



Faculty of Science and Technology

CBOP3103

Object-Oriented Approach in Software Development

CBOP3103
OBJECT-ORIENTED
APPROACH IN
SOFTWARE
DEVELOPMENT

Karin Kolbe

Dr Geoffrey Phipps



Project Directors: *Prof Dato' Dr Mansor Fadzil*
Assoc Prof Dr Norlia T. Goolamally
Open University Malaysia

Developers: *Karin Kolbe*
Dr Geoffrey Phipps
Spirus Pty Ltd

Coordinator: *Dr S L Chung*
OUHK

Production: ETPU Publishing Team

**Adapted for
Open University Malaysia by:** *Assoc Prof Dr Nantha Kumar Subramaniam*
Open University Malaysia

Reviewed by: *R Moganadas Ramalingam*

Edited by: *Aezi Hani Abd Rashid*

Developed by: *Centre for Instructional Design and Technology*
Open University Malaysia

Printed by: Meteor Doc. Sdn. Bhd.
Lot 47-48, Jalan SR 1/9, Seksyen 9,
Jalan Serdang Raya, Taman Serdang Raya,
43300 Seri Kembangan, Selangor Darul Ehsan

First Edition, May 2008
Second Edition, December 2015 (rs)

Copyright © The Open University of Hong Kong and Open University Malaysia, December 2015, CBOP3103

All rights reserved. No part of this work may be reproduced in any form or by any means without the written permission of the President, Open University Malaysia (OUM).



Table of Contents

.....

Course Guide

ix–xiv

Topic 1	Introduction to Object-Oriented	1
1.1	Approaches to Systems Development	3
1.1.1	Predictive Approaches	3
1.1.2	Models and Notations in Predictive Approaches	4
1.1.3	An Iterative SDLC	5
1.2	Object-Oriented Systems Design	6
1.2.1	Thinking in Objects	7
1.2.2	Building a System from Objects	11
1.2.3	The Unified Process	11
1.2.4	Notation	14
1.2.5	Is Object-oriented Systems Design an Adaptive Approach?	18
1.2.6	Analysis and Design	19
1.3	The Importance of Design	20
1.4	Getting Started: The Inception Phase	21
1.4.1	Inception Phase Artifacts	23
	Summary	24
	Key Terms	28
	References	28
Topic 2	Requirement and Use Cases	29
2.1	Requirements Analysis	30
2.1.1	Levels and Types of Requirements	31
2.1.2	Challenges of Writing Requirements	33
2.2	Stakeholders, Actors and Roles	36
2.2.1	Types of Roles	37
2.2.2	Differences in Roles	37
2.2.3	Discovering the Roles	39
2.3	Writing Use Cases	41
2.3.1	Goals and Levels of Use Cases	46
2.3.2	Format for Writing Use Cases	48
2.3.3	Use Case Diagrams	48
2.3.4	Activity Diagrams	49
2.3.5	Potential Problems with Use Cases	51
2.3.6	How to Deal with Hundreds of Use Cases	52

	2.4 Capturing Requirements Iteratively	53
	Summary	54
	Key Terms	55
	References	55
Topic 3	Object-Oriented Modelling	57
	3.1 From Inception to Elaboration	58
	3.2 Use Case Review – System Sequence Diagrams	61
	3.3 Domain Model	62
	3.4 Conceptual Classes	69
	3.4.1 Identification of Conceptual Classes	70
	3.4.2 Issues in Identifying Conceptual Classes	73
	3.4.3 Specification Conceptual Classes	73
	3.5 Associations	74
	3.5.1 Identifying Associations	75
	3.5.2 Multiplicity	78
	3.5.3 Other Issues of Association	79
	3.6 Attributes	81
	3.7 UML Notation, Models and Methods: Multiple Perspectives	83
	Summary	84
	Key Terms	85
	References	85
Topic 4	More Use-Cases	86
	4.1 Relating Use Cases	87
	4.1.1 The Include Relationship	89
	4.1.2 The Extend Relationship	92
	4.1.3 The Generalise Relationship	94
	4.2 Finding All Use Cases	95
	4.2.1 State Diagrams	96
	4.2.2 Showing Super States	99
	4.2.3 Create + Read + Update + Delete = CRUD	100
	4.3 Business Rules	101
	4.4 More Modelling	103
	4.4.1 Robustness Diagrams	104
	Summary	113
	Key Terms	114
	References	114

Topic 5	Dynamic Modelling	116
5.1	The Design Model	117
5.1.1	Classes versus Objects	119
5.1.2	Objects and Messages	121
5.2	Interaction Diagrams	123
5.2.1	Collaboration Diagram Notation	123
5.2.2	Sequence Diagram Notation	126
5.2.3	Designing for Responsibilities	128
5.3	Design Class Diagram	132
5.3.1	Visibility	134
5.3.2	Full Attribute Specification	134
5.3.3	Association	135
5.3.4	Composition (“Uses a” Relationship)	136
5.3.5	Aggregation (“Has a” Relationship)	137
5.4	Inheritance	138
5.4.1	Polymorphism	142
5.4.2	Concrete and Abstract Classes	142
5.5	Design Principles	144
5.5.1	Coupling	144
5.5.2	Cohesion	146
	Summary	148
	Key Terms	149
	References	149
	Case Study	150
	Answers	151

COURSE GUIDE

COURSE GUIDE DESCRIPTION

You must read this *Course Guide* carefully from the beginning to the end. It tells you briefly what the course is about and how you can work your way through the course material. It also suggests the amount of time you are likely to spend in order to complete the course successfully. Please keep on referring to *Course Guide* as you go through the course material as it will help you to clarify important study components or points that you might miss or overlook.

