LEVEL 5

ANALYSIS, DESIGN AND IMPLEMENTATION

Lecturer Guide

Modification History

Version	Date	Revision Description
V1.0	September 2011	For issue

Published by: NCC Education Limited, The Towers, Towers Business Park, Wilmslow Road, Didsbury, Manchester M20 2EZ, UK.

Tel: +44 (0) 161 438 6200 Fax: +44 (0) 161 438 6240 Email: info@nccedu.com
http://www.nccedu.com

CONTENTS

1. Module Overview and Objectives 7

2. Learning Outcomes and Assessment Criteria 7

3. Syllabus 8

4. Related National Occupational Standards 10

5. Resources 10

5.1 Software Requirements 11

6. Pedagogic Approach 11

6.1 Lectures 11

6.2 Tutorials 11

6.3 Laboratory Sessions 11

6.4 Private Study 11

7. Assessment 11

8. Further Reading List 12

Topic 1: Introduction to the Module 13

1.1 Learning Objectives 13

1.2 Pedagogic Approach 13

1.3 Timings 13

1.4 Lecture Notes 14

1.4.1 Guidance on the Use of the Slides 14

1.5 Laboratory Sessions 18

1.6 Private Study 19

1.7 Tutorial Notes 20

Topic 2: Introductory UML modelling with StarUML 21

2.1 Learning Objectives 21

2.2 Pedagogic Approach 21

2.3 Timings 21

2.4 Lecture Notes 22

2.4.1 Guidance on the Use of the Slides 22

2.5 Laboratory Sessions 26

2.6 Private Study 28

2.7 Tutorial Notes 29

Topic 3: Object-Oriented Modelling 31

3.1 Learning Objectives 31

3.2 Pedagogic Approach 31

3.3 Timings 31

3.4 Lecture Notes 32

3.4.1 Guidance on the Use of the Slides 32

3.5 Laboratory Session 36
3.6 Private Study 39
3.7 Tutorial Notes 40
Topic 4: Static Modelling in UML 41
4.1 Learning Objectives 41
4.2 Pedagogic Approach 41
4.3 Timings 41
4.4 Lecture Notes 42
4.4.1 Guidance on the Use of the Slides 42
4.5 Laboratory Sessions 47
4.6 Private Study 49
4.7 Tutorial Notes 50
Topic 5: Dynamic Analysis and Design 51
5.1 Learning Objectives 51
5.2 Pedagogic Approach 51
5.3 Timings 51
5.4 Lecture Notes 52
5.4.1 Guidance on the Use of the Slides 52
5.5 Laboratory Sessions 56
5.6 Private Study 57
5.7 Tutorial Notes 58
Topic 6: OOAD Case Study 59
6.1 Learning Objectives 59
6.2 Pedagogic Approach 59
6.3 Timings 59
6.4 Lecture Notes 60
6.4.1 Guidance on the Use of the Slides 60
6.4.2 The Scenario 60
6.5 Laboratory Sessions 62
6.6 Private Study 76
6.7 Tutorial Notes 77
Topic 7: Design Patterns 1 79
7.1 Learning Objectives 79
7.2 Pedagogic Approach 79
7.3 Timings 79
7.4 Lecture Notes 80
7.4.1 Guidance on the Use of the Slides 80
7.5 Laboratory Sessions 84
7.6 Private Study 88

7.7 Tutorial Notes 89
Topic 8: Design Patterns 2 93
8.1 Learning Objectives 93
8.2 Pedagogic Approach 93
8.3 Timings 93
8.4 Lecture Notes 94
8.4.1 Guidance on the Use of the Slides 94
8.5 Laboratory Sessions 99
8.6 Private Study 102
8.7 Tutorial Notes 103
Topic 9: Elements of Good Design 105
9.1 Learning Objectives 105
9.2 Pedagogic Approach 105
9.3 Timings 105
9.4 Lecture Notes 106
9.4.1 Guidance on the Use of the Slides 106
9.5 Laboratory Sessions 111
9.6 Private Study 116
9.7 Tutorial Notes 117
10.4 Lecture Notes 119
10.7 Tutorial Notes 127
Topic 11: Maintenance and Refactoring 129
11.1 Learning Objectives 129
11.2 Pedagogic Approach 129
11.3 Timings 129
11.4 Lecture Notes 130
11.4.1 Guidance on the Use of the Slides 130
11.5 Laboratory Sessions 135
11.6 Private Study 136
11.7 Tutorial Notes 137
Topic 12: Recap of Module 139
12.1 Learning Objectives 139
12.2 Pedagogic Approach 139
12.3 Timings 139
12.4 Lecture Notes 140
12.4.1 Guidance on the Use of the Slides 140
12.5 Laboratory Sessions 142
12.6 Private Study 144

12.7 Tutorial Notes 145

1. Module Overview and Objectives

The aim of this unit is to develop knowledge, skills and experience in the use of object-oriented techniques for the development of software. The module aims to develop expertise in:

- object-oriented analysis
- object-oriented design
- object-oriented coding
- the testing of systems

2. Learning Outcomes and Assessment Criteria

Learning Outcomes; The Learner will:	**Assessment Criteria;** The Learner can:
1. Understand the seamless transition from OO analysis to OO design.	1.1 Explain the seamless transition from OO analysis to OO design 1.2 Identify and describe OO analysis models 1.3 Indentify and describe OO design models
2. Understand how to convert OO analysis and design models to code	2.1 Explain how to convert OO analysis models to code 2.2 Explain how to convert OO design models to code
3. Understand the quality attributes associated with an OO development	3.1 Explain the developer software quality attributes 3.2 Explain the user software quality attributes
4. Understand the concept of maintenance within an OO development environment	4.1 Describe what is meant by maintenance of software 4.2 Identify and define the different types of software maintenance
5. Be able to produce OO analysis and design models using a case tool	5.1 Use a case tool to produce OO analysis models based on a case study 5.2 Use a case tool to develop OO design models based on a case study
6. Be able to convert OO analysis and design models to code using an appropriate IDE	6.1 Use an IDE to develop code based on an OO analysis model 6.2 Use an IDE to develop code based on an OO design model
7. Be able to refactor an OO program to improve quality	7.1 Refactor code based on standard refactoring techniques.

3. Syllabus

Syllabus			
Topic No	Title	Proportion	Content
1	Introduction to the module	1/12 2 hours of lectures 2 hours of laboratory sessions 1 hour of tutorials	• Introduction to the module • Distinction between analysis and design • The Software Crisis • Recap of key OO concepts ***Learning Outcomes: 1***
2	Introduction to StarUML	1/12 1 hours of lectures 3 hours of laboratory sessions 1 hour of tutorials	• Obtaining and using the module OO Case tool • Turning simple models into code ***Learning Outcomes: 5 & 6***
3	Object-Oriented Modelling	1/12 2 hours of lectures 2 hours of laboratory sessions 1 hour of tutorials	• Discussion of the OO software development process • Use case diagrams • Identifying abstractions • Event Decomposition • Discussion of benefits of OOAD • Discussion of drawbacks of OOAD ***Learning Outcomes: 1 & 5***
4	Static Modelling in UML	1/12 2 hours of lectures 2 hours of laboratory sessions 1 hour of tutorials	• Requirements gathering • Natural Language Analysis • Candidate classes • Class diagrams • Converting class diagrams into code ***Learning Outcomes: 1 & 5***
5	Dynamic Analysis and Design	1/12 2 hours of lectures 2 hours of laboratory sessions 1 hour of tutorials	• Activity diagrams • Sequence diagrams • Converting dynamic models into code ***Learning Outcomes: 1 & 5***

6	OOAD Case Study	1/12 1 hours of lectures 3 hours of laboratory sessions 1 hour of tutorials	• Worked example from problem statement to design ***Learning Outcomes: 1, 3 & 5***
7	Design Patterns 1	1/12 2 hours of lectures 2 hours of laboratory sessions 1 hour of tutorials	• Introduction to design patterns • Factory • Abstract Factory ***Learning Outcomes: 2, 3 & 4***
8	Design Patterns 2	1/12 2 hours of lectures 2 hours of laboratory sessions 1 hour of tutorials	• Model-View-Controller • Flyweight • Strategy • Facade ***Learning Outcomes: 2, 3 & 4***
9	Elements of Good Design	1/12 2 hours of lectures 2 hours of laboratory sessions 1 hour of tutorials	• Software quality attributes • Software component design • Coupling • Cohesion • The Observer design pattern ***Learning Outcomes: 3 & 5***
10	Redesign and Implementation	1/12 2 hours of lectures 2 hours of laboratory sessions 1 hour of tutorials	• Redesign of case study • Incorporation of design patterns • Implementation of elements of previous design case study into code ***Learning Outcomes: 2 & 6***
11	Maintenance and Refactoring	1/12 2 hours of lectures 2 hours of laboratory sessions 1 hour of tutorials	• Impact of change • Refactoring • Refactoring case study ***Learning Outcomes: 4 & 7***

12	Recap	1/12 2 hours of lectures 2 hours of laboratory sessions 1 hour of tutorials	• Recap of module ***Learning Outcomes: All***

4. Related National Occupational Standards

The UK National Occupational Standards describe the skills that professionals are expected to demonstrate in their jobs in order to carry them out effectively. They are developed by employers and this information can be helpful in explaining the practical skills that students have covered in this module.

Related National Occupational Standards (NOS)
Sector Subject Area: 6.1 ICT Professionals **Related NOS:** 4.3.P.1 – Manage, under supervision, information to direct human needs analysis assignments; 4.3.P.2 – Produce, implement and maintain quality human needs analysis activities; 4.3.P.3 – Provide human needs analysis findings to others; 4.4.P.1 – Prepare, under supervision, for a systems analysis assignment; 4.4.P.2 – Carry out, as required, systems analysis activities; 4.4.P.3 – Monitor the effectiveness of systems analysis activities and their deliverables; 4.4.S.1 – Design, implement and maintain systems analysis activities; 4.7.P.1 – Prepare, under supervision, for system/solution/service design activities; 4.7.P.2 – Assist with the design of system/solution/service design; 4.7.P.3 – Monitor the progress of system/solution/service design activities; 5.1.S.2 - Initiate systems development activities; 5.3.S.3 - Manage systems development activities; 5.2.P.2 - Perform software development activities; 5.3.P.2 - Contribute to the communication of the results of IT/Technology solution testing; 5.3.S.1 - Implement the infrastructure for testing activities; 5.3.S.2 - Manage testing activities; 5.3.S.3 - Monitor and control testing activities.

5. Resources

Lecturer Guide: This guide contains notes for lecturers on the organisation of each topic, and suggested use of the resources. It also contains all of the suggested exercises and model answers.

PowerPoint Slides: These are presented for each topic for use in the lectures. They contain many examples which can be used to explain the key concepts. Handout versions of the slides are also available; it is recommended that these are distributed to students for revision purposes as it is important that students learn to take their own notes during lectures.

Student Guide: This contains the topic overviews and all of the suggested exercises. Each student will need access to this and should bring it to every taught session during the module.

Lecturer Code: Some examples of suitable code which students may produce are given in the Lecturer Guide. This supplementary document makes this available in electronic format to avoid the need to re-type all of it. It is available from the NCC Education *Campus* (http://campus.nccedu.com)

5.1 Software Requirements

This module also makes use of StarUML. Students will need to have access to this during laboratory, tutorial and private study time. This is available from:

http://staruml.sourceforge.net/en/download.php.

6. Pedagogic Approach

Suggested Learning Hours					
Lectures:	**Tutorial:**	**Seminar:**	**Laboratory:**	**Private Study:**	**Total:**
22	12	-	26	90	150

The teacher-led time for this module is comprised of lectures, laboratory sessions and tutorials. The breakdown of the hours is also given at the start of each topic.

6.1 Lectures

Lectures are designed to start each topic and PowerPoint slides are presented for use during these sessions. Students should also be encouraged to be active during this time and to discuss and/or practice the concepts covered. Lecturers should encourage active participation wherever possible.

6.2 Tutorials

These are designed to deal with the questions arising from the lectures and private study sessions.

6.3 Laboratory Sessions

During these sessions, students are required to work through practical tutorials and various exercises involving group work, investigation and independent learning. The details of these are provided in this guide and also in the Student Guide.

6.4 Private Study

In addition to the taught portion of the module, students will also be expected to undertake private study. Exercises are provided in the Student Guide for students to complete during this time. Teachers will need to set deadlines for the completion of this work. These should ideally be before the tutorial session for each topic, when private study work is usually reviewed.

7. Assessment

This module will be assessed by means of an assignment worth 25% of the total mark and an examination worth 75% of the total mark. These assessments will be based on the assessment

criteria given above and students will be expected to demonstrate that they have met the module's learning outcomes. Samples assessments are available through the NCC Education Campus (http://campus.nccedu.com) for your reference.

Assignments for this module will include topics covered up to and including Topic 7. Questions for the examination will be drawn from the complete syllabus. Please refer to the Academic Handbook for the programme for further details.

8. Further Reading List

A selection of sources of further reading around the content of this module must be available in your Accredited Partner Centre's library. The following list provides suggestions of some suitable sources:

Bevis, T. (2010). *Java Design Pattern Essentials*. Ability First Ltd.
ISBN-10: 0956575803
ISBN-13: 978-0956575807

Gamma, E., Helm, R., Johnson, R. and Vlissides, K. (1994). *Design Patterns.* Addison Wesley.
ISBN-10: 0201633612
ISBN-13: 978-0201633610

Jackson, R., Burd, S. and Satzinger, J. (2004). *Object-oriented Analysis and Design with the Unified Process.* Course Technology Inc.
ISBN-10: 9780619216436
ISBN-13: 978-0619216436

Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall.
ISBN-10: 9780131489066
ISBN-13: 978-0131489066

McGregor, J. and Sykes, D. (2001). *A Practical Guide to Testing Object-Oriented Software.* Addison Wesley.
ISBN-10: 9780201325645
ISBN-13: 978-0201325645

Topic 1: Introduction to the Module

1.1 Learning Objectives

This topic provides an overview of the analysis and design. On completion of the topic, students will be able to:

- Define the need for software development techniques;
- Discuss the need for analysis in developing complex software;
- Discuss the need for design in developing complex software.

1.2 Pedagogic Approach

Information will be transmitted to the students during the lectures. They will then practise the skills during the laboratory sessions. Tutorials are then used to consolidate students' understanding of the concepts covered and deal with any questions.

1.3 Timings

Lectures:	2 hours
Laboratory Sessions:	2 hours
Private Study:	7.5 hours
Tutorials:	1 hour

1.4 Lecture Notes

The following is an outline of the material to be covered during the lecture time. Please also refer to the slides.

The structure of this topic is as follows:

- Overview of Analysis
- Overview of Design
- The Software Crisis
- Recap of Key OO Concepts

1.4.1 Guidance on the Use of the Slides

Slide 4: *Introduction to the Module* - Many modules that focus on OOAD are interested mainly in the modelling aspects without a focus on implementation. The strategy taken in this module is different in that implementation (turning designs into actual working program code) is a fundamental part of its structure. However, while we will be using Java and the Eclipse IDE to do this, the assumption is that students are already familiar with object-orientation in programming and this IDE in particular.

Slide 5: Software engineering processes are usually very documentation heavy, it is easy to lose sight of why we go through the process at all when the focus is on getting diagrams right. These modelling processes are all about one single thing – effective communication between everyone involved in the process.

Slide 6: A project development team is likely to have several members in it, but it is communication between the development team and the users that is likely to be the most difficult because of the different skill-sets involved. As part of this process, a developer/analyst/designer must talk to many individuals who are not experts in computing, so their mental models of what computers can do will often be very different. The level of specific detail required to create a computer algorithm is considerable, and ambiguity is not something that can be incorporated. Formal modelling allows developers to understand user requirements and provides a vehicle for the developers to demonstrate this understanding to the users. The formal model can also be updated to rectify any misunderstandings between the users and developers.

Slide 7: When we do not understand a system **before** we begin, modelling lets us find out the costly task of implementing it. When dealing with large and complex systems with many interrelating parts, it is very easy to miss edge cases and workarounds; having a modelling process means that we will encounter these early as we try to explain our understanding of a system. If we find out that we do not have a way to deal with a particular situation, we can go back to the users and ask for further clarification – it is an inherently iterative process and we go through it many times in order to arrive at a full and complete understanding of how the problem domain functions.

Slide 8: Communication errors in one way or another are responsible for many of the largest and most costly failures in software development. This slide outlines two examples, but students will be encouraged to investigate others during the seminar.

Slide 9: Analysis is the first step in our model, and analysis is all about understanding what it is we are going to be building as a system. It is often the case that users may not know sufficient detail about what it is they want, and as such the problem statement initially received will only be a vague guide as to what is supposed to be done. Users too will often emphasise what they **want**, but as a professional, it is often necessary for you to explain and convince them instead of what they **need** on the basis of your expert knowledge and experience.

Slide 10: The problem statement is often vague, ambiguous, and contradictory. It is only the first step in the process, and is only very rarely sufficiently detailed or clear enough to be used as a basis for analysis. However, it does give the sense of context needed in order to investigate properly through interviews, examination of what is currently being done, and by asking questions designed to remove ambiguity and contradictions.

Slide 11: Analysis is all about building an understanding of what is currently being done and how the system we are developing should work. However, different practitioners have different views on how analysis should fit into the OOAD process – some stress that analysis is the same as design, except the **emphasis** changes depending on which phase you are in. Some stress that they should be completely separate stages from each other and they should not overlap. In this module, we take a more pragmatic approach, whereby the conceptual purity of the model is sacrificed for something that meets our key goal of communication.

Slide 12: *The Problem Domain* - This defines the subject specialist information around the system – if you are developing an air traffic control system, you need to understand how all the parts fit together and how they communicate. It is more likely that there will be a subset of the problem domain that you must investigate – perhaps you are only focusing on the scheduling of runways. The **problem domain** is the set of information you must master in order to understand the problem. The **project scope,** on the other hand, is limited to those bits of the domain that your system will eventually be incorporated within – scope creep is common, but should be avoided.

Slide 13: The design phase follows the analysis, and this is where we start to build a system to address the problem domain and our project scope. Analysis focuses on understanding, but design must incorporate functional, environmental and social considerations into the model. In analysis we can ignore things like performance, but in design we must make sure we address them.

Slide 14: This slide lists some of the considerations of design – they are not the only considerations and are presented only as an example of things that must be considered. This is the stage when students will usually raise the issue of 'why is this analysis and why is that design?' when it comes to specific things to be included in phases. The reality is that the two phases actually overlap, and while there are pure OOAD models that stress the deliverables for each, it is actually a more useful technique to let the two overlap and switch between them as needed.

Slide 15: In this slide we talk about the overlap – classes are something we often outline in analysis as we try to come to grips with the problem, but these are very much an issue of design. We use class diagrams though, because they are an effective communication tool. We move between the phases – in the past history of software development, things like the Waterfall Method for engineering were considered to be stately progressions from one stage to the other, but the realities of development resulted in redesigns and adaptations, such as the Iterative Waterfall Model, which addressed the need to move backwards and forwards between phases.

Slide 16: It is difficult to enthuse students about formal analysis and design in a computing course, because the case studies must be small enough for students to analyse, but small studies do not really show the need for the tools. Larger studies that did would overwhelm students with their complexity. Students often believe that the techniques they use for building their personal projects will work for large real-world projects, but they simply do not scale up.

Slide 17: The larger a project, the more people will be involved. The more people that are involved, the harder the communication becomes. Formal OOAD techniques allow us to manage both the complexity of a problem, and the communication between different developers and between the development team and the users.

Slide 18: Implementation requires us to decide how to honour what the design model has identified as requirements. We need to decide on algorithms, design patterns, coding structures and performance tradeoffs. The design will often tell us what is required, but implementation is where we decide how to fulfil those requirements.

Slide 19: As mentioned above, formal analysis and design is a hard sell – but students should be aware that these kinds of techniques were born of necessity. The software crisis is a concept that was tremendously influential for decades, but has fallen out of use as the discipline of software engineering has matured. In the next few slides we will explain, for context, why we actually go through these kinds of processes.

Slide 20: There were many things that characterised the software crisis, but the list provided on this slide is a subset of the most important ones. As software developers realised that their ad-hoc approaches did not scale up, a field of software engineering was born so that best practise and techniques for managing these issues could be shared, codified, refined and disseminated.

Slide 21: These techniques exist now because they were needed – the problems of the software crisis still exist, but the symptoms can be managed through the use of formal methods such as the OOAD model that will be explored in the module.

Slide 22: Strictly speaking, an analysis of a problem domain should not mandate an implementation strategy, but that is exactly what OOAD does. This is of a necessity; object-orientation as a system requires a very particular approach to problems, and you cannot retrofit it in if you were thinking in terms of flow of data rather than communication of objects. The analysis we do then will assume that the end result will be an object-oriented program, which frees us up to assume that things like class definitions and such will be relevant.

Slide 23: We will discuss OO concepts and how they relate to OOAD as we go through the module, but if students are not familiar and comfortable with these terms at the beginning of the module, they should make a special effort to learn and understand them because they will be key to understanding why we do certain things in terms of our later analysis and design.

Slide 24: Here we recap on the concept of a class and an object. The nature of large programs is that they will tend to have hundreds of classes that interact in complex ways. We will talk about design patterns that let us manage this complexity later in the module, but for now the simple scale of object communication should be highlighted as an important issue in building OO software.

Slide 25: *Inheritance* – This is primarily a technique for reusability and maintenance, but it also serves as the enabler for polymorphism. We show here the two key ways in which inheritance is applied, through specialisation or through extension.

Slide 26: *Encapsulation* – This is a security and maintenance technique that is important in ensuring that complex systems can be changed and the impact of that change managed. It is going to be important when we talk about maintainability in a later lecture of the topic.

Slide 27: *Polymorphism* – This is perhaps the most powerful and useful technique in object-orientation – treating a specialised object as an instance of its more general case. As a necessity, we will be using polymorphism heavily in example material through the module, and even those students who are familiar with the term may benefit from spending some time refreshing their memory as to how it works.

Slide 28 Conclusion

Slide 29: Terminology

1.5 Laboratory Sessions

The time allocation for the laboratory sessions for this topic is 2 hours.

Lecturers' Notes:

Students have copies of the seminar activities in the Student Guide. Answers are not given in their guide.

It is recommended that you give students an hour to research the set of the topics in activity one, using books, journal articles and the internet as necessary. An hour is then recommended for the group discussions, including time for groups to report back any significant insights to the rest of the class if you feel it is appropriate.

Activity 1:

Investigate the following concepts as discussed in the lecture:

1. The software crisis
2. The reasons why software products fail
3. Problem statements
4. The role of analysis versus design

Activity 2: Group Discussion

Work in small groups as directed by your tutor.

Making use of the insights and material you uncovered during Activity 1, discuss the need for formal analysis and design, and how it contributes to resolving some of the issues associated with the software crisis.

1.6 Private Study

The time allocation for private study in this topic is expected to be 7.5 hours.

Lecturers' Notes:

Students have copies of the private study exercises in the Student Guide. Answers are not provided in their guide.

Exercise 1:

You should begin keeping a journal of important notes and concepts from the lecture. Supplement the material from each lecture with the results from your own studies. Each week, your guide will contain some topics for you to research and summarise - these will be your own personal revision and research guides, so make sure you write them in such a way that you will understand the content. Include examples and interesting slides and web-pages that you encounter during your research.

For this topic, you should research the following incidents, outlining the reasons how failed IT or failed communication (or both) contributed to the way the situation unfolded.

- The Therac-25 Incident
- The Mars Orbiter Incident
- USS Vincennes and the A320
- Black Monday

Exercise 2:

Prepare a short, five minute presentation on the results of your research for Exercise 1 above. If you have found out anything particularly interesting, you should focus on that as a priority.

Exercise 3:

Review the lecture material and ensure that you are comfortable with everything discussed thus far.

1.7 Tutorial Notes

The time allowance for tutorials in this topic is 1 hour.

Lecturers' Notes:

Students have copies of the tutorial activities in the Student Guide. Answers are not provided in their guide.

This tutorial is designed to give students a chance to present the more interesting findings from their private study, partially as a way to share information with others in the class, but also as a way to ensure that they are indeed keeping a journal and researching the topics provided. You should encourage students to bring along their journals to these sessions and make notes on any points of interest that are raised by their classmates.

Exercise 1: Reporting Back to the Class

As a result of the research you did during your private study time, you should have a short five minute presentation ready to give to the rest of the class. There is no need for this to be especially formal - you are simply reporting on anything interesting that you found during your research, or pointing out especially useful resources on the topic. Bring your journal along to the class so that you can make a note of anything especially useful that your classmates mention. This is a knowledge dissemination exercise; you are not being formally assessed on the style or content of the presentation.

Topic 2: Introductory UML modelling with StarUML

2.1 Learning Objectives

This topic provides an overview of using StarUML to build UML diagrams. On completion of the topic, students will be able to:

- Obtain and use the StarUML diagramming package;
- Develop class diagrams from fixed, simple scenarios.

2.2 Pedagogic Approach

Information will be transmitted to the students during the lectures. They will then practise the skills during the laboratory sessions. Tutorials are then used to consolidate students' understanding of the concepts covered and deal with any questions.

2.3 Timings

Lectures:	1 hour
Laboratory Sessions:	3 hours
Private Study:	7.5 hours
Tutorial:	1 hour

2.4 Lecture Notes

The following is an outline of the material to be covered during the lecture time. Please also refer to the slides.

The structure of this topic is as follows:

- A brief introduction to UML
- Class diagrams
- Attributes in StarUML
- Behaviours in StarUML
- Relationships in StarUML

2.4.1 Guidance on the Use of the Slides

Slide 3: All diagramming notations are, in their simplest form, about constructing a standardised vocabulary between stakeholders in a project. The actual notation used is often irrelevant, although there are benefits and drawbacks to each of the various standard methodologies – the important thing is that they succinctly transmit intentions and design to everyone who must understand the way the system is to fit together. Ad hoc diagrams do not fit this role, because they require explanation and interpretation in a way that a fixed, common diagramming vocabulary like UML does not.

Slide 4: UML is the diagramming language we use in this module. There are over a dozen different diagrams, spread over two diagramming families. We won't look at all of these during this module, because some of them have very niche applicability and there is only so much time available here. However, we will look at diagrams from both families, and show how they fit into the software development process.

