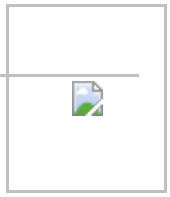


Software construction using objects

4.1 The software development process

4.1.1 How to study

How to study this unit

This unit is structured rather differently from the other units of the course. The bulk of the material can be found in the ten chapters of the booklet, *Software Construction using Objects. [link: TheSoftwareDevelopmentProcess/../Download/Contents.pdf]*

That booklet is your primary information source and the Web material is intended to coordinate and organise your study of it.

Your study material is organised into six topics, each of which subdivides into a number of sections. One topic is intended as representing about one week's study. There is a final, seventh topic that indicates what might come next, in a subsequent course.

We expect you to spend the bulk of your time working from the material provided in the booklet, *Software Construction using Objects* . This unit will offer regular *Learning Process Checks* to tell you how far we expect you to have got in reading and understanding it and any other material.

Learning Process Check

It's rather silly to imagine that all students work at the same rate. We ask you to spend enough time on this topic to complete it in one week.

4.1.2 Software development process

The development process

The business of constructing a computer system is a process typically involving several people and existing over a considerable time-period. Software development is only judged successful if it results in software artefacts that match the demands of the customers who commission and pay for it. Process control involves a continuous monitoring regime including checks on the software artefacts being produced and periodic steering reviews of the projects as a whole.

This unit relates to the analysis and design of software systems using object-oriented ideas. We are not primarily concerned with the intricate and complicated story of software process control, but you should be aware of some of the issues for your own project work.

Learning Process Check

Please read and master the first chapter [link: TheSoftwareDevelopmentProcess/../Download/Chapter1.pdf] of *Software Construction using Objects.*

As always, make such additional notes as seem appropriate to you and try to relate what you have read to ideas that you have encountered elsewhere. As always, if you have problems or queries, or even simple observations or corrections – post them to your tutor group forum.

After doing this, you should have a greater understanding of system development as a process and some feeling for what software models are and how they are used. Your grasp of both of these ideas will improve as this course progresses.

4.1.3 Questions relating to the software development process

Some questions to consider

1. Building a pyramid is a process. Identify *two* products associated with that process.
2. Identify *one* product of the pyramid-building process which is not a final deliverable.
3. Identify *three* process metrics associated with pyramid building.
4. The pyramid-building process would clearly have required some technical communication. Speculate on how some of this may have been arranged.
5. Would a **system diary** have been useful to **Imhotep?** (Imhotep was, allegedly, the architect and overseer in the construction of the first pyramid built at Saqqara around 2680 BC.)

Suggested answers from us

Suggested answers from us

1. Dressed stone blocks and bills for food for the workforce are two products of this process.
2. Design plans (they must have had some). Papyrus was available well before the time of Imhotep, so it is likely that plans would have been drawn on that. We can only speculate about notation but guess that plans would have been broadly diagrammatic rather than expressed in script form.
3. Number of workers injured per month. Number of blocks situated per week. Height of the construction at any given time.
4. Diagrams, dirt sketches, marks on stone blocks, human information repositories.
5. It would have been essential. Its form is something that we can only speculate about. Imhotep was later regarded as a god (something that does not typically happen to software engineers). Representations of him usually show him seated, holding an open papyrus in his lap. Interestingly, he was associated with the Greek god of healing and it

was for that reason that we offered data on injuries to workers in point 2. We wonder whether such diaries were prized by subsequent pyramid builders and confess ignorance on this section – though the fact that Imhotep is so often shown with a papyrus is intriguing.

The process of studying this unit

Studying this unit is a process and it is best approached actively. Suggest *three* process metrics that might be gathered by you to monitor your own process and construct *one* argument in favour of keeping a process diary, and *one* argument against it.

Please post your ideas onto your tutor group forum. We ask you to do this so that you can participate in the studying process with the people with which you will subsequently need to work, but we realise that not everyone has time to do this every time we request it. Please make an effort to do it as often as possible and include some time in your own studies for reviewing what other people have posted. One of the reasons we have for telling you how far you should get, on a week-by-week basis, is so that tutor group postings can be synchronised.

The development process

You need a background understanding of what the software development process entails in order to make sense of this unit but we do not purport to offer you more than basic information on this complicated subject.

Our primary concerns are the diagrammatic models associated with object-oriented development, together with some of the techniques used to analyse and design systems in object-oriented terms. This means that the sort of testing that worries us is always associated with **validation** (making sure that our models represent what is wanted), rather than **verification** (making sure that our code does indeed conform to what we said we would do). Validation testing is most important at the start and in the middle of a development process, while testing focuses increasingly on verification from the middle and towards the end of it.

4.1.4 Software models

Software modelling

A delivered software system ultimately consists of a long string of bits that makes sense to a computer system but is virtually unintelligible to any human in the world. This impenetrable, inaccessible data stream will have been produced by software professionals and a translation process. These take the functionality and facilities demanded by the commissioners of the software and produce the corresponding structures and interactions of a good system. The translation process always involves intermediate structures that abstract and represent system ideas in a way that is actually intelligible to humans – structures that can be used in the process of code construction. These intermediate structures are called **software models**. They can be as complicated as the design of a relational database or as simple as a few lines of pseudocode.

This unit is mostly concerned with software models appropriate to the analysis and design of object-oriented systems. Our models can be regarded as the sophisticated descendants of old-fashioned pseudocode.

Models

Software models represent some aspects of a software system intelligibly, while perhaps obscuring other issues. They are primarily devices for aiding human technical communication – a sort of communicational abstraction. An unintelligible or ambiguous model is worse than useless. Overcomplication is the disease of apprentice modellers.

When you yourself embark on modelling take this advice: *if you can possibly remove something from your model, do!* A model is finished, not when there is no more that one can add but when there is no more that one can remove.

Object-oriented models

Diagrammatic modelling is particularly important in object-oriented systems. You may yourself have some experience of diagrammatic models used in other contexts: flowcharts, entity-relationship diagrams, data flow diagrams, and so on. These ideas grew up organically from various sources over a period of many years.

Object-oriented development is a rather younger subject, and its most popular techniques were actually produced all together, as a result of a collaboration of the most frightening thing that one ever encounters in computing – a group of experts.

UML

The diagrams that we want to show form part of the **Unified Modelling Language (UML).** We believe that it is fair to say that there is disagreement about object-oriented analysis and design, and that there are several conflicting methodologies and opinions expressed with fervour reminiscent of the religious wars of the seventeenth century. Surprisingly, there is actually a widespread consensus regarding the appropriate modelling techniques for these systems and that consensus is centred on the facilities offered by UML.

It is for that reason that we base our account on UML diagrams and you should regard mastering these diagrammatic ideas as a major goal of your studies.

Use case modelling

Our approach to analysis and design is based on **use case modelling**. We do this because use cases are a popular entrée to various object-oriented methodologies and, we believe, highly accessible to people new to the subject.

Use case diagrams are deceptively simple. (We assert that in something of the spirit that led an English football manager to brand Terry Sherringham as ‘deceptively slow’ – that is, *even slower than he looked!*) The net result of hours of use case analysis can sometimes appear to be something so fatuous as a diagram in which a matchstick man contemplates an egg. This apparent simplicity is a real conceptual problem in teaching these ideas. The diagram, of course, is the product of a process of analysis and simplification and would ordinarily be supplemented with quite a body of additional data.