INTRODUCTION

CBOP3103 Object-Oriented Approach in Software Development is one of the courses offered by the Faculty of Science and Technology at Open University Malaysia (OUM). This course is worth 3 credit hours and should be covered over 8 to 15 weeks.

TO WHOM IS THIS COURSE TARGETED?

This course is targeted to all IT students especially those specialising in Computer System or Information System who need to understand how software is developed using the object-oriented approach. Students enrolled in other IT-related specialisations will also find this course useful as this course will answer many of their questions regarding object-oriented system development.

STUDY SCHEDULE

It is a standard OUM practice that learners accumulate 40 study hours for every credit hour. As such, for a three-credit hour course, you are expected to spend 120 study hours. Table 1 gives an estimation of how the 120 study hours could be accumulated.

Table 1: Estimation of Time Accumulation of Study Hours

Study Activities	Study Hours
Briefly go through the course content and participate in initial discussions	3
Study the module	60
Attend 3 to 5 tutorial sessions	10
Online participation	12
Revision	15
Assignment(s), Test(s) and Examination(s)	20
TOTAL STUDY HOURS	120

COURSE OUTCOMES

By the end of this course, you should be able to:

1. Explain the fundamental tenets of object orientation;
2. Explain how to iteratively develop an object-oriented system;
3. Apply object-oriented analysis, design skills and notations to a stated problem;
4. Discuss the use of object-oriented methodologies and notations during system development;
5. Use appropriate skills and techniques to adequately set the scope and document requirements;
6. Develop an actor and use case model;
7. Create a domain model with classes;
8. Draw state and robustness diagrams to validate a domain model;
9. Draw dynamic diagrams to address the behavioural aspects of a system; and
10. Understand the principles of good object-oriented design.

COURSE SYNOPSIS

This course is divided into 5 topics. The synopsis for each topic is presented below:

Topic 1 introduces the fundamentals of object orientation, putting it in the context of developments in computing over the last 40 years.

Topic 2 examines who the users of the system will be. We will study approaches to ascertaining and capturing the requirements for the system.

Topic 3 is the first of four units devoted to various aspects of object modelling. Specifically, *Topic 3* is about determining the correct objects for a system.

Topic 4 looks more closely at the requirements of the system and adds this information to our models.

Topic 5 is about designing the messages that are sent between objects. In addition, you will also learn about inheritance in detail and other related concepts. The principles of good object-oriented design will constitute the last section of this topic.

There will be a number of additional “short” readings for certain topics as shown in the table below. These readings are provided as part of your course package in myVLE.

Topic	Reading
Topic 1	–
Topic 2	<ul style="list-style-type: none"> • Reading 2.1 – Reading 2.3 (downloadable from myVLE) • One reading from e-book chapter (24×7 digital library)
Topic 3	–
Topic 4	<ul style="list-style-type: none"> • One reading from e-book chapter (24×7 digital library)
Topic 5	<ul style="list-style-type: none"> • Two readings from e-book chapters (24×7 digital library)
There is also one case study provided at the end of the module	

It is compulsory for students to read these readings as it will give them a solid understanding of the particular concepts related to the course. These readings are part of the syllabus. Hence, it can be used for tests and the final examination.

TEXT ARRANGEMENT GUIDE

Before you go through this module, it is important that you note the text arrangement. Understanding the text arrangement will help you to organise your study of this course in a more objective and effective way. Generally, the text arrangement for each topic is as follows:

Learning Outcomes: This section refers to what you should achieve after you have completely covered a topic. As you go through each topic, you should frequently refer to these learning outcomes. By doing this, you can continuously gauge your understanding of the topic.

Self-Check: This component of the module is inserted at strategic locations throughout the module. It may be inserted after one sub-section or a few sub-sections. It usually comes in the form of a question. When you come across this component, try to reflect on what you have already learnt thus far. By attempting to answer the question, you should be able to gauge how well you have understood the sub-section(s). Most of the time, the answers to the questions can be found directly from the module itself.

Activity: Like Self-Check, the Activity component is also placed at various locations or junctures throughout the module. This component may require you to solve questions, explore short case studies, or conduct an observation or research. It may even require you to evaluate a given scenario. When you come across an Activity, you should try to reflect on what you have gathered from the module and apply it to real situations. You should, at the same time, engage yourself in higher order thinking where you might be required to analyse, synthesise and evaluate instead of only having to recall and define.

Summary: You will find this component at the end of each topic. This component helps you to recap the whole topic. By going through the summary, you should be able to gauge your knowledge retention level. Should you find points in the summary that you do not fully understand, it would be a good idea for you to revisit the details in the module.

Key Terms: This component can be found at the end of each topic. You should go through this component to remind yourself of important terms or jargon used throughout the module. Should you find terms here that you are not able to explain, you should look for the terms in the module.