Slide 5: UML diagrams are syntactically simple and drawing them by hand is entirely feasible for simple, self-contained systems. However, formal design tools can greatly ease the scaling issues that go along with building large and complex software artefacts, and these tools often come with extras such as code generation and templates that can further ease the burden on designers. UML diagrams however are not supposed to be diagrams that are drawn at the start of the project and then never updated; they are intended as living documents that should be adjusted, altered and expanded as the needs of the project evolve. They should influence the software development project at all stages.

Slide 6: There are dozens of UML packages available. The one we are using, StarUML, is open source and free – development has been patchy over the past few years, but it is stable and available to anyone who wants to use it. During this lecture we will explore how StarUML works by building a simple diagram. It is very worthwhile for you to develop this diagram in front of the students so that they can see how it is supposed to be done. Instructions are given in the slides, but as with all instructions they are no substitute for direct experience or exposure.

Slides 7-8: StarUML has a design that is very similar to that of a programming IDE; it even uses much of the same terminology, such as properties and projects. This fits the design of UML, which is very much a software project that cannot be executed on a

system. Students should be familiar with this from the Designing and Developing Object-Oriented Computer Programs module at Level 4.

The first thing UML does is ask for an 'approach'. Many tools and methodologies that use UML have a particular way they approach the building of the diagrams; some will focus first on the structural aspects, others will begin with an appreciation of dynamic interaction. These are codified within StarUML as 'approaches'. The one we will be using for all our projects is the 'default approach'. This sets up with a set of standard diagrams, the only one of which we will be looking at in this module is the Design Model.

Slide 9: We begin by looking at the class diagram, which documents the structural infrastructure of the program. While it is common to use this to diagram existing projects, in the context of object-oriented analysis and design (OOAD) we must progress from design to implementation. We draw the class diagram before we write any code at all.

Slide 10: This slide introduces a (very) simple scenario that will be used as the basis of our class diagram. In later topics, we will talk about how we can analyse more complicated scenarios to generate class diagrams, but for now a very simple overview is introduced because the focus is on how to use the tool. A useful exercise at this stage is to get students call out what is needed to represent the system, and then you can spend some time slotting their ideas into appropriate classes. This has already been done for the rest of this lecture, but allowing students to consider the problem first will help demonstrate some of the complexities that come with analysing scenarios.

Slide 11: In reality, we are hardly ever given a scenario that is so specific when it comes time to build a system – our problem statements are usually full of ambiguities, lacking in context, missing necessary detail and sometimes even entirely misleading. This scenario however gives us what we need right away – we need three classes to represent the system.

Slide 12: Analysis and design is as much an art as it is a formal process, and as such different people will analyse scenarios in different ways depending on their own preferences and experiences. As such, it is common for different interpretations of a scenario to occur even when the system is very well defined. In this case, separate classes are used to represent each of the different parts of the system because that ensures an architecture that incorporates classes of **high cohesion.** This point will be addressed in more detail in later lectures.

Slide 13: The first step of our design is to draw the boxes that contain the three classes. The graphic on this slide shows what the diagram should look like after this has been done.

Slide 14: Classes in a class diagram have three key sets of data – the classes and their relationships, the attributes of each class, and the behaviours in each class. In our diagram, we have to represent all of these to get a full overview. We don't need it to be fully complete at this point (as a project goes on, new attributes and behaviours will be added on a regular basis), but we should have the structure in place that is required to model the interactions between the classes we have and the minimum set of attributes they will contain.

Slide 15: Classes, attributes, methods and practically everything else in StarUML will have properties. These properties are visible in the bottom right of the screen. Much of

the work of setting up a StarUML diagram is conducted through these property dialogues.

Slide 16: This slide shows the 'collection editor', which is an important part of working with StarUML. Whenever an element of the diagram might have multiple things associated with it, the collection editor is used to represent it. On occasion, these windows might be nested – a class may have many methods (represented via the collection editor) and each of these may have many parameters (similarly represented via the collection editor).

Slide 17: Once an attribute has been added, we can make some simple changes to it by double clicking on the name of the attribute in the class. However, most of the changes we might need to make are handled via the property window.

Slide 18: As a general rule, all attributes should be set as private throughout this module. Students may already have an understanding of what is meant by **encapsulation** or **data hiding**, but if not they should simply be aware that this is a requirement of the module. The explanations of why will be given later when needed.

Slide 19: This slide shows the StarUML interface for modifying the properties of an attribute – **visibility** and **type** are the only properties that matter at the moment. The rest will be discussed as needed in the context of the module.

Slide 20: This diagram shows what the class should look like if the visibility and the type information has been set correctly. We do this for every attribute we have identified in this lecture; this is only a small subset of the attributes that a real world system would have to represent.

Slide 21: There are some complexities here in terms of representing the relationship between attributes of a class, especially Repair, since a car will have only one owner (as far as the garage is concerned), but may need several repairs. This is handled via the **multiplicity** property on the attribute.

Slide 22: This shows how the class diagram should look at the end of this process. Only a small set of attributes have been specified, but each has its visibility set to private and its type information presented.

Slide 23: Next, we look to populate some behaviours. We will only look at accessor methods since these will be sufficient to demonstrate how these can be represented in StarUML. Parameters are added through the collections editor; the only slight complication is that return types are handled as a special case of a parameter. To create a return type, we add a parameter, remove its name, set its type, and change its **direction** to return.

Slides 24-25: These slides show the IDE when setting up operations and parameters. All work is again done through the property manager.

Slide 26: The model explorer permits for easy access to deeply nested elements of the UML diagram. We don't need to go through the nested collection editors to get to a parameter for example, we can simply click on its entry in the model explorer. We need to go through each of our classes and represent the accessor methods for each attribute.

Slide 27: As students become more familiar with StarUML, they will start to develop shortcuts for more efficient manipulation of their diagrams. Some such shortcuts are represented here.

Slide 28: This shows the UML diagram with the behaviours populated. There are still no formal relationships set between the classes; that is still to come.

Slide 29: The relationship between the classes in our diagram is handled via a DirectedAssociation. We will talk about the different kinds of relationship between UML classes in a later lecture. For now, this should just be done to show how a relationship is set up. Multiplicity in a relationship (such as is handled in Repair) is also controlled via properties on the relationship.

Slide 30: This shows the final class diagram for our scenario.

Slide 31: This lecture has been mostly about the technical aspects of building these class diagrams in StarUML, rather than on how or why we do the things we do – that topic is coming later in the module. Practise is the most important thing students should do in the time between this lecture and the next topic, because familiarity with how StarUML works will be assumed from this point onwards. As new technical elements are introduced, they'll be related back where necessary to the tool, but on the whole it will be expected students are comfortable with the application.

Slide 32: Diagramming tools like StarUML usually come bundled with the functionality needed to generate boilerplate code templates. We won't be using this functionality through the module because it is far more important that we see directly how we build code from diagrams. Students should be encouraged to think of this kind of facility as an extra for when they have mastered the techniques, rather than a core part of what these tools are used for.

Slide 33: Conclusion

Slide 34: Terminology

2.5 Laboratory Sessions

The time allocation for the laboratory sessions for this topic is 3 hours.

Lecturers' Notes:

Students have copies of these activities in the Student Guide. Answers are not given in their guide.

For Activity 1, you can have the students work in groups to generate lists of methods, attributes and classes. You should run a short feedback session after this activity to ensure that all of the students have correctly understood methods, attributes and classes in the context of for the scenarios.

For Activity 2, students should work individually to build the diagrams. You may want students to refer to the lecture slides while they do this for assistance on how to use the package. You should monitor closely and assist with any issues. Students can then compare their diagrams with those of other students from their original groups.

Activity 1:

Consider the following scenarios. Identify potential classes, methods and attributes:

1. A library needs to keep track of its books, and which patrons have each of the books.
2. A university needs to keep track of its courses, and its students, and which students are registered to which course (and which courses have which students on them).
3. A shop needs to keep track of the stock it has on the shelves, and also the stock it has in the backroom.

Suggested Answer:

It is not especially important what students come up with at this point, it is only important they go through the process of arriving at answers. When we discuss formally how to generate candidate classes and assess them for their correctness, the starting point will be more important. However, you should ensure that there are at least three classes for each of these scenarios, and that each has at least one attribute and its associated accessor methods.

While there is no suggested answer, the following would serve as useful examples of where the exercise should touch upon. Each of these would have a getX and setX method to go with them.

Library

Patron: *Name, address, postcode, books currently held, status of account, accumulated fines*

Book: *Name, author, ISBN, Publisher, Genre*

Library: *A list of all books and a list of all patrons*

University

Course: *Name, Code, Subject Group, Assessment information, enrolment*

Student: *Name, Address, Course, Date of Birth*

University: *A list of all courses, and a list of all students*

Course Registration: *Name of student, name of course, date of registration, grade*

Shop

Stock: *Name, Type, Price*

Shelf: *A list of stock items*

Backroom: *A list of stock items*

Activity 2:

Start up StarUML. Use this to represent the class diagrams that derive from your analysis in Activity 1.

Suggested Answer:

Again, no suggested answer – this is highly dependent on what students come up with for their potential candidate classes. The purpose of this exercise is to give students experience with constructing StarUML class diagrams.

2.6 Private Study

The time allocation for private study in this topic is expected to be 7.5 hours.

Lecturers' Notes:

Students have copies of the private study exercises in the Student Guide. Answers are not provided in their guide.

Exercise 1:

If you did not complete the diagrams during the laboratory sessions, finish these during your private study time. Remember that the tool you need is open source and can be freely downloaded.

Exercise 2:

As part of your ongoing journal exercise, you should research the following topics:

- StarUML user tutorials
- UML Class diagrams
- Other UML diagramming tools (such as Rational Rose)

Exercise 3:

Consider the following scenario. Identify candidate classes, methods and attributes and then draw the class diagram in StarUML.

A shipping firm has a fleet of ships of which they need to keep track. These ships are registered with a licence code, and classified according to their speed and storage capacity in tons. The firm has a number of contracts with businesses, and ships are assigned to these contracts on a day by day basis. Each of these contracts will have a duration and an amount of freight (in tons) that need to be shipped. A single ship might not be able to make the journey, and so multiple ships may be assigned where appropriate.

Exercise 4:

Prepare a short, five minute presentation on the results of your research for Exercise 2 above. If you have found out anything particularly interesting, you should focus on that as a priority.

Exercise 5:

Review the lecture material and ensure that you are comfortable with everything discussed thus far

2.7 Tutorial Notes

The time allowance for tutorials in this topic is 1 hour.

Lecturers' Notes:

Students have copies of the tutorial activities in the Student Guide. Answers are not provided in their guide.

This tutorial is designed to give students a chance to present the more interesting of their findings from the previous exercise, partially as a way to share information with others in the class but also as a way to ensure that they are indeed keeping a journal and researching the topics provided. You should encourage students to bring along their journals to these sessions and make notes on any points of interest that are raised by their classmates.

Exercise 1: Discussion of Private Study Exercise

Discuss as a group your solution to Exercise 3 in the private study.

Suggested Answer:

As with the laboratory exercises, a suggested answer is not provided here because of the many ways in which it is possible to analyse the scenario. During this part of the tutorial you should ask students to present their own work and discuss it as class. The following serves as example candidate classes for consideration:

Firm: *List of ships, name of firm, list of contracts*

Ship: *Licence code, capacity, speed*

Contract: *Amount of freight, duration of journey, ships assigned*

Exercise 2: Reporting Back to the Class

As a result of the research you did during your private study time, you should have a short five minute presentation ready to give to the rest of the class. There is no need for this to be especially formal - you are simply reporting on anything interesting that you found during your research, or pointing out especially useful resources on the topic. Bring your journal along to the class so that you can make a note of anything especially useful that your classmates mention. This is a knowledge dissemination exercise; you are not being formally assessed on the style or content of the presentation.

Topic 3: Object-Oriented Modelling

3.1 Learning Objectives

This topic provides an overview of object-oriented modelling. On completion of the topic, students will be able to:

- Define the benefits and drawbacks of OOAD;
- Make use of event decomposition;
- Build use case models.

3.2 Pedagogic Approach

Information will be transmitted to the students during the lectures. They will then practise the skills during the laboratory sessions. Tutorials are then used to consolidate students' understanding of the concepts covered and deal with any questions.

3.3 Timings

Lectures:	2 hours
Laboratory Sessions:	2 hours
Private Study:	7.5 hours
Tutorials:	1 hour

3.4 Lecture Notes

The following is an outline of the material to be covered during the lecture time. Please also refer to the slides.

The structure of this topic is as follows:

- An overview of structured analysis and design
- A simple OOAD process
- Use case diagrams
- Event decomposition

3.4.1 Guidance on the Use of the Slides

Slide 3: In this lecture, we introduce the justifications for OOAD as an analysis and design approach. We also show the use case diagram, which is the main tool students will have at this stage for representing the requirements of the systems they design. OOAD is a powerful technique, and we will discuss some of the benefits in the course of the lecture. It is also a technique with drawbacks, and we will discuss those too.

Slide 4: The most popular modelling language prior to UML was SSADM, and this was a structured approach to building system models. The basic goals of the notation were the same as with UML – to improve the quality of software and provide a consistent diagrammatic vocabulary for the stakeholders in a project. However, SSADM was developed at a time when the complexity of computer systems were many orders of magnitude less complex than they are now. The nature of the diagrams was such that the notation struggled to cope with real world requirements. Also, as object-orientation began its ascendance in terms of being the industry standard, the SSADM notation provided a method for designing software that was at odds with the requirements of the industry.

Slide 5: The focus of SSADM was on the flow of data through a system – it demonstrated where data was modified, where it was passed, where it was stored, and where it was entered. This was a useful notation, as understanding what is happening with data is the first step in understanding a system. Unfortunately, it was a very verbose format and the data flow diagrams for systems would soon grow beyond the ability of anyone to understand.

Slide 6: This shows an example of a simple level one data flow diagram. The ovals show **external entities**, the boxes show distinct algorithmic processes, and the boxes labelled as D1 and D2 show data store locations. Arrows show and describe the flow of data through the system.

Slide 7: The design of a DFD was that it would in itself spawn off other diagrams that view each of the algorithmic processes in more depth. These would be known as level two data flow diagrams, and each of the processes in Slide 6 would get a diagram of its own, and each of the boxes in the level two diagram would spawn off level three diagrams. These allowed for the process to be viewed at different levels of abstraction, but lacked maintainability – changes had to propagate all the way through each of the levels.

Slide 8: The focus on data also meant that programs had to come with defined data dictionaries – huge documents that contained descriptions of all the data in the system, where it was used, how it was validated, what kind of format it was in, and so on. The data dictionary had to be kept up to date, or it would become misleading.

Slide 9: As the complexity of software increases, so too does the complexity of the diagrams that must be drawn to describe it (this is true for UML and SSADM, but UML scales better). OOAD permits for a simpler decomposition of systems into self-contained chunks, and these chunks can be developed and understood in isolation in a way that was not easily possible with SSADM. UML incorporates more diagrams, but these diagrams are smaller and more easily understood.

Slide 10: Object-orientation, while a substantial shift in terms of design and implementation, is still only a evolution of procedural programming. Monolithic programs became more modular with the introduction of functions, and objects in turn serve as a way to modularise collections of functions and data. It's all about breaking up programs into properly manageable chunks. By separating objects from their context and ensuring they were self-contained, it became increasingly easy to update, reuse and maintain programs.

Slide 11: There are numerous benefits that accrue from using object-orientation in both analysis & design, and implementation. The whole structure of object-orientation is based around the idea that systems can be easily decomposed into discrete interacting units. If the units are well designed a system will be more maintainable and the individual parts can be reused in related contexts. Data flow diagrams too are an unnatural way of representing systems; they are a representation of systems for the ease of development, rather than as a way to precisely model what is happening or what should happen. Building a data flow diagram often required translation or interpretation of elements to fit the framework.

Slide 12: OOAD comes with drawbacks too. Complexity and size are still a major consideration in programs, and while UML scales better as a diagramming notation than SSADM, it is still large and unwieldy for big projects. This is further complicated by the fact it is very easy to badly design classes, and this will have a knock-on effect that is substantial as other classes have to work around flaws. A good class has high cohesion and low coupling, but these are two ends of a spectrum and as you move further towards one you move further from the other. Finally, while object-orientation is a more natural metaphor than data flow modelling, it is still the case that it does not precisely map on to how people think and representing systems in this way also requires translations and interpretation of elements.

Slide 13: In order to make use of OOAD, we need a process. There are many ways in which we can approach the design of a system, and none of these are especially better or worse than others. What is outlined here is a very simple initial process students can follow – they won't know all the diagrams at the moment, but they will be able to expand their analysis as the notations and concepts are discussed. First, and the focus of this lecture, is that we must identify the needs of users. Second, once we have identified the needs of our users we must work to detail the steps we must go through to fulfil each of those needs. Once we know what it is we must do, we can approach the structural aspects of the system by outlining classes and how these fit together in a component diagram. Most important though is the fifth step – this is an interactive process, we need to keep doing it to make sure that new

requirements are adopted, missing requirements are implemented, and that our design is as solid as it can be.

Slide 14: Good design is user-centric. It should involve the users as often as possible to ensure that problems are caught early and requirements are properly modelled. We'll discuss some dangers of this in a later lecture. At the core of our iteration though is the idea that what we are doing is **incremental** – we don't try to solve the whole problem at once, because that is far too difficult. Instead, we solve parts of the problem and then refine those solutions in each iteration. We focus on broad behaviour before we begin to specialise it.

Slide 15: In order to incrementally develop, we need to understand how to decompose the whole into constituent parts. An important skill is that of **abstraction** – we must be able to think of things at the appropriate level of detail, and sometimes that detail is as high level as representing a complex algorithm with a single descriptive sentence. Our incremental analysis and design will gradually expand that single sentence into the diagramming models we require for people to implement our designs.

Slide 16: The use case diagram is the first UML diagram we are going to properly discuss. In the last lecture we talked about how to build a class diagram in StarUML, but the explanation of the role of a class diagram is still to come. A use case diagram is used to represent the interactions users and subsystems will have with our model; we represent these users and subsystems using stick figures known as **actors.**

Slide 17: Use case diagrams focus on the relationship of actors to systems. They do not cover interactions between actors. The various actions that actors can have with our system are handled via ovals which contain some descriptive text representing what the action is. Applicability of actions to actor is represented by linking an actor to an action via a line.

Slide 18: This shows an extremely simple use case diagram. The actor here is the person, and the system is that of a light switch. A light switch offers two possible interactions – switch on, and switch off.

Slide 19: This is a system which services two actors – one of these is a person surfing the web, the second is a file server (a subsystem or external server) which provides the pages needed to the web server.

Slide 20: Use case diagrams are supposed to be simple and show only the broad strokes of interaction, however they do come with some specialised notation for later in the design process. We often want to represent that an action is made up of several subactions, and we do this by using a line which has the **<<uses>>** notation attached.

Slide 21: This is a use case example of a camera. The 'take picture' action is made up of three separate actions as are detailed on the breakout diagram on the right.

Slide 22: Use case diagrams do not impose ordering; we cannot tell from a use case diagram in what order actions will be performed. This is represented in a different diagram that we will discuss later in the module. One way to think about these kind of diagrams is as a description of what functionality your user interface will have to support. This is slightly inaccurate because some actions won't actually be triggered directly by users, but it does help outline why this kind of diagram is important.

Slide 23: A third kind of syntax provided by a use case diagram is that of inheritance, and that is handled via the **extends** syntax. In a use case diagram, this also implies the use of polymorphism.

Slide 24: This shows an example of the Flash action (as represented on Slide 21) actually being a generalisation of three specialised actions. When the flash action is performed, we are actually performing one of the three specialisations.

Slide 25: Identifying use cases is harder than drawing them, so the last topic of this lecture is how we can go about identifying the cases we may want to represent. There are no hard and fast rules for this, but a number of useful techniques. The one we'll talk about here is event decomposition, and is simply a lens through which we can view interactions with a system by focusing our attention on the events that will be raised.

Slide 26: We treat a system as a black box for this reason: it's just 'the system'. We then think through three event classes – external events (what kind of things will users want to do with this system), temporal events (what kind of things are going to happen on a regular or time-delayed basis) and state events (what is supposed to happen when the state of data is changed).

Slide 27: All we are looking to do from this process is build a list of candidate actions. We won't get them all, and a lot of the ones we get will be inaccurate or irrelevant. That's okay, as when we iterate over our system, we'll refine and modify that which we need to refine and modify.

Slide 28: This shows an example of candidate actions derived from a simple scenario of an ecommerce site. You can usefully spend some time here coming up with an appropriate simple scenario and getting students to collaboratively call out possible events for each of the available actors.

Slides 29-30: Temporal events are those that are either time-lapsed or time-dependent. Many systems come with requirements such as 'must send updates to management on a monthly basis' or 'must email users if their passwords have not been changed in six months', and these are captured as temporal events.

Slide 31: State events are those that occur when the state of data in the system is changed – the events are dependent on the data being set to a particular value, or meeting a certain requirement. Most often, this kind of events occur as a result of one of the other events, such as a temporal event where stock levels are assessed.

Slide 32: The only events we care about are those that directly impact on our system. Going through this process will raise lots of candidates that we can't or shouldn't use, and part of our constant refining will get rid of these, or split them up and combine them as needed. We're not looking to get the answer right from the start, we just want a starting point from which we can work.

Slide 33: Conclusion

Slide 34: Terminology

3.5 Laboratory Session

The time allocation for the laboratory session for this topic is 2 hours.

Lecturers' Notes: Students have copies of the seminar activities in the Student Guide. Answers are not given in their guide.

Activity 1:

Work together in small groups of three or four.

Consider the following simple scenarios, and generate an appropriate list of candidate actions:

1. A web-based book shop (such as Amazon)
2. A web-based social networking site (such as Facebook or Orkut)
3. A desktop word processor

Suggested Answer:

Much of this will be based on how students analyse the scenarios. They are sufficiently simple that anything even tangentially related can be included, and you should permit students to make whatever reasonable assumptions they require. A few possibilities however are outlined below:

1. Web-based book shop

Customer	Search for books by ISBN Search for books by author Search for books by title Add books to shopping basket Remove books from shopping basket Update financial details Create an account Review book Rate reviews

Administrator	Add books Modify book details Delete books Modify customer accounts Delete customers

2. Web-based Social Networking

User	Update details Browse profile Search for friends Add friends Message friends Post status updates Delete status updates Post links Like status updates Create groups
Administrator	Delete accounts Authorise accounts Browse accounts Change passwords

Exercise Three:

User	Open a document Format document Print a document Save a document Spell-check Grammar check Insert tables Insert graphics Convert file formats

Activity 2:

Working alone, take the analysis of the scenarios above and create appropriate use case diagrams for each in StarUML.

Suggested Answer:

No suggested answer – highly dependent on the results of student analysis.

3.6 Private Study

The time allocation for private study in this topic is expected to be 7.5 hours.

Lecturer's Notes:

Students have copies of the private study exercises in the Student Guide. Answers are not provided in their guide.

Exercise 1:

If you did not complete the diagrams from the seminar, finish these during your private study time. Remember that the tool you need is open source and can be freely downloaded.

Exercise 2:

As part of your ongoing journal exercise, you should research the following topics:

- Use case diagrams
- SSADM
- Benefits of OOAD over structural modelling
- Drawbacks of OOAD over structural modelling

Exercise 3:

Prepare a short, five minute presentation on the results of your research for Exercise 2 above that you will present during the tutorial session. If you have found out anything particularly interesting, you should focus on that as a priority.

Exercise 4:

Review the lecture material and ensure that you are comfortable with everything discussed thus far.

3.7 Tutorial Notes

The time allowance for tutorials in this topic is 1 hour.

Lecturers' Notes:

Students have copies of the tutorial activities in the Student Guide. Answers are not provided in their guide.

This tutorial is designed to give students a chance to present the more interesting of their findings from the previous exercise, partially as a way to share information with others in the class but also as a way to ensure that they are indeed keeping a journal and researching the topics provided. You should encourage students to bring along their journals to these sessions and make notes on any points of interest that are raised by their classmates.

Exercise 1: Reporting Back to the Class

As a result of the research you did during your private study time, you should have a short five minute presentation ready to give to the rest of the class. There is no need for this to be especially formal - you are simply reporting on anything interesting that you found during your research, or pointing out especially useful resources on the topic. Bring your journal along to the class so that you can make a note of anything especially useful that your classmates mention. This is a knowledge dissemination exercise; you are not being formally assessed on the style or content of the presentation.

Topic 4: Static Modelling in UML

4.1 Learning Objectives

This topic provides an overview of class diagrams and static modelling. On completion of the topic, students will be able to:

- Use Natural Language Analysis to identify candidate classes and methods;
- Design class diagrams;
- Implement class diagrams in code.

4.2 Pedagogic Approach

Information will be transmitted to the students during the lectures. They will then practise the skills during the laboratory sessions. Tutorials are then used to consolidate students' understanding of the concepts covered and deal with any questions.

4.3 Timings

Lectures:	2 hours
Laboratory Sessions:	2 hours
Private Study:	7.5 hours
Tutorials:	1 hour

4.4 Lecture Notes

The following is an outline of the material to be covered during the lecture time. Please also refer to the slides.

The structure of this topic is as follows:

- Static models
- Natural Language Analysis
- Candidate Classes
- Implementation

4.4.1 Guidance on the Use of the Slides

Slide 2: In this lecture, we are going to explore the process of building class diagrams. We have already seen in a previous lecture how we use the StarUML CASE tool to draw class diagrams, but we haven't yet discussed how we can create them. This lecture is part of the content that addresses that. Class diagrams are part of the static model of a system; it is an architectural notation that does not represent in any way the functionality that underlies a system.

Slide 3: The static view is 'time independent', which means that the passage of time is not represented in any manner in the notation. Thus, user interactions over time, and the changing state of data are not incorporated, and neither is the order in which particular methods may be invoked or which classes must be instantiated in which order. The notation covers only how classes relate to each other, and what data and operations they have as part of their makeup. Class diagrams belong to the structural view of a system.

Slide 4: By far the most difficult thing about class diagrams is actually deciding what classes, attributes and operations are part of a model. There are many formal techniques that can be used to work this out, but a simple process of 'Natural Language Analysis' can go a long way towards creating an effective first draft of how a system is going to be constructed. It's not perfect, and will require redrafting as time goes by, but that is an inevitable consequence of any design process – it will need to take into account the evolving understanding that stakeholders have in the development of the software.

Slide 5: Natural Language Analysis (NLA) is a simple system whereby we identify nouns, verbs and adjectives in a piece of text and assign those to their likely role in a computer system. Nouns are likely to be classes, adjectives are likely to be attributes (although usually not in their basic form) and verbs are likely to be operations that will need to be allocated to a class. Adjectives are a special case because they imply the existence of an attribute, but are not the attribute itself. 'The red car will do X and the yellow car will do Y' implies the existence of a 'colour' attribute that belongs to the Car class, for example.

