



**UNIVERSITY
OF LONDON** | INTERNATIONAL
PROGRAMMES

Software engineering, algorithm design and analysis Volume 1

T. Blackwell

CO2226

2007

Undergraduate study in
Computing and related programmes

This is an extract from a subject guide for an undergraduate course offered as part of the University of London International Programmes in Computing. Materials for these programmes are developed by academics at Goldsmiths.

For more information, see: www.londoninternational.ac.uk

Goldsmiths
UNIVERSITY OF LONDON

This guide was prepared for the University of London International Programmes by:

Tim Blackwell

This is one of a series of subject guides published by the University. We regret that due to pressure of work the author is unable to enter into any correspondence relating to, or arising from, the guide. If you have any comments on this subject guide, favourable or unfavourable, please use the form at the back of this guide.

University of London International Programmes
Publications Office
32 Russell Square
London WC1B 5DN
United Kingdom
www.londoninternational.ac.uk

Published by: University of London

© University of London 2007

The University of London asserts copyright over all material in this subject guide except where otherwise indicated. All rights reserved. No part of this work may be reproduced in any form, or by any means, without permission in writing from the publisher. We make every effort to respect copyright. If you think we have inadvertently used your copyright material, please let us know.

Contents

Introduction	iii
I Principles	1
1 Software Engineering	3
1.1 What is a good system?	3
1.2 The problem	4
1.3 Building systems with objects	5
1.4 OO design	7
1.5 Exercises	9
1.6 Summary	9
2 Development Process and Modelling	11
2.1 Iterative and waterfall processes	11
2.2 Design and modelling	12
2.3 A unified modelling language: UML	13
2.4 Fitting UML into a process	14
2.5 Exercises	15
2.6 Summary	16
II UML	17
3 Use Cases	19
3.1 Requirements capture	19
3.2 The basic technique	20
3.3 When to use use cases	21
3.4 Exercises	21
3.5 Summary	21
4 Class Diagrams: The basic technique	23
4.1 Class identification	23
4.2 Properties	24
4.3 Properties as code	25
4.4 Adding more information to the class model	26
4.5 When to use class diagrams	27
4.6 Exercises	27
4.7 Summary	27
5 Sequence Diagrams	29
5.1 The basic technique	29
5.2 Advanced techniques	30
5.3 When to use sequence diagrams	31
5.4 Exercises	31
5.5 Summary	31
6 Class Diagrams: Advanced techniques	33
6.1 Responsibilities and collaborators	33
6.2 Static operations and attributes	34
6.3 Aggregation and composition	34
6.4 Interfaces and abstract classes	34

6.5	Classification	35
6.6	Association classes and visibility	36
6.7	When to use advanced concepts	36
6.8	Exercises	37
6.9	Summary	37
7	State Machine Diagrams	39
7.1	The basic technique	39
7.2	Implementing state diagrams	40
7.3	When to use state diagrams	41
7.4	Exercises	41
7.5	Summary	41
8	Activity Diagrams	43
8.1	The basic technique	43
8.2	Advanced techniques	44
8.3	When to use activity diagrams	45
8.4	Exercises	46
8.5	Summary	46
9	Summary of UML modelling techniques	47
III	Quality	49
10	Product Quality	51
10.1	Verifying software	51
10.2	Validating software	52
10.3	Testing	52
10.4	Exercises	53
10.5	Summary	54
11	Process Quality	55
11.1	Project management	55
11.2	Project planning	56
11.3	Exercises	56
11.4	Summary	57
IV	Resources	59
12	Analysis and design of a personal Organiser	61
12.1	Scenarios	61
12.2	Use cases	61
12.3	Class Identification	62
12.4	UML diagrams	62
13	Sample examination paper	69
13.1	Advice	69
13.2	Questions	69
13.3	Answers	71

Introduction

A while ago I decided to build a small tool shed. I took a trip to the local construction yard and bought some wood. Back home, I built a wooden frame for my shed, sawing the bits of wood as I needed them. I then nailed sheets of wood to the frame. I left a space for a window - but I had forgotten to buy any glass at the yard, so I returned there. I discovered that the yard had ready-made windows, and they looked rather good, so I bought one. Returning home, I found that the space was too small for the window unit I had just bought, but this was not a problem because I could easily saw some wood away from the wall panel. And so I continued with my haphazard construction, returning to the yard from time to time to buy roof felt, a door, hinges etc. Finally my shed was finished and it was fine. In fact it was so good and I was so proud of my achievement that I commenced another, larger construction project. I really did need a bigger kitchen.

After many trips to the construction yard to replace ill-fitting pipes and window frames, after knocking down a newly-built wall because I had forgotten to install any plumbing, and after countless other minor re-starts, the kitchen extension was finished, and it was OK, even though it had taken me months to complete. My next project was a complete new house, built from scratch in a neighbouring field.

You know the ending to this domestic tale! My construction techniques just could not scale to this big project. The complexity of the supply of materials (bricks, tiles, concrete, plaster, wood...), the relationship between the various systems (heating, drainage, electric, gas) and the ramifications of late design change (perhaps the stairs should really be to the right of the hall - but then the bathroom needs to be moved to the left of the landing - now this means re-routing the water pipes - ...) meant that, if I were lucky, this project might actually have succeeded, but I would certainly overrun and at a very high cost. And would it be the house I actually wanted?

After some calls I assembled a project team. An architect designed the house, producing several plans of her design according to my initial brief. Several rather artistic drawings showed the house from the outside so I was able to judge the overall physical design. Other more formal plans of the interior layout, so I could imagine walking around inside and visiting the various rooms. At this stage I changed a few things. Still other plans were made for the electrician and the plumber detailing where the pipes and cables should be fitted. There was even a special plan of the kitchen showing in detail the work-surfaces and appliances. A surveyor was employed to organise the supply of materials, builders were employed to perform the construction, and engineers checked the work at various stages. A manager oversaw the whole process.

Learning activity

Extract from this tale some general principles of large-scale construction.