References: The References section is where a list of relevant and useful textbooks, journals, articles, electronic contents or sources can be found. The list can appear in a few locations such as in the *Course Guide* (at the References section), at the end of every topic or at the back of the module. You are encouraged to read or refer to the suggested sources to obtain the additional information needed and to enhance your overall understanding of the course.

ASSESSMENT METHOD

Please refer to myVLE.

USE OF CASE STUDIES

The course makes use of two case studies which you will find in the end of Topic 1:

- The NextGen POS system
- Victoria's Videos

Case studies are a useful and increasingly popular form of learning and assessment in the OUM's Faculty of Information Technology and Multimedia Communication.

CONCLUSION

Design is difficult. Good design is even more difficult and requires practice and time. Unlike accounting, there is often more than one correct answer. If you ask your tutor questions, they may frequently answer with "it depends". This can be very frustrating at first, but with the repeated practice involved in the case study, self-tests and tutorials you should gain confidence, and, we hope, enjoy the course. Good luck!

A NOTE ABOUT THE DEVELOPERS AND REVIEWERS OF THIS MODULE

Karin Kolbe (BSc, MSc) has extensive experience in object-oriented consulting and training. She has written and conducted courses on object-oriented analysis and design, writing use cases and usability. Over a wide range of applications, including portfolio management, road traffic control, policing and e-commerce, she has worked in the role of project manager, business analyst, methodologist and programme manager.

Geoffrey Phipps (BSc, PhD) is an experienced consultant and software architect, skilled in Java, J2EE, distributed computing and investment accounting on both Windows NT and Unix. His aims are to continually improve the software process using object-oriented technologies and software metrics.

Nantha Kumar Subramaniam (PhD (CompSc)) has reviewed, moderated and added values for this module so that it is line with the needs of Open University Malaysia (OUM) students. He has wide teaching and research experience in the field of object-oriented analysis and design and object-oriented programming. He has co-authored modules in these areas for OUM students.

TAN SRI DR ABDULLAH SANUSI (TSDAS) DIGITAL LIBRARY

The TSDAS Digital Library has a wide range of print and online resources for the use of its learners. This comprehensive digital library, which is accessible through the OUM portal, provides access to more than 30 online databases comprising e-journals, e-theses, e-books and more. Examples of databases available are EBSCOhost, ProQuest, SpringerLink, Books24x7, InfoSci Books, Emerald Management Plus and Ebrary Electronic Books. As an OUM learner, you are encouraged to make full use of the resources available through this library.

Topic 1 ► Introduction to Object-Oriented Software Development

LEARNING OUTCOMES

By the end of this topic, you should be able to:

1. Describe the fundamental principles of object-oriented systems;
2. Define the terms waterfall and iterative as applied to system development;
3. Differentiate between methodology and notation;
4. Define UML and describe how it is used;
5. Define Unified Process (UP) and describe how it is used; and
6. Describe how phases, disciplines and artifacts in UP are related.

► INTRODUCTION

Welcome to Topic 1 of the course **CBOP3103 – Object-Oriented Approach in Software Development**. This topic is an introduction to the whole course. We will begin with the most important concept: what is an **object**? It is very important for you to understand the concept of an object. Later we will see how an object is related to a class.

At the basic level, an object is a set of data together with some operations that can be performed on that data. An example of an object is a bank account. The data consists of the account number, the name of the account and the branch where the account is held. Operations that can be performed on a bank account include opening the account, withdrawing money, depositing money and giving an account balance.

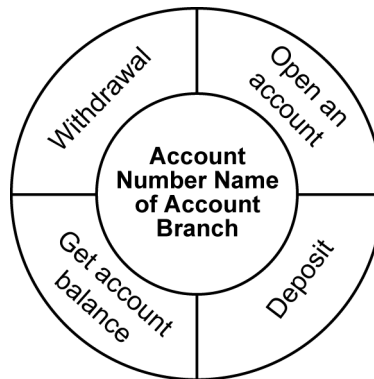


Figure 1.1: A bank account object

An object-oriented system can be viewed as a collection of objects sending and receiving **messages** from and to one another. An analogy is the people in a city. Each person is an object who sends and receives messages to and from other people. Some messages are requests for information ("What is your phone number?"), some messages supply information ("6685 1234") and others are a request to perform a task ("Please photocopy these pages"). A person cannot change the data of another person. All changes to data are done by the individual concerned, either in response to a message from somebody else, or because of a personal decision. I cannot change your name, but you can. Thus, we each have our own data that we can either share or keep private. In this course we make use of the "Victoria's Videos" case study, which you will find in this topic. Please have a look at this case study as we will use it throughout the module.

1.1 APPROACHES TO SYSTEMS DEVELOPMENT

Fowler (2000) considers some disadvantages of previous predictive approaches (such as the waterfall model) to systems design and recommends instead an adaptive approach. An example of adaptive approach is object-oriented design.

1.1.1 Predictive Approaches

One example of the kind of predictive approach is the **waterfall** Systems Development Life Cycle (SDLC) that was introduced to software engineering in the 1970s.