Slide 6: The elements that we discover using this process may or may not be a good fit for a computer system – as such, we call them 'candidates' – they are candidates for inclusion in the system, but we need to assess them first. We do this by examining each of the candidates according to several criteria. If we have candidates that are synonyms for each other, then we keep only one (the one with the most appropriate name). We need to be sure that the scope of our systems is controlled, and so we

get rid of anything that falls outside of our project scope. We also get rid of irrelevant candidates; the documents that we perform such Natural Language Analysis upon are usually written in such a way as to make them easy for humans to parse, but not formally enough to build a system around. People avoid things like repetition in their written text, so they may choose synonyms to ensure that their documents read well.

Slide 7: The text here is for a very simple description of a very simple system. Most Natural Language Analysis exercises are on documents that are several thousand words in length. This paragraph though will serve as a useful baseline to show how we can use NLA to extract actionable information from a document.

Slide 8: First, we identify each of the nouns. We don't make a value judgement at any point of this process – if we see a noun, we include it. In the format of the module, a bold word in a piece of text is used to indicate that a noun (proper or otherwise) has been encountered.

Slide 9: Next, we identify the verbs (there are no adjectives in this simple paragraph). We include these in square brackets. Although we don't formally include the context of the verb (what it is acting upon), we will use that to allocate actions to classes later on in the process.

Slide 10: We end up with a pair of lists at the end of this – one contains classes that may or may not be part of the system, and the other contains functionality that we may or may not wish to include. It is obvious just looking at the lists that they contain quite a lot of extraneous and irrelevant entries, but that is part of the process. Having arrived at our first list of candidates, we need to assess each of the entries and remove those we do not wish to consider for inclusion in our system design.

Slide 11: First, we remove those candidates that are synonyms of each other. Customers and Patrons are a good example of this – they both relate to the people making use of the library. 'Hold' and 'Reserve' is another example, since these are two words relating to the process of placing a block on a book so that it can be checked out later. We then remove those entries that are too high a level of abstraction – 'manipulate the library' is an action that is too abstract to be worth keeping in our design. Manipulating the library properly breaks down into a range of activities that involve editing patron information and book details. We also remove those entries that we have to do anyway – we'll need to keep a database (that is, data about the state of the library) as a matter of course, so we don't need to represent it in our system design. Finally, we remove those elements that are outside the scope of our project.

Slide 12: Once we've gone through this process, we end up with a smaller list of candidates. Again, not all of these will end up being part of the system, and some of the classes will end up being incorporated as attributes, but that's all part of the iterative design process. The important thing is that we have a starting point for productive iteration.

Slide 13: The context of the verbs we encountered in our Natural Language Analysis will help inform us where functionality should be situated. If we say we must be able to 'add a book', we know that we need some book adding functionality located where it is most sensible. These assignments can be difficult, because we want the functionality to be as close to the data it uses as possible, while also being as high up an inheritance chain as it can be to ensure reusability. Situation of functionality

is often a fluid task and the realities of implementation will usually influence where operations end up being located in a system.

Slide 14: The act of constructing a class diagram from our candidate classes will help us refine our list. If we end up with a class that contains only a single attribute and no methods (aside from accessors) for acting on that attribute, it's likely that would be best represented as an attribute in another class. Shelves and storerooms for example are identified as candidate classes in our analysis, but what value there is in having them as full classes is uncertain. As such, when it comes time to build our class diagram, we'll condense these into a 'location' attribute of the book itself. All we are looking to do with our first draft is to cement our understanding of the system and provide ourselves with something we can take back to the client and further refine.

Slide 15: Ambiguity is an inescapable fact of life when developing software. People are not computers, and as such human communication serves a different purpose than computer programs. Our NLA candidates are only a way for us to direct our thinking about a system, they're not the 'right' answer. We are often going to have to add in things that have not been discussed, or remove things that have been. We often need to ask for clarifications, and we often need to infer the existence of some classes, attributes and operations from the information we have. If we are told we must store data, we can infer from this that we will be required to manipulate that data at some point also. Our NLA is a starting point, not an end goal.

Slide 16: Sometimes it is just the word choice that is used that leads to ambiguity – 'manipulate' is an ambiguous term and it will need to be clarified before we can progress. We may have one assumption from the context in which the word was used, but the author of the document we applied our NLA to may have had an entirely different meaning in mind.

Slide 17: This slide shows a first draft of a class diagram; notice it is lacking in much detail. This is the draft we'd use as the basis of further refinement with the client. Since we don't know for sure how much of what we have is correct, we don't spend the time in worrying about data types, visibility, or cardinality of classes. We just want to show the structure and what attributes/operations we have deduced.

Slide 18: Notice too that we don't include the web page here – our class diagram is the model of how the various aspects interact structurally. User front-end details wait until a later part of the modelling.

Slide 19: This shows a second draft of the class diagram, this time being filled out with the necessary information that leads to implementation. We have data types, cardinality and parameters all implemented into the diagram.

Slide 20: It is important to note here that our class diagram is just one of many diagrams we'll construct during the course of this module – it exists only to show us how the various classes will interrelate. We can't use this to build a program, because there is no emphasis on **how** methods will work. All we state at this point with this diagram is that the methods exist and they have a particular structure.

Slide 21: The need for us to state data types and the lists of parameters puts us in an awkward position - because we don't know how the functionality is to be implemented, we have to decide on what is appropriate data without knowing the

code. At this stage, we do a 'best guess' of what type data will be – we will end up changing it as we go through our development model.

Slide 22: Class diagrams imply that we are going to end up implementing the system in an object-oriented programming language. However, beyond the requirement that the language be object-oriented, UML does not mandate a particular language. Any language that has the necessary OO features will work for us. The class diagram simply details the relationship between the classes – we can take these and implement them in any suitable language simply by leaving out functionality and replacing it with stubs.

Slide 23: Normally we wait until our model is more mature before we build the system in code, but there is benefit to be gained by doing this early through a throw-away prototype. Prototyping allows us to identify obvious deficiencies in our system at the earliest point in the design, because if our model is incorrect then we simply won't be able to compile the prototype.

Slide 24: Prototyping is an important part of building software, and it falls into two types. Throw-away prototyping is usually unpopular with developers because people, as a rule, do not like having to do work that is only going to end up being discarded. However, doing so ensures that the benefits of prototyping are gained without the need to support legacy functionality from the early days of a system. Incremental prototyping involves starting with a prototype and continually modifying it until it does what the final system was supposed to do – the final product emerges from the prototype. The latter is usually more popular with developers, but it can lead to a lot of accumulated 'cruft' (discarded technical clutter) in the system.

Slide 25: This slide shows the code that is created from the class diagram. We look only at the Library class here – note that our methods are empty save for that which is required to make the program compile.

Slide 26: All we are looking for at this stage is a program that compiles – we have return values in some functions because the system will not compile without them. It doesn't have to **do** anything when it has compiled, it just needs to compile.

Slide 27: It is easy to fall into the trap of thinking that all the work goes into building the diagram and the developer has no creative input into the project. There are always specific implementation details that the developer is going to have to consider when building the software – these are usually not mandated as part of the design. In the case of our library class, the mechanism by which we implemented the cardinality is through an ArrayList – that has implications for scalability and ease of development, and deciding on the implications is part of the role of the developer. Another developer may have chosen to implement the cardinality as an associative array – neither is objectively 'correct', but they are part of the judgement call that the developer must make when implementing a system.

Slide 28: It is important that developers do not spend too much time in the early stages building these prototypes. Most CASE tools (StarUML included) have functionality that will generate the code files directly from the diagram, and for much of the early stages of the system it is these that should be used to determine whether or not there are structural deficiencies in the diagram.

Slide 29: Class diagrams offer one view of a system, but computer code offers another. The code view, because of the additional error checking of the compiler, will often reveal problems with how the code is to interact – you can see whether or not associations

are honoured, or whether attributes are missing or whether operations that are implied by the existence of attributes are actually represented on the diagram. When making changes to the class diagram, you should regenerate your code view and ensure it compiles.

Slide 30: Conclusion

4.5 Laboratory Sessions

The time allocation for the laboratory sessions for this topic is 2 hours.

Lecturers' Notes:

Students have copies of the seminar activities in the Student Guide. Answers are not given in their guide.

Students may ask you to clarify these (intentionally) ambiguous problem statements, and you should feel free to do so in any way you feel is appropriate. You can invent details, and there is no need to ensure that all students have the same information – the intention of the exercises here is to get students into the habit of thinking through the analysis of systems and generate class diagrams from the result of those analyses. The intention is not for them to develop a perfect class diagram.

Diagrams should be developed using StarUML, with the result of the students' NLA being the basis for the construction of the diagrams.

Activity 1:

Perform NLA on the following paragraph, and create the class diagram that follows from your analysis. You should go through the proper stages of the NLA process discussed in the lecture, including dealing with synonyms and redundancies, and inferring attributes, classes and operations from the information you have available.

> *The system that you are being asked to build is one of a web-based front-end to a hardware shop. The hardware shop contains many different kinds of tools such as hammers, screwdrivers and wrenches. The shop has a customer loyalty program and gives a discount to customers who have spent a certain amount of money over the past year. The software you write must also keep track of stock in the warehouse.*

Once you have generated the class diagram, either write or generate the code that is implied by the diagram. Your program must be able to compile, but it does not have to do anything that is required of the program in the problem statement.

Suggested Answer:

Generated diagrams will vary considerably depending on how students approach the analysis, but the analysed text should look something like this:

*The **system** that you are being asked to [build] is one of a **web-based front-end** to a <hardware> [shop]. The <hardware> **shop** [contains] many different kinds of **tools** such as **hammers**, **screwdrivers** and **wrenches**. The **shop** has a **customer loyalty program** and [gives] a **discount** to **customers** who have [spent] a certain amount of **money** over the past **year**. The **software** you [write] must also [keep track] of **stock** in the **warehouse**.*

Analysis should follow the general pattern in the lecture, and include identifying synonyms (software and system for example), discarding irrelevant information, and removing details out-with the scope of the system. There is much functionality that is implied but not specifically stated – for example, there are certain tasks we would normally associate with a web-based shop – buying items, logging in, and keeping track of our accounts (especially important when thinking about customer loyalty). Keeping track of stock in the warehouse is an ambiguous requirement, and students should either seek clarification of what that means, or infer the functionality from context and comparison,

Students can approach this exercise in many different ways (for example, are the different kinds of tools specialisations from a general tool class, or should the tool class simply contain the type of the tool as an attribute?), and so it is not possible to give a suggested class diagram that would be representative.

Activity 2:

Perform NLA on the following paragraph, and create the class diagram that follows from your analysis. You should go through the proper stages of the NLA process discussed in the lecture, including dealing with synonyms and redundancies, and inferring attributes, classes and operations from the information you have available.

> *The shop is called Charlotte's Web, and specialises in spider-themed jewellery that is sold from a retail outlet in the city's high street. The shop has a high degree of turnover and is looking to commission someone to develop a web-based store that customers can use. It should not be possible for people to buy jewellery directly from the web – all of our jewellery is custom made and all that customers should be able to do is book an appointment with one of our specialised spider jewellers. This is an exciting opportunity and we hope that you are as enthused about this task as we are!*

Once you have generated the class diagram, either write or generate the code that is implied by the diagram. Your program must be able to compile, but it does not have to do anything that is required of the program in the problem statement.

Suggested Answer:

Generated diagrams will vary considerably depending on how students approach the analysis, but the analysed text should look something like this:

*The **shop** is called **Charlotte's Web**, and [specialises] in **spider-themed jewellery** that is [sold] from a **retail outlet** in the **city's high street**. The **shop** has a high degree of **turnover** and is looking to [commission] **someone** to [develop] a **web-based store** that **customers** can [use]. It should not be possible for **people** to [buy] **jewellery** directly from the **web** – all of our **jewellery** is custom made and all that **customers** should be able to do is [book] an **appointment** with one of our specialised **spider jewellers**. This is an exciting **opportunity** and we hope that you are as enthused about this **task** as we are!*

The analysis for this task will primarily revolve around removing unnecessary detail such as the information about the turnover and where the shop is located. However, the usual tasks of identifying repetition, synonyms and irrelevances should be performed also. Students should also be encouraged to infer the functionality that is not specifically stated.

Students can approach this exercise in many different ways, and so it is not possible to give a suggested class diagram that would be representative.

4.6 Private Study

The time allocation for private study in this topic is expected to be 7.5 hours.

Lecturer's Notes:

Students have copies of the private study exercises in the Student Guide. Answers are not provided in their guide.

Exercise 1:

If you did not complete the work from the practical session, finish these tasks during your private study time. Remember that the tool you need is open source and can be freely downloaded.

Exercise 2:

As part of your ongoing journal exercise, you should research the following topics:

- Natural Language Analysis
- Implementing code from class diagrams
- Candidate classes in UML

Exercise 3:

Take the code that you obtained for Activity 1 in the laboratory session. Add in the following functionality, making a note in your journal when you had to add to your design in order to make it work. Make assumptions where required, but note these assumptions in your journal.

- Add a new type of tool that can be sold
- Adjust the quantity of tools in stock
- Allow a customer to buy some of the stock, honouring the discount offered by the loyalty program.

Exercise 4:

Prepare a short, five minute presentation on the results of your research for Exercise 2 above that you will present during the tutorial session. If you have found out anything particularly interesting, you should focus on that as a priority.

Exercise 5:

Review the lecture material and ensure that you are comfortable with everything discussed thus far.

4.7 Tutorial Notes

The time allowance for tutorials in this topic is 1 hour.

Lecturers' Notes:

Students have copies of the tutorial activities in the Student Guide. Answers are not provided in their guide.

This tutorial is designed to give students a chance to present the more interesting of their findings from the previous exercise, partially as a way to share information with others in the class but also as a way to ensure that they are indeed keeping a journal and researching the topics provided. You should encourage students to bring along their journals to these sessions and make notes on any points of interest that are raised by their classmates.

Exercise 1: Discussion of Private Study Exercise

Discuss as a group your solution to Exercise 3 in the private study session.

Suggested Answer:

The code that students come up with for this exercise will be highly dependent on the way that they approach the problem, and so a representative answer cannot be given. However, certain commonalities should reveal themselves – detail was lacking in the scenario (for example, what does a 'certain amount of money' mean with regards to the loyalty scheme?), and the result of the analysed NLA is not sufficient to implement the needed functionality because of ambiguity and missing detail.

Exercise 2: Reporting Back to the Class

As a result of the research you did during your private study time, you should have a short five minute presentation ready to give to the rest of the class. There is no need for this to be especially formal - you are simply reporting on anything interesting that you found during your research, or pointing out especially useful resources on the topic. Bring your journal along to the class so that you can make a note of anything especially useful that your classmates mention. This is a knowledge dissemination exercise; you are not being formally assessed on the style or content of the presentation.

Topic 5: Dynamic Analysis and Design

5.1 Learning Objectives

This topic provides an overview of activity and sequence diagrams. On completion of the topic, students will be able to:

- Make use of activity diagrams;
- Turn activity diagrams into code;
- Develop sequence diagrams.

5.2 Pedagogic Approach

Information will be transmitted to the students during the lectures. They will then practise the skills during the laboratory sessions. Tutorials are then used to consolidate students' understanding of the concepts covered and deal with any questions.

5.3 Timings

Lectures:	2 hours
Laboratory Sessions:	2 hours
Private Study:	7.5 hours
Tutorials:	1 hour

5.4 Lecture Notes

The following is an outline of the material to be covered during the lecture time. Please also refer to the slides.

The structure of this topic is as follows:

- The notation of activity diagrams
- Creating activity diagrams
- Turning activity diagrams into code
- The role of sequence diagrams
- Sequence diagram notation

5.4.1 Guidance on the Use of the Slides

Slide 2: In this lecture, we look at how dynamic models can be used to build up the time dependent view of the system that we need in order to develop functionality. The two diagram notations we discuss during this lecture are only a subset of the models that are available to meet this need.

Slide 3: The first of the diagrams we look at is the activity diagram. The activity diagram is largely a formalised flow-chart with extended notations that allow for it to represent the flow of execution through a multi-object system. Activity diagrams are used to give an easily understood representation of the flow of logic through a given process.

Slide 4: This slide shows some of the notational elements that make up an activity diagram. As with most UML diagrams, there is a lot of complexity here that has been left out; UML is a dense and complicated notation, and not all of it is relevant to novices. The notation elements mentioned here are all represented in the example diagrams to follow.

Slide 5: This slide shows some more of the notation. Forks and joins are outside the scope of our systems to date (they are mostly to do with ensuring a representation for parallelism), but we will have cause to use each of the other notational formats as we progress through the module.

Slide 6: This slide shows an example of a simple activity diagram. The diagram in question is an analysis of an existing workflow – it represents how things are currently done in a system. Activity diagrams can be used for both analysis (getting a full understanding of how processes are undertaken) and design (representing improved or optimised processes prior to implementation).

Slide 7: In their first capacity (as a high level view of activity in a system) they can be used to see how actors in a system interact, where responsibilities are properly allocated, and what kind of decisions are taken through the course of performing an action. This information is vital in building a full representation of how the various aspects of a complex workflow interrelate. In their second capacity, activities serve as a form of graphical pseudocode, with all of the attendant benefits that come from the use of such. Contrary to how pseudocode is created, activity diagrams are notationally consistent and as such the pseudocode that they represent will be instantly understandable to anyone who is familiar with how UML works.

Slide 8: This slide shows the activity diagram for a process that will be implemented in code – specifically, the finding a book operation. Activity diagrams, by virtue of being flow-charts, lack many of the common programming structures we may make use of (loops, for example, are handled via an explicit decision that links back to an earlier part of the diagram). The outline of the programming logic is platform independent, and we as developers can decide, when it comes time to implement the process, exactly what kind of code is most appropriate for our needs.

Slide 9: As with writing computer code, developing an activity diagram is an intensely creative process and one that doesn't have a 'right' answer to go with it. A profitable way to build a computer program is to implement only the simplest, highest level functionality and then specialise that as time goes by – a similar technique can be useful for creating an activity diagram. To begin with, each activity can be represented by a single activity, and that can be continually broken up into discrete steps until the final, fully detailed diagram is constructed.

Slide 10: The two kinds of activity diagrams can serve as a way to direct this specialisation of detail – much as with the construction of a class diagram, it is derived from our understanding of how the system currently functions. Our use case diagrams will serve as a good basis for deciding what must be represented in activity diagrams, with each distinct use case being a candidate for expansion. We begin with a single activity box that contains the text of the use case, and then we expand it in line with our understanding of the process. 'log in' may be a use case, which we then represent as a single 'log in' activity, which becomes three activities (enter name, enter password, validate password), which is then further specialised as time goes by. These diagrams of the process can then serve as what becomes our design – we can look at how systems work, or how they are supposed to work, and refine the flow of the activity until it is as clean and effective as it can be.

Slide 11: The Natural Language Analysis exercise discussed in the previous chapter will go some way towards helping build up all of the diagrams we discuss during the module. Additionally, every diagram we draw as we go through the OOAD process will help in codifying our understanding of the system; sadly, there is no short-cut or simple way in which this can be done. It all involves thinking through how the parts of the system interact, and documenting according to our best understanding. Our best understanding is usually going to be insufficient, at least in early stages of OOAD, and so having someone try to work through the logic we have put in place can be an instructive exercise – they will identify ambiguity, missing steps and imprecise instructions, and each identified problem will increase the effectiveness of the diagrams we create.

Slide 12: The NLA reveals behaviours that will need to be diagrammed, and the use case diagrams will reveal actors that are likely to be involved. The classes identified during NLA will likewise help when it comes time to develop the sequence diagrams that follow. UML is an integrated system, which means that every diagram relates to every other diagram; it can seem like a collection of unrelated notation systems, but each helps inform the development of the others.

Slide 13: It can be a useful first step in developing an activity diagram to outline the process first in structured English (which has a low cost in developing and changing) and then convert it into the formal notation of an activity diagram later. It is important to understand that not all of the detail that will eventually be incorporated into the diagram has to be implemented at the start – as the understanding of the system evolves, so too will the diagrams. Granularity however is a constant difficulty when developing diagrams, especially when working across multiple individuals. The

exact detail that an activity should represent will often vary between different people, and that can make it difficult to integrate diagrams and implement them into code.

Slide 14: This slide outlines a process that can be iteratively followed to help build activity diagrams. First, we decide which process is to be documented (these will either come from the NLA or the use case diagrams). We remove all the extraneous detail that doesn't relate to our activity, and then work through the logic. When we reach a decision, we make sure to include all options. When we encounter a repetition, we make sure to put down precisely how the loop terminates. When we encounter activities, we continually break them down until they represent, at best, a single line of code.

Slide 15: Activity diagrams lend themselves easily to code, because they are really just a graphical representation of program logic. They are focused primarily at the level of the method or operation. If the activity diagram is appropriately constructed, we can render it almost line for line in a suitable OO language.

Slide 16: However, as with all UML diagrams, activity diagrams have a level of abstraction to ensure that they are genuinely platform independent. As such, there is still a degree of value judgement involved in turning the diagrams into actual implemented code. The diagrams don't contain shorthand notation for things like loops, and so we must decide as we implement how we are going to best represent the logic of the activity diagram. One useful way to do this is to begin by writing out the logic in full, without revision, and then condensing it into the best possible code representation.

Slide 17: This slide shows the implementation of the logic outlined in our activity diagram – the loop that handles stepping over each of the books in turn is done long-hand. Java does not contain a 'goto' keyword or any mechanism other than a loop to handle going back and repeating, so we just add a comment at that point.

Slide 18: This slide shows a revised version of the same program, this time using a loop. The choice of looping structure is up to the developer – in other circumstances we may have chosen to use a while loop, or even recursion.

Slide 19: Because of the tightly interrelated nature of object oriented programs, it's often the case that we need to implement everything at once or implement nothing at all. Our findBook method is dependent on there being an implemented getISBN method, as one example. In such circumstances, we begin developing from the inside out – we begin with the simplest methods that exist in isolation with no dependencies on other functions. We then implement those that are dependent on the functions that we just completed, and finally we can start developing those functions that have the highest amount of dependency on existing functionality.

Slide 20: Having discussed the activity diagram, the last topic for this lecture is the sequence diagram. This is a related diagram in that it is also used to give a view of the dynamic aspects of a system. However, its role is different and we will discuss that in the last few slides of the lecture. They are mostly about giving a high level view of how objects and classes interrelate in a complex process. This is the same basic area as that of an activity diagram, but the level of abstraction is higher.

Slide 21: Lifelines are the key mechanism used in a sequence diagram – these are the boxes that extend along the dotted lines that drop down from an object definition. Bars

are used to indicate when the object is in scope, and arrows are used to indicate the messages that are passed between objects as part of their lifecycle.

Slide 22: This slide shows a simple example of a sequence diagram. Note that the messages that are passed are tagged with the name of methods, and the type and name of parameters. This relates to the method information that is available as part of the constructed class diagram such as we built last week.

Slide 23: Sequence diagrams also need to be able to give a high level view of multiple paths of execution through an activity. In order to do this, we use a system of 'frames' which let us indicate parts of the diagram that are conditional on external and internal criteria. These criteria are known as 'guard' conditions, and these too are usually expressed in simple code. When we wish to show that a frame provides alternate courses through a diagram, we tag it with 'ALT'. If we wish to show that a frame is entirely optional (such as, dependent on an if statement), we tag it with OPT.

Slide 24: This slide shows a sequence diagram with frames - the first frame is entered only if the value of 'valid' is true. If it is not, the alt frame marked as 'else' handles the flow of logic.

Slide 25: In a sequence diagram, we are working with objects, rather than classes. Properly, we will indicate this in the standard UML notation of <object> : <Class> so that we can uniquely identify between instances, or when we are working with static methods of a class.

Slide 26: Sequence diagrams are somewhat unusual in that they rarely directly inform the implementation process. Mostly they are used to help ensure that logical and architectural inconsistencies are identified early and that the dependencies between objects are clearly visible. This helps define the order of implementation of methods, because we can see which methods are involved in a particular process.

Slide 27: There is a lot of overlap in what sequence and activity diagrams do, but the key distinction is the role they are supposed to play. Activity diagrams assist in developing the detail of a process, while sequence diagrams give a high level view of interaction. Many analysts and designers will use sequence diagrams as an early 'sketch' of how they see the parts of a system fitting together, much like a throwaway prototype of functionality to come.

Slide 28: Collaboration between multiple developers is always complex, and one of the areas in which multi-developer projects break down are at the margins where sections completed by different individuals must interface. Sequence diagrams, because they show the specific methods and the parameters/return types of each, help in collaboration because everyone can see what public interface an object must honour in order to mesh correctly with the rest of the system.

Slide 29: Conclusion.

5.5 Laboratory Sessions

The time allocation for the laboratory sessions for this topic is 2 hours.

Lecturers' Notes:

Students have copies of the seminar activities in the Student Guide. Answers are not given in their guide.

This is a group exercise that requires several packs of playing cards and a card game to play. Groups should consist of around four individuals. The exact game chosen for the exercise does not matter, but a variation of blackjack would be an appropriate baseline. Groups should be composed of members who know the rules of the game, and also members who do not know – this will influence the choice of game for the exercise. You can also choose to let groups pick their own game. The first half an hour of the session should be given over to individuals playing a few hands of the game selected. Then, an hour should be given over to the second activity, and the final half hour being given over to the third activity.

Activity 1:

This is a group exercise that involves playing a game of cards and then constructing the appropriate activity diagrams. Your group will consist of members who know how to play the game and others who do not. You should spend some time at the beginning of the session playing through the game, with an effort being made to focus on resolving the ambiguities and questions raised by those who are learning for the first time.

Suggested Answer:

No suggested answer – highly dependent on game choice and student questions.

Activity 2:

Construct activity diagrams that represent the major processes of the game. If you are playing blackjack for example, you should construct activity diagrams that handle the dealing of the cards, the order of play, and the evaluation of winning conditions.

Suggested Answer:

No suggested answer – highly dependent on game choice.

Activity 3:

Pass the diagrams you have constructed to another group. You will receive their instructions in return. Try to play the game as they have outlined in their diagrams, **interpreting the instructions exactly**. Do not attempt to fill in ambiguity or missing information from your own understanding – simply play the game according to the processes you have been given.

