robust Object Oriented Designs.

Chapter 15

Inheritance

One of the most well-known and most powerful concepts behind Object Orientation is **Inheritance**. In this chapter, we'll have a basic look at inheritance, when to apply it (and when not to apply it), and in particular we'll look at the UML notation for expressing inheritance.

Inheritance – the basics

Often, several classes in a design will share similar characteristics. In Object Orientation, we can factor out these common characteristics into a single class. We can then "inherit" from this single class, and build new classes from it. When we have inherited from a class, we are free to add new methods and attributes as the need arises.

Here's an example. Let's say we are modelling the attributes and behaviour of Dogs and People. This is what our classes might look like:

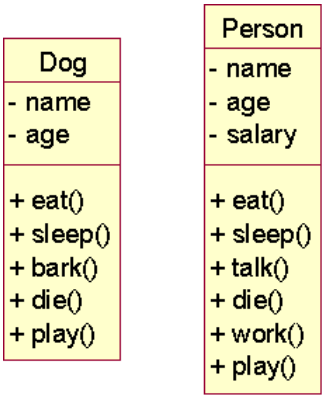


Figure 69 - Modelling Dogs and People

Although the two classes are different, the two classes also share a lot in common. Every Person has a name; so too does every Dog[14]. Similarly with Age. Some of the behaviours are common, such as Eat, Sleep and Die. Talk, however, is unique to the Person class.

If we decided to add a new class to our design, such as "Parrot", it would be tedious to add all of the common attributes and methods to the Parrot class again. Instead, we can factor out all of the common behaviour and properties into a new class.

[14] Let's assume we are modelling *pet* dogs!

If we take out the attribute “Age”, and the behaviours “Eat”, “Sleep” and “Die”, we have attributes and behaviour which should be common amongst *all* animals. Therefore, we can build a new class called “Animal”.

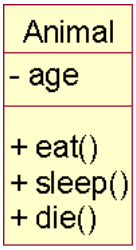


Figure 70 - A more general "Animal" class

So we have now built a more general class than our specific Dog, Person and Parrot class. This process is known as **generalisation**.

Now, when we need to create the Dog class, instead of starting from scratch, we inherit from the Animal class and merely add on the attributes and methods that we need for our specific class.

The following diagram illustrates this in the UML. Note that in the new Dog class we don’t include the old methods and attributes – they are implicit.

The class we started from is called the **base class** (sometimes called the Superclass). The class we have created from it is called the **derived class** (sometime the Subclass).

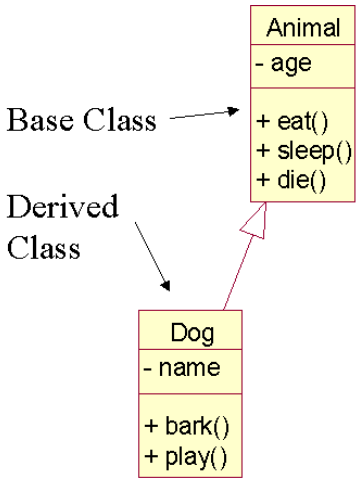


Figure 71 - Creating a Dog class from the Animal Class

We can continue creating new derived classes from the same base class. To create our Person class, we inherit again, as follows:

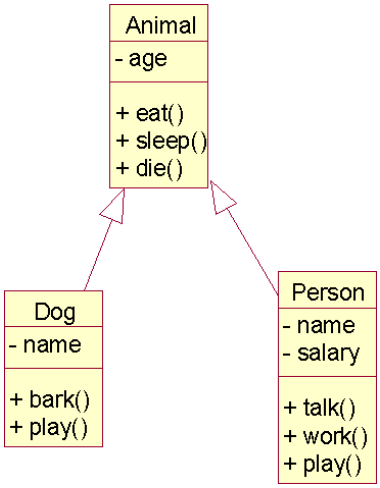


Figure 72 - Person derived from Animal

We can continue inheriting from classes to form a *class hierarchy*.

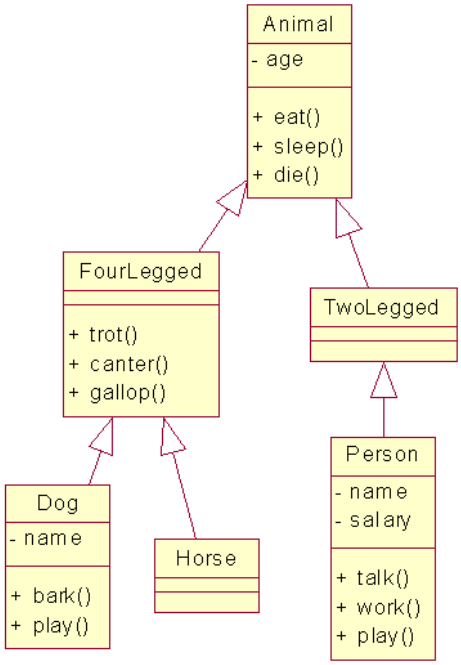


Figure 73 - A Class Hierarchy

Inheritance is White Box Reuse

A common mistake in object oriented designs is to over use inheritance. This leads to maintenance problems. Effectively, a derived class is tightly coupled to the base class – changes to the base class will result in changes to the derived class.

Also, when we use a derived class, we need to find out exactly what the base classes can do. This might mean trawling through a large hierarchy structure.

This problem is known as the *proliferation of classes*.

One common cause of proliferation is when inheritance is used when it shouldn't be. Follow the following rule-of-thumb:

Inheritance should only be used as a generalisation mechanism.

In other words, only use inheritance when the derived classes are a specialised type of the base class. There are two rules to help here:

- The **is-a-kind-of** rule
- The **100%** rule

The 100% Rule

All of the base class definition should apply to all of the derived classes. If this rule doesn't apply, then when you inherit, you are not creating specialised versions of the base class. Here's an example:

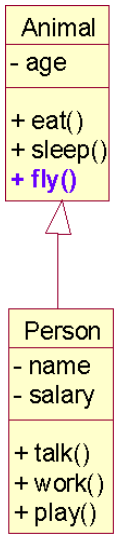


Figure 74 - Poor inheritance

In Figure 74, the fly() method should not be part of the Animal class. Not all animals can fly, so the derived class, Person, has an extraneous method associated with it.

Ignoring the 100% rule is an easy way to create maintenance problems.

Substitutability

In the previous example, why could we not simply **remove** the "fly" operation in the person class? That would solve the problem.

Methods **cannot** be removed in an inheritance hierarchy. This rule is enforced to ensure that the **Substitutability Principle** is upheld. We'll look at this in a little more detail shortly.

The Is-A-Kind-Of Rule

The Is-A-Kind-of rule is a simple way to test if your inheritance hierarchy is valid. The phrase "<derived class> is a <base class>" should make sense. For example "a dog is a kind of animal" makes sense.

Often, classes are derived from base classes when this rule does not apply, and again, maintenance problems are likely. Here's a worked example:

Example - Reusing queues through inheritance

Assume that we have built (in code) a working Queue class. The Queue class allows us to add items to the back of the queue and remove items from the front of the queue.

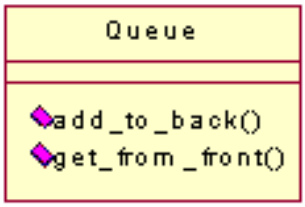


Figure 75 - The Queue Class

After a while, we decide that we need to build a new type of queue – a special kind of queue called a "Dequeue". This kind of queue allows the same operations as a queue but with the additional behaviour of allowing items to be added to the front of the queue and for items to be removed from the back. A kind of "two-way" queue.

To reuse the work we have already completed with the Queue, we can inherit from the Queue class.

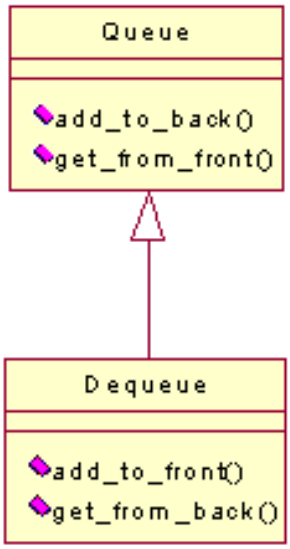


Figure 76 - Building a Dequeue through inheritance

Does this pass the Is-A-Kind-of and 100% tests?

- **100%** : Do all of the methods in Queue apply to Dequeue? The answer is yes, because all of these methods are required.
- **Is-A-Kind-Of** : Does this sentence make sense? "A Dequeue is-a-kind-of Queue". Yes, it does, because a dequeue is a special kind of queue.

So this inheritance was **valid**.

Now, let's go further. Let's say that we need to create a Stack. For a stack, we need to support the methods add_to_front() and remove_from_front().