Comments on the activity

Large scale construction requires: analysis of the project so that the right thing is built; early consultation with the client; plans of the project from various perspectives and levels of analysis; inclusion of pre-built components into the design; management to ensure that parts of the project are built in the correct order; builders; people to check the construction, both for accuracy, and for quality.

All of these principles apply to software projects. Here the constructed artefact is not tangible, but is a logical object. A large piece of computer code is very intricate, far too complex for any person to understand. Additional complexity arises from interaction between the software and an uncertain physical world, and in understanding the changing needs of the client. Software, then, needs to be *engineered*.

This course is strategically placed at the midpoint of your studies. You will already have studied a computer language at an introductory level, corresponding to rudimentary do-it-yourself skills. Later in your studies you will be developing a larger software project and this will require a more systematic and scalable approach. You will no longer be able to sit at the computer and type your programme code in a trial and error fashion. As the programme grows in size you will drown in a sea of complexity. Your software project will have to be designed, you must become a software *architect*. It is the purpose of this course to provide you with the skills and insights to bridge the gap between small scale programming and large commercial applications.

The complexity of software is modelled, rather like an architect models a building, with the use of diagrams. We shall use a technique that has become an industry standard: the Unified Modelling Language (UML). Proficiency at UML, along with the computer languages you have studied, will be amongst the main transferable skills you will acquire throughout your entire degree programme.

This subject guide, which covers one half of CIS226, provides an introduction to software engineering. Part I of the guide begins with a look at some principles of good software design and how these principles fit into a particular developmental process. The main topic of this course unit, covered in Part II, examines how to analyse and design software using the Unified Modelling Language (UML). Part III discusses quality assurance, both of the product and in the process. Part IV shows a specimen software project and examination.

There are two software projects in this guide: a Personal Organiser and Space Game. Each chapter of part II ends with an exercise for each project. These exercises cover the analysis and design stages of the software life cycle. Algorithm design is an important part of the implementation phase of a software project, and this is covered in the second half of this course unit. There will not be any implementation of Personal Organiser or Space Game. The final

stages of a software project are testing and maintenance, which, in the absence of actual code, can only be studied theoretically in this course. Later in your studies you will undertake a complete software project and you will find the ideas and skills of CIS226 invaluable for the successful completion of this project.

Here are the main topics of this course, arranged to correspond to the parts of this guide:

I Principles

- (a) the role of design and modelling in software development
- (b) software development process

II UML

- (a) use cases
- (b) class, sequence, state machine and activity diagrams
- (c) objects and links
- (d) compositions, aggregations and dependencies
- (e) other UML diagrams and concepts

III Quality

- (a) Product quality: verification, validation and testing
- (b) Process quality: project management and planning

How to use this subject guide

This subject guide is not a self-contained account, but is a companion to the course texts *UML Distilled* by Martin Fowler and *Using UML* by Perdita Stevens and Rob Pooley. *It is essential that you obtain these books.* This guide is structured in the form of selected readings from these texts. Learning activities will test your understanding of the main points of each reading. You are strongly advised to attempt these activities as a way of monitoring your own progress. There are more exercises at the end of each chapter and a sample examination paper at the end of the guide.

Each chapter begins with a general statement of the essential reading for this part, and throughout the text you will find specific instructions on what to read. It is important that you read the relevant sections of the course texts when instructed to because text subsequent to a reading will only make sense in the context of the reading. The reading will be cited on the following format:

Read UUML/UMLD, Chapter x, heading [omit]

where *UUML/UMLD* are the title initials of the course texts, *x* is the relevant chapter (identical to the chapter(s) of the essential reading) and *heading* is the chapter section for this directed reading. Each chapter of either course text begins with a short introduction which does not have a section heading and is denoted *introduction*. You are expected to read all the subsections contained in the section you have been directed to. Some directions include an omit guard which tells you to ignore certain subsections or boxes. British English is used throughout this guide, except in quotations and section headings for the set readings, where the original punctuation and spelling (which might be American English) is used.

There is an additional list of books which expand on a number of topics and you are advised to deepen your understanding by referring to these additional texts where directed.

Readings

Here are the details of the course texts and additional materials:

- Essential

1. *UML Distilled (third edition)*, Martin Fowler, Addison-Wesley, 2004
2. *Using UML (updated edition)*, Perdita Stevens with Rob Pooley, Pearson Education Limited 2000

- Supplementary

1. *Project-based Software Engineering*, Evelyn Stiller and Cathie Leblanc, Addison Wesley, 2002
2. *Software Engineering (fifth edition)*, Roger Pressman, McGraw-Hill, 2000
3. *The Mythical Man-Month (anniversary edition)*, Frederick Brooks, Addison Wesley 1995
4. *Effective Java*, Joshua Bloch, Addison Wesley 2001

Aims and objectives

At the end of this course you will be able to:

1. Explain the role of software development in the software life-cycle
2. Produce static and behavioural models of software programs
3. Specify and verify software systems
4. Decompose problems and develop software architectures
5. Implement software models in a structured and efficient way.

These are the *major objectives* of this course unit. Each chapter also lists a number of *learning outcomes*. The learning outcomes break the objectives into manageable tasks.

The examination

You will be assessed on your understanding of objectives 1 - 4. (Objective 5 will be assessed at a later stage of your studies when you will be asked to implement a particular project.) The method of assessment for the complete course unit CIS226 will be one unseen written exam paper of 3 hours (85 per cent) and four equally weighted assignments (15 per cent). Software Engineering (the subject of this guide) will comprise one half of the overall mark for CIS226. There is a sample examination paper for Software Engineering and advice on exam preparation in the final chapter of this guide.

Part I

Principles

Chapter 1

Software Engineering

Essential reading

Using UML Chapters 1 and 2

All large-scale construction projects follow five main phases of activity: analysis, design, construction, testing and maintenance. Software projects are no exception. Software engineering is concerned both with the phases themselves and also with how each phase fits into the overall development of the project (in software engineering-speak, *process*).

A large software project runs like this: *software architects* draw up plans (*models*) at various levels (*abstractions*) of analysis. The models facilitate further *analysis* and *design*. Models also serve as construction guides during the *implementation* phase and as documentation of the end result. To facilitate design and implementation, ready-made units (*components*) are used wherever possible. A *project manager* oversees the whole process.