Suggested Answer:

No suggested answer – highly dependent on game choice.

5.6 Private Study

The time allocation for private study in this topic is expected to be 7.5 hours.

Lecturer's Notes:

Students have copies of the private study exercises in the Student Guide. Answers are not provided in their guide.

Exercise 1:

As part of your ongoing journal exercise, you should research the following topics:

- Sequence diagrams
- Activity diagrams

Exercise 2:

Research the rules of five card draw (a variant of poker), and develop a set of diagrams that show how a hand should be dealt out, and what a player must do in order to play their hand.

Exercise 3:

Review the lecture material and ensure that you are comfortable with everything discussed thus far.

5.7 Tutorial Notes

The time allowance for tutorials in this topic is 1 hour.

Lecturers' Notes:

Students have copies of the tutorial activities in the Student Guide. Answers are not provided in their guide.

For the first part of this tutorial, you will again need packs of playing cards. You should get students together in small groups of four. Have them exchange their diagrams for playing a hand of poker, and then have them attempt to play a game using only the diagrams they have been provided. It is unlikely that a successful game of poker will be accomplished doing this, and this is intentional. Students should spend the rest of the tutorial discussing where their scenarios are not compatible, and refining their diagrams so that the group have a set of rules that can be used as the basis for the game. If a successful game is actually played, then separate that group of students out and assign them as 'consultants' to groups having problems.

Exercise 1: Poker Rules

In Private Study Exercise 2, you put together a set of diagrams that explain how to play a hand of poker. In groups of four, swap your diagrams and attempt to play a hand of poker using only the instructions you have been given.

Suggested Answer

Students should only play a few representative hands of the game in this exercise – the intention is for them to find common ground between the assumptions they have made, the phrasing and terminology they have used, and even the flow of logic they have assumed. Insist that students follow the diagrams exactly, without filling in blanks or discussing what is going on. The game should be unplayable, and students should make notes on the diagrams as to the problems they are encountering.

Exercise 2: Refining Models

As a group, work together to refine the diagrams you have produced until a successful game of poker can actually be played.

Suggested Answer

No suggested answer for this exercise, as it is highly dependent on student diagrams and the problems they have trying to follow the rules. Students should work together to address deficiencies and absences in their diagrams, as per the previous exercises they have done on the topic.

Topic 6: OOAD Case Study

6.1 Learning Objectives

This topic provides an overview of previous content in context. On completion of the topic, students will be able to:

- Analyse a complex scenario;
- Design diagrams based on that scenario.

6.2 Pedagogic Approach

The pedagogic structure for this topic is different from the others in the module. The lecture for this topic must come first and consists of a way for students to approach the provided scenario. It should be used to provide a framework for students to attempt this extended exercise.

Students have copies of the scenario in their Student Guides; however the answers are not provided and should rather be discussed within the tutorial. The scenario is intended to be incomplete and ambiguous so as to simulate the issues that are associated with handling OOAD in real contexts. The scenario also includes a considerable amount of detail that is not relevant to the worked solutions, and this too is intentional.

During the lecture, you should be prepared to answer any questions students have. If they ask for specific clarifications, you should provide them. For this scenario, it does not matter what the details of such clarifications are, only that they are given.

6.3 Timings

Lectures:	1 hour
Laboratory Sessions:	3 hours
Private Study:	7.5 hours
Tutorials:	1 hour

6.4 Lecture Notes

The following is an outline of the material to be covered during the lecture time. Please also refer to the slides.

The structure of this topic is as follows:

- How to approach the case study

6.4.1 Guidance on the Use of the Slides

This is the lecture that should be used to prepare students for the case study and to provide them with a simple framework from which they should work when developing their diagrams. Lecture slides here are not intended to take up the allocated time, only to serve as a brief introduction to what students should be doing. The rest of the time should be taken up with a question and answer session on the scenario that is presented. You should ask students to look at the scenario in their Student Guides during the lecture so that they can consider it and ask for whatever clarifications they feel are needed.

Slide 2-8: Simple framework for approaching the case study.

6.4.2 The Scenario

'Trucks for Hire' are a van and truck rental firm operating in a single city. The business was started several years ago, and has progressed to the point where expansion is a legitimate possibility. However, until now most of the administration of the business has been handled with paper files, post-it notes and conversations between staff members. The owner of the company knows that this is not sustainable, and has commissioned you to develop a piece of software that he can use to manage his organisation.

The company currently offers customers the opportunity to hire one of three kinds of vehicle – combo vans, transit vans, and box vans. When customers show up at the front desk, they must provide a valid driving licence and this determines which of the vans they can hire. Anyone can hire a combo van, anyone who has a driving licence that was issued before 1990 can drive a transit van, and for all other cases the appropriate driving licence credentials are required. The fee for the hire is based on the type of vehicle and the length of time that the individual is to have the vehicle. The company also offers extras, such as satellite navigation equipment and full insurance, but these are additional and not included in the standard price.

The company currently has ten of each of the vans, and customers can book ahead to reserve a vehicle on a particular day for a particular length of time.

For those customers who do not feel comfortable driving one of the vehicles themselves, the company will make available a driver on a per hour basis. However, they have only four drivers on the payroll and these can work a maximum of 40 hours in a week, and a maximum of two jobs a day, of which the sum can be no longer than eight hours (with an hour lunch break scheduled somewhere). Drivers are thus not always available, and customers are required to reserve their vehicle on a time and day when a driver is available. In addition to charging an hourly rate for the driver, the company also charges per mile, with the mileage of the hired vehicle being used in the calculations for the overall cost. The company provides an allowance of 40 miles (rural) or 10 miles (urban) per hour. Transporting a load 20 miles away through the city then is classed as a two hour journey. The maximum load that can be transported per journey is based on the capacity of the vehicle in question.

Your software must be able to handle all of this, and provide both a web-based interface for customers and a desktop application for staff. The owner would like as much of this to be automated as possible. As such your software should handle scheduling and allocation of drivers to jobs, as well as tracking which vehicles are available, when they are available, and which customers have made reservations.

6.5 Laboratory Sessions

The time allocation for the laboratory sessions for this topic is 3 hours.

Lecturers' Notes:

Students have copies of the seminar activities in the Student Guide. Answers are not given in their guide.

During the seminar, students should work on the activities outlined below in groups of three or four. Students have the responsibility for subdividing labour, but you should encourage them to ensure that every member of the group has a task in each of the activities. Do **not** go through any answers during this session, although you can provide guidance and advice. Students should be making use of the framework provided in the lecture to develop their own analysis of the scenario and the software that is to be constructed.

Activity 1:

Review the Natural Language Analysis of the provided scenario.

Suggested Answer:

'Trucks for Hire' are a **van** and **truck** rental **firm** operating in a single **city**. The **business** was [started] several years ago, and has [progressed] to the point where [expansion] is a legitimate possibility. However, until now most of the [administration] of the **business** has been handled with <paper> **files**, **post-it notes** and **conversations** between **staff members**. The **owner** of the **company** knows that this is not sustainable, and has [commissioned] you to [develop] a piece of **software** that he can use to [manage] his **organisation**.

The **company** currently [offers] **customers** the opportunity to [hire] one of three kinds of **vehicle** – <combo> **vans**, <transit> **vans**, and <box> **vans**. When **customers** show up at the **front desk**, they must [provide] a valid **driving licence** and this determines which of the **vans** they can [hire]. Anyone can [hire] a <combo> **van**, anyone who has a **driving licence** that was issued before 1990 can [hire] a <transit> **van**, and for all other cases the appropriate **driving licence credentials** are [required]. The **fee** for the hire is based on the type of **vehicle** and the length of time that the **individual** is to have the **vehicle**. The **company** also [offers] **extras**, such as **satellite navigation equipment** and **full insurance**, but these are extras and not [included] in the standard **price**.

The **company** currently has ten of each of the **vans**, and **customers** can [book ahead] to [reserve] a **vehicle** on a particular **day** for a particular **length of time**.

For those **customers** who do not feel comfortable driving one of the vehicles themselves, the **company** will [make available] a **driver** on a per hour basis. However, the owner has only four **drivers** on the **payroll** and these can work a maximum of 40 hours in a week, a maximum of two **jobs** a day, and the sum of these jobs can be no longer than eight hours (with an hour **lunch break** scheduled somewhere). Drivers are thus not always available, and **customers** are required to [reserve] their **vehicle** on a time and day when a driver is available. In addition to [charging] an hourly **rate** for the driver, the **company** also [charges] per **mile**, with the **mileage** of the <hired> **vehicle** being used in the calculations for the overall **cost**. The **company** provides an **allowance** of 40 miles (rural) or 10 miles (urban) per hour. [Transporting] a **load** 20 miles away through the **city** then is classed as a two hour **journey**. The <maximum> **load** that can be [transported] per **journey** is based on the **capacity** of the **vehicle** in question.

Your **software** must be able to [handle] all of this, and provide both a **web-based interface** for customers and a **desktop application** for **staff**. The **owner** would like as much of this to be [automated] as possible. As such your **software** should handle [scheduling] and [allocation] of **drivers** to **jobs**, as well as [tracking] which **vehicles** are available, when they are available, and which **customers** have made reservations.

Activity 2:

Review the selection of candidate classes from the NLA exercise you conducted.

Suggested Answer:

Initial Candidate Classes:

Candidate Classes	Candidate Operations
Trucks for Hire, Van, Truck, Firm, City, Business, Files, Post-It Notes, Conversations, Staff Members, Owner, Company, Software, Organisation, Vehicle, Customers, Front Desk, Driving Licence, Driving Licence Credentials, Individual, extras, satellite navigation equipment, full insurance, price, day, length of time, driver, payroll, job, lunch break, rate, mileage, mile, allowance, capacity, web-based interface, desktop application	Offer, develop, progressed, commissioned, provide, hire, required, offers, included, book ahead, reserve, make available, charges, transporting, handle, automated, scheduling, allocation, tracking

Our first task in reducing this to a more manageable list is getting rid of synonyms and overlapping candidate classes:

- Trucks for Hire, Firm, Business, Organisation, Company - we'll use organisation
- Van, Truck, Vehicle – we'll use vehicle
- Individual, customer – we'll use customer
- Driving licence, driving licence credentials – we'll use licence
- Day, length of time – we'll use length of time
- Mileage, mile – we'll use mileage

This leaves us with the following:

Candidate Classes	Candidate Operations
City, Files, Post-It Notes, Conversations, Staff Members, Owner, Software, Organisation, Vehicle, Customers, Front Desk, Driving Licence, extras, satellite navigation equipment, full insurance, price, length of time, driver, payroll, job, lunch break, rate, mileage, allowance, capacity, web-based interface, desktop application	Offer, develop, progressed, commissioned, provide, hire, required, offers, included, book ahead, reserve, make available, charges, transporting, handle, automated, scheduling, allocation, tracking

Next, we get rid of classes that are out-with the scope of our project:

- City
- Post-it notes, conversations, files
- Software, lunch break, web based interface, desktop application

Removing these leaves us with the following:

Candidate Classes	Candidate Operations
Staff Members, Owner, Organisation, Vehicle, Customers, Driving Licence, extras, satellite navigation equipment, full insurance, price, length of time, driver, payroll, job, rate, mileage, allowance, capacity	Offer, develop, progressed, commissioned, provide. Hire, required, offers, included, book ahead, reserve, make available, charges, transporting, handle, automated, scheduling, allocation, tracking

Next we get rid of synonyms or overlapping operations:

- Book ahead, reserve – we'll use 'reserve'

Candidate Classes	Candidate Operations
Staff Members, Owner, Organisation, Vehicle, Customers, Driving Licence, extras, satellite navigation equipment, full insurance, price, length of time, driver, payroll, job, rate, mileage, allowance, capacity	Offer, develop, progressed, commissioned, provide. Hire, required, offers, included, reserve, make available, charges, transporting, handle, automated, scheduling, allocation, tracking

And then operations out-with the scope of our project:

- Develop, progressed, commissioned, offer
- Transporting, handle
- Required, included

Candidate Classes	Candidate Operations
Staff Members, Owner, Organisation, Vehicle, Customers, Driving Licence, extras, satellite navigation equipment, full insurance, price, length of time, driver, payroll, job, rate, mileage, allowance, capacity	provide. Hire, reserve, make available, charges, automated, scheduling, allocation, tracking

Next, we must decide on each of these in turn to see whether they make sense for inclusion – the removal on criteria can only go so far, we need to think through the rest. First the classes:

Staff Member	Explicit mention is made of the fact that we have to handle allocation of drivers to vehicles and time slots, so there is certainly a role for certain kinds of staff members to be represented in the system. It seems that this is likely to be worth including.
Owner	The owner is likely to be a specialisation of the general case 'staff member'. We will include it in the class diagram, but as a specialisation of staff member and with the understanding that it may not be required later.
Organisation	The organisation itself is likely to be represented by all the constituent classes in the system. It will serve a useful role for linking all the other classes to each other.
Vehicle	We are certainly going to need to represent vehicles in our system, so this remains.
Customers	Customer details and reservations are explicitly mentioned in the brief, and so this class remains.
Driving Licence	We only need certain details about the driving licence to be stored, and as such we probably don't need a full class to represent it. Instead, this will be represented as an attribute of the customer.
Extras	We will need to store the extras that a customer has purchased, but we perhaps do not need a full class to do it. This is a borderline case – for the first draft, I would recommend dropping it in favour of attributes on the customer's reservation.
Satellite Navigation Equipment	As with extras, best represented perhaps as an attribute on the customer's reservation.
Full insurance	See satellite navigation equipment.
Price	We will need to record the price, but most likely this will be an attribute on the reservation.
Length of time	Most likely an attribute on the reservation
Driver	We will need to represent drivers, which we will do as a specialisation of the staff member class.
Payroll	We will need a class that stores the list of staff members, and so the payroll class will be appropriate for that.

Job	The job is perhaps better referred to as the customer's reservation. We'll rename the class when it comes time to do the diagram.
Rate	Best represented as an attribute of the vehicle
Mileage	Best represented as an attribute of the reservation
Allowance	The allowance will emerge from a calculation that we perform later – it should be an operation of its own.
Capacity	Best represented as an attribute of the vehicle.

Next, the operations:

Provide	In the context of the system we are building, this is too general a phrase to be meaningful and so we won't represent it.
Hire	Hiring should be the operation that creates a new customer reservation that is to be auctioned instantly, and so it should be represented in our system.
Reserve	Reserve should allow us to create a reservation for the future, and so it should be represented in our system.
Make Available	Too general to be included in our system
Scheduling	Scheduling will emerge naturally as a result of the way our system works, and as such does not need to be represented.
Allocation	As with scheduling
Tracking	As with scheduling

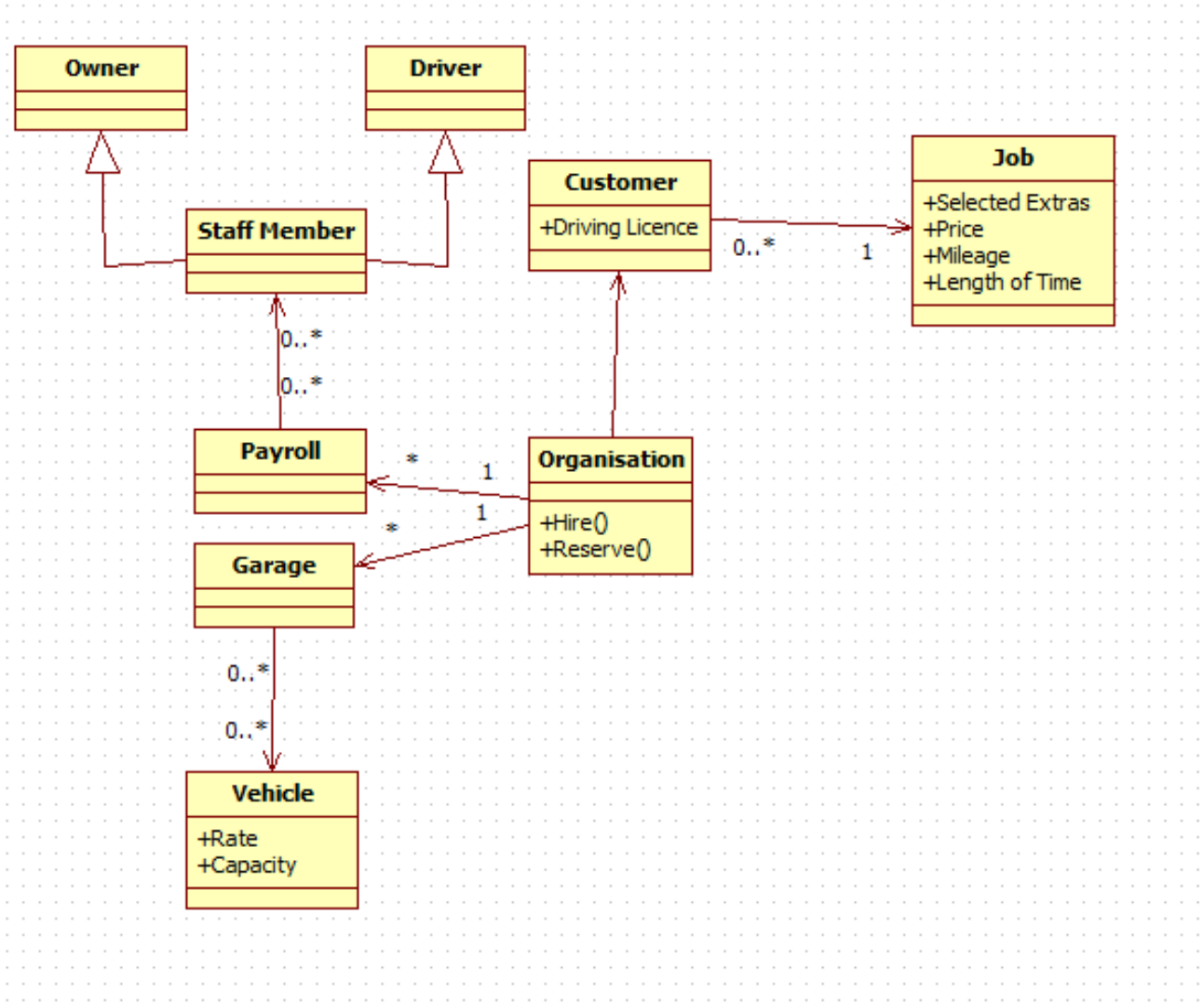
What we end up with at the end of this process is a manageable list of classes, operations and attributes. It is not complete, and we will need to fill in the missing blanks ourselves and add in some classes to support our structural needs, but it's an important first step in being able to create a suitable class diagram:

Candidate Classes	Candidate Operations	Attributes
Staff Members, Owner, Organisation Vehicle, Customers, driver, payroll, job	Hire, reserve	Driving Licence, Selected Extras, Price, Length of Time, Rate, Mileage, Capacity

Activity 3:

Review the class diagram that emerged from your Natural Language Analysis.

Suggested Answer:



Note that there is a Garage class here that was not present in the NLA. This is a convenience class that will manage a list of vehicles in the same way that the payroll will manage a list of staff members. The structural attributes of each of the classes are not shown here – at this stage of analysis, we are only looking to gain an understanding of how things will fit together and what role classes will play in the final system.

Activity 4:

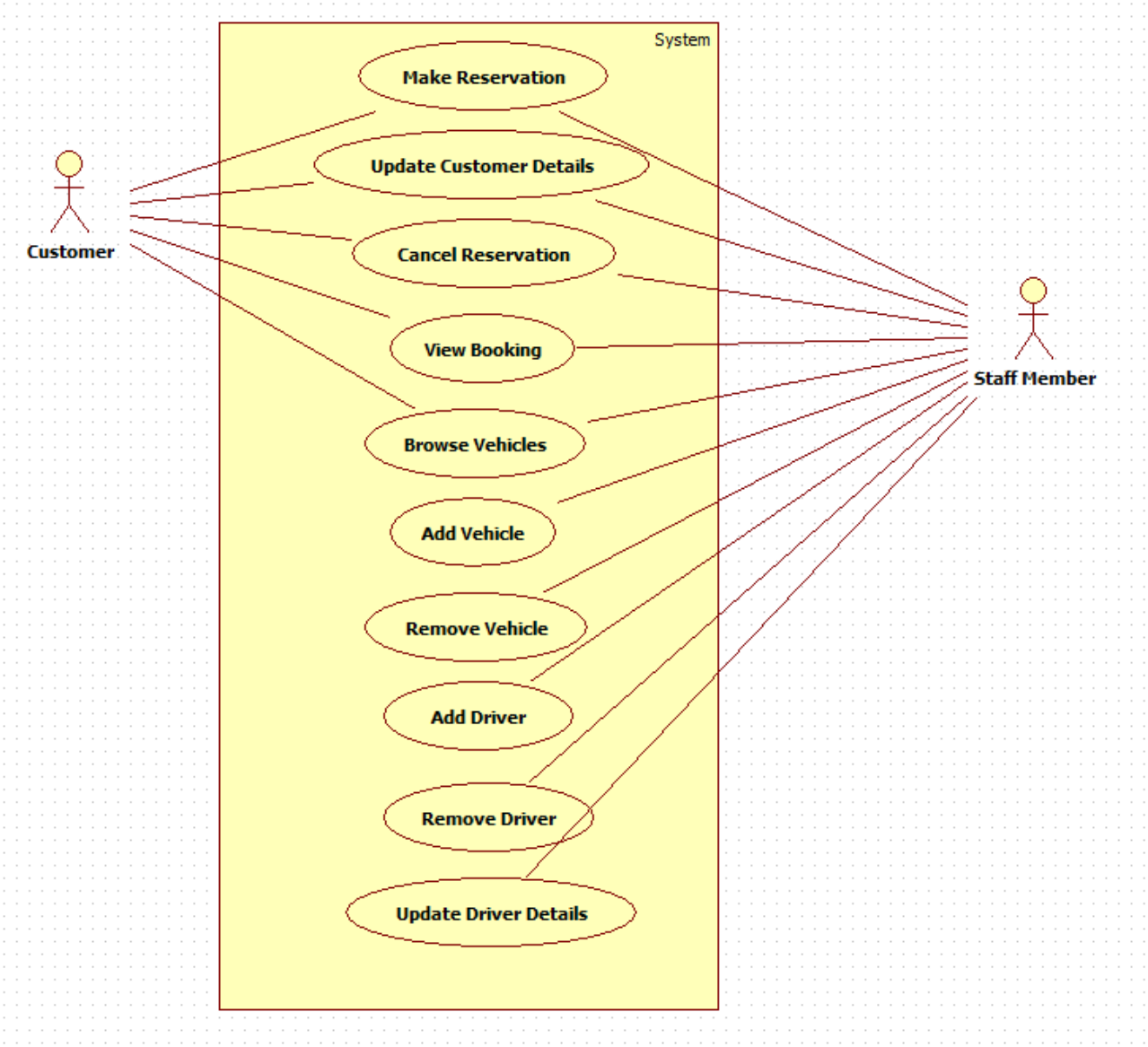
Review the use case diagram that you developed from the scenario.

Suggested Answer:

For the use case analysis, we must consider who the actors are going to be. We have at least two – customers, and staff members. The former is obvious because it is explicitly mentioned in the scenario. The latter emerges from consideration of how people will interact with the system – there will need to be a mechanism by which data can be put into the system, corrected and updated.

For the customer, we need to provide ways for them to make reservations, cancel reservations, update their details, and browse the available vehicles. For the staff, we need to provide ways for them to add vehicles, remove vehicles, make bookings (for those customers who still make use of the front desk staff), add drivers, remove drivers, and a range of other upkeep activities.

Our use case diagram then will look as follows:



Activity 5:

Review the activity diagrams that you developed from the scenario.

Suggested Answer:

We should have an activity diagram for each of the processes that we identify as use cases. These will first centre on how the system currently works, and then on how the implemented system should function. We won't go through all of these in the worked example because there is not enough time to do so. We will however show the full workflows of the processes for making a booking, cancelling a booking, and viewing a booking. Students will be encouraged to consider the other workflows during their private study time.

First we must understand how the system works. The problem statement, like all problem statements, is ambiguous as to what the current processes are but we can infer from context. If you are asked questions by students, clarify in whatever way you see fit. It is not important that a particular workflow be identified, only that a workflow **can be** identified.

Making a booking at the front desk involves the following steps:

1. Give details of booking
2. (Optional) Request a driver
3. Provide your driving licence details
4. Make a deposit

Each of these steps requires further refinement:

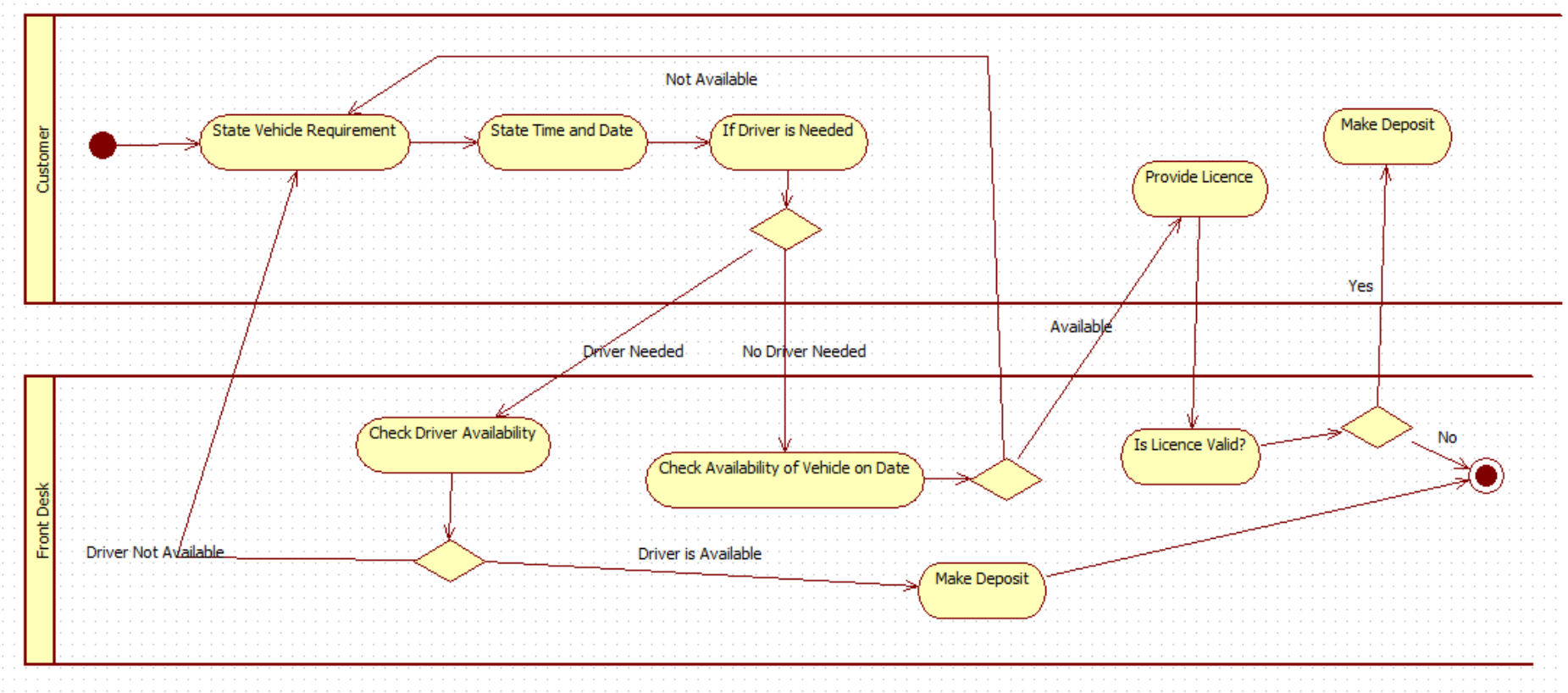
1. Give details of booking
    a. Provide the type of vehicle you want
    b. Give time and date. If the vehicle is available and you don't need a driver::
        i. Provide driving licence details
        ii. Make a deposit
    c. If vehicle is not available, go to a
    d. If vehicle is available but you need a driver:
        i. Check availability of driver on day. If driver is not available:
            1. Go to a
        ii. If the driver is available:
            1. Make a deposit

This gives a simple workflow that describes how individuals interact with the front-desk staff. The order in which actions are performed is mandated as part of this flow, as is the way in which we deal with problems with the booking. Part of what we will do when we are developing our design is optimise this structure to simplify it and remove unnecessary complexity – for example, this could be two workflows, one for hiring a van for yourself and one for hiring a van and driver.