Rather than writing the stack from scratch, we could simply inherit from the dequeue, as the dequeue provides both of these methods.

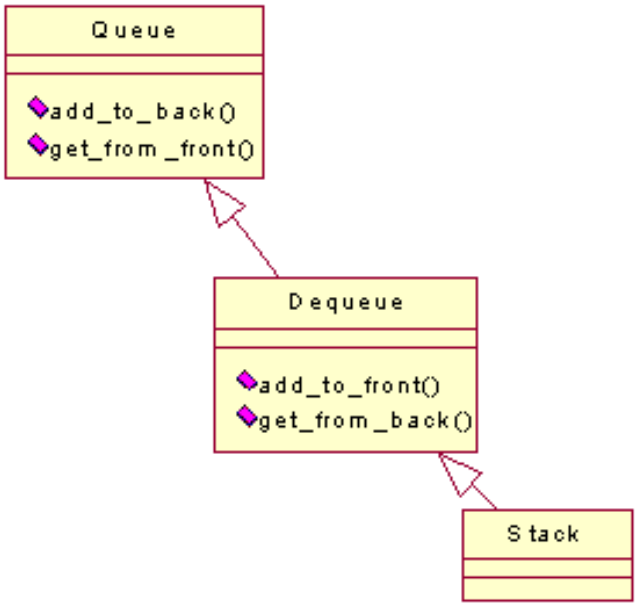


Figure 77 - Creating a Stack...no work needed!

We can feel very smug that we have re-used the Dequeue and created a stack with no further work required. Any code using the stack can add items to the front, and remove them from the front, and therefore we have a working stack.

We shouldn't feel too smug though. *This is a very poor design indeed*. Don't forget that the stack has also inherited the methods add_to_back() and remove_from_back() – these are two methods that are meaningless in a stack! So the stack fails the **100%** test. In addition, the stack fails the **Is-A-Kind-Of** test, because a stack isn't really a kind of Dequeue at all!

So we've created a maintenance problem – namely that any code using the stack can erroneously add items to the back of the stack! How do we get around this?? There really isn't any way, other than to bodge the stack class. Remember that we cannot remove methods from a class when we inherit.[15]

[15] The **substitutability** principle

The solution is to use aggregation rather than inheritance. We create a new class called **stack**, and include a Dequeue as one of its private attributes.

We can now provide the two public methods, add_to_front() and remove_from_front() as part of the stack class. The implementation of these methods are simple calls to the same methods contained in the Dequeue.

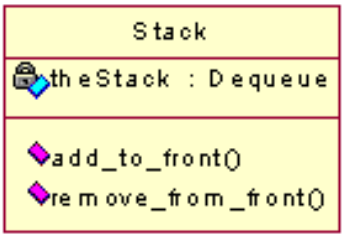


Figure 78 - The Stack class reusing the Dequeue efficiently

Now, users of the Stack class can only call the two public methods – the methods contained within the "hidden" Dequeue are private. This ensures that the Stack class has a highly cohesive interface, and will be much easier to maintain and understand.

Problems With Inheritance

Although Inheritance looks like a powerful mechanism to achieve reuse, Inheritance should be approached with care. Overusing inheritance can lead to very complex and difficult to understand hierarchies. This problem is known as the *proliferation of classes.* The problem is made even more acute when inheritance is used incorrectly (as described above). So ensure that inheritance is used sparingly, and make sure the 100% and is-a-kind-of rules apply.

In addition, Inheritance is White Box Reuse. Encapsulation between the base class and the derived class is quite weak - generally, a change to the base class could impact any derived classes, and certainly, any user of a class also needs to know about how the chain of parent classes above the class works.

The user of a class that is buried at the foot of an 13-class deep inheritance chain is going to have a real headache when working with that class - how many classes can *you* juggle in your head at once??

Visibility of Attributes

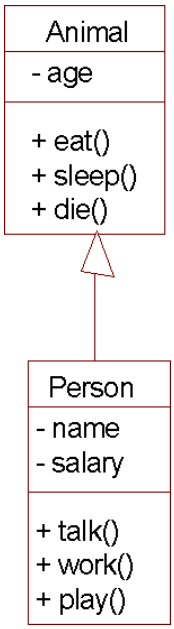


Figure 79 - Simple Inheritance

Consider the inheritance tree in the figure above. It is important to realise that the private members of the base class, Animal are **not** visible to the derived class, Person. So the methods talk(), work() and play() cannot access the age attribute.

This makes sense in a way, because you can argue that the methods that are only relevant to the Person class should not be able to fiddle with the attributes of the Animal class.

However, this restriction is sometimes too tight, and you need to allow a derived class to be able to "see" the attributes in the base class. Of course, we could make the attributes public, but that would break encapsulation and open up the attributes to the entire world. So OO provides a "middle ground", called **protected visibility**.

A protected member is still private to the outside world, but will remain visible to any derived classes. Most OO languages support protected visibility.

The UML notation for a protected member is shown below:

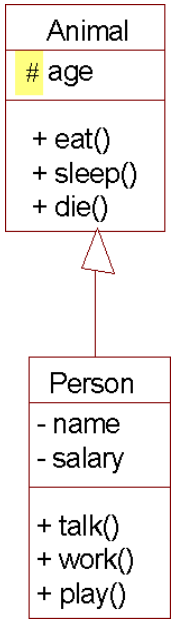


Figure 80 - The Age attribute is now protected, and is therefore visible to the Person class. It is still "private" as far as other classes are concerned.

Polymorphism

Derived classes can redefine the implementation of a method. For example, consider a class called “Transport”. One method contained in transport must be move(), because all transport must be able to move:

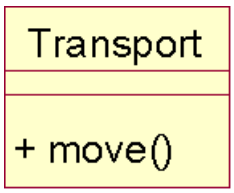


Figure 81 - The Transport Class

If we wanted to create a Boat and a Car class, we would certainly want to inherit from the Transport class, as all Boats can move, and all Cars can move:

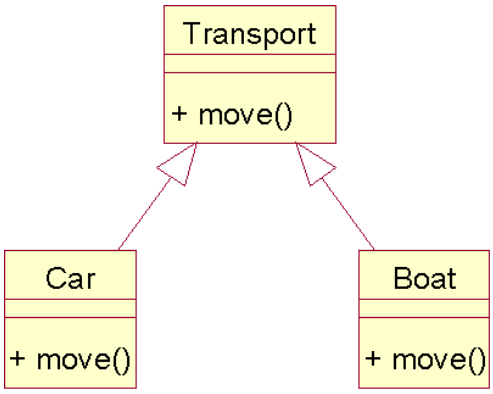


Figure 82 - Boat and Car derived from Transport. The Is-A-Kind-Of and 100% rules are satisfied.

However, cars and boats move in different ways. So we will probably want to implement the two methods in different ways. This is perfectly valid in Object Orientation, and is called **Polymorphism**.

Abstract Classes

Often in a design, we need to leave a method unimplemented, and defer its implementation further "down" the inheritance tree.

For example, back to the situation above. We added a method called "move()" to Transport. This is good design, because all transport needs to move. However, we cannot really implement this method, because Transport is describing a wide range of classes, each with different ways of moving.

What we can do is make the Transport class **abstract**. This means that some, or maybe all, of the methods are unimplemented.

When we derive the car class from Transport, we can then go ahead and implement the method, and similarly in boat.

The UML Syntax for an abstract class and an abstract method is to use italics, as follows:

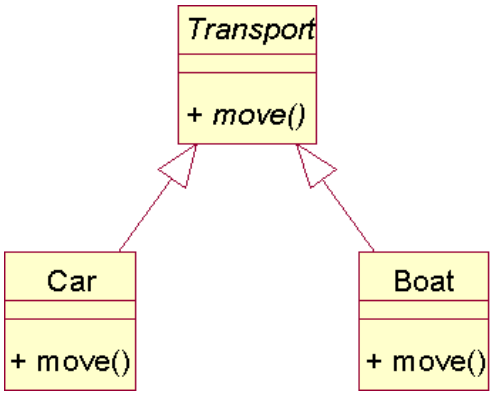


Figure 83 - We don't implement the move() method in Transport

This is one area of the UML that I don't think has been too well thought out. Italics are often difficult to spot on a diagram (and difficult to produce if you are writing the diagram on paper). The solution is to use a UML Stereotype on the abstract class as follows:

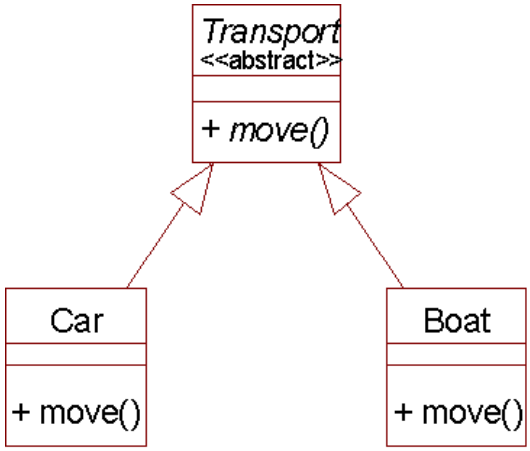


Figure 84 - Clarifying an Abstract class using a stereotype

The Power of Polymorphism

Polymorphism comes into its own when we apply the principle of substitutability. This principle says that any method we write that expects to "work on" a particular class, **can happily work on any derived class too**.