We attempt to give you some feel for this intensity of this process of analysis by relating our UML diagrams to the production of the software models that might occur during the analysis and design of a real system – the Porterhouse

Library [link: TheSoftwareDevelopmentProcess/../Download/ThePorterhouseLibrary.pdf].

4.1.5 Introducing objects

Introducing objects

You have probably already suffered many ‘introductions’ to these ideas in the past, none of which will have produced anything along the lines of a close personal relationship!

We therefore avoid the tedious business of regurgitating ideas you already well understand and, instead, request that you read through Chapter Two of *Software Construction using Objects* . You should read actively, making notes of things that are unclear to you and raising any issue that arise with your tutor or the other members of your tutor group.

We expect that most of the chapter will involve material that is already known to you and that you will not need to spend a great deal of time on it. The chapter closes with a small quiz that tests your appreciation of the information given.

Knowledge of objects is a prerequisite of the entire unit. We ask that you cover Chapter 2 as thoroughly as possible.

Points to note

These ideas occur in Chapter 2. You need to be familiar with them and have a clear grasp of them – that is, you should be able to explain them succinctly, and without effort.

1. Objects
2. Classes
3. Attributes
4. State
5. Behaviour
6. Methods
7. Messages
8. Method signatures
9. Polymorphism
10. Interfaces
11. Encapsulation
12. Inheritance
13. Overriding
14. Static methods and attributes
15. Constructors
16. Object references
17. Garbage collection

Discussion point

There are many different ways to represent an alphabetic character in computing – lots of different codes, fonts, sizes and styles. Use your tutor group forum to discuss whether these representations are best considered as attributes of a single object (such as an abstraction of the letter ‘a’) or as distinct objects (so that ‘a’ in 12-point Arial is a different object from ‘a’ in 10-point Times New Roman Italic). Your discussion should centre on the construction you place on the adjective ‘best’. Appropriate contexts might include data transfer, data representation and programming, and programming languages like Java do have primitive data types corresponding to individual letters of the alphabet.

Learning Process Check

Please cover all of the material up to the end of Chapter 2 [link: TheSoftwareDevelopmentProcess/../Download/Chapter2.pdf] of *Software Construction using Objects* .

After doing this you should be conversant with the main ideas of object-oriented programming.

4.2 Use cases

4.2.1 The Porterhouse Library

UML diagrams are intended to be accessible and simple, but the analysis involved in producing them is not. We hope to give you a feel for that work by relating our modelling ideas to a system specification for a library system in a small university.

The specification in question has been used by us to teach analysis and design ideas for a number of years; although one would expect a real system to have a much clearer **statement of requirements** because that would form the basis for any contract between the commissioners of the software and its developers. Nevertheless, the Porterhouse Library [link: UseCases/../Download/ThePorterhouseLibrary.pdf] offers more than enough scope to illustrate the important ideas that follow.

You should read the library specification and you will probably find it helpful to keep a printed and annotated copy of that for the work that follows.

Making sense of the Porterhouse Library specification

Please read through the library specification. You will probably find that there are several things in the specification that are unclear to you. These are the sorts of things that you would normally feed back to the system commissioners during the process of **requirements elicitation** but, of course, there is no possibility of that here.

Report *two* such issues to your tutor group. In each, explain what the problem is, and how you recommend that

it be resolved.

Learning Process Check

Please print out and read the specification for the Porterhouse Library. Make your own notes on the printout.

After doing this, you should have:

- Experience of a software system specification.
- Knowledge of what is wanted for the Porterhouse Library system.

We ask you to devote sufficient time to this topic to complete it in one week. The first two topics of this course should be covered in the first two weeks of your study.

4.2.2 Designing the system

Designing the Porterhouse Library system

Our goal is to construct an object-oriented implementation of the Porterhouse Library system. Here are some of the issues that we shall need to address.

1. What objects and classes do we need?
2. How are our objects to behave?
3. What data do our objects need to store?
4. How are we to implement data persistence and archiving?
5. What sort of user interface should we have?
6. What programming language and operating system should we use?

First steps

We address each of the questions that we have just raised in what follows but we think it would be very useful for you to consider what is needed, before we explain how one can treat these questions in a systematic way. Each of the questions asked is related to the others. It would be inappropriate to treat the questions sequentially and in isolation.

Please take about one hour to sketch out your own ideas relating to the answers to these questions, bearing in mind that your ideas must indicate the following:

- a clear understanding of system functionality and how it can be expressed in object-oriented terms
- some means of recording your ideas that would be intelligible to other people working on the same project as part of a development team

- some techniques for checking that your ideas make sense, do not contradict each other, and express the demands of the system commissioners

You are encouraged to post some of your analysis to the tutor group conference. If you do this, you can obtain feedback from the rest of your group and from your tutor.

4.2.3 Drawing UML diagrams

We shall discuss a number of different sorts of UML diagram in what follows and expect you to acquire expertise in constructing them yourself.

You can draw UML diagrams using the drawing facilities already available in Microsoft *Word* or, better still, the facilities in Microsoft *PowerPoint*. *PowerPoint* drawings can be saved in an image format: gif files would normally be appropriate. We produced some of the drawings in *Software Construction using Objects* (the activity diagrams) in this manner.

There are a number of free UML drawing tools available on the Internet and there are also some well-known, more sophisticated UML editors that one can purchase.

Violet is one particular UML drawing tool that we have found useful. *Violet* is available as a Java archive (JAR) file which you may download from Violet [link: http://www.horstmann.com/violet/].

We run *Violet* by means of a Microsoft batch file which contains the single line of text:

```
java -jar violet-0.15-alpha.jar
```

We built the batch file in *Notepad* and, of course, gave it the extension **bat**. You should also be able to run *Violet* simply by typing ‘Violet’ into your **Start .... Run** menu.

This only works if you have the appropriate version of the Java run-time environment available. If you do not have this, then you will also have to download it from: Sun Microsystems [link: http://www.java.com/en/download/manual.jsp]. We needed to do this ourselves and can report that the process was entirely painless!

Violet

Violet is a useful little tool. You can use it to construct all of the UML diagrams that we show you in this unit, with the exception of **activity diagrams.** We produced most of our UML diagrams with *Violet*, saving the images as jpeg files. (At present, *Violet* does not support gif.) We encourage you to try it out but offer no help in using it. Our opinion is that you should be able to sort it out for yourself and that if you run into problems, the students in your tutor group will probably be able to help you.

There are other free UML editors available and some IDEs have facilities for producing UML.

Installing Violet

This is an optional exercise. Download and install *Violet* (updating your version of the Java runtime environment if you need to do so). Practise drawing the following sorts of UML diagram without worrying about their significance or meaning:

1. Use case diagrams
2. Class and association diagrams
3. Sequence diagrams
4. State diagrams

(You can find examples to copy in *Software Construction using Objects* .) You should be informed that one creates new diagrams in *Violet* by taking the **New** option from the **File** menu.

Save any one of your diagrams as a jpeg file by means of the **File … Export Image** option. (You do this by setting the file extension to jpeg.)

When you finally manage to create a jpeg file, embed it within a *Word* document using **Insert … Picture.**

Learning Process Check

Please read through Chapter 3 [link: UseCases/../Download/Chapter3.pdf] of *Software Construction using Objects* . Please also experiment with drawing UML diagrams. You will find many of them in the course booklet. You don't yet need to be able to understand them, but it is important that you are in a position to draw your own and you would be well advised to consider using *Violet*.