A major aspect of software engineering is therefore concerned with the development process. However, in order to judge how well our software is engineered, we need to have an idea of software quality. In most domains of engineering, we understand fairly well what a successful construction is, and how to achieve it. For example, bridges should not fall down (and they rarely do), and there are accepted design principles, construction techniques and quality assurance assessments. The situation with engineered software is not so clear, however.

This chapter begins by considering what a high quality software system should be like and outlines some important design principles. Software design with objects is an important manifestation of these principles and the important aspects of this paradigm are explained in the remaining sections of this chapter.

1.1 What is a good system?

Read UUML, Chapter 1, *What is a good system?*.

Stevens and Pooley, the authors of UUML point out that a good system is surely one that meets its users' needs. They list five desirable attributes (namely: usability, reliability, flexibility, affordability and availability) of a good system and consider how well current systems satisfy these criteria.

Although successful systems do exist, there are a number of spectacular failures. These are informally known as software horror stories and a web search will reveal many sources of information. Remarkably, many large projects are cancelled, and of those that do survive, most will significantly overrun their initial planning period. Three quarters of large projects that are actually completed will ultimately fail!

Learning activity

Suppose that you are managing a large software project which is slipping behind schedule. Should you employ more people to work on the project? Explain your answer.

Comments on the activity

The surprising answer to this question is that increasing the size of the team will not necessarily have any benefit. The reason for this is that as the team increases in size, more and more of the project's costs and time are consumed in the overhead of inter-person communication. In fact, the time needed to ensure that all the information is available to keep the work consistent can actually extend a project. This idea is expanded at some length in Fred Brooks' The Mythical Man Month.

1.2 The problem

Read UUML, Chapter 1, What are good systems like? [omit Architecture and components, Component-based design: pluggability]

What is behind this gloomy state of affairs alluded to above? Quite simply, software is extremely complex: there is a limit to how much we can understand at one time.

If your only experience of programming has been an introductory course on Java, you will have been the sole author of your programmes and should (in principle) have been able to understand everything about your code. At first you probably just sat in front of a screen and typed code directly into an editor, rather like me and the shed. However, as an application grows in size, this unstructured approach becomes increasingly hard to sustain - the programme becomes a single monolithic unit and is just too complex and difficult to develop and maintain. It seems that some time must be spent planning how you will code your solution to a software problem. Then you will implement your solution. Good quality systems are surely *designed*.

A number of fundamental design concepts are broadly agreed upon. See *Software Engineering* by R. Pressman, sections 13.4 and 13.5 for a full discussion. Above all, the division of software into smaller *modules* is universally accepted as the main way to control complexity. There is nothing special about this idea; it is the way that we seek to understand anything. For example, science works by modularising the world in a process called reductionism. There is a hierarchy of software modules: from methods to classes to packages to subsystems. Note however that an excessive modularisation of a

system could actually increase overall complexity due to the overhead of integrating very many modules.

Abstraction is the thinning away of detail from a problem domain, leaving a more general representation. At the highest level of abstraction, a software solution is provided in broad terms, using the language of the problem domain. And at the lowest level, the language is the programming code itself. In between there are many levels. The software architect will abstract and model the system in various ways in order to focus on the essential programme elements.

A *cohesive* module performs a single task and requires little interaction with other modules in order to accomplish this. For example, it is a common dictum that methods should only do one thing i.e. return or alter a single value with no other side-effects. Similarly, a cohesive class will represent a single concept, or type of real-life object.

Coupling refers to the connectivity between modules, and should be kept to a minimum if the software is to be understood. The elimination of the notorious *goto* statement has done much to reduce 'spaghetti code' and the complexity of a programming unit. There is no such check on modular spaghetti networks however, and software designers must guard against high coupling.

It is generally agreed that all modules should present a simple, public *interface*. The interface defines what tasks the modules can perform, but not how the modules may accomplish this task. These details are private to the module; a client of the module need never know them. Each module in a 'good' system is only linked to a few others and every client is not able to know more about the module than is contained in the interface. This is the important principle of *encapsulation*.

A module with high cohesion and low coupling can be re-used, either in later systems, or in other modules of the present system. The object oriented paradigm supports re-usable modules as we will later explain. High cohesion and low coupling also mean that modules can be replaced without too much disruption to the system as a whole. They are pluggable. Re-usable, replaceable pluggable modules function rather like component parts in a mechanism. That is why they are called *software components*.

Learning activity

Write, *in your own words*, a paragraph on what a good system should be like.

1.3 Building systems with objects

Read *UML*, Chapter 2, *What is an object?*.

This reading relates programming with objects to the reusable, replaceable modules which make good components and ultimately, we hope, a good system.

The fundamental module of object oriented (OO) programming languages is (unsurprisingly) the object. Here we summarise, from a software engineering perspective, some properties of objects, and introduce a small amount of the unified modelling language (UML).

Objects are programming modules that group related data items together. A set of objects with a similar role in a system is grouped into a class. Blocks of code which perform common operations on this data are also grouped together and are maintained by the class. Objects are supposed to be the software analogue, or representation, of an actual object or of a concept. In this way, we hope to map the tangible real world of things onto the logical, abstract world of computer code.

The class is an object factory, responsible for making however many objects the system requires. (An object can be referred to as an instance of a class.) The *state* of an object is the value of all the variables in that object and the *behaviour* of an object is the way that an object responds to messages. In the OO world, a system is a collection of objects which are sending messages to each other in order to accomplish a certain task.