An example in code illustrates this point - I have written this snippet in Java-Pseudocode-ish(!), but the principle should be clear:

```
public void accelerate (Transport theTransport, int
acceleration)
{
    -- some code here
    theTransport.move();
    -- some more code
}

--
--
--
accelerate (myVauxhall);
accelerate (myHullFerry);
```

Figure 85 - Java code illustrating substituability

In this example, I have written a method called accelerate. It works using a parameter of type "Transport", and presumably speeds up the transport using the move() method.

Now, I can safely call this method, and pass it a Car object (because it is a subclass of Transport), and I can safely pass it a Boat object too. The function I have written simply doesn't care what the actual type of the object is, as long as it is derived from Transport.

This is extremely flexible. Not only have I written a general purpose method that can work on a whole range of different classes, I have also written a method that could in

future be used on a class that isn't even designed yet. Later, someone might create a new class called "Aeroplane", derived from Transport, and the accelerate() method would still work, and happily accept the new Aeroplane class, **without modification or recompilation!**

This is the reason why we cannot remove a method when we derive a new class. If we were allowed to do so, the aeroplane class could conceivably remove the move() method, and all the benefits listed above would be destroyed!

Summary

The Notation for Inheritance in UML is simple.

Classes can be arranged into an “inheritance hierarchy”

A sub-class *must* inherit all of the parent class’ public behaviour

Protected methods and attributes are also inherited

Polymorphism is an incredibly powerful tool to achieve code reuse

Chapter 16

System Architecture - Large and Complex Systems

So far in this book, we have considered relatively "small" systems. Generally, everything we have said so far would be easy to apply on a project with, say, 3 or 4 developers, with a handful of iterations lasting a couple of months each.

In this chapter, we'll have a look at some of the issues surrounding larger, more complex developments. Is the UML within an Iterative, Incremental Framework scaleable? And what else can the UML offer to help contain the complexity of such developments?

The UML Package Diagram

All UML Artefacts can be arranged into "UML Packages". A package is basically a logical container into which related elements can be placed - exactly like a folder or directory in an operating system.

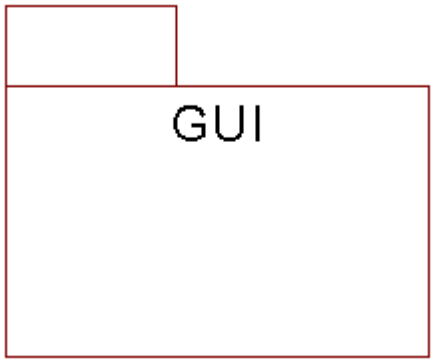


Figure 86 - Notation for a UML Package

In the above example, I have created a package called "GUI". I will probably be placing UML artefacts relating to Graphical User Interfaces inside the package.

We can display groups of packages, and the relationships between them, on the UML package diagram. The following is a simple example:

Figure 87 - Three UML packages,

In the example above, I have notated the classic "three tier model" of software development. Items inside the "Presentation" package are dependent upon the items inside the "Business" package.

Note that the diagram does not show what is actually inside the package. Therefore, the Package Diagram provides a very "high level" view of the system. However, many case tools allow the user to double-click on the package icon to "open up" the package and explore the contents.

A package can contain other packages, and therefore packages can be arranged into hierarchies, again, exactly as with directory structures in operating systems.

Elements Inside a Package

Any UML artefact can be placed inside a package. However, the most common use of a package is to group related classes together. Sometimes, the model is used to group related Use Cases together.

Within a UML package, the names of the elements must be unique. So, for example, the name of every class within the package must be unique. However, one major benefit of packages is that it doesn't matter if there is a name class between two elements from different packages. This provides the immediate advantage that if we have two teams working in parallel, Team A does not need to worry about the contents of Team B's package (as far as naming goes). Nameclashes will not occur!

Why Packaging?

So why do we bother with packaging? Well, by careful use of packages, we can:

- Group large systems into easier to manage subsystems
- Allow parallel iterative development

Also, if we design each package well and provide clear interfaces between the packages (more on this shortly), we stand a chance of achieving code reuse. Reusing classes has turned out to be somewhat difficult (in some sense, a class is quite small and a bit fiddly to reuse), whereas a well designed package can be turned into a solid, reusable software component. For example, a graphics package could be used in many different projects.

Some Packaging Heuristics

Let us assume for this section that we are using the package diagram to partition classes into easy to understand and maintain packages.

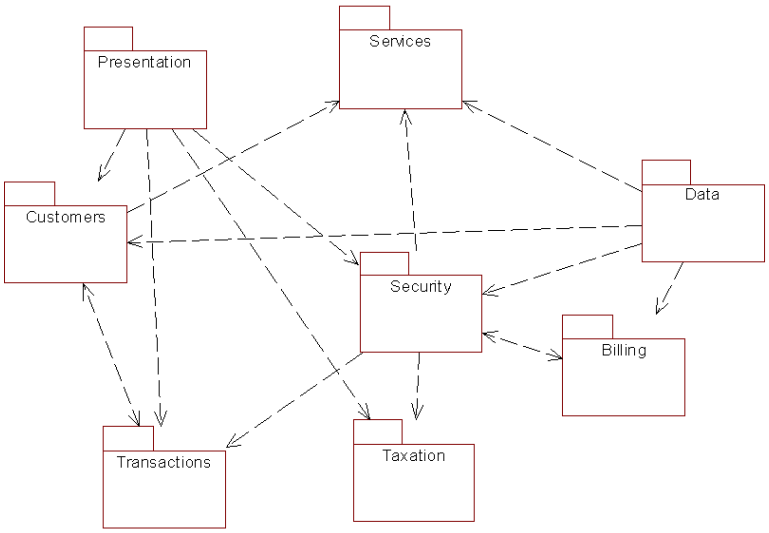


Figure 88 - A Well Designed Package Structure?

Several of the Heuristics from the GRASP chapter apply equally well to packaging. Three in particular stand out:

Expert

Which package should a class belong to? It should be obvious where each class belongs - if it isn't obvious then the package diagram is probably lacking.

High Cohesion

A package shouldn't do too much (or it will be difficult to understand, and certainly difficult to reuse).

Loose Coupling

Dependencies between packages should be kept to an absolute minimum. The diagram above is a made up example, but it looks fairly horrendous! Why is there so much cross package communication?

Handling Cross Package Communication

Assume we have two packages, each containing a number of classes.

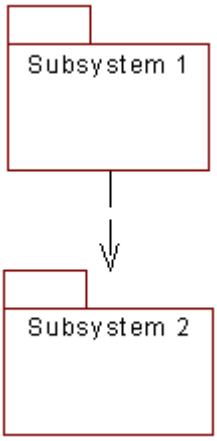


Figure 89 - Two Subsystems, modelled as UML Packages

From the dependency arrow, we can see that classes in the "Subsystem 1" package make calls to classes in the "Subsystem 2" package.