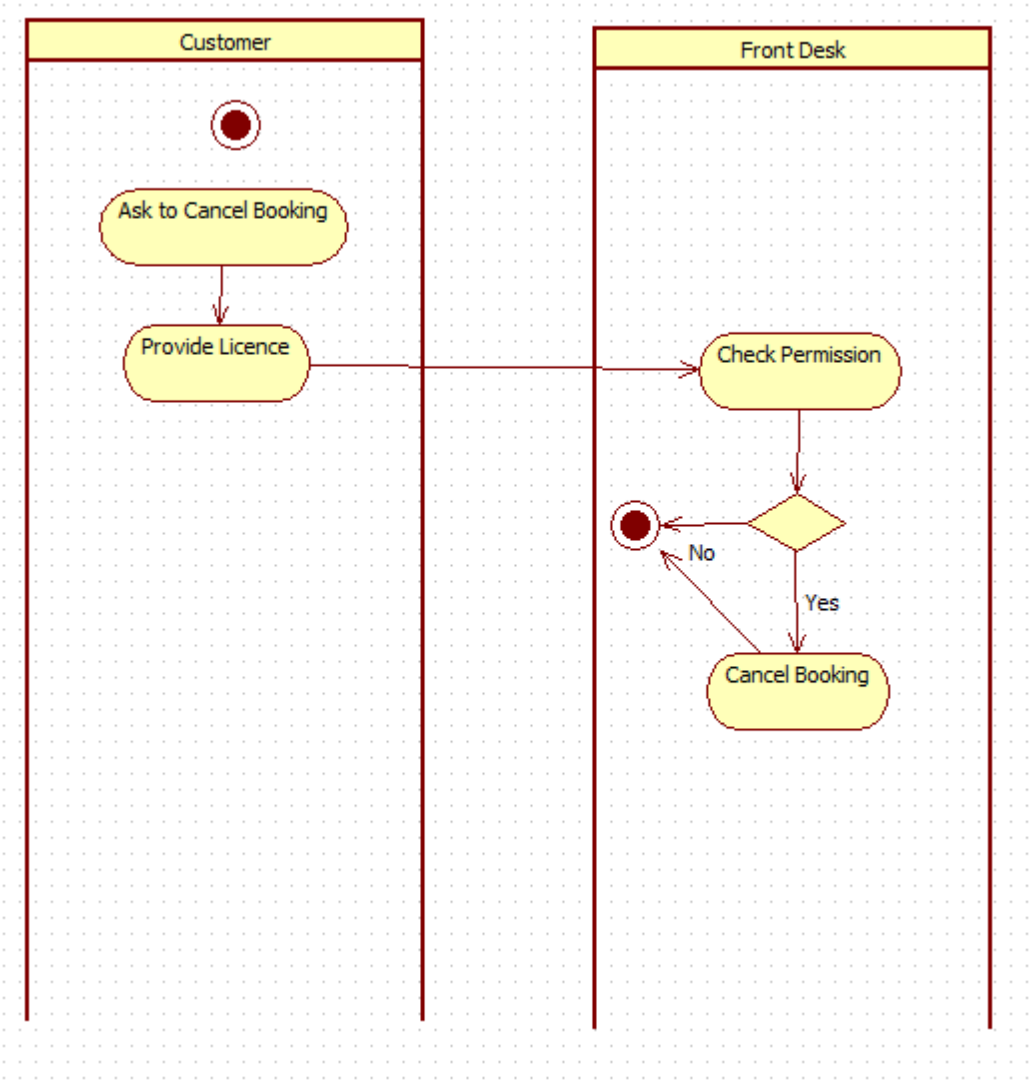
We use this structured text as the basis for the activity diagram that follows.

For cancelling a booking, the process is much simpler – the customer asks to cancel the booking and provides their licence. The front-desk checks the licence against the details in the booking, and if they match the booking is cancelled. Similarly for viewing the details of a booking, we make sure that the customer has permission to view the booking (using the licence) and then we either permit or deny the access accordingly. This leads us to the following diagrams:

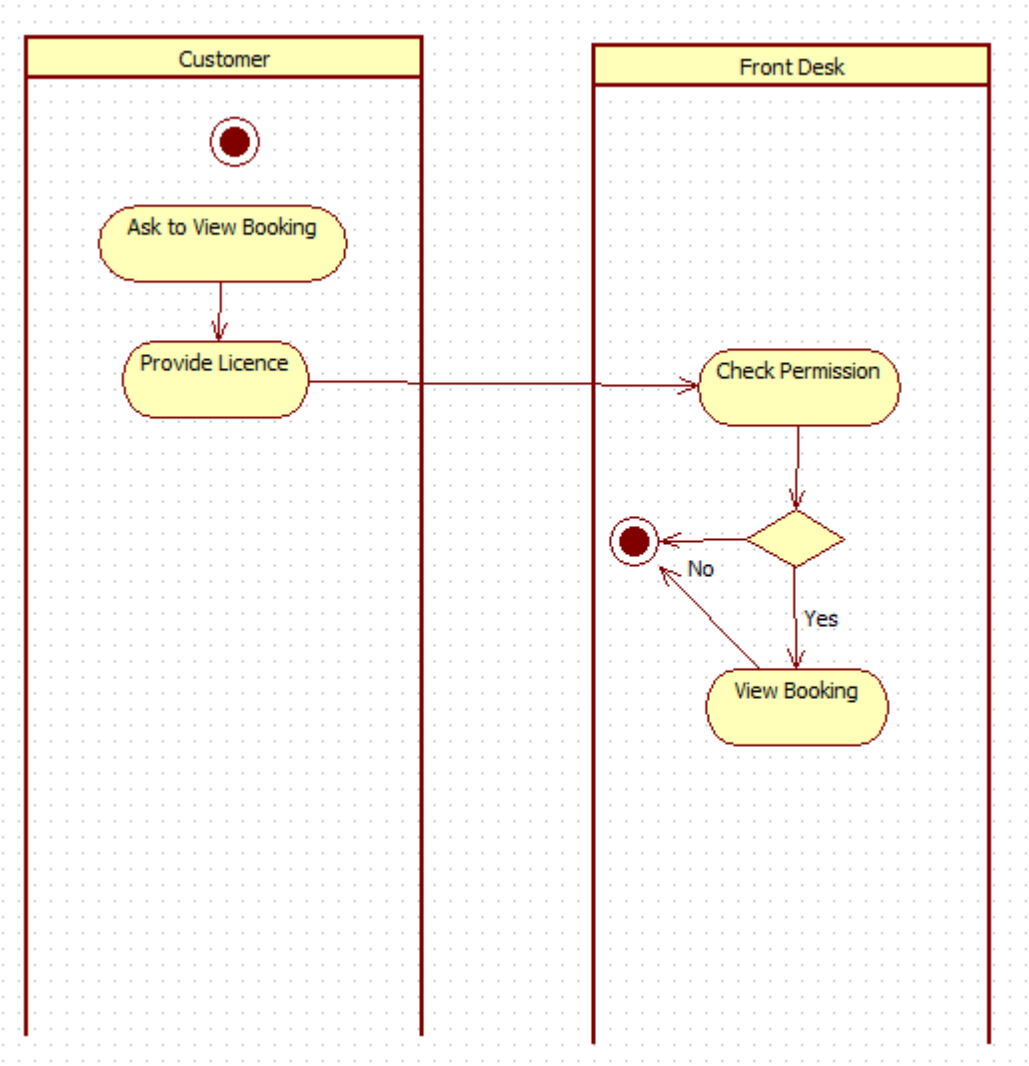
Process Activity Diagram for Booking a Van



Process Activity Diagram for Cancelling a Booking



Process Activity Diagram for Viewing a Booking



Designing the System

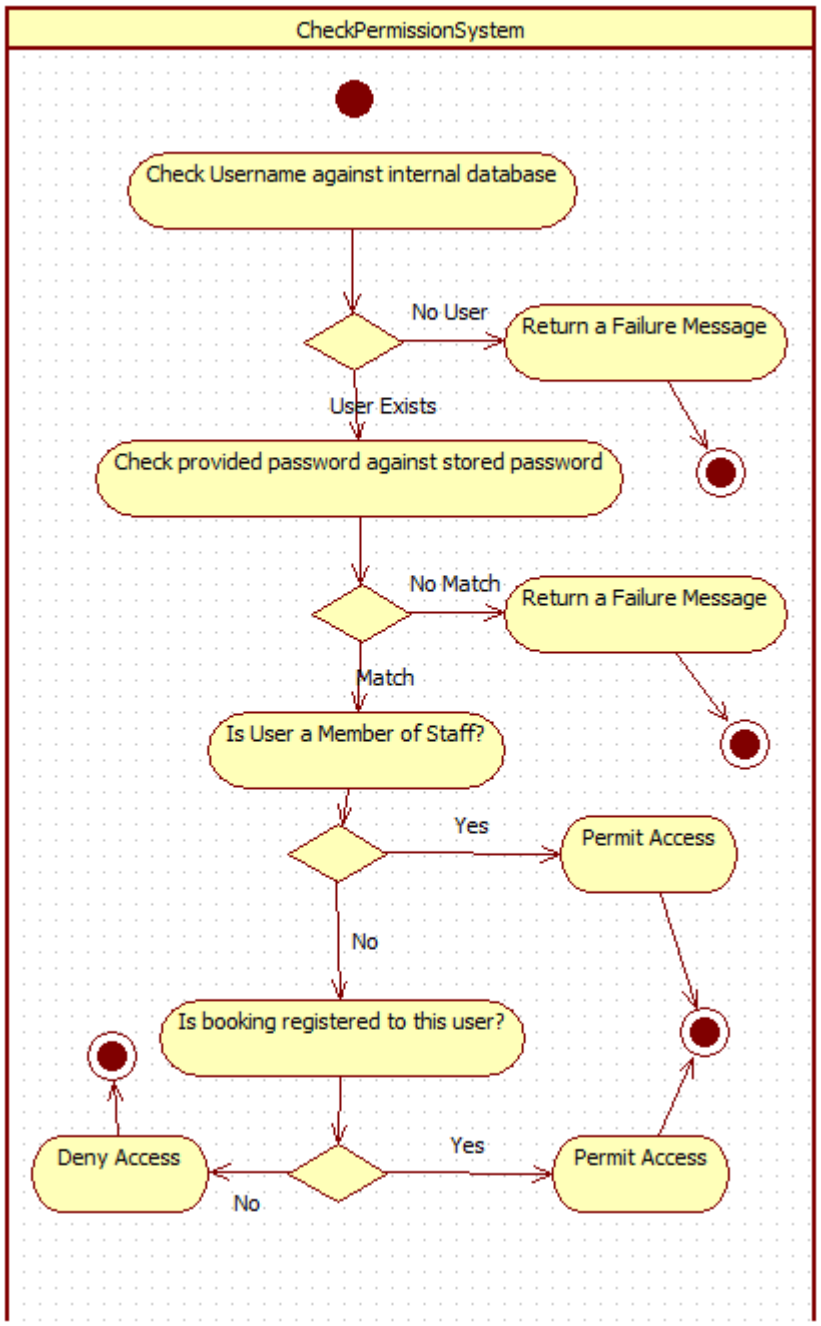
These diagrams give us our broad level understanding of how the processes in the system work – the next step is deciding how we are to implement this in our design. In the design phase we are looking to ensure that the details of the process are kept, but those elements that can be improved **are** improved. We also have to ensure that we compensate for those things that we cannot get access to through a remote interface (like the driving licence). We also have to put in place activity diagrams that allow us to handle those things that would be handled by real people at the front desk (like checking the permission of an individual). We should be thinking of these processes more like functions at this point. Let's begin with the process of checking permission. We won't have access to a driver's licence on the internet, so instead we require people to login using a username and password for which they register.

Once we have a username and password as part of our system, we can implement the structured English for handling permissions:

- Check username exists
- If it doesn't, indicate failure

- Check provided password against stored password
- If they don't match, indicate failure.
- If user is a member of staff, permit access.
- Otherwise if the booking details are registered to this username, permit access.
- Otherwise, deny access.

That there are security implications with permitting all members of staff to do anything to any booking is self-evident, but these can be explored in the classroom during the lecture, seminar or tutorial if it is appropriate. The logic above provides us with the following activity diagram:



This then gives us the structured diagram we can follow to replace the previously human task of checking for permission. While we're doing our design, we must be mindful of the implementation context that will follow. We want to make the logic as simple as possible because that will be the most straightforward for us to implement in code. This brings us back to the flow of logic for making a booking. We can profitably break that into two activities, booking a van and booking a driver. That gives us two structured English flows:

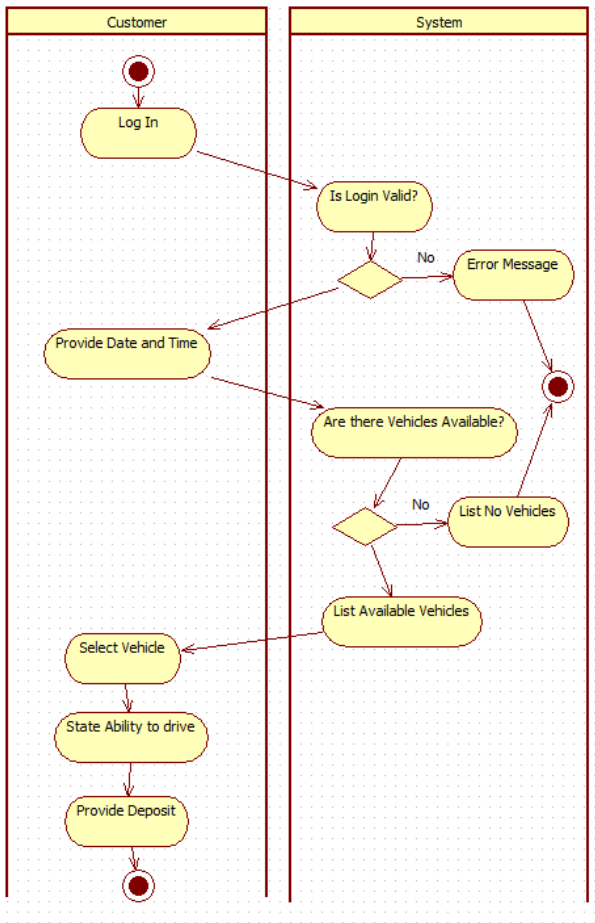
Booking a Driver

- Log in
- Provide a date and time
    - If there are vehicles and drivers free, list available vehicles.
    - Select vehicle
    - Provide Deposit

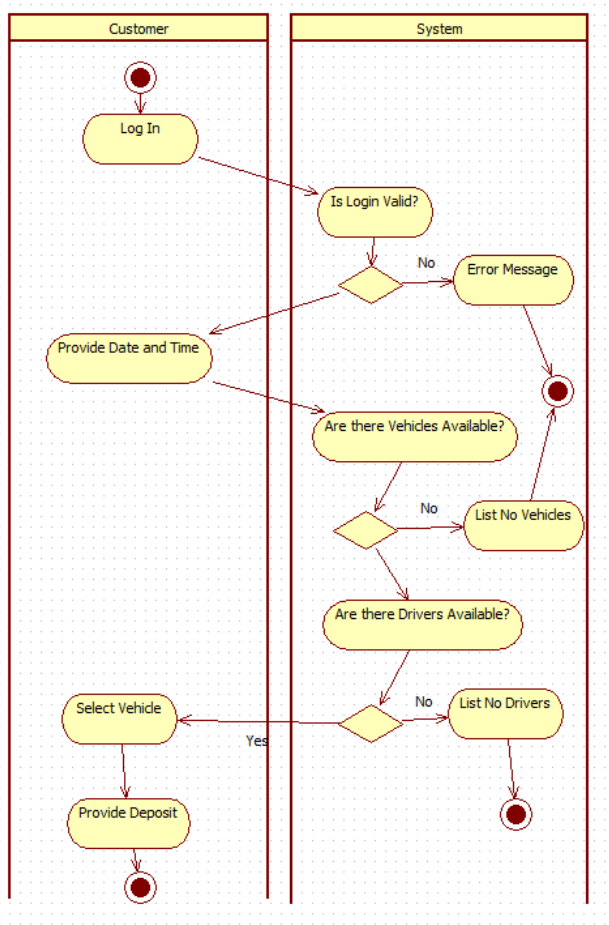
Booking a Vehicle

- Log in
- Provide a date and time
    - If there are vehicles free, list vehicles
    - Select Vehicle
    - State authorisation to drive selected vehicle.
    - Provide deposit
    - Show licence on collection

In doing this, we greatly simplify the flow of logic through the system. Design is about more than turning an existing system into code - it's about doing it in such a way that will lead to a well engineered and structured program that people can work within and future developers can maintain. The impact that this makes on the activity diagrams is shown below:



Booking a Vehicle



Booking a Driver

6.6 Private Study

The time allocation for private study in this topic is expected to be 7.5 hours.

Lecturer's Notes:

Students have copies of the private study exercises in the Student Guide. Answers are not provided in their guide.

Exercise 1:

If you did not complete the work from the practical session, finish these tasks during your private study time. Remember that the tool you need is open source and can be freely downloaded.

Exercise 2:

The worked example in the seminar and lecture did not fully explore the scenario. In your own time, work up activity diagrams for the following processes:

- Logging In
- Identifying if any vehicles are free on a specific date
- Identifying if any drivers are free on a specific date
- Registering a booking

Exercise 3:

Review the lecture material and ensure that you are comfortable with everything discussed thus far.

6.7 Tutorial Notes

The time allowance for tutorials in this topic is 1 hour.

Lecturers' Notes:

Students have copies of the tutorial activities in the Student Guide. Answers are not provided in their guide.

During this tutorial, you should discuss the worked solution to the scenario, providing students with copies of the diagrams and worked analyses that have been provided under 'Suggested Answers' in the Laboratory Session section above. This solution will differ from that of the students, but it will serve as a baseline for the discussion and you can profitably incorporate student feedback into your discussion.

If there is time left at the end of the seminar, discuss the diagrams students put together during their private study.

Exercise 1: Overview of Scenario Solution

Discuss as a class the solution to the exercise that your lecturer will provide.

Exercise 2: Reporting Back to the Class

As a result of the work you will have done for your private study, you should have activity diagrams for each of the four processes listed. Be prepared to discuss your diagrams with others in your class, and to help your peers refine and correct the diagrams they have themselves developed.

Topic 7: Design Patterns 1

7.1 Learning Objectives

This topic provides an overview of creational design patterns. On completion of the topic, students will be able to:

- Understand the use of design patterns;
- Design and use factory design patterns;
- Design and use abstract factory design patterns.

7.2 Pedagogic Approach

Information will be transmitted to the students during the lectures. They will then practise the skills during the laboratory sessions. Tutorials are then used to consolidate students' understanding of the concepts covered and deal with any questions.

7.3 Timings

Lectures:	2 hours
Laboratory Sessions	2 hours
Private Study:	7.5 hours
Tutorials:	1 hour

7.4 Lecture Notes

The following is an outline of the material to be covered during the lecture time. Please also refer to the slides.

The structure of this topic is as follows:

- The need for design patterns
- The factory design pattern
- The abstract factory design pattern

7.4.1 Guidance on the Use of the Slides

Slide 2: In this lecture we are going to begin our exploration of implementation issues in software development. Design patterns are a tool that sits between design and implementation. They help us to create software that is maintainable and flexible enough to meet our immediate needs as developers and the future requirements of those who must work to extend and support the software systems that we develop.

Slide 3: One of the useful aspects of analysis and design is that it allows us to leave difficult implementation decisions until we have fully understood the nature of the problem. This means that we can have a high level view of how the system works before we begin to consider how the coding will need to be done. This high level understanding feeds into our knowledge of design patterns, because these patterns are designed to help resolve many of the complex situations that emerge.

Slide 4: Design patterns offer many benefits over simply working out our own solution. It's important to realise that design patterns are not code themselves – they are a high level, abstract description of the interrelationship of classes and objects. Developing good object-oriented solutions can be difficult because of the various competing constraints with regards to elements such as coupling and cohesion. Design patterns give a 'good' (although usually not the 'best') solution for particular problems.

Slide 5: In addition, design patterns don't enter the common developer vocabulary until they have been tried in many real life situations. They are battle tested so that their deficiencies can been identified and repaired. It is also important to realise that design patterns are general solutions. They are not simply code samples that can be dropped into a program; they are ways of approaching problems. We need to write the code ourselves, but the pattern gives us a framework to do it.

Slide 6: Design patterns are usually documented as the interactions between several classes. Not all design patterns are multi-class (the singleton for example is not), but most design patterns are a high level way to document interrelating functionality between classes. Again, we stress here that design patterns don't give you the solution to a problem; they just represent a generalised way of thinking about how to solve certain recurring issues in software development.

Slide 7: As with UML, design patterns give us a common vocabulary through which we can discuss engineering solutions to common problems. These problems are such a part and parcel of software development, and their solutions more complex than can be expressed in a simple algorithm, that we use the shorthand jargon of design patterns to discuss their application. 'Use a factory to do this' is high-signal method

by which we can indicate what the solution to a problem is. Design patterns represent the best practise of software developers – they have been developed over decades as programmers have tried to write solutions to problems and then iteratively improved those solutions. In short, they are a way to get the benefit of years of continual development without having to actually develop continually.

Slide 8: Design patterns fit easily into modern software development routines in a way that algorithms do not. They are naturally expressed in object-oriented terms, often using UML to fully explore the interaction of each part. Design patterns also reduce the maintenance burden in future. We know the design patterns work because they have been tested, and they have been refactored constantly over years until they have entered the common vocabulary of software developers.

Slide 9: However, patterns are not flawless – they are often workarounds for things that some languages support as part of their syntax. As such, they may be more or less applicable for particular languages. Novice pattern users also tend to use patterns whenever they can, and this can lead to systems that are heavily over-engineered. Most design patterns come with a cost in terms of overall system complexity – a pattern will usually increase the class count in a system by at least one, and often will increase it by many more. Design patterns must be used with care. They should only be implemented when there is a gain to be had, and not just because one **could** be used.

Slide 10: Additionally, it is tempting for novices to use patterns rather than properly analyse the needs of a system. Effective analysis is always more important than using an 'off the shelf' solution. Finally, because patterns come complete with implementation problems already solved, they can prevent individuals from building expertise. Reliance on design patterns can frustrate a developer's ability to understand **why** they work.

Slide 11: Design patterns fall into three broad categories – these are not fixed and strict, but they serve to give the main intention of the pattern. Structural patterns are used to simplify connections between classes, especially in terms of allowing them to interrelate more flexibly when a program is running. Creational patterns are used when dealing with the instantiation and configuration of many objects over the course of a program's lifetime. Behavioural patterns are used to change the way in which objects work as a result of the context in which they are operating. The factory and abstract factory patterns we will see in this lecture are examples of creational patterns.

Slide 12: Creational patterns allow us to deal with several realities of software development. Instantiation of objects is often more complicated than the 'new' keyword will allow, especially when objects must be instantiated with regards to the context the user is operating within. Instantiation may be a complex affair, and the more complex it is the more important it becomes that the instantiation of an object is handled somewhere central. Sometimes the creational needs of our program dictate that we only have a single object instantiated (for example, if we have an object that stores the state of our data, we only ever want one instance of that class to be in use at any time).

Slide 13: The factory pattern is used to provide a consistent way for objects to be requested and configured. Rather than using the 'new' keyword throughout our code, we keep its use to a single class that is responsible for creating an object and configuring it on the basis of data we provide. In most cases, a factory is represented by only a single class that contains a single static method.

Slide 14: This slide shows the class diagram for a simple Shape class. This is the abstract class, and will be specialised as we see in Slide 15.

Slide 15: The base Shape class is specialised into Circle, Rectangle and Triangle. This is important for how the factory works.

Slide 16: The scenario here is for a tremendously simple drawing package – the user selects a shape, clicks on the screen, and then the application draws the selected shape at the selected location. This does not require us to use a design pattern, but a design pattern tremendously simplifies the necessary logic. We could hard code all of the possible combinations of shapes into our system, but the more shapes we have, the more difficult it becomes. A factory on the other hand can be used to let us separate the logic a little, simplify our code, and improve the readability of our programs.

Slide 17: The maze of 'if' statements we would have to use would lead to complex flows of data through our project. The simpler the flow of data is, the easier a system is to code and the more maintainable it becomes. As such, we use a factory to accomplish the goal of letting us request particular shapes. The class structure we use becomes the replacement for a complex mesh of if statements, each of which have to handle the logic for drawing the shape. Polymorphism is the technique that drives this – the general case of Shape means that our factory can instantiate the appropriate specialisation, but the rest of the program can make use of the most general case.