After doing this you should:

- Have some knowledge of what UML diagrams are.
- Have some idea how you are going to draw the UML diagrams that you need.

4.2.4 Use cases

The primary way to approach analysis and design is through consideration of system functionality.

The functionality of a system relates to what the system actually does. When people first began to consider the process of requirements elicitation, their initial response was to describe systems in terms of function – activities that engendered code. Only later was it realised that other aspects of the system were vitally important; including the non-functional requirements associated with interfacing, response times, support, extensionality, maintainability and so on.

One way to specify system functionality is **use case analysis**. This originated with object-oriented analysis and design but is much more widely applicable than that and does not really imply an object-based approach at all.

Use cases specify the following things:

- a specific use of the system
- entities external to the system which have this use of it (which are called **actors** and do not necessarily correspond to human beings)
- the system boundary (useful in scoping)

Actors and scoping

People starting out in system design often fail to appreciate how important scoping is.

Actors are formally *outside* the system. If you were designing an ATM database system, you might conceivably regard individual ATM machines as system actors – because it would not be your business to design their code.

So, use case analysis requires you to decide about actors, and because actors are not part of your design considerations, your analysis clarifies what is and what is not within the scope of your design.

Actors are defined by their rôles. A particular human being might take several rôles in their intercourse with the system. Systems recognise actors, not humans.

Use case analysis and use case descriptions – an overview

The system requirements should give you an idea of the system's main functionality and that should permit you to produce a list of use cases describing the main facilities that it is to offer.

There is some skill to acquiring a list of use cases. To illustrate: a typical use case would be **logging into the system ,** but a moment's thought will tell you that that particular piece of functionality actually splits up into several possibilities (called **scenarios**).

The **main success scenario** is that logging in is actually achieved. Passwords can be wrong or facilities suspended, however, so many other possibilities can occur, although they might be deemed less likely.

In this context, it would be almost certainly be appropriate to handle the different scenarios in the one, logging in, use case and you will be able to guess how closely related the scenario coding is likely to be.

Use case analysis involves a detailed consideration of what is involved in a use case. Along the way, you might decide that the use case is inappropriate, should be subsumed by an existing use case, or should be split into separate use cases.

The product of your use case analysis will be a comprehensive account of what happens. You might document this in the form of a **use case description** – there are examples in Chapter 4 of the course text.

Members of your development team need to have the various use cases identified in some clear way. This can be done by means of a simple **use case diagram** – the matchstick man staring sadly at an egg that we mentioned earlier. These diagrams can also indicate system boundaries. Don't be fooled by them because they are surprisingly useful, but they must be accessible to every member of your team, and kept up to date.

When not to use use cases

Use case modelling is probably inappropriate in some systems, since not all systems are so clearly associated with

actors. (This often applies to systems associated with scientific processing of data, depending on what it is that they do.) Most of the problems that you are likely to see will probably best be handled by this technique, however, and it forms the basis of much of our subsequent work, so you are advised to take pains to master it.

Learning Process Check

Please read Chapter 4 [link: UseCases/../Download/Chapter4.pdf] of *Software Construction using Objects* . Please practise drawing use case diagrams.

After doing this, you should:

- Understand what use cases are.
- Be able to draw use case diagrams.
- Have some feeling for what use case analysis entails.
- Have encountered your first activity diagram.

Use case analysis is the cornerstone of the analysis and design process that we are describing. It is vital that you master it.

4.2.5 First steps in use case analysis

Some issues arising from Chapter 4

Please consider the following issues carefully. We offer our own solutions at the end of this page.

1. We assert that it would probably be misleading to have an actor in the Porterhouse Library system with the title ‘Borrower’. Why do you think we say that?
2. Describe a scenario of the suggested ‘Borrow Books’ use case in which books are not actually borrowed.
3. The ‘Locate Account’ use case mentioned in Figure 4.3 of *Software Construction using Objects* might occur in several use cases. Name one other one.
4. We insisted that you should only use the notation we showed you in your work for this unit. Give two possible reasons for our doing that.

You do not have to post your answers to your group forum. You are welcome to do so, and to make whatever comments you choose about our questions and solutions.

Return Books

We ask you to conduct a preliminary analysis of a Porterhouse Library use case called ‘Return Books’.

You should attempt to deliver:

- an identification of the actor involved

- a use case diagram, similar to Figure 4.3 of *Software Construction using Objects*
- a list of possible scenarios similar to Scenarios 4.1–4.6 for the use case ‘Borrow Books’ in *Software Construction using Objects* .
- identification of the main success scenario
- a use case description similar to the tabular one given in Chapter 4

Our answers to the issues raised above.

Our answers to the issues raised above.

1. In the system described, borrowers do not address the system to borrow books. What they actually do is talk to library staff, and it is library staff that deal with the system. They are the actors, and it seems odd to describe a librarian as a borrower!
2. Scenarios 4.1–4.5 cover this. More importantly, our data definition is going to have to decide what a Book actually is. The view we shall take is that a Book is an abstract idea and that the library actually deals in things that we shall call **Volumes** (individual copies of a Book). So, a particular Book might exist as several Volumes. Customers don’t borrow Books – they borrow Volumes. However, Customers probably do reserve Books!
3. Other possible use cases include: ‘Return Books’, ‘Pay Fine’ and ‘Make Reservation’.
4. Possible reasons include extreme laziness on the part of the author (not wishing to explain other ideas), the fact that group work is a central part of this course (and you must all speak the same language), the fact that our poor tutors cannot be expected to spend time making sense of whatever you throw at them, and the fact that organisations often place restrictions on what you can use, for conformity reasons.

4.3 Modelling objects

4.3.1 Class diagrams

Object modelling

Use cases describe system functionality in high-level terms. We now have to start considering how we make that functionality happen. Points 5.1–5.5 of Chapter 5 of *Software Construction using Objects* gets us started on this; and you should read Chapter 5 before starting this topic.

Since we are dealing in objects, it becomes necessary to discuss how to represent the system we are designing in their terms. This is another sort of modelling – using software structures as representatives of real-life things. Object-oriented systems are commonly supposed to facilitate representations of this kind.

Take pains to distinguish the models that we are building as artefacts of the design process, which are used to communicate and record our ideas, from the objects that we shall code to represent our take on the real-life structures

of the system we are designing. There are two sorts of model here, with the design artefacts informing the construction of the software data structures.

So what objects does our system need and how do we obtain them? At present, we could imagine having:

- a complete list of use cases and their descriptions, representing system functionality
- the original Porterhouse Library statement of requirements

These assets can be used to construct a list of the objects that our system is likely to require.

What we really need to do is design the objects that we need. What concerns us is the construction of classes. Points 5.6–5.15 of Chapter 5 gave a list of candidate classes for the Porterhouse Library system. As you will see, our later analysis will reject some of them.

Our analysis : Your analysis

We said that our analysis of the Porterhouse Library system would reject some of the classes listed in Chapter 5. We do not want you to imagine from this that the classes we reject would be rejected by everyone else looking at the system. Our particular viewpoint (as expressed in *Software Construction using Objects*) throws out some classes that others might actually embrace; and it's also possible that another designer would come up with classes that had not occurred to us. There is no right or wrong way to produce a system design – the only criterion for success is utility.

Learning Process Check

Please read and take pains to understand Chapter 5 [link: ModellingObjects/../Download/Chapter5.pdf] of *Software Construction using Objects.* This is probably the densest chapter of the booklet and it contains quite a lot of information that may be new to you. You should expect to take five or more hours reading and re-reading it, and you are very welcome to read the chapter interactively, posting questions to your tutor group forum as they occur to you. Please do answer similar questions posted by your fellow students, if you can. Don't be afraid of being wrong – it never worries us!