An example of an object message is:

```
resetTime( newTime: Time )
```

This message is written in UML notation. Here, *newTime* is an object of type *Time* i.e. an instance of the *Time* class. Notice that the message `resetTime(newTime: Time)` is language independent. Suppose that *myClock* is an instance of a *Clock* class and we would like *myClock* to accept and respond to a `resetTime` message. In Java, we would ensure that a *Clock* object would accept this message by implementing a method in *Clock* with the signature:

```
public void resetTime( Time newTime )
```

The class of an object also defines its interface. In the *Clock* example, *myClock* is an object of class *Clock*. *Clock*'s public interface might specify that each *Clock* object should provide *reset-time* and *report-time* operations. A *Clock* object will also have a private time attribute. Private attributes are only available to the object itself: clients of *Clock* cannot access time directly. Instead, they can only ask the time by sending the message *report-time* to a *Clock* object. This is known as *data-hiding*. In contrast, a public attribute or operation is accessible by any client. Hidden-data is often accessed using specific accessor (getters and setters) methods. Objects from the same class have the same interface. The public and private interface to *Clock* might be:

```
-time : Time
+ reportTime() : String
+ resetTime( newTime: Time )
```

where the symbols + and - refer to public and private *visibility*.

Learning activity

1. What is an object's interface? What is the difference between a public and a private interface?

2. Why bother grouping similar objects together into a class?
3. Write, in pseudo-Java, a class outline for `Clock` based on its declared interface.

Comments on the activity

1. See section UUML, What is an object [subsection: Interfaces].
2. See the digression UUML, What is an object? [subsection: Digression: why have classes?]. A basic answer to the problem of maintaining consistent copies of any software artefact is to structure code so that no copies are necessary! For example, all my `Clock` objects need to be reset. It's far easier to maintain (i.e. revise and re-use) a single `reset` method in the class `myClock`. A further and more technical point is that a class can be thought of as a type, and the language compiler can spot possible errors when type-checking.
3. The pseudo-Java code, shown below, is a bridge between the design and the implementation phases of a software project.

```
class Clock
    private Time time

    public String reportTime()
        return time.toString()

    public void resetTime(Time newTime)
        time = newTime
```

Read UUML, Chapter 2, How does this relate to the aims of the previous chapter?.

The previous chapter proposed that reusable, replaceable, component-like modules will lead to reduced development time and cost, ease of maintenance and greater reliability. OO is expected to deliver on these counts, but objects themselves do not make good modules, because there can be many similar objects in a system, and the requirement to ensure consistency between them would be a very high price to pay. A house is made from many bricks but a single `Brick` class provides the necessary generalisation.

Another benefit of OO is that it is natural to objectify the world, so that an OO system can provide a better match between abstract computer code and the problem domain of the customer. An object model can help engineers in capturing requirements, following changes in user requirements and allowing for more naturally interacting systems.

OO, of course, is not the only possible approach, but it is one that takes modularity, encapsulation and abstraction as fundamental.

1.4 OO design

Read UUML, Chapter 2, Inheritance.

Inheritance, polymorphism and dynamic binding are the three features that make OO special.

Inheritance is an important concept in OO design and you should make sure that you understand the explanations given in this reading. Notice in particular how a subclass can inherit attributes and operations from its superclass, a subclass can add extra attributes and operations, and a subclass may override some superclass methods. (Less usefully, a subclass may also shadow superclass variables.)

Apart from the direct advantage of code re-use, inheritance implies another defining feature of the OO paradigm. A subclass object can be freely substituted for its superclass parent. This means that an object can have any one of several types. The ability of an entity to exist in different forms is called polymorphism.

Since methods may be overridden in sub-classes, what code should be executed as the result of some message? For example, a `Lecturer` class can be sub-classed by a `DirectorOfStudies` class with the overridden method `canDo(Duty duty)`. In the following pseudo-code

```
for each o in lecturers
    o.canDo( seminarOrganisation )
```

`lecturers` is a list of `Lecturer` objects, and includes a `DirectorOfStudies` object, `o` is a specific object selected from this list and

```
canDo( seminarOrganisation: Duty)
```

is an operation on `Lecturer` objects. Note also the dot notation signifies message sending: `o` is the recipient of the message `cando(seminarOrganisation)`. When the program execution reaches the `DirectorOfStudies` object, the overridden `canDo` method is invoked, as we would expect. The runtime selection of the correct method due to a specific message is known as dynamic binding.

Learning activity

Inheritance clearly favours code re-use, and it would therefore appear to be a desirable feature of OO. Do you think that inheritance leads to greater or less encapsulation?

Comments on the activity

Item 14 of Effective Java by Joshua Bloch warns against inheritance across package boundaries. A subclass depends on the implementation details of its superclass, clearly in conflict with the principle of encapsulation. Composition is a better solution - we shall return to this in a later chapter.

Read *UML*, Chapter 2, *Polymorphism and dynamic binding*.

Clearly, polymorphism promotes code re-use. But polymorphism also makes code more flexible. For example, the Java classes `Vector` and `ArrayList` both implement the Java interface `List`. Remember that an interface, in general, defines what tasks a module can do, but not how it may do it. In Java, an interface type is a programming implementation of this general notion: `Vector` and

`ArrayList` implement certain methods which are specified by the `List` definition. Hence they share the same public interface. Furthermore, an interface defines a type which all implementing classes share. For example, wherever a `Vector` is instantiated

```
Vector subscribers = new Vector();
```

we can use a `List` object:

```
List subscribers = new Vector();
```

and a `Vector` instance method such as `add(Object o)` will still work:

```
subscribers.add( newSubscriber );
```

This is more flexible because we can later switch implementations

```
List subscribers = new ArrayList()
```

and all the surrounding code will still work. Item 34 of *Effective Java* explains this point in some more detail.

1.5 Exercises

1. Would individual objects make good modules?
2. Supporters of OO claim that a major benefit of this paradigm is that it is inherently natural to look at the world in terms of objects. Why should this be beneficial for software design?
3. Suppose that OO enables us to build a good system. How much of this is due to the intrinsic nature of OO techniques?

Answers

1. No. See *UML*, Chapter 2, *How does this relate to the aims of the previous chapter? [first three paragraphs]*.
2. The short answer is that the system can better match the users' model of the world. But you should try to understand why software architects consider this to be desirable.
3. If OO succeeds, it is because this paradigm takes modularity, encapsulation and abstraction as fundamental.