Slide 18: This shows the code for a simple factory. The code here is not complex, and that is the case for most design patterns. It is the idea that is important, not the code structures. In this case, we have greatly simplified our logic by having a method that allows us to simply pass in the details of what we want, and then we get the appropriate specialised object back. We don't need to handle the logic for drawing the shape in our main program, because each specialisation of the shape class can have its own specialised version of the drawShape method. Our flows of logic are greatly optimised, and maintainability is increased by locating functionality where it should belong.

Slide 19: The more options and more configuration choices that are handled in a program, the more complex the logic becomes. The main benefit of a factory is that it reduces this hard-coded complexity. If we want to add in a new shape, all we must do is add in a new class to handle it, and then provide an appropriate case for it in the factory. If we want to make sure all objects have a particular method called on them before we use them (such as in setDrawingColour on Slide 18), that is functionality we get for almost free – we change it one place and we don't need to change it anywhere else. The factory is a tremendously flexible design pattern, but sometimes situations are more complicated and we need a factory **for** factories.

Slide 20: The next layer of abstraction is the **abstract factory**, and that can be thought of as a factory **for** factories. We use this design pattern in the same way as we use a factory – we say what kind of factory we want out of it, and then when we get the factory we say what kind of object we want.

Slide 21: We could use a simple factory to handle the examples on Slide 20, but this introduces the same hard-coded complexity problems that come from not using a factory at all. Hard-coding combinations of things within a program is usually bad design, and bad design has a habit of causing problems later on. We should

always ensure that 'production' code is properly engineered because we may very well be the ones who must maintain it in the future.

Slide 22: This slide shows the problems that come with using a factory to create a graphical interface that can be selected from different themes. We start having to nest our structures, and the more choices we have the more they must be nested. This is going to cause problems later, especially if we want to then add in perhaps a third option that needs to be supported.

Slide 23: We can use an abstract factory to greatly improve the design of this system. The abstract factory produces only a factory object – one that returns properly configured objects for its own specific theme. We need a slightly more complex class structure to support this (thus the warning about over-engineering on Slide 9).

Slide 24: This shows the classes that would be required in order to implement an abstract factory. Everything stems from a single base Factory class, and each one is outputting a GUI Widget object. GUI Widget objects are specialised into swing, Windows and Macintosh widgets, and further specialised to be the actual GUI component they represent.

Slide 25: This slide shows the code for an abstract factory – in terms of functionality it is identical to the factory we have seen before. The only difference is how we treat the factory that comes out of the process – it's an intermediate object that we then call upon to give us the component we wanted.

Slide 26: This shows a simple implementation for one of the factories, and the abstract class from which those factories would extend. Polymorphism is again important here, and we need to be sure that each factory will expose the same interface.

Slide 27: This slide shows how the two interact to give a two layer system of abstraction. We don't need to hard code the new statement anywhere into our program outside of the factory, we simply request the factory we want, and then use that factory to generate the component we want. At the cost of a few extra classes, we have greatly reduced the internal logical complexity of the code we write. The end result is code that is tighter, more maintainable, and more readable.

Slide 28: Patterns link in partially to the design phase of our projects. When we are analysing the systems, we will encounter situations that will lend themselves well to a pattern. When we are designing our systems, we will be able to incorporate the patterns we are going to use into the class structures that we develop. They don't emerge from NLA or use case analysis, they are inserted by us as a response to certain implementation requirements that will occur in development.

Slide 29: Working out when these patterns can be used is unfortunately not straightforward. As with much in software development it depends very much on our own abilities to analyse and generalise patterns. In order to build that ability, we must experiment with patterns and see where they have been used in the past. It is often difficult to look at a system and instantly realise which pattern will be appropriate, and often for novice programmers the consequences of bad design won't be appreciated until the code has been written. Nonetheless, all developers should be actively looking for opportunities to increase the robustness of their code. Simply going through the list of patterns known and checking them off against required functionality identified in the analysis can be a useful technique for this.

Slide 30: Conclusion

7.5 Laboratory Sessions

The laboratory time allocation for this topic is 2 hours.

Lecturers' Notes:

Students have copies of the laboratory exercises in the Student Guide. Answers are not provided in their guide.

Activity 1:

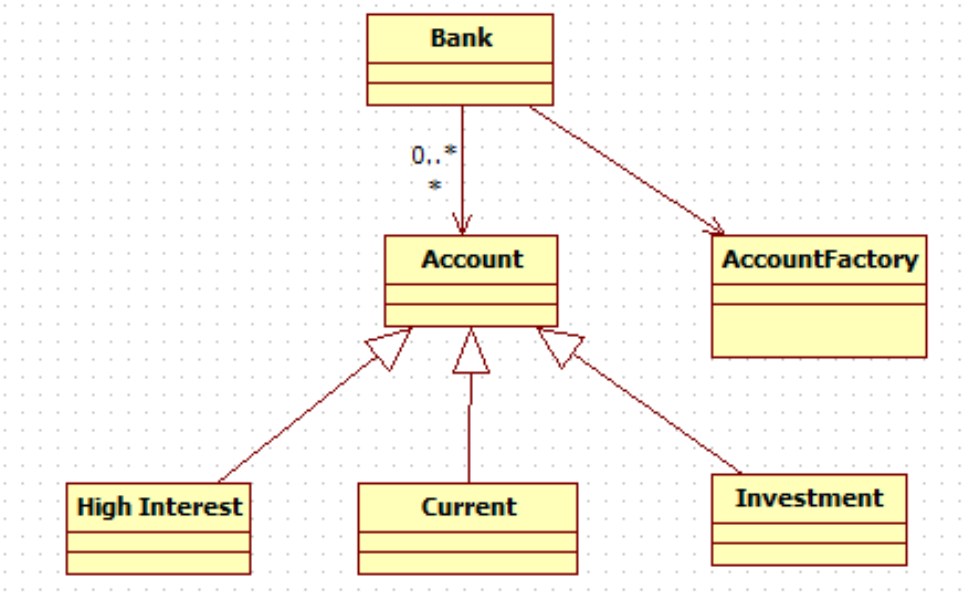
Consider the factory and abstract factory patterns you encountered during the lecture. Draw the class diagram for a factory that supports a bank account system containing three kinds of account. Each customer in the bank should have an account, and the bank should be able to track a number of these at the same time.

The following account types should be supported:

- A High interest account which has a 10% interest rate provided no withdrawals are made. If a withdrawal is made, the interest rate is reduced to 2% for a month.
- A current account with a 3% interest rate but no penalties on withdrawals.
- An investment account which allows deposits only, with a 15% interest rate.

Suggested Answer:

The class structure students should come up with will look like this (attributes and operations are omitted):



Activity 2:

Write the code for your factory as outlined above. You should write a main program that incorporates at least two accounts of each type, and sets them up with random starting values. The

program should allow you to withdraw and deposit from any account. Your front end should also include a way for the user to move the date onwards a month and apply all interest on accounts.

Suggested Answer:

Only the code for the factory and the necessary infrastructure are provided here:

```
abstract class Account {
      private int balance;
      private double interestRate;

      public Account(int bal) {
            balance = bal;
      }

      public boolean deposit (int val) {
            balance += val;
            return true;
      }

      public boolean withdraw (int val) {
            balance -= val;
            return true;
      }

      public int getBalance() {
            return balance;
      }

         public void setBalance (int b) {
             balance = b;
         }

      public void setInterest (double i) {
            interestRate = i;
      }

      public double getInterest() {
            return interestRate;
      }

      public void applyInterest() {
            interestRate = getBalance() * getInterest();
      }

      abstract public void monthlyUpkeep();
}
```

```
public class HighInterest extends Account {
      public HighInterest(int bal) {
            super(bal);
            setInterest (0.1);
      }

      public boolean withdraw (int val) {
            setInterest (0.02);

            return super.withdraw (val);
      }

      public void monthlyUpkeep() {
            setInterest (0.1);
      }
}
public class CurrentAccount extends Account {
      public CurrentAccount(int bal) {
            super(bal);
            setInterest (0.03);
      }

      public void monthlyUpkeep() {
      }
}

public class InvestmentAccount extends Account {
      public InvestmentAccount(int bal) {
            super(bal);
            setInterest (0.15);
      }

      public boolean withdraw (int val) {
            return false;
      }

        public void monthlyUpkeep() {
      }
}
```

```
public class AccountFactory {

    public static Account setupAccount(String type, int amount) {
        Account acc;

        if (type.equals("current")) {
            acc = new CurrentAccount(amount);
        } else if (type.equals("investment")) {
            acc = new InvestmentAccount(amount);
        } else {
            acc = new HighInterest(amount);
        }

        acc.setBalance(amount);

        return acc;
    }
}
```

7.6 Private Study

The time allocation for private study in this topic is expected to be 7.5 hours.

Lecturer's Notes: Students have copies of the private study exercises in the Student Guide. Answers are not provided in their guide.

Exercise 1:

If you did not complete the work from the practical session, finish these tasks during your private study time. Remember that the tool you need is open source and can be freely downloaded.

Exercise 2:

As part of your ongoing journal exercise, you should research the following topics:

- Design patterns
- Applications of the factory design pattern
- Applications of the abstract factory design pattern

Exercise 3:

Making use of the factory that you developed during the laboratory session, expand this to incorporate code to handle an overdraft limit, and an abstract factory that allows you to generate accounts for particular banks. You should have two banks, and they should both have a different overdraft limit.

Exercise 4:

Prepare a short, five minute presentation on the results of your research for Exercise 2 above. If you have found out anything particularly interesting, you should focus on that as a priority.

Exercise 5:

Review the lecture material and ensure that you are comfortable with everything discussed thus far

7.7 Tutorial Notes

The time allowance for tutorials in this topic is 1 hour.

Lecturers' Notes:

Students have copies of the tutorial activities in the Student Guide. Answers are not provided in their guide.

This tutorial is designed to give students a chance to present the more interesting of their findings from the previous exercise, partially as a way to share information with others in the class but also as a way to ensure that they are indeed keeping a journal and researching the topics provided. You should encourage students to bring along their journals to these sessions and make notes on any points of interest that are raised by their classmates.

At this stage of the module, you should also introduce the assessed assignment to students. Assignments for the relevant assessment cycle are available from the NCC Education Campus (http://campus.nccedu.com). You will need to ensure that each student has a copy of the assignment and understands the requirements. Assignments would normally be submitted for marking during Topic 9 or 10, depending on how much time you feel you need for marking.

Exercise 1: Abstract Factory

Discuss as a class the solution to the abstract factory exercise outlined as Exercise 3 in your private study session.

Suggested Answer:

The following shows an implementation of the exercise, using an abstract factory to generate an account factory:

```
public class AbstractFactory {
    public static AccountFactory getBank (String bank) {
        AccountFactory af;

        if (bank.equals ("bank 1")) {
            af = new BankOne();
        }
        else {
            af = new BankTwo();
        }

        return af;
    }

}
```

```
public class AccountFactory extends Bank{

    public Account setupAccount(String type, int amount) {
        Account acc;

        if (type.equals("current")) {
            acc = new CurrentAccount(amount);
        } else if (type.equals("investment")) {
            acc = new InvestmentAccount(amount);
        } else {
            acc = new HighInterest(amount);
        }

        acc.setBalance(amount);
        acc.setOverdraft(getOverdraft());
        return acc;
    }
}

public class Bank {
    private int defaultOverdraft;

    void setOverdraft (int o) {
        defaultOverdraft = o;
    }

    int getOverdraft() {
        return defaultOverdraft;
    }
}
```

```
public class BankOne extends AccountFactory {
    public BankOne() {
        super();
        setOverdraft(1000);
    }
}

public class BankTwo extends AccountFactory {
    public BankTwo() {
        super();
        setOverdraft(2000);
    }
}
```

Exercise 2: Reporting Back to the Class

As a result of the research you did during your private study time, you should have a short five minute presentation ready to give to the rest of the class. There is no need for this to be especially formal - you are simply reporting on anything interesting that you found during your research, or pointing out especially useful resources on the topic. Bring your journal along to the class so that you can make a note of anything especially useful that your classmates mention. This is a knowledge dissemination exercise; you are not being formally assessed on the style or content of the presentation.

Topic 8: Design Patterns 2

8.1 Learning Objectives

This topic provides an overview of the MVC, the Facade, the Strategy and the Flyweight. On completion of the topic, students will be able to:

- Make use of the Model-View-Controller design pattern;
- Make use of the facade design pattern;
- Make use of the strategy design pattern;
- Make use of the flyweight design pattern.

8.2 Pedagogic Approach

Information will be transmitted to the students during the lectures. They will then practise the skills during the laboratory sessions. Tutorials are then used to consolidate students' understanding of the concepts covered and deal with any questions.

8.3 Timings

Lectures:	2 hours
Laboratory Sessions	2 hours
Private Study:	7.5 hours
Tutorials:	1 hour

8.4 Lecture Notes

The following is an outline of the material to be covered during the lecture time. Please also refer to the slides.

The structure of this topic is as follows:

- The Model-View-Controller design pattern
- The Facade design pattern
- The Strategy design pattern
- The Flyweight design pattern

8.4.1 Guidance on the Use of the Slides

Slide 2: Knowing how and when to use patterns is largely based on knowing the range of them that are available. In this lecture, we introduce four new patterns, all of which are useful in a wide range of circumstances. However, there are many dozens of other design patterns in common use, and we cannot spend the entire module exploring them. Students should be encouraged to research those patterns that we do not have the time within the module to discuss.

Slide 3: The first two patterns we will look at are structural patterns, and these define the relationship between classes in a design. All structural patterns derive from two basic guidelines: the first is to identify those parts in a program that are variable and likely to change, then, rather than hard-coding that variability, to create a class that represents it. We can then make use of dynamic binding when the program is running to create interesting relationships between objects. Sometimes the variability is during the development (maintainability is an aspect of this) and as such we identify those parts of the system that are likely to be changed as time goes by and separate them out for our later convenience.

Slide 4: The Model View Controller (MVC) pattern is a structural pattern aimed at separating systems out into the three key roles they all must fill. We have avoided discussing user interfaces so far because we have been waiting to introduce this pattern. The application of this pattern has important implications for how we write software, and we will discuss that when it becomes time to implement the design of our case study. When novice developers write software, they often make no distinction between presentation of data and manipulating that data – this tightly binds the functionality to the context, which has numerous implications for development. The primary problem is that it becomes difficult to change either part without changing the other. This kind of tight binding also creates problems with assumptions – if working within a text-based interface you may assume all manipulation of data is to be done textually, which would create problems if you had to represent them graphically later.

Slide 5: The MVC design pattern resolves this problem by cleanly separating systems out into three roles. The model covers what we'd traditionally think of as the 'business logic' – those bits of the program that manipulate the data in the system. The view handles the presentation of the data to the user, and the controller handles user input.

Slide 6: In most simple programs (and the ones we look at during this module) the view and controller are combined into a single class so as to avoid the issue of over-engineering. Many real world programs however will have separate classes for each of these roles.

Slide 7: The model handles the state of the system and the functionality that acts on that state. It does nothing relating to user input – it exposes a set of functions, and the view/controller will make use of those functions to manipulate the system. The model should be written in as general a way a possible so as to ensure it can be used for all kinds of different potential user interfaces. You should never store a list of strings as a combobox GUI object for example, because that becomes difficult to convert into an array of strings for a text display.

Slide 8: The view handles the presentation. It consists of the user interface, and the exact makeup of that interface will differ depending on how it is to be presented. The only code present in the view is that code needed to turn general data representation into a more specialised version. Taking an array of strings and populating a combo box with those strings would be an example of such code – it is code that doesn't relate to the state of the data, just how that state is represented.

Slide 9: The controller handles the user connection between the view and the model. In a GUI front-end, it would be represented by the event-handlers attached to components. As a matter of convenience, the event handlers are usually implemented directly in the class that makes use of them, but for a proper MVC application they would be stored in classes of their own and those classes would be instantiated as event handlers in the presentation.

Slide 10: Decoupling is important for several reasons. One is that it makes it much easier to run development in parallel – nobody has to wait for another team to finish before they can make their contribution. All that must be agreed upon is the interface between different parts of the system, and that is something that would be documented in the class diagrams and the sequence diagrams.

Slide 11: The most important benefit though is that it allows flexibility of both deployment and maintenance. If you want to deploy your text-based command line tool as a GUI application all you must do is implement a new View and it can be converted across with minimum difficulty.

Slide 12: One of the consequences of object-oriented development is that systems can contain many different classes, and often these classes must interact in complex ways to create the appropriate connections between them. The more classes in a system, the harder it is to work within. The facade design pattern is used to help manage that complexity by creating a new, simpler class that has 'helper' methods. It sits between the view and the model, and gives a set of methods that handle common, complex tasks. This however comes at a cost of cohesion (facades, because of the fact they must offer a diverse range of functionality, are not cohesive classes). Similarly, because they must link together many classes, they are also highly coupled.

Slide 13: However, the benefits that can come from this are considerable. Facades make software libraries much easier to use – instead of learning how forty or fifty classes interact, you can focus only on the facade. Code becomes more readable because there is a tighter link between the intention of the code and the statements required to make that happen. Facades can hide implementation details, allowing for

greater maintainability (in much the same way that the OO principle of data hiding does). It can also work to 'fix' badly designed code – when working with a software API that is difficult, you can write your own facade to ease your development burden.

Slide 14: This slide shows a code example of a facade. The lower class shows the interaction between three separate classes – we instantiate one from a string, then use the object that we get there as part of the constructor of a second object. This second object returns an instance of the third object, which is the one we actually want. Complex layering of objects like this are a common aspect of object-oriented programs (for example, creating a PrintWriter object in Java), and so the class above makes use of the facade rather than hard-coding this relationship each time the object is required.

Slide 15: Facades are most powerful when they are used in a consistent way – the more code that goes through the facade, the greater the 'multiplier' on development. A facade can be used by multiple classes, and the more classes using it the more impact changes to the facade will have. Making a 10% efficiency improvement on one instantiation of a complex object chain will be unlikely to make much difference. If that 10% improvement is demonstrated over fifty different classes, the improvements will start to add up. Facades allow for opportunities for increasing the impact of streamlining such processes as well as simplifying the code needed to make use of a complex API.

Slide 16: However, as with all things a facade has its downsides too. In order to use a facade, you must sacrifice a lot of understanding of how an API works, and that can make things difficult when code malfunctions. By definition, a facade is about simplifying code interfaces, and in order to do so you have to lose the flexibility that comes from setting up the objects yourself. Facades increase the structural complexity of our systems by adding in a new class that does not deliver any actual system functionality, and they have detrimental impact on coupling and cohesion as discussed in the notes for Slide 10.

Slide 17: Both the MVC and the facade are structural patterns. The strategy pattern is a behavioural pattern, and one of the most powerful patterns available. It serves to remove the hard-coding of functions in a class replacing them instead with invocations of methods on objects that we configure at runtime. The code that would belong to a function in a class is instead moved out into an object that contains only that function (or related functions).

Slides 18-20: Here we outline a simple scenario in which a strategy is the only effective solution. The key aspects of the scenario are that each character shares some functionality with other types of character, but not with all of them. This creates a structural difficulty that cannot be resolved effectively with the tools we have available.

Slide 21: Inheritance is not appropriate, because the character types share only some of their functionality. Abstract classes and interfaces will work, but the problem is that we then need to implement the same code in multiple specialisations, and that violates a core principle of good software engineering. We can combine these, doing some of the work with interfaces and some with inheritance, but that creates a complex class structure that is difficult to extend. The Strategy pattern however neatly resolves this problem by allowing us to 'hotswap' functionality and set it up at runtime.

Slide 22: This slide shows the base class we use to implement the solution. Note that performAttack, performDefence and performSpell have no functionality associated – they just call the appropriate method on the object they store. This is the core of the strategy pattern.

Slide 23: Here we see the four different character types set up – they instantiate objects directly into the constructor, and that is all that differentiates them structurally. The effect is that we can 'slot' specific kinds of functionality into an object without having to hard-code any of it.

Slide 24: C# and Java have no facility for multiple inheritance, which is part of the problem that the Strategy pattern solves. However, it is not a simple substitute for a syntactic deficiency – you can change the behaviour of the character types at runtime simply by configuring them with a new object. You could have a wizard that shape shifted into a rogue, without having to worry about complex conditional logic. The cost in terms of code clarity can be considerable, but the benefits are substantial.

Slide 25: Multiple inheritance would not allow for the 'hot swapping' of methods, so even in languages where it is supported the strategy is still very valuable. This particular pattern works beautifully with a state machine, whereby as the state of an object changes the strategy pattern allows for invocations of a method to do the appropriate action. We'll see an example of this in the next lecture.

Slide 26: The last pattern that we look at in this lecture is the flyweight. This pattern helps resolve some of the memory and CPU footprint associated with object-orientation by only instantiating objects when necessary. Objects are computationally expensive to create, and often consume much more memory than we need to get the benefit out of them (because of the need for them to be configurable). In an ideal world, an object would hold only the data we were using. Since we don't live in an ideal world, the best we can do is instantiate objects only when they are required.

Slide 27: This is a scenario in which a flyweight would be useful – every letter in a word processed document can have its own font information to go with it, but even an average page of 500 words (at, let's say, 5 letters per word) requires a potential 2500 objects even though many of them will be identical (you're unlikely to be using 2500 different styles in a single document). It would be much better if we just created one font object for a particular style and when we wanted to change a letter we simply pointed it to the appropriate object we already created. That's what the flyweight pattern is for.

Slide 28: The flyweight works through creating a new object which acts as a cache – when requesting an object, we do it through this flyweight (which may also be a factory). The flyweight cache will check to see if an appropriate object has already been created, and if it has, we return the one we had before rather than a new instance. If there isn't one, we instantiate the appropriate object and then store it in the cache.

Slide 29: This can dramatically reduce the memory footprint of an object because it is much more memory efficient to store a reference to an existing object than to have everything with its own object.

Slide 30: This slide shows the code before and after. The code for the FlyweightCache is not provided since it is simply an existence check (which may be stored in a HashMap or such) – students will be asked to implement this in their lab session.

Slide 31: The pattern itself has no specific implementation assumptions – all that is required is that the basic structure shown on this slide is followed.

Slide 32: Flyweights are limited however in that they only work when there is no need to differentiate between objects. A font object that specifies a certain font and size doesn't care what letter it is attached to, but you couldn't use the pattern for customer objects or for anything that is going to manipulate the object once it has been attached.

Slide 33: Conclusion

8.5 Laboratory Sessions

The laboratory time allocation for this topic is 2 hours.

Lecturers' Notes:

Students have copies of the laboratory exercises in the Student Guide. Answers are not provided in their guide.

Activity 1:

Making use of the flyweight pattern, create a program that handles customers ordering things from a restaurant. The restaurant should have a menu of four starters, four main courses and four desserts as well as four drinks. Each menu item should have a description and a price. Each customer should have a list of their ordered items stored, and a mechanism by which the total price can be calculated.

Suggested Answer:

The following code shows a skeleton implementation of the above:

```
import java.util.*;

public class FlyweightFactory {
    HashMap<String,MenuEntry> items;

    public FlyweightFactory() {
        items = new HashMap<String,MenuEntry>();

    }

    public void addItem (String name, String item, double price) {
        MenuEntry me;

        me = new MenuEntry (item, price);

        items.put (name, me);
    }

    public MenuEntry getItem (String name, String description,
double price) {
        if (items.get (name) == null) {
            addItem (name, description, price);
        }

        return items.get (name);
    }
}
```

```
public class Customer {
    ArrayList<MenuEntry> orders;

    public Customer() {
        orders = new ArrayList<MenuEntry>();
    }

    void addOrder (MenuEntry me) {
        orders.add (me);
        System.out.println ("Bing!");
    }

    double getCost() {
        double total = 0.0;

        for (MenuEntry me : orders) {
            total += me.getPrice();
        }

        return total;
    }

    ArrayList<String> getOrders() {
        ArrayList<String> desc = new ArrayList<String>();

        for (MenuEntry me : orders) {
            desc.add (me.getDescription());
        }

        return desc;

    }
}


public class MenuEntry {
    String description;
    double price;

    public MenuEntry(String d, double p) {
        description = d;
        price = p;
    }

    public double getPrice() {
        return price;
    }

    public String getDescription() {
        return description;
    }
}
```

```
public class MainClass {
    public static void main (String args[]) {
        FlyweightFactory myFact = new FlyweightFactory();
        Customer cust = new Customer();
        MenuEntry me;
        ArrayList<String> desc;

        me = myFact.getItem ("starter 1", "It's a starter", 10.0);

        cust.addOrder (me);

        me = myFact.getItem ("starter 2", "It's another starter",
10.0);

        cust.addOrder (me);

        me = myFact.getItem ("starter 1", "It's a third starter",
10.0);

        cust.addOrder (me);

        desc = cust.getOrders();

        for (String str : desc) {
            System.out.println (str);
        }

        System.out.println ("The total price is " + cust.getCost());

    }
}
```

8.6 Private Study

The time allocation for private study in this topic is expected to be 7.5 hours.

Lecturer's Notes:

Students have copies of the private study exercises in the Student Guide. Answers are not provided in their guide.

Exercise 1:

If you did not complete the work from the practical session, finish these tasks during your private study time.

Exercise 2:

As part of your ongoing journal exercise, you should research the following topics:

- The singleton design pattern
- The chain of responsibility design pattern
- The memento design pattern
- The composite design pattern

Exercise 3:

Implement a facade class for the flyweight program you developed in the laboratory session. This facade should hide the implementation of the flyweight, and provide methods for adding menus, adding orders, and querying the price of an order.

Exercise 4:

Implement the facade class you developed for Exercise 3 as a singleton.

Exercise 5:

Review the lecture material and ensure that you are comfortable with everything discussed thus far.

8.7 Tutorial Notes

The time allowance for tutorials in this topic is 1 hour.

Lecturers' Notes:

Students have copies of the tutorial activities in the Student Guide. Answers are not provided in their guide.

You should also allow time during the tutorial to check that students are working on their assignments and answer any general questions on the expected scope of the work. You may also wish to remind them of the submission deadline and documentation requirements

Exercise 1: Facade

Discuss as a class your solutions to Exercise 3 from the private study session.