After doing this, you should:

- Have some idea of the central concerns of the analysis process that we are describing.
- Understand associations between classes, and their multiplicities.
- Understand collaborations between objects.
- Understand class and object responsibilities.
- Be able to use, use cases and specifications to produce lists of candidate classes.
- Understand what walk-throughs are, and how to use them.
- Be able to understand and draw UML class diagrams.
- Be able to understand and draw UML sequence diagrams.
- Know about CARC cards, be able to make sense of them, and construct your own.
- Understand what system glossaries are.

This is a difficult topic and we ask you to complete it in one-and-a-half weeks of study.

4.3.2 First steps in class and association analysis

Test yourself on Chapter 5

1. Reproduce Figure 5.1 but do this using the UML convention for representing the class or type of variables rather than the Java notation.
2. The section of the chapter that we called ‘Responsibilities’ asserts: ‘Commercially, your job would be rather different from that – it is to deliver software that does what you agreed it would do and not very much more.’ Write down *five* of your own observations relating to this assertion.
3. What is the distinction between a system diary and a system glossary?
4. Construct a CARC card for System_User and then post it to your tutor group forum for comments from your fellow students. Please attempt to do this – student participation is a vital aspect of the course!
5. Post your own comments on the CARC cards posted by your fellow students.
6. Raymond Flood kindly offers to buy Bob Lockhart a large cup of Java coffee. He goes to the shop and returns with a cup; but despite the coffee having been ground that morning, it tastes like mud. Raymond returns to the shop and requests another one. He brings it back, and it is just right. (We hope this scenario makes you feel happy.) Record this whole transaction as a UML sequence diagram.

Answers to the self-test

Answers to the self-test

1. You can check for yourself whether the figure you produce does indeed mimic Figure 5.1. It’s important that you acquire some expertise in drawing these diagrams, which is why we asked you to do it.
2. Here are our musings on this.
    a. If you individually include facilities different from those specified, then it is unlikely that they will be properly documented and tested.
    b. If you stop your programmers doing anything creative they will probably be very poor, and the good ones will leave. (Herein lieth the disease of British software development!)
    c. Additional functions need to be maintained, supported and documented. This costs money.
    d. You cannot order a programmer who is going to be the one hauled out in the middle of the night when things go wrong *not* to put in comfort coding that makes the code more stable and reliable; and designers are often incredibly unimaginative.

e. All the books agree that you code exactly to agreed specification.

f. Books are written by authors, not programmers.

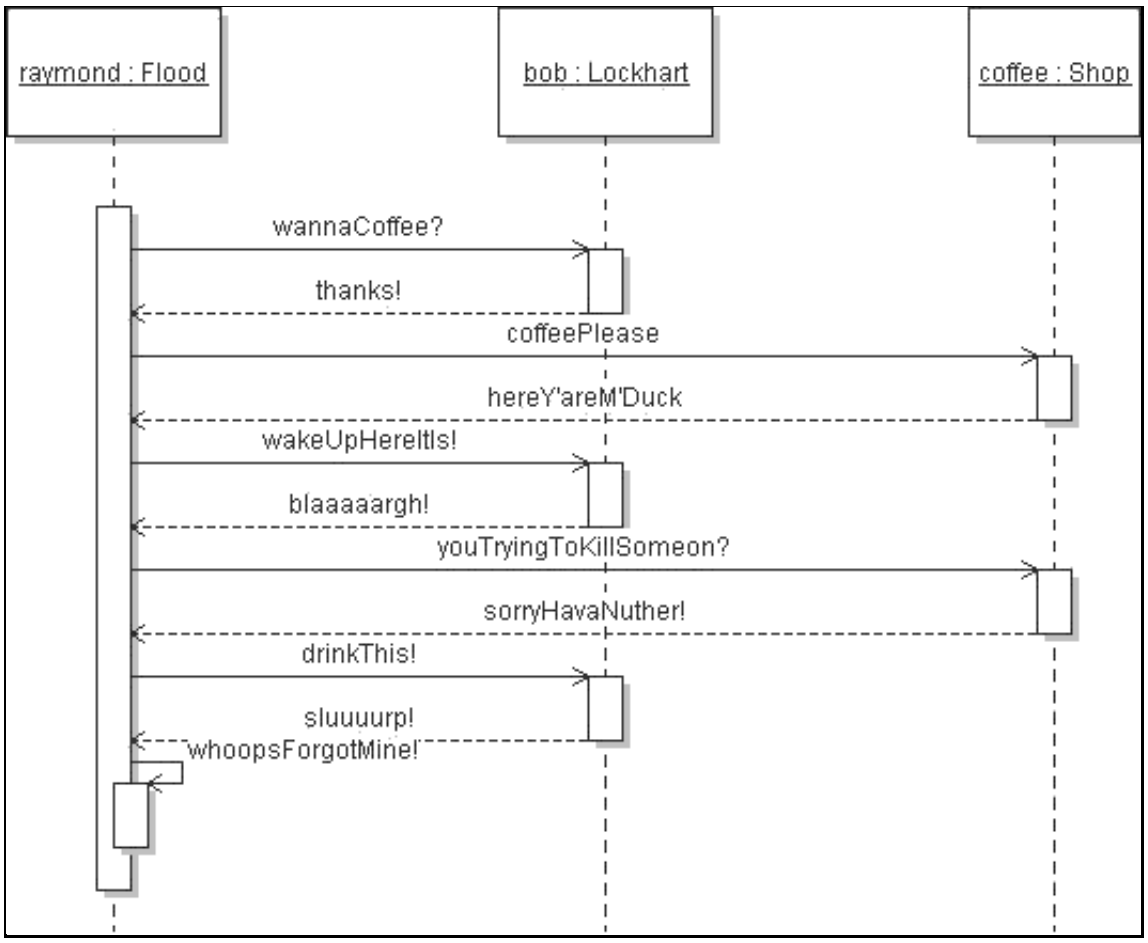
3. System diaries are dynamic; they record the development process. System glossaries are static; they record the final agreement on the meaning of the data and its services.

4. Covered by your fellow students and tutor.

5. Covered by your fellow students and tutor.

6. Here is our diagram.

Sequencing coffee



Happy days in Summertown.

Image created by Bob Lockhart for this course.

4.3.3 Classes associations and responsibilities

Functionality and classes

One problem of the object-oriented way of looking at things is that it invites one to concentrate on the individual class or object at the expense of taking a wider view of system activity. In object-oriented analysis, one must translate system functionality into interplay between objects and classes but the stress on classes makes this fairly difficult to do.

When we have a clear idea of how it is to be done, we may use sequence diagrams to record it. Sequence diagrams, like all models, introduce simplifications. It would not be practicable to show *all* the internal messages passed between an object and itself on the way to achieving something (although we do show an internal message in Figure 6.2 of *Software Construction using Objects* , for example). What one would often do in a sequence diagram is show the **interface** methods (essentially, the Java **public** methods) and explain that they themselves would organise further internal messages appealing to **private** or **protected** methods that are only available to a restricted set of objects of the system.

One would probably not show class messages at all. These tend to be restricted to the provision of utility services; and their use can be recorded in annotation given elsewhere.

Connections between objects

Modularity is implicit in object-oriented systems and hinges on classes in some sense representing a semantically integrated concept or functionality – most usually, something directly related to the entities that are being modelled.