If we were to drill down and look inside the two packages, we might see something like the following (the attributes and operations have been removed for clarity):

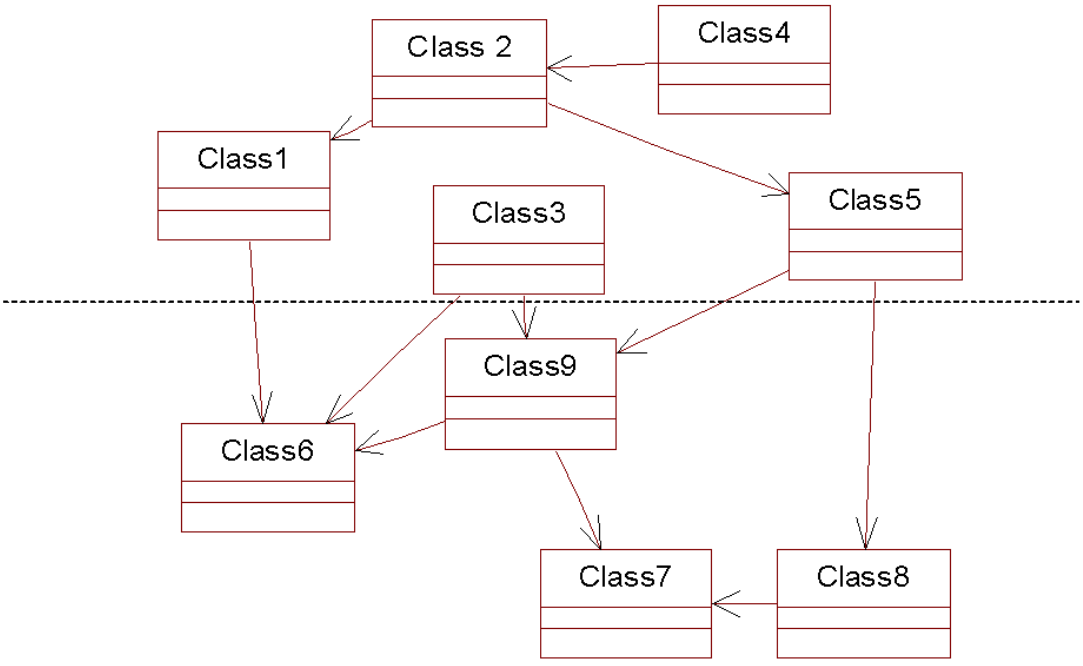


Figure 90 - Classes across two subsystems. The dashed line represents the subsystem boundary

Basically, we have a situation where any class from the "Subsystem 1" package can call any class from the "Subsystem 2" package. Is this a good design?

Clearly, this idea leaves something to be desired. What if we needed to remove the Subsystem 1 package and replace it with a new subsystem (let's say that we are

removing a terminal based user interface and replacing it with an all singing, all dancing graphical interface).

There would be a lot of work involved to understand the impact of the change. We would have to ensure that every class in the old subsystem is replaced with a corresponding class in the new subsystem. Very messy, and very inelegant. Luckily, there exists a design pattern called a Facade to help us with this problem.

The Facade Pattern

A better solution is to employ an extra class to act as a "go between" between the two subsystems. This type of class is called a Facade, and will provide, via its public interface, a collection of all of the public methods that the subsystem can support.

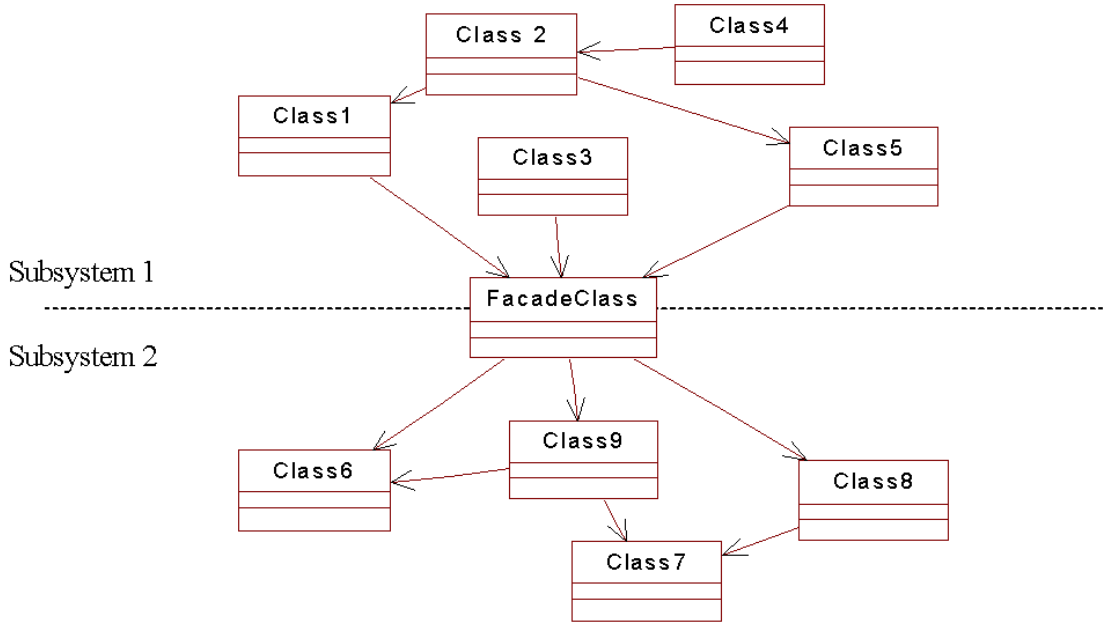


Figure 91 - The Facade Solution

Now, calls are not made across the subsystem boundary, but all calls are directed through the Facade. If one subsystem were to be replaced, then the only change required would be to update the Facade.

The Java language has excellent support for this concept. As well as the usual Private, Public and Protected class visibilities, Java provides a fourth level of protection called **Package Protection**. If a class is designated as package rather than public, only classes from the same package may access it. This is a very strong level of encapsulation - by making all classes except the Facades **package**, each team building the subsystem really can work independently of each other.

Architecture-Centric Development

The Rational Unified Process strongly emphasises the concept of Architecture-centric development. Essentially, this means that the system is planned as a collection of Subsystems from a very early stage in the project development.

By creating a group of small, easy to manage subsystems, small development teams (maybe of just 3 or 4 people) can be allocated to each subsystem, and, as much as possible, can work in parallel, independent of each other.

Clearly, this is far easier said than done. To underline the importance of this architecture activity, a **full time** architecture team is appointed (this could be a single person). This team is charged with the management of the architectural model - they would own and maintain everything related to the high level "big picture" of the system.

In other words, this team would own the package diagram. In addition, the architecture team would also own and control the interfaces (the Facades) between the subsystems. Clearly, as the project progress, changes will need to be made to the Facades, but those changes must be performed by the central architecture team, and not the developers working on individual subsystems.

As the architecture team maintain a constant "high level" view of the system, they are best placed to understand the impact changes to the interfaces between subsystems might have.

Example

For a major command and control system, the architecture team make a first cut of the system architecture by identifying the major areas of functionality to be offered by the system. They produce the following package diagram:

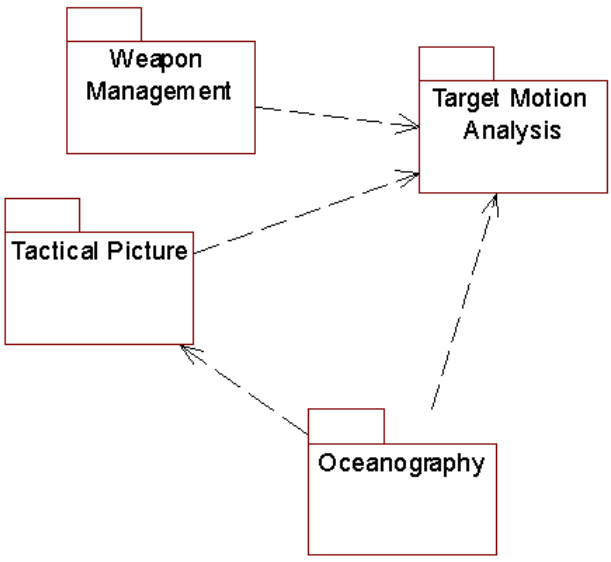


Figure 92 - First Cut Subsystem Plan using a UML package diagram

Note that the architecture is not set in stone - the architecture team will evolve and expand the architecture as the project progresses, to contain the complexity of each subsystem.

The team would continue setting up subsystems until the size of each subsystem is not too complex, and is easy to manage.

Use Cases may well then be built for each subsystem. Each subsystem is treated as a system in its own right, exactly as we did in the early stages of the book:

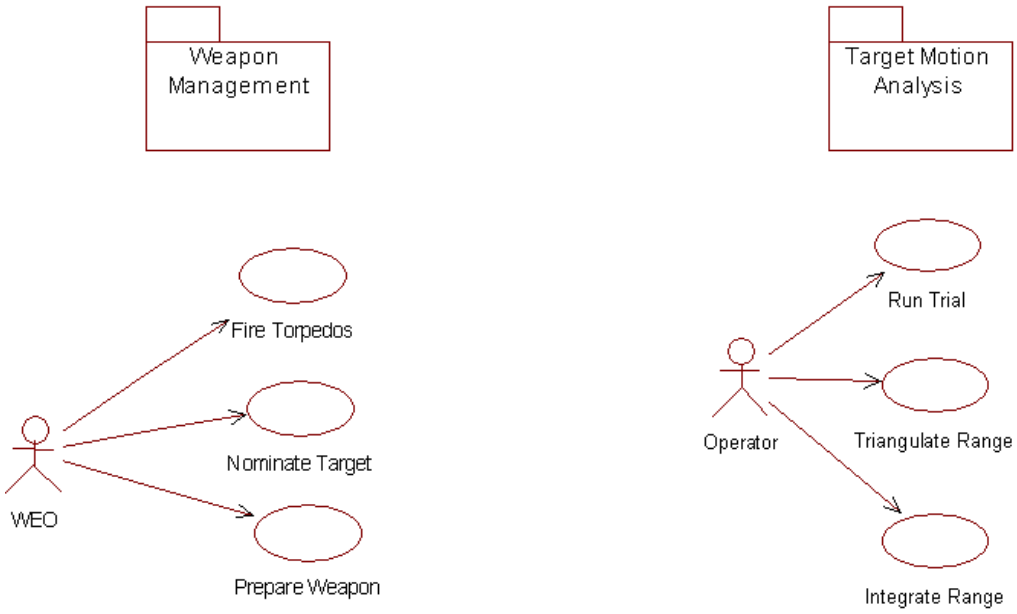


Figure 93 - Parallel Use Case Modelling

Handling Large Use Cases

Another problem with such large scale development is that these "first cut" use cases identified at the Elaboration phase may well be far too big to develop in a single iteration. The solution is **not** to make the iterations longer (this would cause complexity to rise again). Rather, the solution is break the Use Case into a series of easier to manage "versions".