Suggested Answer:

The following shows an example implementation of a facade:

```
public class MenuFacade {
    private ArrayList<Customer> myCustomers;
    private FlyweightFactory myFact;
    public MenuFacade() {

        myFact= new FlyweightFactory();
        myCustomers = new ArrayList<Customer>();
    }

    Customer getCustomer (int number) {
        teturn myCustomers.get(number);
    }

    MenuEntry getMenuEntry (String str, String desc, float price) {
        MenuEntry me = myFact.getItem (str, desc, price);
        Return me;
    }

    void addOrderToCustomer (Customer cust, MenuEntry me) {
        cust.addOrder (me);
    }
}
```

Exercise 2: Singleton

Discuss as a class your solutions to Exercise 4 from the private study session:

Suggested Answer:

The following shows an example implementation of a singleton:

```
    public class MenuFacade {
    private ArrayList<Customer> myCustomers;
    private FlyweightFactory myFact;
    private MenuFacade menu;

    // Implements the singleton.
    public static MenuFacade getMenu() {
        if (menu == null) {
            menu = new MenuFacade();
        }

        return menu;

    }
    private MenuFacade() {

        myFact= new FlyweightFactory();
        myCustomers = new ArrayList<Customer>();
    }

    Customer getCustomer (int number) {
        teturn myCustomers.get(number);
    }

    MenuEntry getMenuEntry (String str, String desc, float price) {
        MenuEntry me = myFact.getItem (str, desc, price);
        return me;
    }

    void addOrderToCustomer (Customer cust, MenuEntry me) {
        cust.addOrder (me);
    }
}
```

Topic 9: Elements of Good Design

9.1 Learning Objectives

This topic provides an overview of assessing the quality of software. On completion of the topic, students will be able to:

- Analyse and assess the quality of software;
- Assess the architectural quality of an object-oriented program;
- Make use of the observer data pattern to reduce coupling.

9.2 Pedagogic Approach

Information will be transmitted to the students during the lectures. They will then practise the skills during the laboratory sessions. Tutorials are then used to consolidate students' understanding of the concepts covered and deal with any questions.

9.3 Timings

Lectures:	2 hours
Laboratory Sessions	2 hours
Private Study:	7.5 hours
Tutorials:	1 hour

9.4 Lecture Notes

The following is an outline of the material to be covered during the lecture time. Please also refer to the slides.

The structure of this topic is as follows:

- System measures
- Architectural measures
- Project measures
- Assessing measures
- Software component design

9.4.1 Guidance on the Use of the Slides

Slide 2: Part of what we want to do as software developers is produce the best software we can. Our analysis of a system will outline how things currently work (very few computer programs are written completely independently of an existing context. Even entirely web-based phenomenon like Facebook are heavily based on social interactions in the real world), and as part of our design we should be looking at how we can put together a software system that improves on the existing processes. However, improvement is often difficult to quantify and different software developers will have differing opinions on what constitutes an improvement. There is general agreement though on the value of those things that are discussed during this lecture.

Slide 3: There are many competing taxonomies of software quality attributes – they have overlapping terms, but they all have their unique elements. The qualities that are discussed during this lecture are only a sampling, and students will be encouraged to explore alternatives during their private study. On the whole though, they tend to break down into three main categories – those that are relevant to the system as it executes, those that are relevant to the system as it is designed, and those that are relevant to the administration of the building of the software. The latter will only be touched upon as it is out-with the scope of the module.

Slide 4: System measures relate to the system as it is running. These are the most easily quantified as they can be assessed through existing and bespoke tools. Key amongst these measures are whether the system does what it is supposed to do (a subset of this may be how well the provided functionality meets the required functionality without bloat). The performance of the software is important, although how important will vary depending on the role it is playing. We probably don't care very much if an infrequent operation takes 6 seconds as opposed to 5, but we do care that frequent operations are quick to perform. The security of the running system can be critical in certain circumstances, and the reliability of the system (expressed sometimes as 'system uptime') can be an important measure for mission and safety critical products.

Slide 5: The usability of systems is often an afterthought for developers, but it has a huge impact on how well received a system will be. If the system must interact with other existing subsystems, we need to measure how well it does this, and we need to measure how correct the output is and whether it has the necessary amount of precision to meet user needs.

Slide 6: Architectural measures are those that are most relevant to our design. They relate to how 'good' the software is from the perspective of the infrastructure. The maintainability of the software is a major measure of this, as is the reusability of the code we write.

Slide 7: Testability is a valuable measure, since it will influence how easily we can apply test driven development and benchmarking (discussed later in the lecture). Finally, portability can be a huge issue – modern abstracted languages like Java are highly portable, but even languages that work through a virtual machine will encounter platform specific difficulties in some circumstances. When using a language that is tightly bound to the operating system (such as assembler), portability can be a major consideration.

Slide 8: The project measures are largely outside the scope of this module, but they are relevant since they were part of the reason that the software crisis (addressed all the way back in the first lecture) was such a major issue for the field. Issues of cost and scheduling are still important, although marketability may not be a measure that is valid for those that happen to be developing the software.

Slide 9: Assessing any of these measures is sometimes entirely based on what people say (qualitative feedback) and sometimes measurable (quantitative feedback). Design isn't 'fire and forget' (something you do once and move on); we need to revisit it constantly both during and after implementation, drawing in newly available information to refactor the end product. We can't run test cases until we have functioning software for example, and that may reveal flaws in our logic that were not identified during the design. User testing requires at least some kind of prototype (whether it be paper or actual code), and that will help us identify deficiencies in our use cases or in our assumptions for how the system should work.

Slide 10: In almost all cases, trade-offs are required. In order to get more of one measure, you are most likely going to have to sacrifice another. There is a famous concept of the 'project triangle', whereby you have a triangle with the points marked as 'fast', 'good' and 'cheap'. The punch line is 'Pick any two' – you can have any two of these at the cost of the third. It can be fast and good, but that won't be cheap. It can be cheap and fast, but it won't be good. The project triangle is a simplified version of the decisions that analysts, designers and developers must make on a range of measures. The portability of a solution will impact both on its cost and schedule as well as its efficiency and architectural design. You can't have everything, and so you must decide what you **need** and then work to redesign your system so that you get that.

Slide 11: This is the Project Triangle as described above,

Slide 12: System measures can only usually be assessed when something is actually in place in terms of code. It may not be the full solution, but it should be enough so that you can actually start running evaluations to determine the impact of what has been done and what must be done. Two ways of doing this are discussed in this lecture – test driven development and benchmarking.

Slide 13: Test driven development works by writing tests before you write code. Testing is usually considered to be an 'end on' process, but a good testing framework will identify when new bugs are introduced as a part of development, and test driven development takes advantage of that by incorporating testing into the development loop.

Slide 14: Before you write a piece of software, Test Driven Development (TDD) emphasises that you should write the tests that will determine if your functionality is working. Then you run all the tests in the system (to make sure that the new test cases you have added aren't mysteriously passed before you add the functionality – that would indicate a problem in your program before you even begin to write the code). You then write the code that your test-cases were for, and then run the tests again. If any of them fail, you fix the code and retest until all tests pass. Once you've done this, you can refactor your solution and then go back to the start for the next piece of functionality. At all times, you know exactly how far your system is from being fully working. At this point, you should make sure that students are familiar with the concept of refactoring, and if not explain it to them in broad terms before proceeding. It will be covered in further detail in Topic 11.

Slide 15: Benchmarking is an important aspect of measuring performance. Optimisation is, in general, a noble goal for a software developer but it is often very inexpertly applied. Optimisation often involves a major trade-off in terms of readability and maintainability of code, and as such it should only be applied when an actual performance problem is encountered, and even then only where the problem can be effectively addressed. Sometimes you can use industry standard tests to assess the performance of your code (there exist numerous suites for testing graphical performance, for example), but more often you'll need to write your own benchmarking routines.

Slide 16: There are two main stages to benchmarking – profiling the system to see exactly where the CPU is being used, and then benchmarking the performance of the most costly of these subsystems. The former shows you where you should be directing you attention, and the latter quantifies what difference your optimisation tweaks are making to the performance of the system.

Slide 17: This slide shows the code for a simple benchmarking harness. It is important that students realise this is not a transferable, or even precise measure – there is huge amount of variation when you perform an operation, caused by things such as thread scheduling, processor load, time of day, heat of CPU, and so on. As such, we perform the operation we want to benchmark many times (in this case, we are assessing the cost of declaring and instantiating a string object), and ideally we do this itself several times and then average out to get a rough measure of performance **for the system on which the code was written.** We can then quantify what difference it would make changing parts of the system to be more efficient by comparing these measures. We can't say with certainty that 'the first way took 1000 milliseconds and the latter way took 800 milliseconds, so that is a 20% reduction' because this is an imprecise system. We can however say that the second way was **around** 20% faster.

Slides 18-19: Having identified the parts of the system that are using the most CPU, we can optimise that – but we must always be aware of the 80/20 rule. This is an informal guideline that '80 percent of your processing time is spent in 20% of your code'. You don't get much benefit from optimising the 80% of your code that only 20% of the CPU time is spent in, but you get massively disproportionate benefits by optimising the 20% that handles 80% of your processing. It is to these parts of your system that your efforts should be directed. Several ways for improving the efficiency of code are listed on this slide, although optimisation is as much an art form as it is an engineering process.

Slide 20: One of the problems novices have with object-orientation is that it is very difficult to tell whether one particular OO design is better than another. However, there are

ways in which we can formally assess the quality of our models, and this is done through assessing the level of coupling and the amount of cohesion. Within the context of this topic, when we use the word 'module' we are referring to methods, or classes, or even combinations of classes – the level of abstraction at which we are viewing the system will define what is the most sensible in context. We want modules to have low coupling and high cohesion, but unfortunately to emphasise one is to de-emphasise the other. However, that is not to say we cannot optimise either. Often the coupling or cohesion in a system is simply badly done, and we can gain great improvements by identifying and fixing structural deficiencies.

Slide 21: These slides lists some of the most common coupling types used between modules. They are listed from worst to best. Content coupling would be observed if, for example, we had our attributes in a class set as public and an external object changed their state. Common coupling is when we make use of a shared data store such as a global variable or an object that gets passed around an entire program.

Slide 22: Data coupling is what we traditionally use, whereby we communicate via parameters and return types. Callback coupling is the best kind of coupling, and we'll see how that works later in the lecture.

Slide 23: Coupling is problematic because of its impact on our future development. It becomes hard to maintain programs because if we change one module, we often need to change the others (part of a ripple effect). It greatly reduces the reusability of our classes too, since it becomes difficult to extract a class from the context in which it is located. It is however not always bad to have coupling – a system with no coupling is one in which no part can communicate with another, and that's not possible even if it were desirable. We need coupling, we just need to make sure that it's the right kind and is not over-used.

Slide 24: Cohesion is used to describe how tightly a module is bound to a particular role. All parts of a module should be aimed at solving the same kind of problem – they should be part of a single processing transaction, or be part of managing a single subsystem role. Cohesion makes it easier to understand what classes do because you do not need to understand so much of their context, and this in turns makes them easier to reuse and maintain.

Slide 25-26: As with coupling, cohesion breaks down into different types. These slides list them, again from worst to best.

Slide 27: It is one thing to understand that coupling and cohesion are important measures, and another to turn that knowledge into action. First of all, we need to identify where we have structural problems in our system. To do this we go over each class in turn, profiling its coupling and its cohesion. We then need to identify the nature of the coupling and cohesion and identify which parts are using the 'bad' kinds. We can prevent a number of structural problems simply by making use of encapsulation and data hiding (this is why we insist all class attributes are set to private – it eliminates the possibility of content coupling). When we identify classes that are not cohesive, we can refactor our design to either merge (when two classes are performing a single role) or divide (when a class is performing more than one role).

Slide 28: When coupling is identified, the important thing is to either remove it (if feasible) or make it a 'better' kind of coupling. If we find we are using content coupling, we should refactor our design so that we are instead using at least data coupling, and ideally callback coupling. When doing this though we need to be mindful of the

consequences – data coupling is ‘good enough’ for most purposes, but callback coupling (as we’ll see in a few slides) can be very highly over-engineered if used constantly over an application.

Slide 29: One of the ways in which we can ensure well written programs is to enforce component design. We cluster related classes together into subsystems and separate them away from the other parts. Thus, we may have a ‘vehicle management subsystem’ and a ‘customer management subsystem’, and the two never interact except through callback coupling. It is easier to ensure the correct design of two sets of three classes than it is one set of six classes.

Slide 30: The observer design pattern is one that is used often in software development, especially in event driven programming languages. It works by allowing objects to register themselves as observers of another object – when the state of the latter object changes, it sends a notification to all its observers. In this way, there is no hard-coded coupling between objects, and the nature of the coupling can be changed at runtime.

Slide 31: This slide shows two classes of an example. The first is the interface that all observer objects will have to implement. The second is the Account object that will be observed. Notice how it exposes methods for adding and removing observer objects – each of the observers will manage this part themselves. The notifyListeners method calls the method defined on the interface on each object in turn, allowing them to provide their own implementation for the handling code.

Slide 32: This shows the second part of the system. InterestedParty is an observer object, and the main program shows the attaching and detaching of the observer object onto the Account. This can be done as the program is running, making coupling a fluid and easily altered property of the system. This is the loosest kind of coupling, but the need to have an interface for each element of communication can greatly increase the class count in a system.

Slide 33: Software components allow you to subdivide your system so that each part becomes a more easily addressed problem. Limiting the scope of each component and ensuring that all communication is done via loose coupling can greatly increase the maintainability of an OO design. High quality software is something that you must constantly work at, because every change you make will have an impact throughout the entire project. You are never ‘done’ when creating a high quality software project - refactoring is a constant process.

Slide 34: Conclusion

9.5 Laboratory Sessions

The laboratory time allocation for this topic is 2 hours.

Lecturers' Notes:

Students have copies of the laboratory exercises in the Student Guide. Answers are not provided in their guide.

Activity 1:

Create a benchmarking harness, and benchmark the following operations:

- Appending the word "blue" to a string 100, 1000, and 10000 times
  - First using a string and the append operation
  - Secondly using a StringBuffer
- Calculating the Fibonacci number at positions 10, 20, 30, 40 and 50 in the sequence.
  - Firstly using a loop within a function
  - Secondly using recursion
  - Thirdly using recursion and a cache

Suggested Answer:

Append operation:

```
    public static void main(String[] args) {
              Date time = new Date();
              int iterations[] = {100, 1000, 10000, 100000};
              long now = time.getTime();
              long then;
              double total;
              String str  = "blue";
              for (int j = 0; j < iterations.length; j++) {
                  for (int i = 0; i < iterations[j]; i++) {
                       str += "blue";
                  }
                  time = new Date();
                  then = time.getTime();
                  total = then - now;
                  System.out.println ("Method took " + total + "
milliseconds for " + iterations[j] + " iteration.");

              }
        }
```

For Stringbuilder:

```
public static void main(String[] args) {
            Date time = new Date();
            int iterations[] = {100, 1000, 10000, 100000};
            long now = time.getTime();
            long then;
            double total;
            StringBuilder str  = new StringBuilder ("blue");
            for (int j = 0; j < iterations.length; j++) {
                for (int i = 0; i < iterations[j]; i++) {
                     str.append ("blue");
                }
                time = new Date();
                then = time.getTime();
                total = then - now;
                System.out.println  ("Method  took  "  +  total  +  "
milliseconds for " + iterations[j] + " iteration.");

            }
    }
```

Factorials

```
  public static HashMap<Integer, Integer> cache;

    public static int fibb(int num) {
        if (num <= 2) {
            return 1;
        } else {
            return (fibb(num - 1) + fibb(num - 2));
        }
    }

    public static int fibbCache(int num) {
        int c;
        if (cache.containsKey(num) == true) {
            c = cache.get(num);
            return c;
        }

        if (num <= 2) {
            return 1;
        } else {

            c = (fibb(num - 1) + fibb(num - 2));
            cache.put(num, c);
            return c;
        }
    }
```

```
    public static int fibbLoop(int num) {
        int next = 0, current = 1, old = 0;

        for (int i = 0; i < num - 1; i++) {
            next = current + old;
            old = current;
            current = next;
        }

        return current;
    }

    public static void main(String[] args) {
        cache = new HashMap<Integer, Integer>();
        Date time = new Date();
        int iterations[] = {10, 20, 30, 40, 50};
        long now = time.getTime();
        int fact;
        long then;
        double total;
        for (int j = 0; j < iterations.length; j++) {
            fact = fibbCache(iterations[j]);
            time = new Date();
            then = time.getTime();
            total = then - now;
            System.out.println("Method took " + total + "
milliseconds for " + iterations[j] + " iteration: " + fact);

        }
    }
}
```

Activity 2:

Suggest three ways in which the following function can be optimised. Perform a benchmark for each way in isolation, and then on the function with all your optimisations incorporated:

```
public void doTheThings() {
        int[] sizes = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
        double circumference, half, doub, quart, four;
        for (int i = 0; i < sizes.length; i++) {
            for (int j = 0; j < 1000; j++) {
                circumference = Math.PI * sizes[i];
                doub = Math.PI * sizes[i] * 2;
                four = Math.PI * sizes[i] * 4;
                half = (Math.PI * sizes[i]) / 2;
                quart = (Math.PI * sizes[i]) / 4;

                System.out.println ("Stats for Circle " + j + ": "
                + quart + ", " + half + ", " + circumference + ", "
                + doub + ", " + four);
            }
        }
    }
```

Suggested Answer:

There are several ways in which the efficiency of this can be improved. Some that your students should consider will be:

- Use a StringBuilder to build the output, and then System.out.println the full string at once.
- Move the calculation of the circumference out of the inner loop.
- Base the calculation of each of the values on the circumference.

You should be willing to accept any other suitable efficiency improvements. One view of an optimised solution would be the following:

```
public static void doTheThings() {
    int[] sizes = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    double circumference, half, doub, quart, four;
    StringBuilder str = new StringBuilder();
    for (int i = 0; i < sizes.length; i++) {
        circumference = Math.PI * sizes[i];
        for (int j = 0; j < 1000; j++) {
            doub = circumference * 2;
            four = circumference * 4;
            half = circumference / 2;
            quart = circumference / 4;
```

```
            str.append ("Stats for Circle " + j + ": "
            + quart + ", " + half + ", "
            + circumference + ", " + doub + ", "
            + four);
        }
    }
    System.out.println (str.toString());
```

9.6 Private Study

The time allocation for private study in this topic is expected to be 7.5 hours.

Lecturer's Notes:
Students have copies of the private study exercises in the Student Guide. Answers are not provided in their guide.

Exercise 1:

If you did not complete the work from the practical session, finish the tasks during your private study time.

Exercise 2:

As part of your ongoing journal exercise, you should research the following topics:

- Additional software quality attributes
- Benchmarking
- Test Driven Development
- Coupling and Cohesion

Exercise 3:

Prepare a short, five minute presentation on the results of your research for Exercise 2 above. If you have found out anything particularly interesting, you should focus on that as a priority.

Exercise 4:

Review the lecture material and ensure that you are comfortable with everything discussed thus far.

9.7 Tutorial Notes

The time allowance for tutorials in this topic is 1 hour.

Lecturers' Notes:

Students have copies of the tutorial activities in the Student Guide. Answers are not provided in their guide.

This tutorial is designed to give students a chance to present the more interesting of their findings from the previous exercise, partially as a way to share information with others in the class but also as a way to ensure that they are indeed keeping a journal and researching the topics provided. You should encourage students to bring along their journals to these sessions and make notes on any points of interest that are raised by their classmates.

Exercise 1: Reporting Back to the Class

As a result of the research you did during your private study time, you should have a short five minute presentation ready to give to the rest of the class. There is no need for this to be especially formal - you are simply reporting on anything interesting that you found during your research, or pointing out especially useful resources on the topic. Bring your journal along to the class so that you can make a note of anything especially useful that your classmates mention. This is a knowledge dissemination exercise; you are not being formally assessed on the style or content of the presentation.

Topic 10: Redesign and Implementation

10.1 Learning Objectives

This topic provides an overview of applying software quality attributes to design. On completion of the topic, students will be able to:

- Follow through the process of applying design patterns;
- Implement a solution from a design.

10.2 Pedagogic Approach

Information will be transmitted to the students during the lectures. They will then practise the skills during the laboratory sessions. Tutorials are then used to consolidate students' understanding of the concepts covered and deal with any questions.

10.3 Timings

Lectures:	2 hours
Laboratory Sessions	2 hours
Private Study:	7.5 hours
Tutorials:	1 hour

10.4 Lecture Notes

The following is an outline of the material to be covered during the lecture time. Please also refer to the slides.

The structure of this topic is as follows:

- Redesign of previous case study
- Assessment of design patterns
- Implementation

10.4.1 Guidance on the Use of the Slides

Slide 2: In this lecture we are going to incorporate those elements of design that have been discussed in the previous topics into our case study. You may wish to give students a brief recap of the aims and background of the case study before continuing. Design documents are not a formal contract, and the best practise is that they should evolve as our understanding evolves. As we gain further understanding as to what our system should do, we should adapt our documentation to match. As we learn more about the implications of how we are implementing the design, we should adjust it accordingly.

Slide 3: Refactoring is a process that we will talk about in more depth in the next lecture. In the context of this lecture, refactoring is the process of taking that which exists and improving it without losing any of the functionality that we have already put in place. It's about making this better in terms of design, performance or flexibility. As a process it falls somewhere between design and implementation – we need to know a fair bit about how the system is to be implemented (and ideally already have bits of it in place) before we can properly address it.

Slide 4: This slide shows the class diagram we had at the end of the design process in Topic 6. This is our starting point for this topic.

Slide 5: Our first task is to assess this design – we have talked about how we can rate the architecture of a system, and we apply those metrics to this. We know that we are going to implement connections between elements as data couplings, because that's what we have done in the past. We also know that we do not have too much coupling in our model – for the most part, classes are coupled to only one other instance. We could do better than we have done, but we need to redesign the system and that may result in more cost for less benefit.

Slide 6: Our class diagram does not show the methods and attributes (students should be comfortable at this point with adding those in themselves), but we can see that each class has a well defined role. We can see by the existence of classes like Payroll and Garage that we have a proper separation of responsibility between 'being a thing' and 'being a collection of things' – that's good design. However, we spoke in the last lecture about developing systems as collections of components, and we should assess whether this is appropriate for this particular system.

Slide 7: There is no 'right' answer for redesign, although we should be able to tell when something is **not** right. In some cases, we end up over-engineering a solution in order to get a benefit, and often that benefit does not justify the extra developmental complexity and effort. Deciding on when this is true is a judgement call, but we can

assess the intention of a change in light of what adjustment it makes to our system, and then decide whether or not it is warranted. It is valuable to do this at the design stage since it is easier to change a diagram than it is to alter a program.

Slide 8: Software component design would mean breaking this one system down into three separate systems, each working like an independent black box inside the larger context. This would involve implementing coupling through the observer design pattern, and most likely implementing a facade to act as the public API of the system. For large systems with complex interactions this can greatly reduce the internal complexity of a design because you can apply the principles of divide and conquer – it's much easier to implement a vehicle management system than it is to implement the whole system.

Slide 9: This shows the class design of a simple version of the component solution. We add in three new classes (StaffFacade, CustomerFacade and VehicleFacade) to act as the public API. These classes, by virtue of being facades, are likely to need access to all the classes in their subsystems. The result is a system that introduces low cohesion classes and requires much more intricate coupling. Our system is too small and too simple to see much benefit from software component design at this point, and the cost is too high. For this particular project, it's not an appropriate strategy.

Slide 10: Design patterns require the same assessment process. We need to see how they will adjust our design and whether they give more benefit than cost. For this, we need to decide (for ourselves) how much extra flexibility or convenience is going to be gained by applying them, and whether they are a valuable addition and not just 'added for the sake of being added'.

Slide 11: The model view controller architecture is one that is almost always appropriate – it is very low cost, and very high benefit. So far, we have ignored the role of the front-end in our systems because we couldn't integrate them cleanly. With our discussion of MVC, we can start to include them in our design. We have two front-ends that are required of the system, and so this instantly justifies the use of the MVC.

Slide 12: Here, the class 'Model' is used to stand in for the system as we currently have it. We need to ensure a conceptual separation of these to ensure that the roles are properly differentiated. To give a consistent API for the view/controller classes, we will want to incorporate a facade to act as the entry point to the model.

Slide 13: To use a black box component (like our model) we must ensure that we encapsulate it properly and restrict access to its internals. That gives us maximum flexibility when making changes to how it functions. This is done in this situation by restricting access to the model through a facade. It may seem that the Organisation class would be able to fulfil this role since it is already responsible for linking everything together. Architecturally the setup of the facade is similar, but the difference is in the roles – for high cohesion, we should separate out the architectural requirements of the Organisation class from the API role of the facade.

Slide 14: This shows the class diagram incorporating front-end classes and access through a facade. The cost is in one extra class, and this class is highly coupled (linking to Garage, Organisation, Payroll and Customer), and this creates low cohesion. However, that is the cost we must pay for the benefits we obtain.

Slide 15: Next, we need to work out if there is a role for the other design patterns that we have discussed. Not all will have a role in all projects, but most projects will benefit from at least one (the MVC) and potentially more. We shouldn't implement patterns just because we can, we should implement them when they are appropriate. Here, there seems to be a role for a design pattern in simplifying the creation of different kinds of vehicles, as well as allocating jobs to customers. We should consider a factory each to meet these design goals.

Slide 16: This slows the class diagram incorporating a VehicleFactory and a JobFactory. These classes do not have to be complex – their role is quite simple, but it will allow for easy instantiation and configuration of the objects we need.

Slide 17: We **could** use a strategy pattern to handle the workflow of booking a car versus booking a driver, but the extra class complexity would be considerable for the benefit of simplifying a single if-else. Whether or not this would be a sensible trade-off would depend on future requirements – if, later in the process, we were going to need perhaps twenty possible workflows, we would do well to design for our future requirements. In the case of this project, with the information we have available, it is probably not appropriate.

Slide 18: The flyweight pattern is not appropriate for our project because every object we have will have a context in which it operates. Vehicles will be linked to specific drivers and jobs, customers will have their own information and so too will staff members. We should never feel bad about not including a pattern even though we have a good knowledge of it – it is common for novice developers to want to include every pattern they know, but that is what leads to over-engineered solutions. We must balance between what we need now and what we could **potentially** need in the future. The bulk of our effort must always be expended in the former, without worrying in advance about those aspects of the system that are unlikely to be changed.

Slide 19: The observer pattern is useful in most systems for ensuring loose and flexible coupling between objects. However the difference between callback coupling and data coupling is not so huge that we must always seek to eliminate the latter. Instead, we apply callback coupling when we are likely to get significant benefits from it – for example, at the intersection between software components, or between the various aspects of the MVC. Here, we'll use it to communicate between our model and each of the View/Controller classes. In doing so, we remove the structural connection between the view and the controller – this allows us to ensure differentiation between the roles.