The client-server view of things is another cliché of object modelling: an object requires a service of another object and obtains it by sending a message. In order for that to work, there has to be some prior connection between the two objects. More specifically, the client must have previous knowledge of the existence of the server.

It sometimes happens that this prior knowledge comes from the client actually constructing the object whose services it needs. This kind of functionality is vested in the creation of an object whose constructors do the job required. You will see an example of it in our detailed analysis of the 'Issue Loan' use case, at the start of Chapter 7. In our system, Customer objects are created when needed and their attributes are set up from the persistent storage offered by relational databases. You should now re-read the addendum at the end of Chapter 5 and make sure that you understand what it is we are saying there.

More commonly, objects actually store references to other objects with which they have dealings. This is how the organising object **admin : Admin** (first mentioned in Chapter 5) arranges the access of the system by librarians. The single Admin object has an instance variable we called **system_Users ,** which stores a collection of references to System_User objects. This attribute is typed to some kind of Java collection class in our example. (We actually used a Vector.) It is set up by the Admin class constructor, when the system starts.

All this comes down to an **association** between the Admin and System_User classes. Admin **knows_About** System_User(s). The single admin : Admin object will store references to anything between zero and 'many' System_Users. This particular association is represented in Figure 5.3 and, accordingly, has **multiplicity** one to many.

Collaboration

Objects having a client-server relationship are said to be **collaborators**. Collaboration occurs when a class (or its objects) needs the services of another class (or its objects) in order to discharge its own responsibilities.

It is important to be able to identify collaborations between classes because collaboration is a rather more general idea than association. A class can collaborate with another class simply by providing services, and without any other connection between them. Of course, if such collaboration exists it is vital that changes that might affect collaborators are identified and managed.

Organising objects

Our response to the process of translating functionality into object behaviour makes quite a fuss about objects knowing about each other, and the client-server view of things. One aspect of this particular viewpoint is that systems designed in this way often have single organising objects which control what happens, and this turns out to be the case in the system we are designing for the Porterhouse Library (where the special object is **admin : Admin**).

This asymmetry in object design is somewhat against the spirit of distributed computing and probably not very appropriate for some circumstances. It is not an essential aspect of this viewpoint, but it is a common by-product of it. You should be informed that there are other design methodologies which don't tend to exalt a particular organising object and which are perhaps more suited for distributed programming. We shall not say more about this topic here.

4.3.4 Responsibilities of classes

Responsibilities

The client-server view of things requires that the client 'has knowledge of' the server and that the server can offer the services required. It places a responsibility of **knowing** onto the client and a responsibility for **doing** onto the server.

In the Login use case, handled by the sequence diagram in Figure 5.4, admin : Admin **knows_About** System_User(s), and System_User(s) are able to respond to a message called **checkLogin** by checking the proffered user name and password Strings with their own **userName** and **passWord** attributes. (Figure 5.3 gives more information about parameters; we left them out in Figure 5.4.) Here the admin : Admin object is client to several servers (the System_User(s)).

We could list quite a few responsibilities for objects from the System_User class (or its concrete subclasses). They must:

- remember their own user name
- remember their own password
- be able to respond to a message checking a proffered user name and password

Clearly, class members have other responsibilities; but these are the ones that relate to the Login use case.

Summary

We have not given a very detailed account of the idea of responsibilities. Important analysis and design methodologies lay great stress on the idea. We hope some aspects of it are useful to you in the design work that follows.

4.3.5 CARC cards

CARC cards

Class, Association, Responsibility and Collaboration cards record everything that we know about individual classes. Since one continually refines that knowledge during the analysis and design process, they build in version control and identify an individual with responsibility for them.

Our CARC cards are much more complicated than typical ones because we include a great deal of additional information in them and make their construction a major part of our analysis and design process. They contain the following ingredients:

- The class name and status.
- The date of construction of *this* card.
- An identified author for *this* card.
- Information about superclasses and subclasses.
- An informal textual description of what this class is for and what it does.
- Information on associations involving this class (that is, links between this class and other classes that may translate into possible relationships between the objects of this class and the other classes).
- Information about the responsibilities that this class undertakes (that is obligations imposed on the class by our design).
- Information about collaborating classes (classes or objects from those classes that provide services for this class associated with the class discharging its responsibilities).
- Interface methods – **public** accessors associated with this class or its objects. (We are happy that you include **constructor** information here, if it is appropriate.)
- Additional notes. This should be structured text (numbered points). Its key characteristic is that is should be clear to those who are likely to be reading it.

Visibility

The analysis and design methods that we have been discussing involve the production of some key documents such as use case diagrams and CARC cards. There is no point in producing these documents unless they are easily accessible to the people involved in producing the system.

4.4 Interfaces and Data

4.4.1 Interfacing

Separable User Interfaces

The rise of graphical user interfaces has occurred over the past 20 years and has resulted in a degree of commonality with respect to the way in which people access systems.

The effect of this commonality has been to separate interfacing ideas from the processing associated with any particular application. The resulting design pattern – **separable user interfaces** – enforces that separation. User interfaces are standardised and produced by standard techniques. Any particular **domain model** may have a number of

possible interfaces, none of which has any direct relationship to functionality.

Learning Process Check

Please work your way through Chapter 6 [link: Interfaces/../Download/Chapter6.pdf] of *Software Construction using Objects.*

After doing this, you should:

- Have some knowledge of Windows, Icons, Menus and Pointing devices (WIMPs).
- Have an overview of Java events and how user input is transferred to Java programs.
- Understand the drag-and-drop way in which interfaces can often be produced.
- Have some appreciation of the **Model View Controller** design pattern.
- Appreciate how sequence diagrams can be used to represent interface storyboarding.

This is a fairly light topic. We ask you to devote sufficient time to it to complete it in half a week of study. You should reach Topic 5 by the start of week five.

4.4.2 Practical interfacing

Interface proving

We have advocated separating interface concerns from the part of the system that handles processing, but the interface is probably the aspect of the system that most influences human perceptions. Our separation policy means that one could potentially trial various interfaces with the humans who have to use them without influencing functionality in any way; but it is as well to remember, that although the users are probably very different from the people who commission the system, their ability to 'vote with their feet' can snatch defeat from the jaws of any functionality victory. It is not unheard of for systems to fulfil contractual requirements but never get used.

Designing a good interface is an HCI (Human Computer Interaction) issue and not something for which we have space here, but trialling various possibilities is something we can talk about. It's easy to do, makes users feel consulted and can often reveal considerations that may even influence the domain model (a reality that the separable interface pattern suggests is unlikely to occur). You should trial interfaces as a matter of course in any system you design. If you do conduct trials, you should take them seriously, which typically involves recording user response quite formally, reacting to it and regression-testing the results.

We should stress that it is the formal separation of the interface and the domain model that makes interface trialling so easy. Trial interfaces can be mocked up – storyboarded, perhaps even in paper form. This gives the users an idea of what to expect and focuses concentration on interface design issues.

4.4.3 Taking stock

Where we are now.

We have now covered the essentials of what we have to say about analysis and design using objects. You should have covered the first six chapters of *Software Construction using Objects* and acquired some skill in drawing most of the UML diagrams that concern us.

We face another four chapters of the course booklet, but only Chapter 10 is likely to be completely new to you. We invite you to pause now, review how far you have come, and revisit ideas that still seem difficult or unclear. Now is the time to sort out problems and you should use your tutor and fellow students to help.