For example, the "Fire Torpedoes" Use Case pictured above is identified, after the elaboration phase, to be a particularly large and difficult Use Case. The Use Case is therefore divided into separate versions, as follows:

- Version 1 - allows the opening of bow caps
- Version 2 - allows interlocks to be set
- Version 3 - allows the discharge of weapons

The aim is to ensure that each version is easy to understand, and achievable in a single iteration. So the Fire Torpedoes Use Case would take three iterations to complete.

The Construction Phase

The construction phase carries on as described in earlier chapters, but with each subsystem being developed, iteratively, by separate teams, working in parallel and as independently as possible.

At the end of each iteration, a phase of integration testing will take place, where the interfaces across subsystems are tested.

Summary

This chapter looked at some of the issues surrounding large scale system development. It is clear that although the UML is designed to be scaleable, transferring the Iterative Incremental Framework to large projects is far from a simple exercise.

The best approach at the moment seems to be the Architecture Centric approach proposed by Rational Corp:

- Define subsystems from an early stage
- Keep complexity as manageable as possible
- Iterate in parallel but don't hack interfaces
- Appoint a central architecture team

The package model provided by the UML provides a way of containing the large complexity, and this model should be owned by the architecture team.

More reading : The Rational Website at www.rational.com provides several interesting whitepapers on scalability issues such as multi-site working and systems requiring multiple variants. In addition, reference [1] is an excellent introduction to the Rational Unified Process and how the architecture centric approach can help contain complexity.

Chapter 17

Modelling States

After taking a break to consider Inheritance and System Architecture, we are now going to return to the design stage of the construction phase and consider state modelling.

State Diagrams allow us to model the possible states that an object can be in. The model allows us to capture the significant events that can act on the object, and also the effect of the events.

These models have many applications, but perhaps the strongest application is to ensure that odd, illegal events cannot happen in our system.

The example given in the introductory chapter of the book (page 31) talks about a situation that seems to happen an awful lot, if local newspapers are anything to go by - a gas bill is sent to a customer who died five years ago!

Carefully written state diagrams should prevent these kind of erroneous events occurring.

Example Statechart

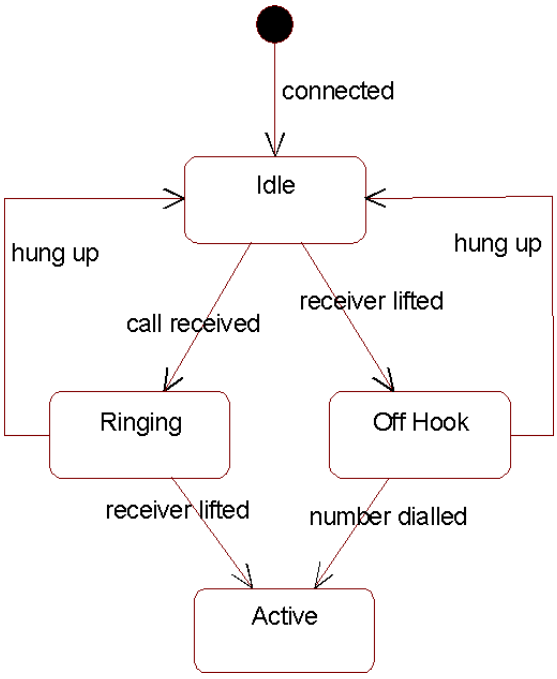


Figure 94 - Example Statechart; Telephone

We will look at the syntax of this diagram in detail shortly, but the basics of the diagram should be obvious. The sequence of events that can occur to the telephone are shown, and the states that the telephone can be in are also shown.

For example, from being idle, the telephone can either go to being "Off the Hook" (if the receiver is lifted), or the telephone can go to "Ringing" (if a call is received).

State Diagram Syntax

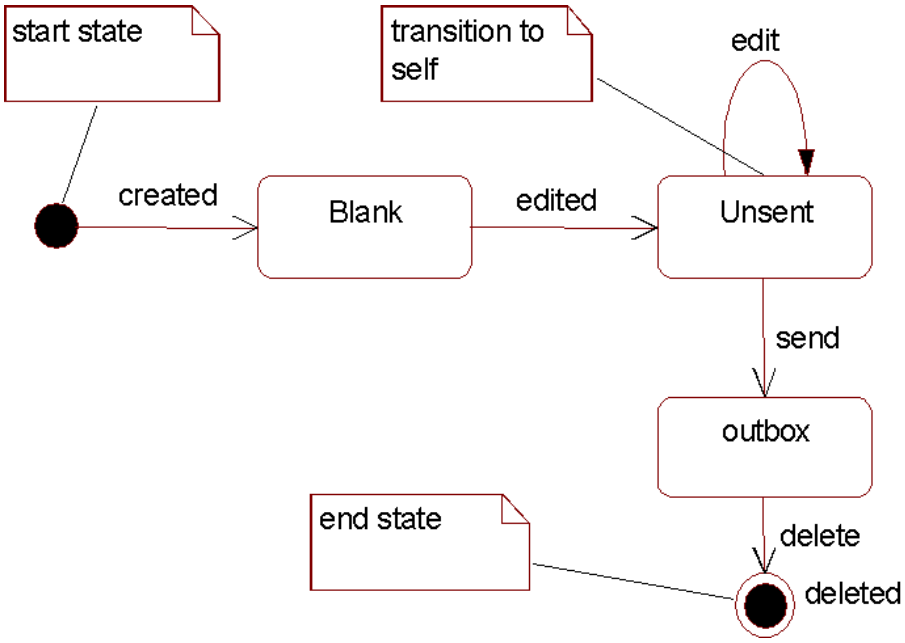


Figure 95 - Syntax of the State Diagram - an E-Mail example

The diagram above shows most of the state diagram syntax. The object will have a start state (the filled circle), describing the state of the object at the point of creation. Most objects have an end state (the "bullseye"), describing the event that happens to destory the object.

Some events cause a state transition that causes the object to remain in the same state. In the example above, the e-mail can receive an "edit" event only if the status of the object is "unsent". But the event does not cause a state change. This is a useful syntax to illustrate that the "edit" event can not happen in any of the other states.

Substates

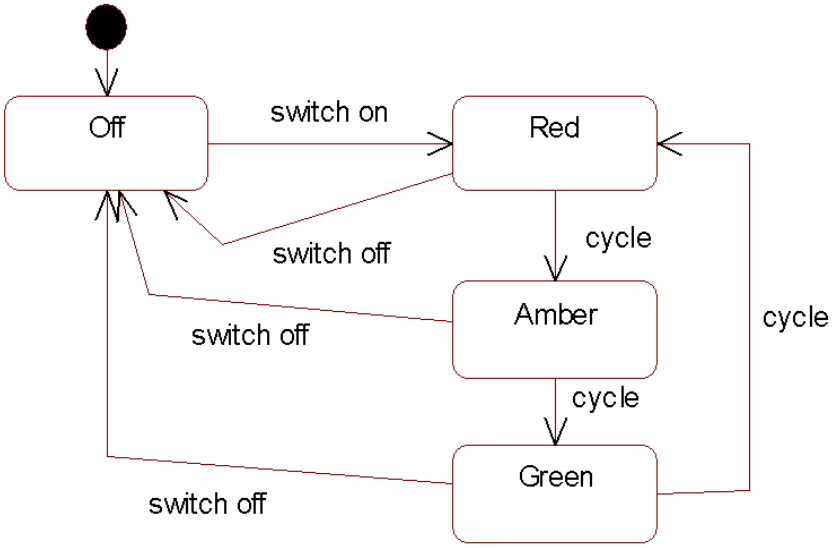


Figure 96 - Messy State Model

Sometimes, we require a model that describes states within states. The above statechart is perfectly valid (describing a traffic light object's states), but it is hardly elegant. Essentially, it can be switched off at any time, and it is this set of events that is causing the mess.