1.6 Summary

After studying this chapter you should be able to:

- state what a high quality software system should be like and comment on to what extent we have such systems
- state the basic problem faced by software developers
- list and explain the fundamental design concepts of software engineering (modularity, abstraction, high cohesion, low coupling, simple public interfaces, encapsulation, components)
- explain what software objects are, and how they communicate
- use UML to specify a class interface

- translate object messages into Java methods, and write a class skeleton based on a declared interface
- list and explain the three special features of OO (inheritance, polymorphism, dynamic binding)

Chapter 2

Development Process and Modelling

Essential reading

UML Distilled, Chapters 1-2
Using UML, Chapter 4

Additional reading

R. Pressman, Software Engineering, a Practitioner's Approach, Chapter 2

The first chapter argued by analogy that software, like any other engineering project, proceeds by phases of development. The phases fit together into a process. In this chapter we consider the development process in more detail. We also look again at the fundamental problem of software systems - the intangibility and complexity of computer code. Good design, we have discovered, is modular. But how do we arrive at these modules? We can't play with a lump of plasticine or sketch pictures in order to brainstorm new ideas. However, the system must be modelled in some way. We introduce software modelling languages and the chapter finishes by demonstrating how the most prevalent modelling language, UML, integrates back into the developmental process.

2.1 Iterative and waterfall processes

Read UMLD, Chapter 2, Iterative and Waterfall Process.

The iterative and waterfall methods are the two major categories of software development processes. Both methods agree that the process must be subdivided into phases. The *waterfall* emphasises discrete activities that must be completed sequentially. And just like a real waterfall, back flow is impossible (or at least exceptional). Iterative techniques break the project down into chunks of functionality, called *iterations*. Each iteration is a complete *life-cycle*: analysis, design, implementation and testing. Each iteration outputs a system of production quality, even if it is not actually released at this stage. *Time boxing* forces an iteration to be of fixed duration, even at the expense of slipping functionality. Generally speaking it is better to slip a function rather than a date because the team can then learn just what the most important functional requirements actually are.

Learning activity

Which process is recommended by the OO community? Why is this?

Comments on the activity

See UMLD, Chapter 2, Iterative and Waterfall Processes [last half of this section].

(Martin Fowler, the author of UMLD recommends "You should use iterative development only on those projects that you wish to succeed!")

2.2 Design and modelling

Read UUML, Chapter 4, Defining terms [omit: Process and Quality].

As we have mentioned before, a model is an abstract representation of a specification, a design or a system. The model itself is usually represented visually, as a diagram, but can be textual. The model is precise according to the implicit point of view. Room plans, wiring diagrams, elevations and other architectural plans strip away unnecessary detail in order to focus more clearly on what they are representing. The modelling language has syntax and semantics, just as a natural (spoken) language does. The unified modelling language, UML, is quickly becoming the *lingua franca* of software architects. One reason for this is that the language is very flexible and is not tied to a particular process.

Learning activity

What are the attributes of an ideal modelling language?

Comments on the activity

A modelling language should be expressive, easy to use, unambiguous, supported by tools and widely used - see the list of points in UUML, Chapter 4, Defining terms [subsection: Why a unified modelling language?].

Read UUML, Chapter 4, system, design, model, diagram.

The developmental process will produce (we hope) a collection of programmes which work in an appropriate environment to fulfil the users' needs. The architecture of this system abstracts away many details and embodies just how the system should be built.

The plans for a house do not include each brick, each floorboard and every electrical socket, but they do show the relative positions of the walls, floors and ceilings, the electrical wiring, the landscaping of the garden...There are many plans, each one is incomplete in itself, but taken together they describe the project at just the right level of detail for the builders, electricians, joiners and landscape gardeners.

Plans are roughly analogous to *models* in software engineering. Many models are needed even with a simple design; it is impossible that a single diagram can capture every design aspect. There are three groupings of software models, a use case model, a static or structural model and a dynamic or behavioural model. A building plan may be drawn from different perspectives. Similarly, four software “views” have been suggested: logical, process, developmental and physical. A design requires several models, and each model might require several diagrams from various views, and all of these must be consistent.

We are now ready to begin our study of UML.

2.3 A unified modelling language: UML

Read UMLD, Chapter 1, What Is the UML?.

The fundamental imperative behind all graphical techniques is that programme code is not at a high enough abstraction to promote meaningful discussions about design. UML is a family of graphical notations, unified by a single meta-model. These notations help in describing and designing OO software. UML is an open standard, controlled by the Object Management Group, and is a unification of many pre-existing OO graphical techniques.

Read UMLD, Chapter 1, Ways of using UML [omit Model Driven Architecture and Executable UML].

This reading suggests that UML can be used in different ways. This is because people use older graphical modelling languages in different ways, and that legacy has passed to the newer, unified language. The three uses or *modes* outlined are sketching, blueprinting and programming. One criticism of the blueprinting is that it takes too much time to prepare and that textual languages are already very effective for most programming tasks. In programming mode, UML becomes the source code, and this mode demands very sophisticated tools.

Learning activity

Write a few sentences on each of the three modes of use of UML.

As well as the three modes of use, there are two *perspectives*, the software perspective and the conceptual perspective. Some tools will turn a UML diagram into source code. This is very much a software perspective. Alternatively, in the conceptual perspective, the diagrams are language independent representations of structures and relationships.

Read UMLD, Chapter 1, UML diagrams.

Tables 1.1 and 1.2 of *UMLD* show classifications of the thirteen diagram types in UML 2.0. You do not need to learn these tables! They will be useful, though, for you to refer back to as we introduce various diagrams in part II of this guide. The UML authors do not in

fact see the diagrams as the central part of UML and as a result the diagram types are not rigid and there is much flexibility in their use.

Read UMLD, Chapter 1, What is legal UML?

Most people regard that the UML rules are descriptive rather than prescriptive. This means that we infer language rules by looking at how other people have used them, just as we have all done in learning our own first spoken language. The UML does have a standard, although even this is imprecise because UML is so complex. Information may be suppressed in UML, so that nothing can be inferred from the absence of any particular detail in a diagram. But most important, it is better to have good design rather than strictly legal and precise UML.

Read UMLD, Chapter 1, The meaning of UML.