Learning Process Check

Please look through the first six chapters of *Software Construction using Objects* .

You should be able to summarise the content of each of these chapters with a single sentence, and then to list a few key ideas. We invite you to try to do this, and to post your answer to the tutor group forum. This activity make sense in relation to the response it attracts. Our assumption is that different students will identify different features of the text and that a lot of the value of this exercise comes from noting other people's ideas. You might also like to report on anything that seems unclear to you at the same time. Unit 5 of this course involves joint work with your fellow students, so take this opportunity to get to know them and to have a first go at working with them.

4.4.4 Data

Persistent data

The Porterhouse Library system will need to maintain data. Its requirements specification speaks of book catalogues and borrower sheets, and the system we are designing will have to sustain information about the user names and passwords of the people able to access it.

Looking at the data requirements

- Make a list of the different sorts of data storage requirement the system has and how you suggest they be arranged.
- One data requirement raised but not answered in the specification involves informing borrowers of the return dates for Volumes that they borrow. Suggest how this could be arranged.

Learning Process Check

Please read Chapter 7 [link: Interfaces/../Download/Chapter7.pdf] of *Software Construction using Objects* .

This topic will occupy the fifth week of your study and involve you in reading Chapters 7 and 8. We only ask

you to read Chapter 7 at this point.

The topic will involve some postings to your tutor group conference. The group-working part of this course is getting nearer. We need you to start talking to your fellow students and to exorcise any reticence you might have about ill-considered ideas or errors. Please do make an effort to respond when we request it.

4.4.5 Appearance and reality

The traditional view of an object-oriented system involves objects that are physically present at all times (whatever that means!). We have suggested a rather different scheme for handling objects that relate to Volume(s) and Customer(s).

- Explain what we have suggested.
- Criticise it.

It would be helpful if you could post your comments to your tutor group forum and read what others in your group say about this. At this point in the course, you are probably rather weary of doing this sort of thing. We ask it of you for three reasons.

1. a. We think the issue is important and need you to have a clear view of it.
   b. Group working becomes increasingly important in this course and we need you to interact with your fellow students.
   c. You will have to express ideas of this sort when you do interact. Practice helps!

Don't spend a lot of time on this; we are after a fairly instantaneous reaction based on your response to Chapter 7. The issue is one of clarity and communication, not one of correctness and assessment!

4.4.6 Relational data

Learning Process Check

Please read Chapter 8 [link: Interfaces/../Download/Chapter8.pdf] of *Software Construction using Objects* .

4.4.7 The Porterhouse Database

You have now read our arguments in favour of having a relational database within the Porterhouse Library system. Such a facility would offer processing facilities that are quite independent of the system we have been designing - so that one could arrange book-searching or loan statistics queries without any recourse to our software and interfaces. Of course, this might not be a desirable thing to do because it might intrude on our processing.

The problem of riches

We ask you to imagine that you are a software designer and that you have just informed your customers that you recommend that the library system include a relational database. Their technical expert is not very technical, and not much of an expert, but he sees a world of database queries opening up before him, and gets visibly excited. He knows little about objects, but is a king of *Access* (or perhaps, a prince in waiting).

What should you do about this?

- Ignore it as something beyond the scope of the system being commissioned.
- Regard it as something likely to impact on the perceptions people have of what you deliver and include written warnings about database use in your delivery.
- Arrange a set of procedures relating to exterior access of the database so that the *Access* king at least does no harm.
- Explore including software checks to preclude danger of this kind (involving your company in additional expense that, arguably, is off-contract).
- Perhaps something else, that has not occurred to us.

Post your comments to your tutor group forum.

4.4.8 System glossaries

Understanding the data

We mentioned system glossaries in Chapter 5 and while we are talking about the persistent data the *system* requires, it might be worth going back to the persistent data our *development process* requires.

You should certainly be able to give a brief textual account of the following ideas.

- System_User
  - System_Manager
  - Issue_Clerk
    - Head_Issue_Clerk
- Admin
- Customer
  - Academic
  - Non_College_Member (dreadful name!)
  - Student

They correspond to Java class definitions.

You should also be able to give an account of things like:

- Entry Frame
- Loan Frame

These are the frames that appear as part of the GUI. The system works by means of a dialog involving those frames. The dialog is centrally organised by admin : Admin.

You should be able to describe such things as:

- Volume
- Book
- Loan

These appear as database tables and are not actually realised as Java classes but might be in some future development of the system.

Your glossary should give an informal account of the system's use cases, its actors, and every class. It should define the terms used in the specification (such things as reservation, loan, fine, etc.) and cover any additional technical terms that seem important. (There might, for example, be an entry for ISBN.)

It might be appropriate to add some structure to the system glossary. If you are going to include interface methods, for instance, you might well be advised to do that in a more technical expansion that could be referenced from the main definition area.

Your glossary might well include **procedures relating to system use** (something that partially came up in the last session when we discussed direct access to the database by non-system means). You will be clearer about this after reading Chapter 9 when we discuss system start-up in relation to state diagrams. Our policy will be that the system itself starts in a state that precludes loans. A privileged user needs to login after start-up and set the system state to permit normal operations. This procedure needs to be recorded in the documents that are delivered to the commissioner and relate to system use (a handbook, perhaps), but they might well be included in an extended system glossary. Procedures relating to system use are sometimes called **Business workflows**. (You can find a bit more about them in Chapter 9 of *Software Construction using Objects*).

System glossaries

- System glossaries must be comprehensive.
- System glossaries must be intelligible.
- System glossaries are semi-technical documents.
- System glossaries are intended for everyone associated with the development process and, perhaps, the technical staff of the commissioners.
- System glossaries need to be easily accessible.
- One person in the development team should take personal responsibility for the glossary.
- A prime consideration of the glossary is that it should be up to date.
- Another is that it should be internally consistent.
- A third is that it should become clearer and more readable as the development process continues (something

that, in our experience, is rarely the case!).

Learning Process Check

You should now have reached the end of your fourth week of study and covered the material up to the end of Chapter 8 [link: Interfaces/../Download/Chapter8.pdf] of *Software Construction using Objects* .

4.5 Workflows, errors and testing

4.5.1 Further issues

Learning Process Check

Please read Chapter 9 [link: WorkflowsErrorsAndTesting/../Download/Chapter9.pdf] of *Software Construction using Objects* .

The work for this topic should take you about one week and should occupy the fifth week of your study.

4.5.2 State diagrams and activity diagrams

State diagrams represent the state behaviour of objects from a single class and how they change state in relation to events.

Our account of state diagrams probably emphasises the individual object rather than the class and we give a somewhat watered down account of these ideas. State diagrams are closely connected to **activity diagrams** – which describe system and business workflows and how they are coordinated.

Violet has facilities for drawing state diagrams but the latest version does not permit you to construct activity diagrams. We produced ours using the drawing facilities of *PowerPoint*, and we saved it from *PowerPoint* to a gif file.

Constructing an activity diagram

Construct an activity to describe the **business workflow** (see Topic 4 Section 8) relating to a library member either borrowing or returning a book. (This activity is based on a diagram we noticed in the book *Using UML*, by Rob Pooley and Perdita Stevens (referenced in *Software Construction using Objects*). Our solution is a slightly modified, re-drawn version of their diagram.)