There is a "superstate" present in this model. The traffic light can be either "On" or "Off". When it is in the "On" state, it can be in a series of substates of "Red", "Amber" or "Green". The UML provides for this by allowing "nesting" of states:

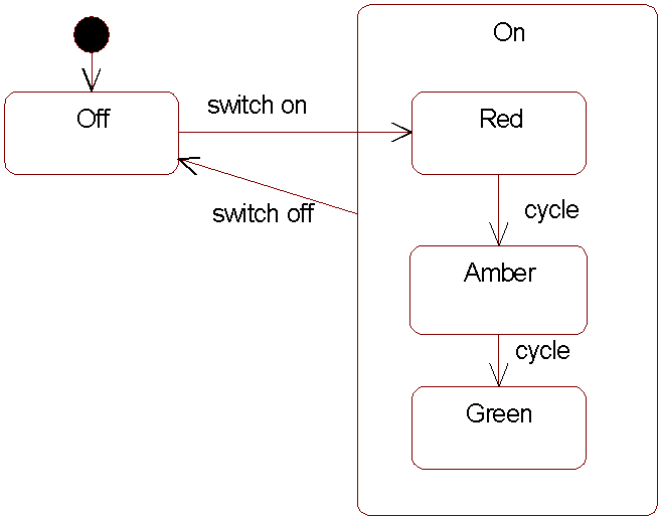


Figure 97 - Simpler state model using substates

Note that in the diagram above, the small arrow pointing in to the "red" state indicates that this is the default state - on commencement of the "on" state, the light will be set to "Red".

Entry/Exit Events

Sometimes it is useful to capture any actions that need to take place when a state transition takes place. The following notation allows for this:

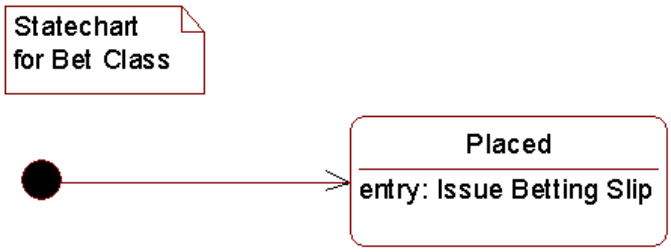


Figure 98 - Here, we need to issue a betting slip when the state change occurs

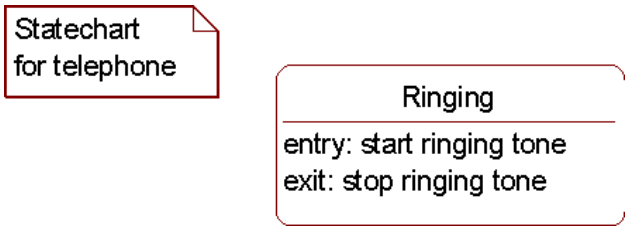


Figure 99 - Here, the ring tone starts on entry to the state - the ring tone stops on exit

Send Events

The above notation is useful when you need to comment that a particular action needs to take place. Slightly more formally, we can tie this approach to the idea of objects and collaboration. If a state transition implies that a message has to be sent to another object, then the following notation is used (alongside the entry or exit box):

```
^object.method (parameters)
```

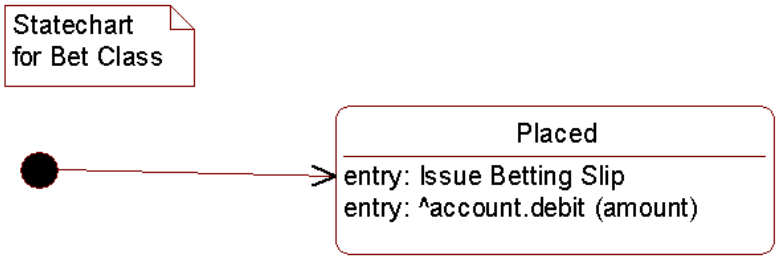


Figure 100 - formal notation indicating that a message must be sent on the state transition

Guards

Sometimes we need to insist that a state transition is possible only if a particular condition is true. This is achieved by placing a condition in square brackets as follows:

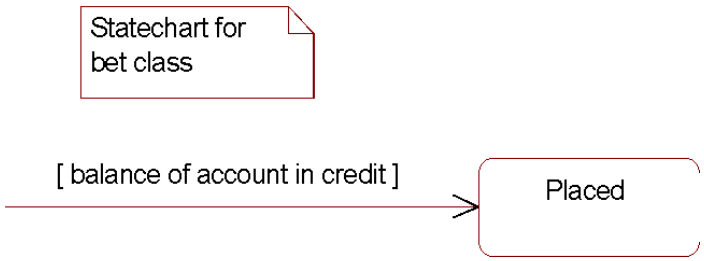


Figure 101 - Here, the transition to the "Placed" state can only occur if the balance of the account is in credit

History States

Finally, returning to substates briefly, it is possible to notate that if the superstate is interrupted in some way, when the superstate is resumed, the state will be remembered.

Take the following example. A criminal investigation starts in the "pending" state. Once it switches to the "Being Actioned" state, it can be in a number of substates.

However, at random intervals, the case can be audited. During an audit, the investigation is briefly suspended. Once the audit is complete, the investigation must resume at the state from where it was before the audit.

The simple "history notation" (a "H" in a circle) allows us to do this, as follows:

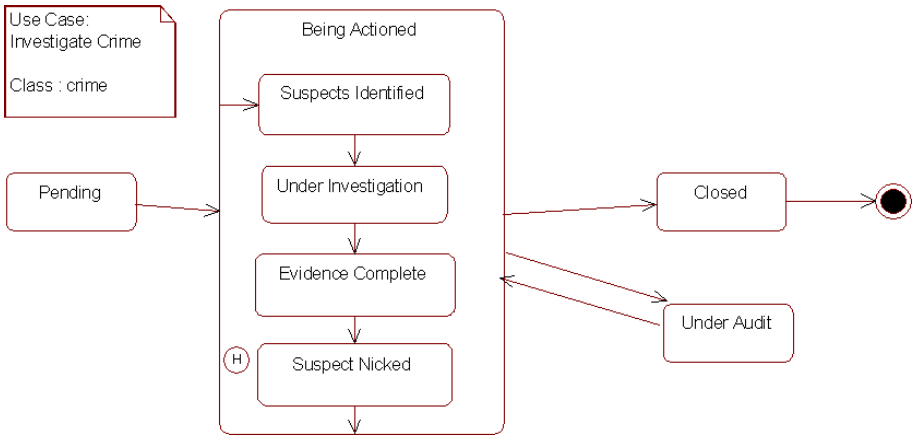


Figure 102 - History State

Other Uses for State Diagrams

Although the most obvious use for these diagrams is to track the state of an object, in fact, statecharts can be used for any state-based element of the system. Use Cases are a clear candidate (for example, a use might only be able to proceed if the user has logged on).

Even the state of the entire system can be modelled using the statechart - this is clearly a valuable model for the "central architecture team" in a large development.

Summary

In this chapter, we looked at State Transition Diagrams.

We saw:

- The syntax of the diagram
- How to use Substates
- Entry and Exit Actions
- Send Events and Guards
- History States

Statecharts are quite simple to produce, but often require deep thought processes

Most commonly produced for Classes, but can be used for anything : Use Cases, entire Systems, etc

Chapter 18

Transition to Code

This brief section describes some of the issues surrounding the move from the model to code. For the examples, we'll use Java, but the Java is very simple and can be easily applied to any modern Object Oriented language.

Synchronising Artifacts

One of the key problems of design and coding is keeping the model in line with the code.

Some projects will want to totally separate design from code. Here, the designs are built to be as complete as possible, and coding is considered a purely mechanical transformation process.

For some projects, the design models will be kept fairly loose, with some design decisions deferred until the coding stage.

Either way, the code is likely to "drift" away from the model to a lesser or greater extent. How do we cope with this?

One approach is to add an extra stage to each iteration - Synchronising artifacts. Here, the models are altered to reflect the design decisions that were made during coding in the previous iteration.