An SDLC is a management tool for planning, executing and controlling various activities involved in the systems development process. More specifically, it is a tool for managing complex processes by breaking them down into a series of phases and activities. It outlines step-by-step activities for each phase of SDLC, the role that stakeholders play in each activity and the **deliverables** expected from each activity. The key characteristic of the waterfall SDLC is that each step is completed before moving on to the next. The drawback of this approach is that we cannot be sure if we are building the right system until the end of the process. Also, it is almost certain that during a long development process, the requirements will change.

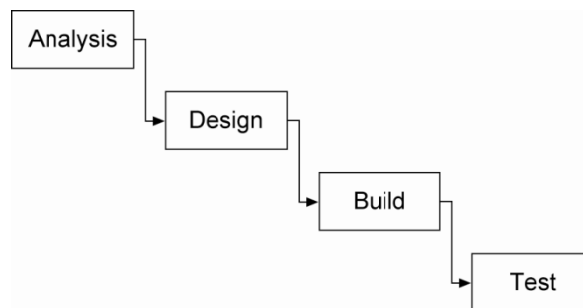


Figure 1.2: The waterfall approach

In the 1980s, various approaches under the heading of rapid, or prototyping or spiral development tried to change this approach. All these techniques had some merit, but also some drawbacks and so systems, especially large systems in major organisations (such as a banking application system) continued to be built by the waterfall approach.

1.1.2 Models and Notations in Predictive Approaches

A model is a simplified analogy of a real entity (a system in this course). It can be a document, a diagram or even a wooden model that allows us to see certain details or aspects of the real entity, while omitting others. For example, a paper model of a new building will show the size of the building in relation to other buildings. If a light is provided we can even see the shadows that the sun will create. However, we will not have much of an idea of the textures of the building – glass, brick, steel, etc.

In the 1980s, modelling an application before trying to build it became common practice.

The models were typically in any of these forms:

- (a) Entity-Relationship diagrams (ER);
- (b) Data Flow Diagrams (DFDs); and
- (c) Flow charts.

Do not worry if you are not sure what these are, as it is unlikely that you will need to deal with them these days especially in large system or software development projects. The important thing to note is that these models separate the data and the process or functions that were done on the data throughout all the analysis and design processes. The resulting systems were generally hard to maintain over time. One reason that maintenance was hard was that all the data and processing relating to one entity could be spread throughout the system, so a change required a lot of work. For example, say we wanted to change the employee ID field from six to eight digits. This would mean looking at every screen or report to see what the impact would be. The Year 2000 problem is another example where date-based calculations were spread throughout a system, each needing to be manually checked. From all these models the programmes were then written.

Later in this topic and later in this course you will see that object-oriented modelling keeps the data and processes together.

According to Fowler (2000), the main limitation to the predictive approach is that it fails to take account of inevitable changes in the lifetime of the development process.

Iterative development makes sense in predictable processes as well, but it is essential in adaptive processes because they must be able to deal with changes in required features.

As you will see, one of the key characteristics of object-oriented systems design is that it uses an iterative SDLC.

1.1.3 An Iterative SDLC

With an iterative SDLC, we build a small section of the system and then “try it out”. Depending on what has been built, the words “try it out” may mean something different. Perhaps we have enough of the system for the users to do a small part of their work, or to work on simple cases. Perhaps we have some software that the users can play with using test data. Very early in the project there may only be some draft screen layouts that mimic certain actions.

From the results of the tests we may need to make changes. Once we are satisfied that everything is working well, we then add some more functionality and repeat the whole process. Each iteration may either improve or add some functionality.

1.2 OBJECT-ORIENTED SYSTEMS DESIGN

We will start this section with a short reading that presents the argument for object-oriented systems design. Just as approaches to Information Technology (IT) have changed in the past thirty years, so too has the role of IT in organisations.

IT is used in an organisation in one of two ways. Firstly, IT can simply *support* the business by streamlining and speeding up various tasks. For example, a book publisher could use IT for the invoicing and accounting side of the business. Secondly, a more sophisticated use of IT is to *enable* new services or products. An example here is a publisher who publishes books on the Web and charges a fee per viewing. Increasingly, more and more applications are enabling rather than supporting the business. In today's volatile business environment, new customer requirements and new ways of doing business are emerging on a daily basis. Thus, the need for flexible systems that can be constructed and changed quickly is very important. **This is where object-oriented technology is critical.** However, of course, we need to be mindful that no technology is the “silver bullet” panacea for all development issues.



ACTIVITY 1.1

Take the organisation where you work as an example. What is the role of IT in the organisation? What IT applications can be categorised as “support”? What IT applications can be categorised as “enabling business”? Talk to the IT personnel in your organisation to find out about the different uses of IT in these two categories. Post a summary of your findings on the myVLE discussion board. Compare your findings with your course mates.

1.2.1 Thinking in Objects

As we stated in the introduction, learning to “think in objects” is difficult. So we want you to start thinking about objects now, even though they are not discussed until *Topic 3*.

There are few important terms to object orientation that are explained in this section:

- (a) **Class** and **object**;
- (b) **Encapsulation** of data and **methods**;
- (c) Support for **polymorphism**;
- (d) **Inheritance** within class hierarchies; and
- (e) Object **interface**.

A **class** is a blueprint that defines the variables (or attributes) and the methods common to all objects of a certain kind. In the real world, we often have many objects of the same kind. For example, your car is just one of many cars in the world. In an object-oriented context, we say that your car object is an instance of the class of objects known as cars. Cars have some state (current gear, current speed, four wheels) and behaviour (change gears, change speed) in common. However, each car’s state is independent of and can be different from that of other cars.