This exercise gives you a chance to use synchronisation. Your diagram should show:

- two swim lanes: one for the library member, one for the librarian
- activity relating to book borrowings and book returns (so, a borrower must be shown locating the book first)

- the member having to queue for service
- the librarian cancelling the loan and organising the physical return of the book in the case of returns
- the librarian issuing the loan in the case of borrowings
- the librarian either ceasing the activity or returning to a state of waiting for the next customer

We shall give you our solution (based on that of Pooley and Stevens) in the Topic 5 Section 2.

4.5.3 Errors and exceptions

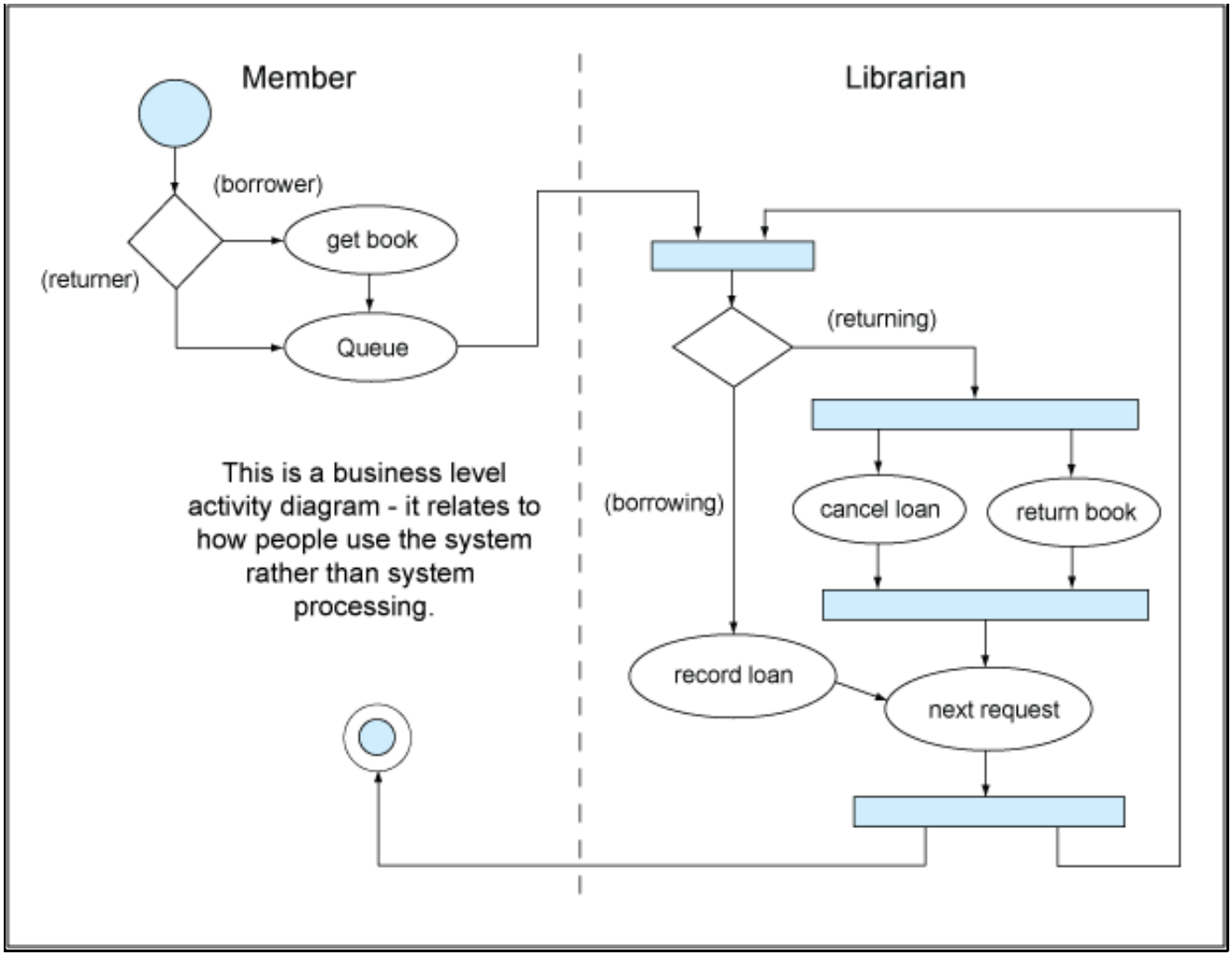
Our library system involves the maintenance of static data and that always raises operational problems. Data can be inconsistent, missing and corrupted, while data-set problems increase when data-sets are distributed over several locations. Moreover, state-related behaviour offers the possibility of situations arising that had not been expected. (This happened with the Ariane 5 spacecraft disaster.)

Modern programming languages, particularly the object-oriented ones, offer facilities for handling some classes of run-time problems (particularly those associated with the sort of integrity problems that remote access involves). You should formally consider how to cater for errors in any system you design and, in our opinion, place your system users at the forefront of your considerations.

One solution to the activity diagram question posed in Topic 5 Section 2

This is a slightly modified version of the diagram that we saw in Pooley and Stevens.

Business workflow – borrow/return books.



An activity diagram copied from Pooley and Stevens.

This diagram is based on one cited in Topic 5 Section 2 but was constructed by Bob Lockhart.

4.5.4 Testing

Much of this unit has concerned itself with techniques used in object-oriented analysis and design. Predictably, we have paid insufficient attention to testing – all authors do.

The artefacts of our development process have mostly been software design models partially expressed as UML diagrams. The primary question has always been 'Is that really what is wanted?' and we hope that it is fair to say that we have attended to that throughout *Software Construction using Objects* and throughout this unit. In that sense, we have actually said quite a lot about **validation** testing.

What would come next is the implementation of these models. Then the issue becomes one of checking that they do in fact code what we said they would – we would need to go on to emphasise **verification** in our testing process.

We said, right at the beginning of *Software Construction using Objects* , that the armoury of weapons provided by UML had been partially constructed with consistency in mind. Consistency is a major issue in the development of the sort of modular systems that object-oriented ideas encourage – the advantages of being able to localise issues are partly outweighed by the problems involved in working in isolation. As always, we must strive for a happy medium in these concerns – and proving the consistency of complicated or distributed systems may often resemble the conduct

of a séance!

4.5.5 Summary

Where we are

We have now covered our introductory course on object-oriented analysis and design notations. Along the way, we have treated you to an informal account of an analysis methodology based on use cases, and mentioned such things as responsibility-based design. The process we have described results in a clear set of products – UML diagrams, structured technical descriptions, glossaries and reports. We have also discussed some informal validating and trialling methods and introduced you to ideas that could blossom in further courses.

We have attempted a common-sense view of things and we hope that you could now list a number of skills that you have acquired in the process of working through this unit. More than anything else, we hope you have enjoyed reading it – because a good way to obtain a poor system is to employ sullen, bored, regimented people.

Where we might go

Further courses on object-oriented issues would pursue some of the things we have discussed more rigorously and introduce fiercer methodologies. Our account is short with respect to the development process itself; and we are conscious that the requirements elicitation and the interfacing stages have been dealt with tersely – there is so very much more to tell but unfortunately we must leave that to other courses.

Learning Process Check

You have now covered the major part of this unit and obtained all the information that you will need for your project work.

The final topic we discuss relates to distributed computing and is highly introductory.

You should now be at the end of week five of your studies.

4.6 Distributed computing

4.6.1 Distributed computing

Learning Process Check

Please read Chapter 10 [link: DistributedSystems/../Download/Chapter10.pdf] of *Software Construction using Objects* . This chapter is fairly technical but probably contains a lot of material that you already know about. You don’t need to have more than an overview of it.