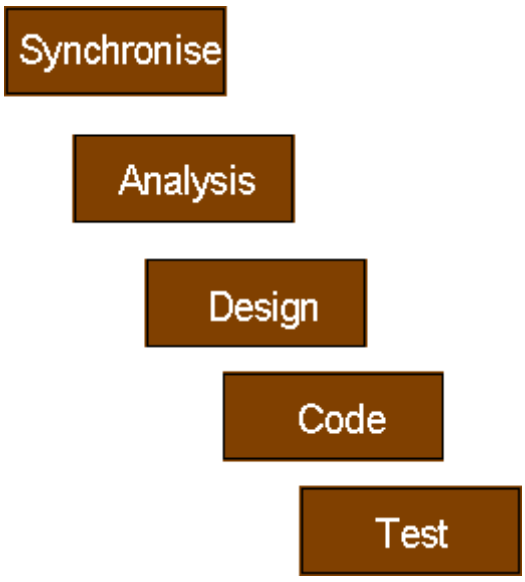


Figure 103 - Extra stage in the waterfall - synchronisation

Clearly, this is a far from simple solution, as often, major changes will have been made. However, it is workable as long as the iterations are short and the complexity of each one is manageable. Well, that's what we've been aiming for all along!!

Some CASE tools allow "reverse engineering" - that is, the generation of a model from code. This could be a help with synchronising - at the end of iteration 1, regenerate the model from the code, and then work from this new model for iteration 2 (and repeat the process). Having said that, the technology of reverse engineering is far from advanced, so this may not suit all projects!

Mapping Designs to Code

Your code's class definitions will be derived from the Design Class Diagram. The method definitions will come largely from the Collaboration Diagrams, but extra help will come from the Use Case descriptions (for the extra detail, particularly on exception/alternate flows) and the State Charts (again, for trapping error conditions).

Here's an example class, and what the code might look like:

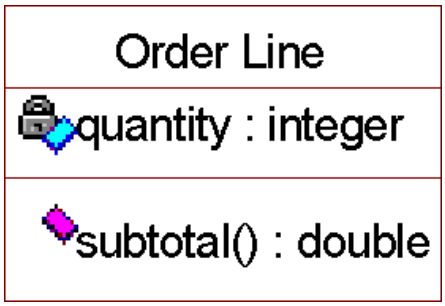


Figure 104 - The Order Line class, with a couple of example members

The resulting code would end up looking something like this (following a mechanical conversion process):

```
public class OrderLine
{
   public OrderLine(int qty, SKU product)
   {
      // constructor
   }
   public double subtotal()
   {
      // method definition
   }

   private int quantity;
}
```

Figure 105 - Sample Order Line Code

Note that in the code above, I have added a constructor. We omitted the create() methods from the Class Diagram (as it seems to be a convention these days), so this needed to be added.

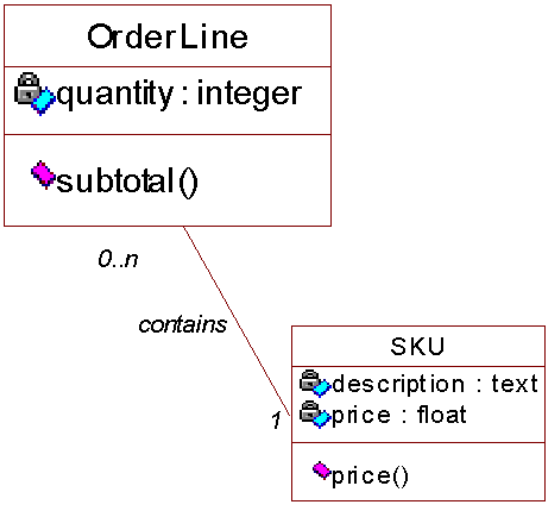


Figure 106 - The aggregation of Order Lines and SKU's

An order line contains a reference to a single SKU, so we also need to add this to the class code:

```
public class OrderLine
{
   public OrderLine(int qty, SKU product);
   public float subtotal();

   private int quantity;
   private SKU SKUOrdered;
}
```

Figure 107 - Adding the reference attribute (method blocks omitted for clarity)

What if a class needs to hold a list of references to another class? A good example is the relationship between Purchase Orders and Purchase Order Lines. A Purchase Order "owns" a list of lines, as in the following UML:

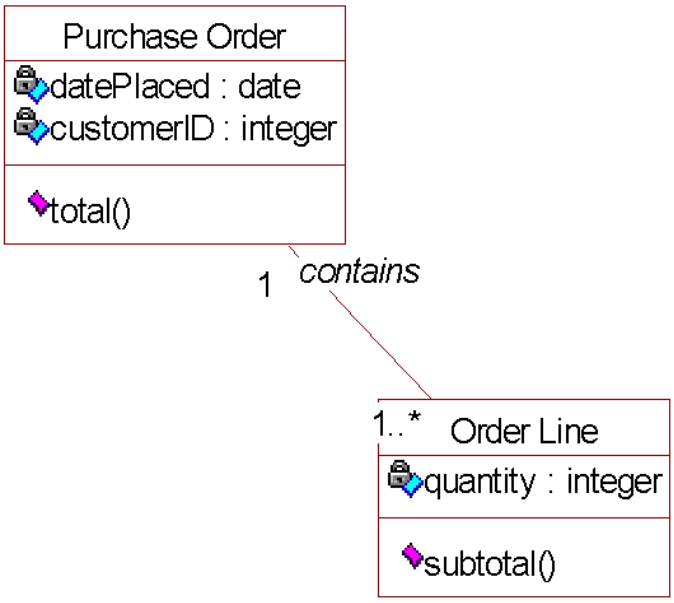


Figure 108 - A Purchase Order holds a list of Order Lines

The actual implementation of this depends upon the specific requirement (for example, should the list be ordered, is performance an issue, etc), but assuming we need a simple array, the following code will suffice:

```
public class PurchaseOrder
{
   public float total();

   private date datePlaced;
   private int  customerID;
   private Vector OrderLineList;
}
```

Figure 109 - Adding a list of references

Initialising the list would be the job of the constructor. For non Java and C++ coders, a Vector is simply an array that can be dynamically resized. Depending on the requirement, a bog standard array would have worked too.

Defining the Methods

The collaboration diagram is a large input into the method definitions.

The following worked example describes the "get total" method for the Purchase Order. This method returns the total cost of all of the lines in the order:

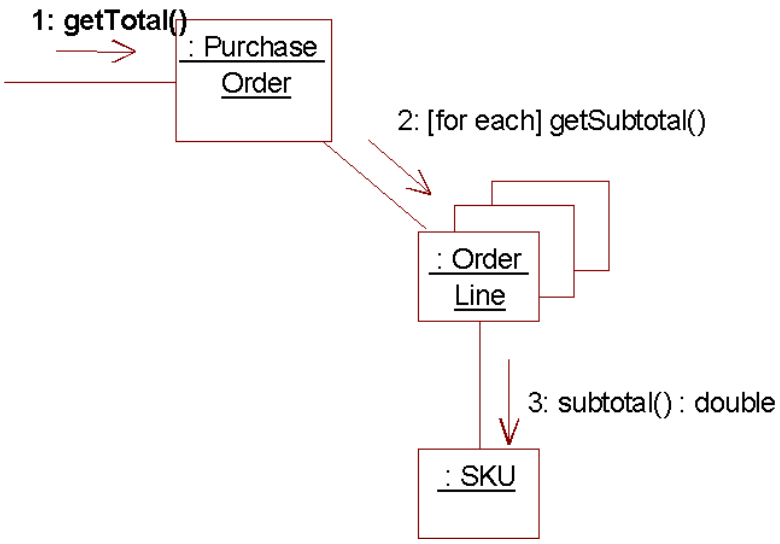


Figure 110 - "Get total" collaboration

Step 1

Clearly, we have a method called "getTotal()" in the purhcase order class:

```
public double getTotal()
{

}
```

Figure 111 - method definition in the Purchase Order Class

Step 2

The collaboration says that the purchase order class now polls through each line:

```
public double getTotal()
{
   double total;
    for (int x=0; x<orderLineList.size();x++)
    {
       // extract the OrderLine from the list
       theLine = (OrderLine)orderLineList.get(x);

       total += theLine).getSubtotal();
    }
    return total;
}
```

Figure 112 - code for getting the total, by polling all purchase order lines for the order.

Step 3

We have called a method called "getSubtotal()" in the OrderLine class. So this needs to be implemeted:

```
public double getSubtotal()
{
   return quantity * SKUOrdered.getPrice();
}
```

Figure 113 - implementation of getSubtotal()

Step 4

We have called a method called "getPrice()" in the SKU Class. This needs implementing and would be a simple method that returns the private data member.

Mapping Packages into Code

We stressed that building packages is an essential aspect of system architecture, but how do we map them into code?

In Java

If you are coding in Java, packages are supported directly. In fact, every single class in Java belongs to a package. The first line of a class declaration should tell Java in which package to place the class (if this is omitted, the class is placed in a "default" package).