When building them, manufacturers take advantage of the fact that cars share characteristics and they can manufacture many cars from the same blueprint. It would be very inefficient to produce a new blueprint for every car manufactured. In object-oriented software, it is also possible to have many objects of the same kind that share characteristics: employee records, video clips, students record and so on. Like car manufacturers, you can take advantage of the fact that objects of the same kind are similar and you can create a blueprint for those objects.

As discussed above, an object is an instance of a class. The object is the most important thing in the object-oriented paradigm. Objects in object-oriented software collaborate and work together by sending **messages** to implement the behaviour of the software. To be specific, we can define objects as a thing that has state, behaviour and identity. Table 1.1 gives explanation and examples of these object characteristics.

Table 1.1: Characteristic of Object

Object Characteristic	Explanation
State	Each object has properties that collectively represent the state. A state can have dynamic values or permanent values. Examples of states that are dynamic are total money in the machine, number of can drinks that are not yet sold, age, etc. Examples of permanent states that do not change are name, birth date, etc. In programming, states are represented by attributes.
Behaviour	<p>Object behaviour is an object's response when the object receives a message. There are probably two types of actions that occur which change the state of the object that receives the message or the state of the object that sends the message.</p> <p>Example: Consider the vending machine object. One of the messages that it probably receives is: release the can drink selected by the buyer. This message will be sent to the vending machine when the buyer pushes a button on the machine. The message received causes the object to show a behaviour, that is, the drink can selected by the buyer is released. Besides showing behaviour, an object can also send messages to other objects in response to a received message.</p> <p>There is a close relationship between behaviour and state. An object's behaviour is influenced by the object's state. For example, assume that the number of cans in the machine is zero, that is, all the cans in the machine are sold out. If the machine received the message to release a can drink, the object certainly cannot fulfill this request because of its state where the number of cans is zero. In programming, methods are used to define the behaviours that an object can perform.</p>
Identity	Identity is the property that distinguishes an object from all other objects. Each object has its own unique identity. For example, a group of student objects could be identified with their unique identity such as StudentA, StudentB, StudentC, etc.

Please refer to <http://java.sun.com/docs/books/tutorial/java/concepts/class.html> which has a good diagram explaining the concept of class and object.

In object-oriented programming, the term **method** refers to a piece of code that is exclusively associated either with a class (called **class methods** or **static methods**) or with an object (called **instance methods**). A method usually consists of a sequence of statements to perform an action, a set of input parameters to

parameterise those actions and possibly an output value (called **return value**) of some kind. The purpose of methods is to provide a mechanism for accessing (for both reading and writing) the private data stored in an object or a class.

Consider the following scenario:

You are entering a restaurant. After going through the menu, you ordered a plate of fried rice from the server. Then, the server will go straight to the kitchen to convey your order to the cook. After ten minutes, the server will come back to you with the food you have ordered.

In the above scenario, you do not bother how the fried rice has been cooked. You just want the fried rice to be served. This is an example of encapsulation. **Encapsulation** is the term given to the process of hiding all the details of an object that do not contribute to its essential characteristics. Encapsulation hides the implementation details of the object and the only thing that remains externally visible is the interface of the object (that is, the set of all **messages** the object can respond to). Once an object is encapsulated, its implementation details are not immediately accessible any more. Instead they are packaged and are only indirectly accessible via the interface of the object. The only way to access such an encapsulated object is via message passing: one sends a message to the object, and the object itself selects the method by which it will react to the message, determined by **methods**. All objects are associated by the list of messages understood by them. This list of messages collectively becomes the **interface** of the object. For example, the interface for the ATM machine object is all the valid messages a user can send to it such as to withdraw money, check balance, funds transfer, etc. Invalid messages sent by the user which is not part of the ATM machine object interface will not be accepted by the object.

Inheritance is the capability of a class to use the properties and methods of another class while adding its own functionality. An example of where this could be useful is with an employee records system. You could create a *generic* employee class with states and actions that are common to all employees. Then more *specific* classes could be defined for salaried, commissioned and hourly employees. The generic class is known as the parent (or *superclass* or base class) and the specific classes as children (or *subclasses* or derived classes). The concept of inheritance greatly enhances the ability to *reuse* code as well as making design a much simpler and cleaner process.

Polymorphism is the capability of an action or *method* to do different things based on the object that it is acting upon. Overloading, overriding and dynamic method binding are types of polymorphism.

We will revisit some of the concepts introduced in this section in Topic 5.