We suggest that you cover this unit as quickly as you can and devote whatever time remains to you to your assignment and the consolidation of the previous work.

4.6.2 HTTP

Enterprise computing

Enterprise computing is essentially web computing. It's presently an area where programming skills are in demand and it involves an understanding of object-oriented ideas and the problems of distributed computing.

Java has a large set of pre-written facilities, and pre-designed structures, that target Enterprise computing – this is usually referred to as **Enterprise Java**.

HTTP

The engine that drives Enterprise applications is the **Hypertext transfer protocol (HTTP).** It has the following features:

1. It is an application layer protocol sitting above the Transport services provided by TCP and UDP.
2. It was originally designed as a simple document retrieval system.
3. It is based on the client-server paradigm.
4. It is stateless.
5. Protocol information is transported in plain text headers and is very simple.

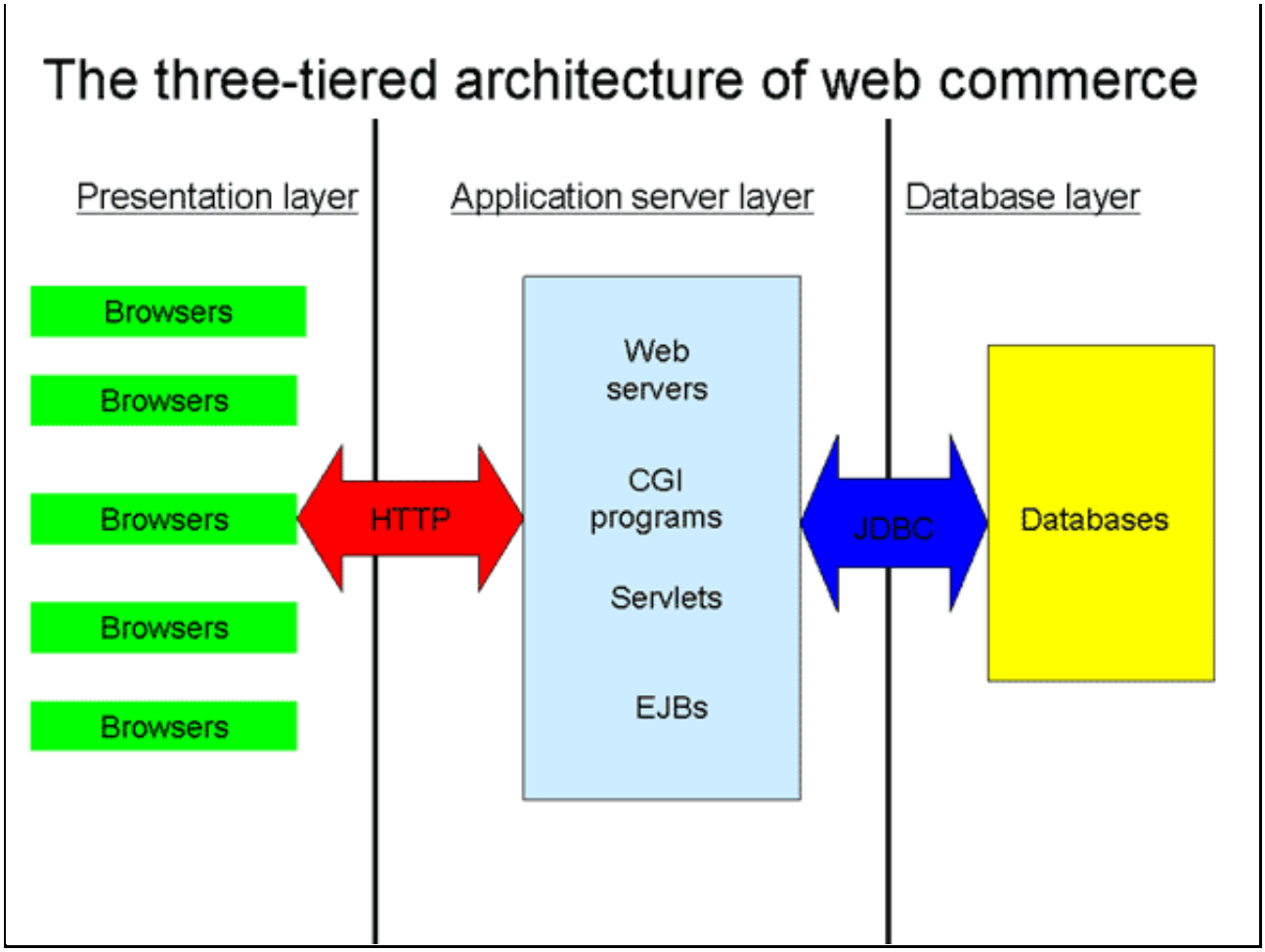
These features of HTTP have influenced the past 15 years of Web processing development.

4.6.3 The 'three-tier' model

One consequence of using HTTP is that client-side processing is problematic. It is done, but only in a limited way, and there are also security issues associated with any serious attempt to extend client-side processing.

A typical web application involves a three-tiered architecture. The client browser really just controls request dialogs and handles FORM data. The server side splits logically into some sort of presentational and processing function, and a repository of information. We reproduce Figure 10.3 from *Software Construction using Objects* to illustrate this.

The three-tiered architecture of web commerce



An illustration of the three tiers.

Image constructed by Bob Lockhart for this course.

4.6.4 The distributed library

Extending the library system to the web

Chapter 10 makes a few hesitant steps in the direction of extending some of the facilities of the Porterhouse Library to the web. Of course, it's inappropriate to imagine that all use cases could be handled in that way – some of them require the physical presence of books! You could certainly imagine the following sorts of function being available over the web.

- Interrogating your own account for loan and credit information.
- Paying fines.
- Searching the book catalogue.
- Determining whether a particular Volume was available for loan.
- Reserving books.
- Making acquisition requests.
- Suspending borrowing, or reinstating it (the chief librarian might like to do this from home).
- Issuing information newsflashes (another facility librarians might enjoy)

Our analysis of the Porterhouse Library system was based on use cases and, as so often happens in that kind of analysis, involved a coordinating object (admin : Admin). This is less appropriate in distributed settings and there is some discussion of this in Chapter 10.

The preliminary solution we offered there involved Java **servlets** (server-side programs) accessing databases and returning web pages relating to state progressions.

Our discussion went on to mention that our solution was sub-optimal, and that we would be more likely to use a related Enterprise framework technology called **Java Server Pages**. The link to Sun Microsystems in Topic 2 Section 3 actually identifies a Java Server Page (JSP) that handles the download transaction.

Our preliminary solution also attempted to reproduce the look and feel of the existing library interface frames in HTML. We would like to champion that as an illustration of **separable user interfaces** , but the fact is that we actually suggested server-side processing from servlets or JSPs, rather than simply bolting this page on to the existing functionality of the Porterhouse system.

4.6.5 Final words

Summary

You should make sure that you have covered both of the texts for this unit by the final week. This has been an introductory account of topics which are of burgeoning commercial and social importance. If you were to continue the technical side of the story you would be well advised to download your own copy of the Apache Web server and obtain practical experience of the technologies that we have described. If you wished to pursue the managerial and developmental side, you might investigate some of the nationally and internationally recognised standards and perhaps some of the courses offered by our sister institutions.

In the final unit of this course you will get a chance to try out some of these techniques in something like a practical setting.