Slide 20: This shows the class diagram once it has incorporated the observer pattern between the model and the view/controller classes – we do this through the use of an interface which the front-ends implement. We'll then hook these up to the model through the add/remove methods we'll need to expose.

Slide 21: This design is now suitable for implementation. We know what the classes are, we know what roles they will perform, and we have incorporated the design patterns we need to ensure flexibility in our design. We have already spoken about how to turn specific diagrams into code through the module, but we need to address the structural aspects of this particular diagram since it's the first relatively complex example we have seen.

Slide 22: For brevity, and because there is nothing new we can tell students about them, we have omitted attributes and operations for the diagram. Students can look at these

during the laboratory sessions. We will implement these partially during the lecture to show the impact on our development. We need to begin developing with classes that have no structural dependencies on other classes – that way, we always have a system we can compile.

Slide 23: This slide shows the implementation of the vehicle class.

Slide 24: Once we have the vehicle implemented, we can then implement classes that are dependent on its existence – in this case, the VehicleFactory. The factory is going to take in the type of the vehicle and then give out the appropriate object. We're going to do this as a static method so that we never need to instantiate an object of type VehicleFactory – we can invoke it wherever we need it.

Slide 25: This shows the first part of the implementation. This is a skeleton implementation that exposes some static constants to ensure that all classes are making use of the options that are available. We could just as easily have the factory spit out vehicles using a string description, but the use of constants removes the possibility of typos.

Slide 26: This shows the full implementation of the getVehicle method in the factory.

Slide 27: Once we have the VehicleFactory, we can create the class that is dependent upon its existence – the Garage. There are many ways that we can implement the list of vehicles that the Garage must contain, but the method we are going to use for this example is a HashMap – that gives us ease of access to each of the elements.

Slide 28: This shows the implementation of the garage. We have an addVehicle method that makes use of the factory. We could have another method that then used this method in a loop to create the 30 vehicles available in the system.

Slide 29: Implementation progresses according to a simple structure – create the basic classes that do not have any dependencies, and then implement those that are dependent. There is no need to have everything coded to begin with. In fact, it is better to handle the development incrementally. Implement the most basic functionality (all that is needed to make the system compile) and then start implementing the simplest version of a method. You can return to these and refine them as you go along to ensure that they are suitably robust and featureful.

Slide 30: Conclusion

10.5 Laboratory Sessions

The laboratory time allocation for this topic is 2 hours.

Lecturers' Notes:

Students have copies of the laboratory exercises in the Student Guide. Answers are not provided in their guide.

Activity 1:

Using the case study discussed in the lecture, implement the observer structure required for the link between the model and the view/controller. Consider what information is likely to be needed by the view/controller classes and define a method in your interface for each of these.

Suggested Answer:

An example implementation that handles four kinds of event is shown below:

```
import java.util.*;
public interface ModelObserver {
    public void vehicleAdded (String licence);
    public void vehicleRemoved (String licence);
    public void customerAdded (String id);
    public void customerRemoved (String id);
}
public class ModelFacade {
    private static final int EVENT_VEHICLE_ADDED = 0;
    private static final int EVENT_VEHICLE_REMOVED = 1;
    private static final int EVENT_CUSTOMER_ADDED = 2;
    private static final int EVENT_CUSTOMER_REMOVED = 3;

    ArrayList <ModelObserver> myListeners;

    public ModelFacade() {
        myListeners = new ArrayList<ModelObserver>();
    }

    public void addModelListener (ModelObserver m) {
        myListeners.add (m);
    }

    public void removeModelListener (ModelObserver m) {
        myListeners.remove (m);
    }
```

```
    public void notifyListeners(int type, String data) {
        for (ModelObserver m : myListeners) {
            switch (type) {
                case EVENT_VEHICLE_ADDED:
                    m.vehicleAdded (data);
                break;
                case EVENT_VEHICLE_REMOVED:
                    m.vehicleRemoved (data);
                break;
                case EVENT_CUSTOMER_ADDED:
                    m.customerAdded (data);
                break;
                case EVENT_CUSTOMER_REMOVED:
                    m.customerRemoved (data);
                break;
            }
        }
    }
}
```

Activity 2:

Using the case study discussed in the lecture, implement the Payroll, Staff, Owner and Driver classes with appropriate methods and attributes.

Suggested Answer:

```
import java.util.*;

public class Payroll {
    ArrayList<StaffMember> myStaff;

    public Payroll() {
        myStaff = new ArrayList<StaffMember>();
    }

    public void addStaffMember (String type, String ID, String name) {
        StaffMember sm;

        if (type.equals ("driver")) {
            sm = new Driver();
        }
        else {
            sm = new Owner();
        }

        sm.setID (ID);
        sm.setName (name);

        myStaff.add (sm);
    }
```

```
    public StaffMember findStaffMember (String ID) {
        for (StaffMember sm : myStaff) {
            if (sm.getID().equalsIgnoreCase (ID)) {
                return sm;
            }
        }
        return null;
    }
}

abstract class StaffMember {
    private String ID;
    private String name;

    public void setID (String s) {
        ID = s;
    }

    public String getID() {
        return ID;
    }

    public void setName (String s) {
        name = s;
    }

    public String getName() {
        return name;
    }

    abstract String getType();
}

public class Owner extends StaffMember {
    String getType() {
        return "Owner";
    }
}

public class Driver extends StaffMember {
    String getType() {
        return "driver";
    }
}
```

10.6 Private Study

The time allocation for private study in this topic is expected to be 7.5 hours.

Lecturer's Notes:

Students have copies of the private study exercises in the Student Guide. Answers are not provided in their guide.

Exercise 1:

If you did not complete the code from the lab session, finish this in your private study time. Remember that the tool you need is open source and can be freely downloaded.

Exercise 2:

You will previously have worked up diagrams for the implementation of several workflows. This was done as part of the private study for Topic 6. Assess these for validity, coupling and cohesion, and the suitability of design patterns, and then implement them along with your implementations from the laboratory session.

Exercise 3:

Review the lecture material and ensure that you are comfortable with everything discussed thus far

10.7 Tutorial Notes

The time allowance for tutorials in this topic is 1 hour.

Lecturers' Notes:

Students have copies of the tutorial activities in the Student Guide. Answers are not provided in their guide.

You can profitably approach this tutorial as a further class discussion, calling on individuals to present their worked solutions and collaborating on refining and improving them.

Exercise 1: Discuss Designs

Discuss as a class your reviews of the work you did for Exercise 2 in the private study session.

Exercise 2: Reporting Back to the Class

As a result of the work you will have done during private study, you should have code implementations for each of the workflows you designed in Topic 6. Be prepared to discuss your implementations with others in your class, and to help your peers refine and correct the code they have themselves developed.

Topic 11: Maintenance and Refactoring

11.1 Learning Objectives

This topic provides an overview of maintenance and refactoring. On completion of the topic, students will be able to:

- Identify different categories of maintenance activity;
- Identify the need for refactoring;
- Refactor methods and classes.

11.2 Pedagogic Approach

Information will be transmitted to the students during the lectures. They will then practise the skills during the laboratory sessions. Tutorials are then used to consolidate students' understanding of the concepts covered and deal with any questions.

11.3 Timings

Lectures:	2 hours
Laboratory Sessions	2 hours
Private Study:	7.5 hours
Tutorials:	1 hour

11.4 Lecture Notes

The following is an outline of the material to be covered during the lecture time. Please also refer to the slides.

The structure of this topic is as follows:

- Types of maintenance activity
- Issues of refactoring
- Refactoring examples

11.4.1 Guidance on the Use of the Slides

Slide 2: The majority of this lecture is given over to the topic of refactoring as it is prerequisite for most kinds of maintenance. Almost all maintenance tasks, regardless of what category into which they fall, will involve refactoring as part of the process. There are many studies on how much time software developers spend, generally, on maintenance activities but they all disagree to a considerable extent. The only real agreement is that maintenance takes up a lot more time than the actual implementation of software. Sometimes that maintenance is done by a different team of developers (other than the ones who wrote the software originally), and as such maintenance tends to be a 'deferred reciprocity' task – we make maintenance easier to do for other people in the hope they in turn make it easier for us in the future.

Slide 3: The four basic categories of maintenance activities are listed on this slide. The first two of these (adaptive maintenance and corrective maintenance) take up the largest amount of developer time. These are reactive maintenance processes, and are performed as a result of need or identified defects. Perfective and preventive maintenance are progressive processes, and are done to improve software, not to deal with emerging problems.

Slide 4: Adaptive maintenance is that which is aimed at bringing existing functionality in line with changes in the context within which the software works. Financial software may need to be adjusted in line with new legislation, networking software may need to be adjusted to work with new protocols, and all software may need to be changed when the processes they are modelled upon are changed. Adaptive maintenance does not include new features – it is about making existing features work correctly since the context has changed.

Slide 5: Corrective maintenance is what we traditionally think of as 'maintenance' – fixing defects that have been identified as the system runs. No matter how strict a testing regime you have, when a system goes live the number of bugs that have been identified in the system will multiply simply as a result of more people working through the system and more parts of the system being stressed. Corrective maintenance is almost always a long term, ongoing process. It involves prioritising bugs by how quickly they must be fixed, implementing the fixes, and then integrating those fixes into the program. The latter can be very complex in itself, especially if the fixes are done on a 'testing' version of the software and not the live product.

Slide 6: Perfective maintenance is something akin to a limited version of the software development phases that preceded it. It involves identifying user requirements,

analysing and designing the processes that will drive them, and then implementing these into an existing system. It also involves making changes to the system measures, such as improving performance and reliability, or interoperability with other programs.

Slide 7: Preventive maintenance is that which is aimed at fixing potential problems before they become actual problems. There is always more work than time when developing a software system, and often parts of the system will not be implemented at the maximum level of correctness because of time considerations. We usually know where these weak points in a system are, and preventive maintenance is about refactoring these before they actually cause problems. Preventive maintenance can also be about assessing system benchmarks and future proofing for capacity requirements. A lot of what happened in the early days of Google was of this kind of maintenance – systems worked perfectly well for current capacity, but needed to be optimised for future capacity.

Slide 8: Almost all maintenance involves refactoring, and refactoring can be a useful first step even in those situations where maintenance is not required. Refactoring is an invisible process where the intention is to turn bad code into good code, while a system is running and without impacting on any of the directly observable elements of the software. Thus, it involves internal restructuring to aid in future maintenance issues, or replacing one algorithm with an equivalent, but more efficient, variant. These kind of tasks fall into the category of refactoring. Refactoring itself does not add new features but refactoring is often a precursor to perfective maintenance.

Slide 9: Maintenance and refactoring are both complicated by issues relating to the impact of change. Much of what we do as software developers is aimed at limiting the impact of any change that we make. That is why we avoid things like high coupling in programs – we want to be able to make changes without having to alter lots of other subsections of the program, and highly coupled classes prevent this.

Slide 10: The process of data hiding is a useful way to limit impact of change, because we can enforce compile-time checks that prevent us exposing too much of the internal mechanisms of objects. This is why we insist that all attributes in classes are private, and why on the whole we should make methods as visible as they have to be but no more. The public APIs - exposed by the facade design pattern for example - have extremely high impact of change, in that we can't change an exposed method without having to change every invocation of that method through our entire system. Impact of change then is a rough measure of how much effort it will be to implement a fix.

Slide 11: Impact of change is partially a quantifiable measure, but it's mostly a rule of thumb. We have to assume that if someone has access to a method or variable that we have in an object that they will make use of it. For very small developer teams you can probably know for sure where this is the case, but for huge projects with large developer teams this isn't always something you can check. As such, if something is available, it must be treated as if it is also used somewhere in the system.

Slide 12: There are several firm rules that must be observed when refactoring if we wish to do so courteously and without upsetting fellow developers. These rules are outlined here.

Slide 13: While the rules are firm, they can be broken in cases where someone has the remit and will to fix all of the problems their refactoring will cause. Additionally, sometimes changes have to be made that violate the rules (because existing class

structures are causing problems, for example). This can be permitted provided enough notice is given to all other developers, and the process of refactoring is handled gradually. First, an announcement is made that part of the system is going to change, and those developers affected should update their code accordingly. Some languages, such as Java, permit for you to flag parts of a system as 'deprecated' which causes a compile time warning when functions are used, and this can be a useful way to make sure everyone is informed. Having provided an announcement and enough time for people to make changes (the size, scope and impact of the change will influence how much time is appropriate), the change can be made. Almost certainly it will result in problems anyway, but hopefully the majority of issues will have been fixed beforehand.

Slide 14: The exact act of refactoring is very individual and varies from task to task. The process however usually includes those elements indicated on this slide. While refactoring at its best is a proactive process whereby we continually improve code that we have authority over, more often it is simply something we do when it's needed in order for us to do something else.

Slide 15: This slide lists some common structural tasks performed during the refactoring process. These are changes to the static elements of the system – they are changes to class and object structures.

Slide 16: This slide lists some of the common tasks that are performed at the object and method level. They all involve turning code that is problematic into code that is less so.

Slide 17: This slide shows a simple example of an easy refactoring. Because the variable is private, we can very easily change its name using nothing more than a search and replace. The impact of change is limited to the object in which the variable is defined. We should change the name of this variable, because 'bing' is not a descriptive term that helps us understand what the contents are. Give students a few moments to study this class and suggest refactoring.

Slide 18: This is the same problem as the previous slide, except that the variable is set as public. Give students a few minutes to think about what the refactoring needs are, and what the impact of change may be. To refactor this properly, we would make the variable private and change its name, but this would require a deprecation process something akin to what is outlined in Slide 13. We could do it by creating a second variable that is private, and for a limited period of time we could support both of these within the functions before eventually deleting bing entirely.

Slide 19: This brings us back to the issue of impact of change – changing a public variable to a private variable is a structural change. It's safe to be **more** permissive (private to protected, or protected to public) but not to be less permissive. It is safe to specialise functionality (providing overridden versions of methods in children objects) but not necessarily to generalise (removing a specific implementation in favour of a general one). In all cases, we want to make sure that as far as our fellow developers are concerned (and specifically, as far as the code they have written is concerned) there is no impact unless it cannot be avoided.

Slide 20: This slide gives another example of a refactoring task – we want to change getValue so that it takes in a parameter. Perhaps it is going to change so that it gives the value modified by the parameter. How would we make this change? Give your students a little time to think about this and make suggestions.

Slide 21: There are several ways in which we can do this. We can simply add the parameter to the class, but that will break every other class that uses the method because they are no longer providing the necessary number of parameters. This is a discourteous way of handling the change. A more graceful way would be to overload the method, but this has the risk that people won't update their code to use the new overloaded method. Some parts of the system may use it, others may not, and this creates behavioural and maintenance inconsistencies. Incremental adjustments to systems take time to ripple through the code, and so sometimes it becomes necessary to simply delete the deprecated code and fix those things that end up breaking.

Slide 22: How easily you can refactor systems is based on your remit, that is, how much of the system you can change without having to consult others. For a small program, you may have a remit that operates over the whole system. For a large program you are likely to have a remit that extends over a subset of the whole system. You can unilaterally change those parts of the system that you have the remit to support.

Slide 23: It is tremendously frustrating as a developer to find someone has refactored a system and broken code you have written, because this goes very much against the grain of courteous development. You assume it is a problem you mistakenly introduced, and only after hours of profitless debugging do you eventually end up finding the part of the system causing the problem.

Slide 24: This shows another example of a function that needs to be refactored. Give students some time to suggest refactoring – an example solution can be found on Slide 27.

Slide 25: Code formatting is something that is usually insisted upon, but often inconsistently applied. The aesthetics of code are important though – they give a visual cue as to problematic code. It should be obvious on Slide 24 that there is a lot of nesting and that suggests that the code could be profitably refactored. However, in doing so we must be careful that we don't mistake ugly code for **battle-hardened** code. Sometimes code becomes complex because it has to do complex things, and that complexity gets introduced incrementally as the simpler routines encounter problems.

Slide 26: As time goes by, it becomes easier to tell which is which. Identifying which functions have poor aesthetics is a matter of taste and practise, but you should direct your students to http://basildoncoder.com/blog/2008/03/21/the-pg-wodehouse-method-of-refactoring/ for one interesting example of a technique.

Slide 27: One of the most uncontroversial ways to handle the code aesthetics is to break functionality into separate methods. It is generally accepted that a function should do one thing only, and that complex logic flows are usually a warning sign of the need to refactor. The design patterns we have discussed during the module can help approach the implementation of this in a way that has been proven to work.

Slide 28: This shows the first part of the refactoring of the method on Slide 24. Note that the amount of code has gotten larger, but that's okay, as refactoring isn't about reducing code count (although it can be). If we gain more from a more verbose representation, that is how we should represent it. This is one of the reasons why optimisation and maintenance are trade-offs – concise representation is usually more efficient, but more difficult to work with.

Slide 29: This slide shows the altered logic for the function. All the nesting has gone, and the method becomes much easier for developers to work with.

Slide 30: The process of test driven development that was alluded to in an earlier lecture is valuable here, because when refactoring, we run the risk of introducing new problems (for every two bugs we fix, we tend to introduce another one). If we can run our test cases each time we make a change, we can find these bugs quickly and easily and fix them early. Refactoring does not include any functionality changes, so we can use the test cases that already exist without having to add any new ones.

Slide 31: The ideal case is that we refactor code on a continual basis, but the realities of development are that we have to take a more pragmatic approach. We generally end up refactoring code that gets in the way of what we want to do (that is why it is often a precursor to other kinds of maintenance). We also tend to have a wish-list of things that we'd like to refactor if and when we can, and when we have some spare time available we can address those subsystems on our list.

Slide 32: Generally though, we refactor when code 'smells bad'. Interesting articles on that subject can be found at:

- http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm
- http://www.codinghorror.com/blog/2006/05/code-smells.html

The things listed in this slide are good examples of 'bad smells' in code that need to be addressed with refactoring.

Slide 33: Conclusion

11.5 Laboratory Sessions

The laboratory time allocation for this topic is 2 hours.

Lecturers' Notes:

Students have copies of the laboratory exercises in the Student Guide. Answers are not provided in their guide.

Activity 1:

Go back over code you have written for this module. Profile each of your functions for 'bad smells' and refactor them accordingly.

Suggested Answer:

No suggested answer – highly dependent on student code. Some particular problems you may wish to encourage students to look for include:

- Bad commenting. Comments should explain 'why' something is done, rather than 'what' is being done. Student comments often fall into the category of the latter rather than the former. If there are no comments at all, that's something else that can usefully be addressed.
- Duplicating code is usually easier than properly engineering objects, and so students should be on the lookout for code that is doing the same thing as any other code, and seek ways to generalise it.
- When developing code for systems that are only partially understood, experimentation is often key. Dead code is the code that is left over after the experimentation – unused variables, methods that aren't used any more, or functionality that is never triggered. This should be removed.
- Long methods that need to be broken up into smaller, more precise methods.
- Inconsistent naming, parameter types and parameter lists.

All of the code smells should be considered, although these are more common in student code.

11.6 Private Study

The time allocation for private study in this topic is expected to be 7.5 hours.

Lecturer's Notes:

Students have copies of the private study exercises in the Student Guide. Answers are not provided in their guide.

Exercise 1:

If you did not complete the work from the practical session, finish this in your private study time.

Exercise 2:

As part of your ongoing journal exercise, you should research the following topics:

- Bad code smells
- Common refactoring tasks
- Maintenance processes

Exercise 3:

Prepare a short, five minute presentation on the results of your research for Exercise 2 above. If you have found out anything particularly interesting, you should focus on that as a priority.

Exercise 4:

Review the lecture material and ensure that you are comfortable with everything discussed thus far.

11.7 Tutorial Notes

The time allowance for tutorials in this topic is 1 hour.

Lecturers' Notes:

Students have copies of the tutorial activities in the Student Guide. Answers are not provided in their guide.

This tutorial is designed to give students a chance to present the more interesting of their findings from the previous exercise, partially as a way to share information with others in the class but also as a way to ensure that they are indeed keeping a journal and researching the topics provided. You should encourage students to bring along their journals to these sessions and make notes on any points of interest that are raised by their classmates.

Exercise 1: Reporting Back to the Class

As a result of the research you did during your private study time, you should have a short five minute presentation ready to give to the rest of the class. There is no need for this to be especially formal - you are simply reporting on anything interesting that you found during your research, or pointing out especially useful resources on the topic. Bring your journal along to the class so that you can make a note of anything especially useful that your classmates mention. This is a knowledge dissemination exercise; you are not being formally assessed on the style or content of the presentation.

Topic 12: Recap of Module

12.1 Learning Objectives

This topic provides an overview of module content. On completion of the topic, students will be able to:

- Explain what was covered during the module.

12.2 Pedagogic Approach

This lecture is primarily a recap of existing material, and as such it is only a broad overview of what was discussed within each of the previous lectures. You should field questions from students at each distinct topic of the recap, ensuring that you capture any issues they have with the material and that you are prepared to contextualise it depending on student queries. It would be useful to keep the other lecture slides available for the summary so that you can refer back to the originals when asked.

12.3 Timings

Lectures:	2 hours
Laboratory Sessions	2 hours
Private Study:	7.5 hours
Tutorials:	1 hour

12.4 Lecture Notes

The following is an outline of the material to be covered during the lecture time. Please also refer to the slides.

The structure of this topic is as follows:

- Recap of module

12.4.1 Guidance on the Use of the Slides

Slide 2: As this is a recap of the module and there is no original content to be discussed, the guidance here will focus on which topics provide the fuller explanation of the points raised. The approach taken here is a mostly contextual overview of the previous eleven lectures, but a few points with regards to integrating later weeks with earlier weeks are made also.

Slide 3-4: We begin with a discussion of analysis. This slide is primarily related to the content of Topic 1.

Slide 5: Use case diagrams were discussed in Topic 3. Activity diagrams were discussed in Topic 5. The first part of Topic 5 discussed activity diagrams in relation to analysis, with the latter part of the lecture addressing activity diagrams in relation to design.

Slide 6: Decomposition was addressed in Topic 3, although most of what analysis and design does is about decomposition, and so it is a theme that extends throughout the entire module content.

Slide 7: The class diagram was discussed in Topic 4, as was the process of Natural Language Analysis. Sequence diagrams were discussed in Topic 5 along with activity diagrams.

Slide 8: Each of the diagrams we use is intended to explore either a different aspect of the system, or the same aspect of the system from a different level of abstraction. In this way, we can gain whatever level view of the system we require to meet our needs.

Slides 9-10: The case study was discussed first in Topic 6 (in terms of its design) and then again in Topic 10 (where we discussed the redesign and the implementation issues that go with it). Topic 6 has very little actual lecture content, as the majority of the content is located in the laboratory sessions section of the Lecturer Guide, with the fully realised analysis and design documents. The key element of the case study is that it sits between the trivial examples of the lectures, and the often considerable complexity of real world systems.

Slides 11-12: We discussed design patterns in Topics 7 and 8. Topic 7 covered the intention of patterns along with the specific patterns of the Factory and the Abstract Factory. Topic 8 covered the other patterns with the exception of the observer pattern (that was discussed in Topic 9 instead).

Slide 13: Assessing our designs and our implementations was discussed in Topic 9. The structural aspects of this were the elements relating to coupling, cohesion and component design.

Slide 14: Topic 10 covered the redesign of our case study in light of the patterns we had discussed, and the coupling and cohesion heuristics we identified in Topic 9.

Slides 15-17: Topic 9 discussed the elements of assessing the quality of running software.

Slide 18: The topic of maintenance and refactoring was covered in Topic 11.

Slide 19: Conclusion

12.5 Laboratory Sessions

The laboratory time allocation for this topic is 2 hours.

Lecturers' Notes:

Students have copies of the laboratory exercises in the Student Guide. Answers are not provided in their guide.

Students should use this session finish up any exercises they have remaining from previous topics. For those who have completed all of the work, they should complete the review activity below.

As this is the final laboratory session, you may also wish students to attempt the sample examination paper, which can be found on the NCC Education Campus (http://campus.nccedu.com).

Activity 1

Provide a paragraph of explanation and explanatory code snippets and diagrams (where appropriate) for the following concepts. This represents a condensing of both module content and your journal entries into a form suitable for revision and later reference:

- Refactoring
- Coupling
- Cohesion
- Analysis
- Design
- Design Pattern
- Candidate Class
- Class Diagram
- Activity Diagram
- Sequence Diagram
- Singleton Design Pattern
- Facade Design Pattern
- Strategy Design Pattern
- Refactoring
- Maintenance

Suggested Answer

There is no suggested answer for this exercise since it is a synthesis of previous topics. The intention here is for students to construct what are essentially their own revision slides using the results of their own research, their own development, and the module slides. An example of the kind of thing that students should be writing is shown below:

Coupling

Coupling is the extent to which components in a program are connected. In general, we aim for low coupling in a program, although we cannot aim for zero coupling. Coupling has many different forms, and some of these forms are better than others:

Content coupling - When a module makes use of the local data of another. The worst kind of coupling.

Common coupling - When two modules share the same global data store.

Data coupling - When modules share data via parameters.

Callback coupling - Such as in the observer design pattern.

12.6 Private Study

The time allocation for private study in this topic is expected to be 7.5 hours.

Lecturers' Notes:

Students have copies of the private study exercises in the Student Guide. Answers are not provided in their guide.

Exercise 1:

If you did not complete the laboratory activity during the allotted time, complete it now.

Exercise 2:

Review the lecture material and your revision notes from Activity 1 in the laboratory session and ensure that you are comfortable with everything discussed thus far. If there are topics that you are not entirely sure of, make a note of them so they can be discussed during the tutorial.

12.7 Tutorial Notes

The time allowance for tutorials in this topic is 1 hour.

Lecturers' Notes:

Students have copies of the tutorial activities in the Student Guide. Answers are not provided in their guide.

During this tutorial, select students at random and get them to recount the revision notes that they constructed for a particular topic. You may wish to have them do this as a short presentation where they recite their answer and show code/diagram samples. This will give students an opportunity to reflect on what others have taken away as the important points in the material. You may also wish to consider making available the results of student work on a website, or perhaps as printed handouts.

Exercise 1: Report back to class

As a class exercise, you should discuss the revision topics that you constructed during the laboratory exercise.

Exercise 2: Revision

As a result of your reviewing in your private study, you may have come up with revision questions to discuss in class. Raise them now with your lecturer.