ACTIVITY 1.2

Add the following labels to the diagram or the sentences below:

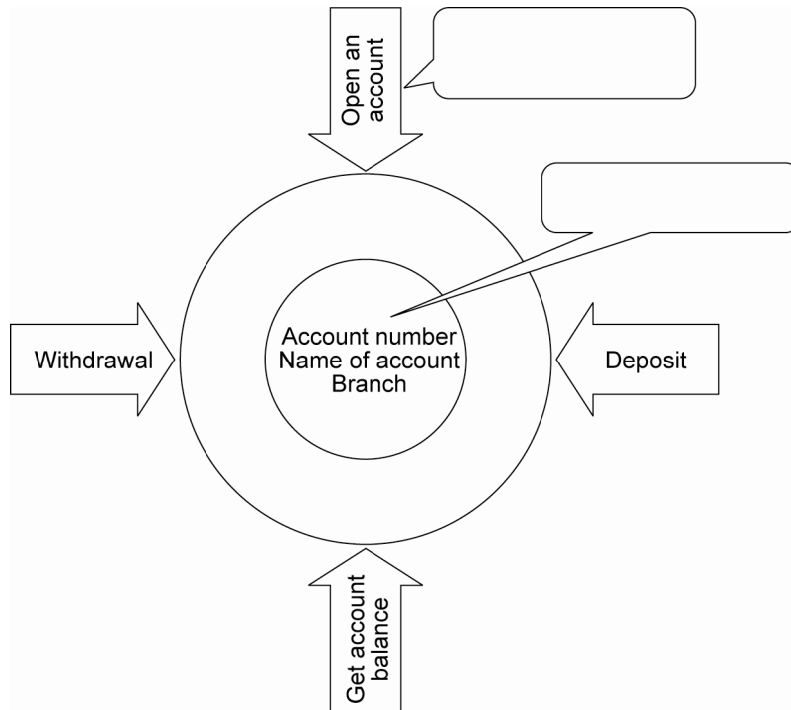
class

data

instance

message

messages



- Objects communicate by sending _____ to each other.
- A _____ is a software template that defines the methods and variables in a particular set of objects.
- Another word for “object” is _____.

1.2.2 Building a System from Objects

From the reading in the previous section you should have an idea of how objects can be used to implement real-world objects like bank accounts and vehicles. However, you are probably still mystified as to how a complex commercial application is built entirely using objects.

In later topics you will see other examples of objects, such as transactions, processes, and calculators. In an object-oriented system, every piece or building block is an object. Furthermore, each object needs to fulfil exactly one purpose or role. Over the past decade, object technology specialists have learned that systems in which this rule is adhered to are more adaptable to change and have fewer defects.

Now we are ready to see how object-oriented technologies are applied in application development projects. You have already encountered the **iterative SDLC**. Now we are going to look at the **methodologies** and **notations** used to develop object-oriented systems. If you need to remind yourself what these terms refer to, review the previous section.

(A word of warning: not everything in this section is strictly object-oriented nor does every project utilising some aspect of object orientation use all of these approaches. What we are discussing here are current best practices in object-oriented software engineering.)

1.2.3 The Unified Process

Some textbooks describe the **Unified Process (UP)** as an object-oriented methodology. This is not quite accurate, as it can be used for other approaches to systems design, but it is the methodology that we are using, so in our context it is object-oriented. It guides the key deliverables (documents, models, etc.), the composition and skills of the team and their responsibilities, and the length and timing of the project's phases. The process is a published industry standard for software development process especially for object-oriented systems. A variant of UP known as **Rational Unified Process (RUP)** which is a detailed refinement of UP has been widely adopted. This course will focus on the normal UP. UP is an iterative development organised over two dimensions – phases and disciplines (originally known as workflows). Each discipline has its own artifacts.

- (a) UP has four phases – **inception**, **elaboration**, **construction** and **transition**. The purpose for all these phases are described in the following table:

Table 1.2: Phases of UP

Phase	Purpose
Inception	The primary goal of the inception phase is to establish the case for the viability of the proposed system.
Elaboration	Elaboration phase is a phase where the project starts to take shape. In this phase the problem domain analysis is made and the architecture of the project gets its basic form.
Construction	In construction, the main focus goes to the development of components and other features of the system being designed. This is the phase where the bulk of the coding takes place.
Transition	In the transition phase the product has moved from the development organisation to the end user. The activities of this phase include training of the end users and maintainers as well as beta testing of the system to validate it against the end users' expectations.

It is important for you to take note that all these phases above are subdivided into iterations.

- (b) UP has disciplines which represent the project's focus areas – business modeling, requirements, analysis and design, implementation, test, deployment, project management, configuration and change management as well as environment. Each discipline has its own artifacts.

Figure 1.5 illustrates the changing relative effort of disciplines with respect to the phases in UP as suggested by Larman (2002). As highlighted by the diagram, one iteration of a phase work is carried out in most or all the disciplines but the relative effort across these disciplines changes over time. Early iterations in a phase tend to focus on requirements and design, and later ones less so, as the requirements and design stabilises through a process of feedback and adaptation (Larman (2002).

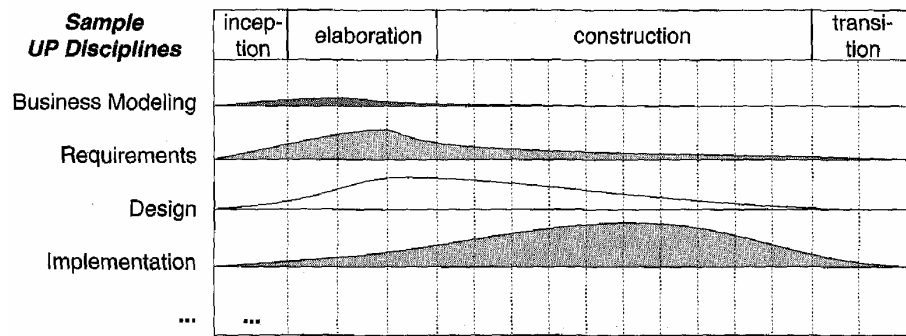


Figure 1.5: Changes in UP relative effort

Source: Larman (2002), p. 22

You have been alerted in the beginning of this section that each discipline has its own **artifacts**. In UP, artifact is the keyword used for any work product such as code, text documents, diagrams, etc. The artifacts chosen for a system development project could be written up in a brief document known as **Development Case**. The Development Case itself is an artifact of an Environment discipline. The following diagram shows the Development Case explaining the artifacts for the “NextGen POS” case study to be introduced at the end of this topic.