So if the SKU class was in a package called "Stock", then the following class header would be valid:

```
package com.mycompany.stock;

class SKU
{ ...
```

Figure 114 - Placing classes in packages

Best of all, Java adds an extra level of visibility on top of the standard private, public and protected. Java includes **package** protection. A class can be declared as being visible only to the classes in the same package - and so can the methods inside a class. This provides excellent support for encapsulation within packages. By making all classes visible only to the packages they are contained in (except the facades), subsystems can truly be developed independently.

Sadly, the syntax for package protection in Java is rather poor. The notation is to simply declare a class with no **public, protected** or **private** preceding the class definition - exactly as in Figure 114.

In C++

There is no direct support for packages in C++, but recently the concept of a namespace was added to the language. This allows classes to be placed in separate logical partitions, to avoid name clashes between namespaces (so I could create two namespaces, say Stock and Orders, and have a class called SKU in both of them).

This provides some of the support of packages, but unfortunately it doesn't offer any protection via visibilities. A class in one namespace can access all of the public classes in another namespace.

The UML Component Model

This model shows a map of the physical, "hard", software components (as opposed to the logical view expressed by the package diagram).

Although the model will often be based on the logical package diagram, it can contain physical run time elements that weren't necessary at the design stage. For example, the following diagram shows an example logical model, followed by the eventual software physical model:

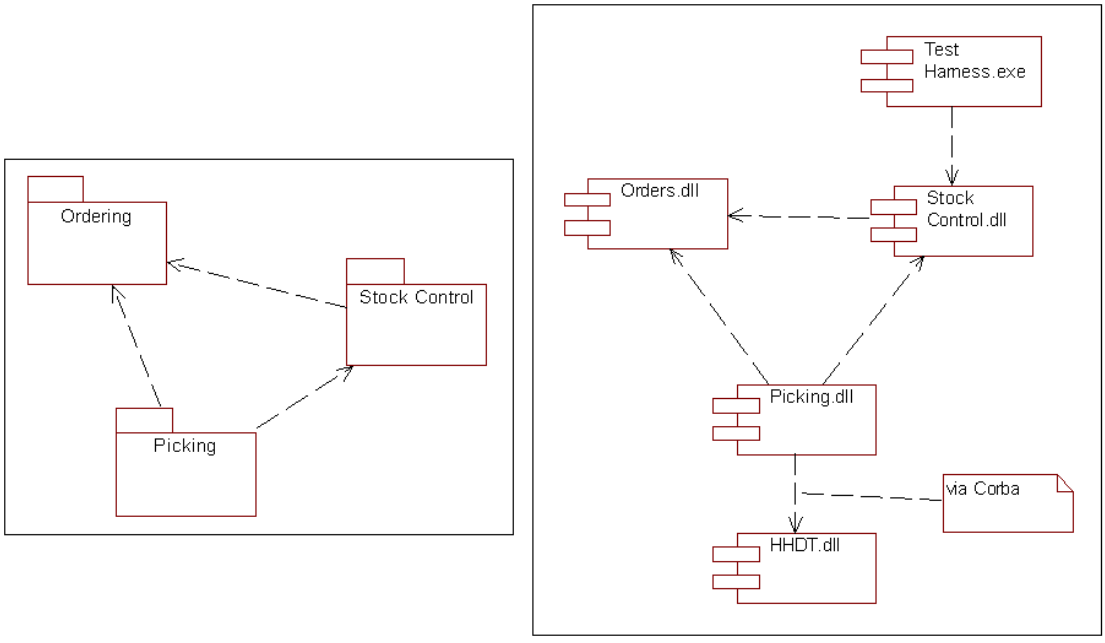


Figure 115 - the logical compared to the physical view

The Component Model is very simple. It works in the same way as the package diagram, showing elements and the dependencies between them. However, this time, the symbol is different, and each component can be any physical software entity (an executable file, a dynamic link library, an object file, a source file, or whatever).

Note that the Component Model is based heavily on the package diagram, but has added a .dll to handle the Terminal Input/Output, and has added a test harness executable.

Ada Components

Some extra component icons are available through Rational Rose that seem to be heavily influenced by the Ada language (presumably through the input of Grady Booch). These icons work in exactly the same way as the components above, but notate more specific software components:

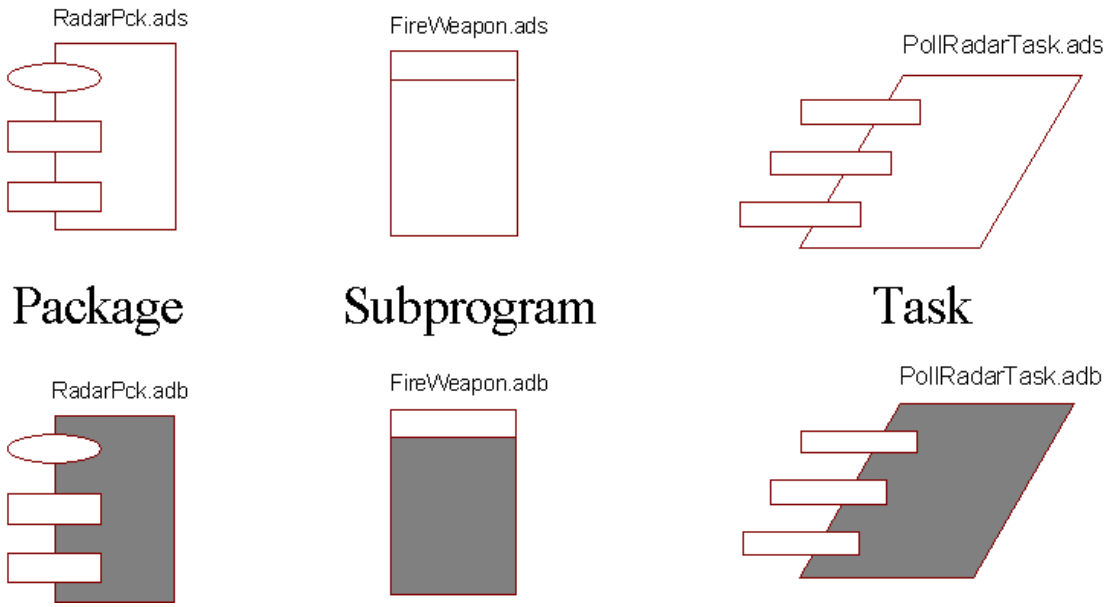


Figure 116 - Extra Components

For readers from a non Ada background, a Package (not to be confused with a UML package) is a collection of related procedures, functions and data (roughly the same as a class), a Subprogram is a procedure or function, and a Task is a subprogram that can run concurrently with other tasks.

These symbols may be of use to you even if you are not working in Ada - in particular the Task symbol is useful to denote that the software element is going to run in parallel with other tasks.

Summary

This chapter has described, in rough terms, the general process of converting the models into real code. We looked briefly at the issue of keeping the model synchronised with the code, and a couple of ideas on how to get around the problem.

We saw the component model. The model is not heavily used at present, but it is helpful in mapping the physical, real life software code and the dependencies between them.

The UML Applied Course CD shows how the Case Study followed on the course can be transformed into Java code - please feel free to explore it for more details.

Bibliography

[1] : Krutchten, Philippe. 2000 *The Rational Unified Process An Introduction Second Edition* Addison-Wesley

A brief introduction to the Rational Unified Process, and its relationship with the UML

[2] : Larman, Craig. 1998 *Applying UML and Patterns An Introduction to Object Oriented Analysis and Design* Prentice Hall

An excellent introduction to the UML, applied to real software development. Used as the basis for this course.

[3] : Schmuller, Joseph. 1999 *Teach Yourself UML in 24 Hours* Sams

A surprisingly comprehensive introduction to UML, including details of the metamodel. The first half concentrates on UML syntax, and the second half applies the UML (using a RUP-style process called GRAPPLE)

[4] : Collins, Tony. 1998 *Crash : Learning from the World's Worst Computer Disasters* Simon&Schuster

An entertaining collection of case studies exploring why so many software development projects fail

[5] : Kruchten, Phillipe 2000 *From Waterfall to Iterative Lifecycle - a tough transition for project managers* Rational Software Whitepaper – www.rational.com

An excellent, and short, description of the problems project managers will face on an iterative project

[6] : Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1995 *Design Patterns : Elements of Reusable Object Oriented Software* Addison-Wesley

The classic "Gang of Four" catalogue of several design patterns

[7] : Riel, Arthur 1996 *Object Oriented Design Heuristics* Addison-Wesley

Rules of Thumb for Object Oriented Designers

[8] : UML Distilled

Martin Fowler's pragmatic approach to applying UML on real software developments

[9] : Kulak, D., Guiney, E. 2000 *Use Cases : Requirements in Context* Addison-Wesley

An in depth treatment of requirement engineering, driven by Use Cases