There is no formal definition of how UML maps to any particular programming language. The meaning of UML (within the sketch mode) is to provide a rough idea of what the code will look like. We can trust the bricklayer to actually make the wall, but a plan tells her where to put it.

Read UMLD, Chapter 1, Where to start with UML.

Luckily, Fowler recommends that we don't need to use all thirteen UML diagrams! A team will work well with just a subset of UML. The most common and useful diagrams are class and sequence diagrams. Chapter 9 makes further suggestions.

2.4 Fitting UML into a process

Read UMLD, Chapter 1, Fitting the UML into a process [omit Documentation].

This section of *UMLD* suggests which types of diagrams are useful in the analysis and design phases, and also for documentation. The section also discusses how to use UML in sketch or blueprint mode.

Learning activity

Contrast the different uses of UML in the waterfall and the iterative process.

Comments on the activity

In the waterfall approach to development, the diagrams are prepared as part of each phase and are included in the end of phase documents. UML is used for blueprinting, and the diagrams are definitive.

In the iterative process, the UML diagrams may be sketches or blueprints, but they may be modified at each iteration. The analysis diagrams will be drawn in the iteration preceding the iteration that adds functionality to the software. Blueprint design can be made early in the iterations for targeted, principal functionality, and any iteration can make changes to existing models rather than building a new model. The model itself is not reworked.

2.5 Exercises

1. What are the similarities and differences between the waterfall and iterative development processes?
2. How can you tell if a process is truly iterative?
3. A project manager is overheard saying “This iteration’s buggy, but we’ll clean it up at the end”. What software development process is being used on this project? Explain your answer.
4. (a) What is code rework?
 (b) In many domains, such as manufacturing, rework is seen as being wasteful. Is this true in the software domain?
 (c) Which technical practices can make rework more efficient?
5. (a) Which UML techniques are useful in the requirements analysis phase?
 (b) What is the main objective during this phase?
 (c) How does this impact on the UML techniques?
6. Which diagrams are useful during the design phase, and how are they used?

Answers

1. Two paragraphs summarising the material in *UMLD, Chapter 2, Iterative and Waterfall Processes [first page]* would be a good answer. Diagrams similar to *R. Pressman, Software Engineering: A Practitioner’s Approach, Figures 2.4 and 2.7* will gain bonus marks.
2. Each iteration must produce tested, integrated code that is of (or close to) production quality. The test is that any iteration that has not been scheduled for release, *could* in fact be released.
3. A waterfall process, because the manager admits the current iteration is not of production quality, contrary to the definitive test for a genuine iterative process (see above answer).
4. (a) Rework means that existing code is revised, possibly even deleted.
 (b) Far from being wasteful, code rework saves time (and hence money) in the long run because it is rarely a good idea to patch badly designed code.
 (c) Automated regression tests, re-factoring and continuous integration.
5. (a) See *UMLD, Chapter 2, Fitting the UML into a Process [subsection: Requirements Analysis]*.
 (b) Use cases, class, activity and sequence diagrams are all useful in this phase.
 (c) The overriding aim is communication between architects and users (just as in house building projects), so UML rules may have to be broken because clients - who are the domain experts - will not necessarily be at all familiar with software engineering.

6. See *UMLD, Chapter 2, Fitting the UML into a Process [second page]*. Class diagrams - these show the classes and how they interrelate. Sequence diagrams model scenarios (scenarios will be explained in a later chapter, but basically they are short stories illustrating how a user may interact with the system). Package diagrams - these show the large-scale organisation of the software. State diagrams are useful for any class with a complicated life history. Deployment diagrams show the physical layout of the software.
-

2.6 Summary

After studying this chapter you should be able to:

- list the four phases of software development (analysis, design, implementation, testing)
- explain why developmental phases must be managed in a process
- describe and distinguish the waterfall, iterative and agile development processes
- explain the importance of modelling and finding a good modelling language
- list the features of a good modelling language (expressive, easy to use, unambiguous, supported by tools, widely used)
- state the three groups of software models (use case, static or structural and behavioural or dynamic)
- state and explain the three modes of use of UML (sketch, blueprinting and programming)
- explain the difference between a software perspective and a conceptual perspective
- recommend types of diagrams for the analysis and design phases

Part II

UML

Chapter 3

Use Cases

Essential reading

UML Distilled Chapter 9

You have been introduced to the idea of a modelling language. Part II of this guide is largely devoted to the unified modelling language (UML). However, we begin with a technique that UML does not standardise, and whose value derives from text rather than diagrams.

3.1 Requirements capture

The hardest single part of building a software system is deciding precisely what to build. No other single part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later ¹.

¹Frederick Brooks in *No Silver Bullet: Essence and accidents of software engineering* IEEE Computer, pp. 10-19, 1987

The essential phases of software development are: requirements analysis, design, implementation, testing and maintenance. These phases are common to all software development processes, but each process differs in the duration, completeness and sequence of these steps. But all processes agree on the very first step, namely *deciding what needs to be built*.

Learning activity

Why do you think deciding precisely what needs to be built is so difficult? What are the ramifications for not making these decisions?

Comments on the activity

The important thing is not to decide what to build, but to decide what precisely to build. This is hard because

- *the client might only have a vague and unrealistic idea of what they want*
 - *the software engineers are not experts in the application domain*
 - *of shifting requirements as the project develops*
-

3.2 The basic technique

The aim of requirements analysis is a description of how the system should behave in certain situations, i.e. a set of *functional requirements*. Use cases are a technique for capturing these requirements. The use cases describe typical interactions between the users and the system. This narrative is built from *scenarios*. A scenario is one thing that might happen, and several scenarios, all tied together by a specific user goal, constitute a use case.

The users of the system are called *actors*, a mistranslation of the Swedish word for role. An actor is a role that a user has when engaging with the system. Actors need not even be people: an external computer system could also be an actor.

Read UMLD, Chapter 9, Content of a Use Case.