Table 1.3: Development Case

Discipline	Artifact	Incep. I1	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	s		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

s – start, r – refine

Source: Larman (2002), p. 24

In the diagram you can see some of the common artifacts used for the disciplines in UP. Please take note that different software or system development projects may need different artifacts. The focus of this course will be on the disciplines of business modeling, requirements and design. So, try to remember the artifacts used in these disciplines.



ACTIVITY 1.3

1. List three advantages of an iterative rather than a waterfall approach.
2. List three challenges associated with an iterative approach.

One of the most important practices of UP is **iterative development**. Another very important aspect of the UP is that it needs to be tailored or modified for every project. As an industry we now realise that no single method will work for every project in every organisation. For example, a small company building an intranet will need different approaches than a large company working on a multi-million dollar defence contract. This customised approach is both a strength and a weakness. It is a strength as it provides guidance, but it is also a weakness in that the project team still needs to decide how to use the methodology.

The UP makes extensive use of the Unified Modeling Language (UML). At the core of the UML is the **model**, which in the context of a software development process is a simplification of reality that helps the project team understand certain aspects of the complexity inherent in software. The UML was designed to help the participants in software development efforts build models that enable the team to visualise the system, specify the structure and behaviour of that system, construct the system and document the decisions made along the way. Many of the tasks that the UP defines involve using the UML and one or more models.

Now let us look at the notation that is used with the UP methodology.

1.2.4 Notation

In this course we will be using the industry standard notation called **Unified Modelling Language (UML)**. This standard was the result of the collaboration between three methodologists: Jim Rumbaugh, Grady Booch and Ivar Jacobson. What they published in 1994 was an amalgamation of their own notations plus the notations of several other authors.

UML defines standardised models and notations for expressing different aspects of an OO design. Version 1.1 was submitted to the Object Management Group (OMG) in 1997 and they published it as version 1.2. Version 1.4 is current. Listed are the other versions of UML:

- (a) UML 2.0 released in July 2005;
- (b) UML 2.11 released in August 2007; and
- (c) UML 2.12 released in November 2007 (latest version).

While this standard is not perfect for all situations, it does mean that all practitioners are able to read each other's diagrams. Most IT professionals use only a tiny part of UML. (Please refer to Figure 1.6)

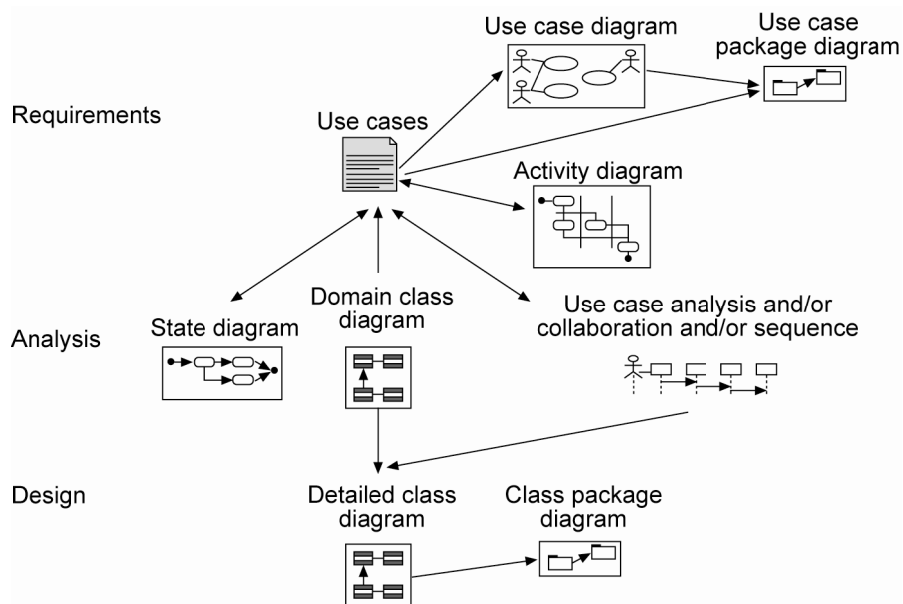


Figure 1.6: Overview of the main UML diagrams

In the next section you will be briefly introduced to you some of the UML notations to be used intensively in this course.

(a) **Use Cases**

A use case describes event sequences for an actor to use the system. It is a narrative description of the process. A use case is normally actor or event-based. An actor will begin a process or an event will happen that the system must respond to.

Use Case Example:**Borrow a Video**

The customer takes the videos he or she has selected and goes to the checkout counter. There the clerk confirms his or her membership and checks for any overdue videos. The total cost is calculated, the customer pays and leaves with his or her videos.