Figure 9.1 of *UMLD* shows a possible format of a use case. UML itself does not specify a standard. Notice that the use case is textual rather than diagrammatic. Each statement or step is an element of the interaction between the actor and the system, showing the intention, but not the mechanism. Each step may suggest alternative situations, and these are logged as extensions to the use case. Sometimes a sequence of steps can occur in other use cases as well. These steps can be factored out and replaced with a single more complex step. Pre-conditions, guarantees and triggers can also be added to use cases, but the aim should always be for simplicity.

Learning activity

1. What is a scenario, and why is it used?
2. What is the relationship between scenarios and use cases?

Comments on the activity

1. *A scenario is a sequence of steps describing an interaction between the user and the system. It is written as a paragraph of normal text. They are used for requirements capture. Since they are textual, they are easily understood by the domain expert. A number of scenarios will eventually comprise a single use case. The use case is the principal UML expression of system requirements.*
 2. *Several scenarios may share a single goal, but in some of them the user does not succeed in achieving this goal. These scenarios, linked by a common user goal, become a single use case. The main success scenario becomes the main body of the use case, the other scenarios enter the use case as extensions.*
-

Read UMLD, Chapter 9, Use Case Diagrams.

As you can see in Figure 9.2, UML does specify a diagram for use cases, but really such things serve best as a table of contents to the textual descriptions. The diagram shows the actors, the use cases, and any inclusions.

Read UMLD, Chapter 9, Levels of Use Cases.

The reading distinguishes business use cases from system use cases. The system use case concerns a (relatively short) interaction between an actor and the system. These are ‘sea-level’ use cases. Interactions that have complex ramifications for the entire business are at a higher (kite) level. And use cases that only exist because they are included in other level 1 cases are at fish level.

3.3 When to use use cases

Read UMLD, Chapter 9, When to Use Use Cases.

Use cases help us to grasp the basic functional requirements of the system and should be generated early in the project. Later, immediately prior to implementation, more detailed versions of these cases can be derived. The use case diagram is of little value compared to the textual specification. As with all good UML, always strive for simplicity and clarity.

3.4 Exercises

1. Your development team has been asked to develop a Personal Organiser. The organiser should allow users to enter appointments in a calendar, receive alarms and maintain an address book of contacts. You might be able to think of other desirable functions. To begin this project, write some scenarios. Form a few simple use cases from these scenarios and draw a use case diagram. Now, in a mini-iteration, study your use cases and include any pre-conditions, guarantees and triggers that might seem appropriate.
2. Your development team has been contacted by a client who wishes to produce a Space Game. This game will be based on classic video games such as Space Invaders, Galaxians and Asteroids *as a single application*. See <http://www.spaceinvaders.de/> for information on Space Invaders and links to other retro arcade games.

To start, develop a set of scenarios and use cases for Space Invaders. If you have time, develop some use cases for at least one more game. Are there common features? Can you now develop some use cases for a generic Space Game?

3.5 Summary

After studying this chapter you should be able to:

- state what a scenario and a use case are
- explain the relationship between a use case and a scenario
- compose scenarios and use cases for software projects
- draw a use case diagram
- understand the different levels of use cases
- add common information such as pre-conditions, guarantees, triggers to a use case

- be able to factor out common steps from use cases and represent this on a use case diagram
- know when (at what stage of the software development process) to compose and amend use cases.

Chapter 4

Class Diagrams: The basic technique

Essential reading

UML Distilled Chapter 3

Using UML Chapter 5

Read UMLD, Chapter 3, Introduction.

The class diagram is a very widely used modelling technique. A class diagram describes the types of objects in the system and the static relationships between them. This chapter shows you the basic techniques of class modelling. We begin by asking: how do we know which classes a system might have in the first place?

4.1 Class identification

Read UUML, Chapter 5, Identifying objects and classes.

The objectives for any software project are to minimise expense and development time whilst maximizing system maintainability and adaptability. These objectives are often in conflict, but they may be met if the class model provides all required system behaviours, and if the classes represent enduring domain objects, irrespective of the particular functionality required during development. A Book class in a library system represents just such a domain object.

Any technique that leads to a good class model is satisfactory. It is unlikely that you will find all the correct classes at the first iteration. The most important domain objects (such as books, catalogue, library member) should be easy to spot, because these belong to the problem. More elusive, though, are other classes that will have to be introduced in order to help solve the problem. A common technique is noun identification. Start with the requirements specification and make a candidate list of all nouns and noun phrases. Then eliminate inappropriate classes, and rename, if necessary the remaining classes. *UUML* lists some reasons why a candidate class might be eliminated and you should study this list carefully.

Ultimately, what you will be left with is a list of tangible or real-world things (book, journal, catalogue) and less tangible things such as roles (librarian, library-member). Ideally a software object, as generated by a class, *represents* a real-world object. This correspondence between the world and its representation as code

helps us to conceptualise the system and deal with complexity. The system itself is the sum total of all the objects, and of all their interactions. OO design seeks to avoid the monolithic top-down approach where a single module knows and does everything and you should be careful that your own architecture does not fall into this trap, for example by using a single object to represent the core of the system.

Learning activity

1. What is so bad about top-down design?
2. Suppose you are refining a candidate class list. Under what criteria would you reject any candidates?

Comments on the activity

1. *Top-down systems with a single monolithic class are difficult to maintain and have built-in assumptions about how the system will be used.*
 2. *An inappropriate class will have one or more of the following undesirable attributes:*
 - *redundancy*
 - *vagueness*
 - *an event or an operation*
 - *too abstract, not representing an object in the problem domain*
 - *outside the scope of the system*
 - *an attribute with no interesting behaviour.*
-

4.2 Properties

Read UMLD, Chapter 3, Properties.

Properties are the structural features of a class, and appear on the class diagram as an *attribute* or an *association*. The attribute appears inside the class box, and only the name of the attribute is necessary. At the first attempt, you might wish to restrict attribute descriptions in your class diagram to name:type, e.g. price:Money. This means that price is an object of class Money. (In fact there are other possibilities, because an object's type is quite a complicated thing.)

An association is a solid line between classes, directed from the source to the target class. In this case, the name of the attribute appears at the target end of the line. In Figure 3.3, dateReceived, lineItems and isPrepaid are properties of Order, represented in the diagram by associations. Following the advice in this reading, you may wish to reserve associations and class boxes for the most significant classes - it all depends on what you want to emphasise by any particular diagram.

Learning activity

Study the associations in Figure 3.3.

1. How many orders may be made on any day?
2. Can we have an order line without an order?

Comments on the activity

1. Any number, including none at all (consider the association between the source *Date* and the target *Order*).
 2. No. The *OrderLine* (source) - *Order* (target) association has multiplicity 1. Any number of order lines are associated with one, and only one, order.
-

4.3 Properties as code

Read UMLD, Chapter 3, Programming Interpretation of Properties.

A simple way of implementing properties in code is with private data fields. Clients of the class can then retrieve or alter this data with public getter and setter methods. However, the property can also be computed rather than stored. In this case a getter method could calculate the data (e.g. multiply the number of items by their unit price). Multi-valued attributes are coded with a collection, and in this case you will certainly choose to update an item with a setter method. An alternative, and dangerous, implementation might return the entire collection to the caller.

The class skeleton for *Order* shows how the *lineItems* property can be implemented by a *HashSet*, which is a particular sort of unordered collection, or *set*. In contrast, this same property is marked as *ordered* in Figure 3.3, and it would be implemented by a *List* or an *Array*. In practice, for consistency, the class diagram and class skeletons should really agree.

Note that the getter/setter methods will retrieve/alter a single item from *lineItems*, rather than pass the whole set to the requesting client object. The client would not need to know that line items are stored in a list, or an array; all the client wishes to do is inspect an item. This is the essence of *encapsulation*.

Learning activity

Why should you be afraid of classes that are nothing but a collection of fields and their accessors?

Comments on the activity

OO design is concerned with objects that have rich behaviour. Objects are not just parcels of data, but also have operations and behaviour. A heavy use of an object's accessors indicates that some behaviour should be moved to this object.

4.4 Adding more information to the class model

Read UMLD, Chapter 3, Multiplicity.

The multiplicity of a property is the number of objects that fill this property. Make sure that you understand the meaning of the notation [1], [0..1], [0..2] etc. and [*].

Read UMLD, Chapter 3, Bidirectional Associations.

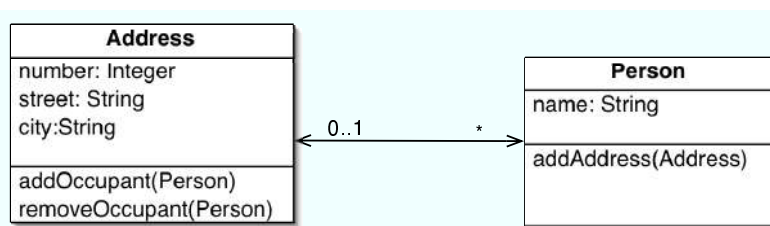
The bidirectional association, signified by navigability arrows at either end, refers to a pair of properties that are linked together, as in the Person - Car association. Navigating from Car to Person and back again returns us to the same car. Implementing bidirectionally is difficult due to synchronisation of the data.

Learning activity

1. Could you mark a Person - Address association as bidirectional?
2. Write down the property pair that represents the actual property 'lives at' for Person - Address. Draw two class boxes, one for each for Person and Address and link these boxes with an association. Mark multiplicities on the ends of the association. Add further properties to the diagram.

Comments on the activity

1. Yes. An object can ask an Address instance who lives there. Address returns a number of Person objects. Take one of these Persons and ask where they live. The returned object will be the original Address.
2. The Address class has properties `dweller: Person[*]` and the Person class has a property `home: Address[0..1]`. (We assume that no-one has more than one address.)



Read UMLD, Chapter 3, Operations.

Operations are the actions of a class, and correspond to *methods* in Java. There is a subtle distinction between the two: an operation is a procedure which might be carried out by any one of several methods in an inheritance tree. Notice that operations appear in the lower third of the class boxes in Figure 3.1.

UML distinguishes between *query* and *command* operations. Technically, a query does not change the *observable* state of an object. A useful programming tip is to separate commands from queries by ensuring that commands do not return any value to the caller.

Read UMLD, Chapter 3, Generalization.

Generalisation means that two objects of different types may be freely substituted for each other. The obvious OO mechanism is *inheritance*. Subclasses of a parent inherit all the features of that parent, and may override any method of the parent. In general, a *subtype* can replace a *super-type* in any code block and the programme will still function (although the output may be different, of course). This is a manifestation of the OO principle known as polymorphism. A subclass is therefore a subtype. Different languages might have additional mechanisms for sub-typing, for example interfaces in Java. Interfaces are studied in Chapter 5 of *UMLD*.

4.5 When to use class diagrams

Read UMLD, Chapter 3, When to Use Class Diagrams.

Class diagrams are the single most important modelling idea in UML and you will use them all the time. However they can be *very* detailed. Too much detail can detract from the underlying purpose of the diagram. It is best therefore to start simply and add extra information only when necessary. Don't draw models for everything. A few class diagrams should cover the breadth of the system. Each class diagram should be used in conjunction with a behavioural diagram so that the dynamic relationship between objects can be grasped.

4.6 Exercises

1. Examine the scenarios and use-cases that you wrote for the Personal Organiser (Section 3.4). Construct a candidate class list. Refine this list by elimination and renaming until you are confident that you have discovered the base classes. Now construct a class diagram for one or more of the use cases that you have previously identified (see Section 3.4).
2. Draw up a class list for Space Game. Refine this list and construct a class diagram for one or more of the use cases that you have already identified in Section 3.4.

4.7 Summary

After studying this chapter you should be able to:

- draw up a candidate class list from a requirements document
- state what characteristics a candidate class should have

- recognise class boxes, attributes and associations on a class diagram
- be able to use the full textual description of an attribute
- understand how an association on a class diagram relates to message-passing between objects
- draw simple class diagrams
- write class skeletons for simple class diagrams
- add multiplicities to associations
- use, where appropriate, a bi-directional association
- distinguish command operations from query operations
- state the meaning of *subtype*, *super-type* and *inheritance*.