(b) Domain Model

Domain model is produced during the analysis and it is not a description of software objects. It is a visualisation of concepts in the real-world domain. It has identification of the concepts, attributes and associations as shown below:

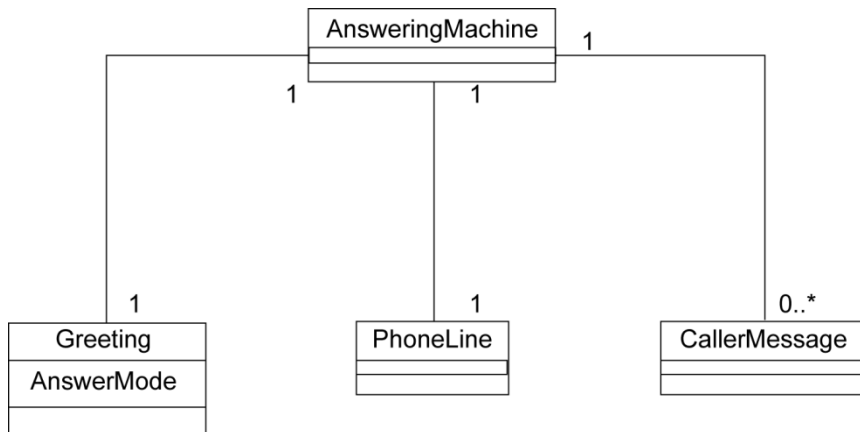


Figure 1.7: Domain model

Source:

http://www.apl.jhu.edu/Classes/605404/stafford/oointro/process_overview/sld004.htm

(c) Interaction Diagram

Interaction diagram is produced in design and is concerned with defining software objects and their collaborations. It has flow of messages between software objects and also the invocation of method.

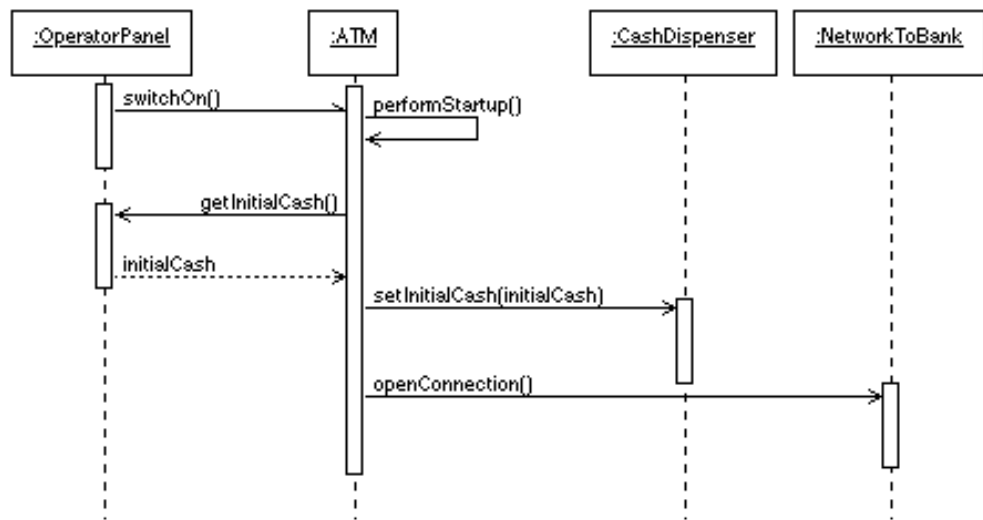


Figure 1.8: Interaction diagram

Source:

<http://www.cs.gordon.edu/courses/cs211/ATMExample/Interactions.html>

(d) **Design Class Diagram**

A design class diagram which is produced during design shows the static view of the class definitions. It has methods and attributes of classes. Unlike the domain model, this diagram does not illustrate real-world concepts, it shows software classes.

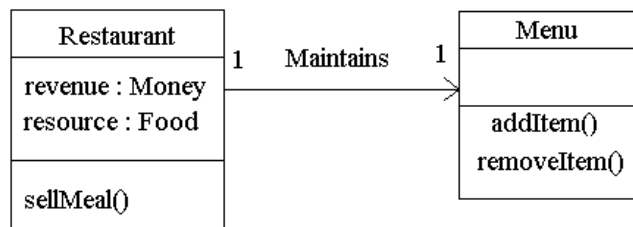


Figure 1.9: Design class diagram

Source: <http://www.comptechdoc.org/independent/uml/begin/umldcd.html>

As you work through this course you will see how the diagrams shown in this section fit together. Topics 2 to Topic 5 will look at these notations in detail.

1.2.5 Is Object-Oriented Systems Design an Adaptive Approach?

Now that you have read about the methodology and notation, what do you think? Is object-oriented systems design an adaptive approach?

You might first think that, since Fowler (2000) is quite dismissive of UML and is of the opinion that a systems design process that uses it cannot be adaptive. However, UML is just a notation; it is how it is used that counts. Fowler is discussing UML as used in formal methodology. You can elaborate UML by drawing a twenty-page model that gets formally reviewed. You can also use very sparse UML notation on a white board to quickly communicate a few ideas. This is quite likely to be the way it is used in the UP.

Note, too, that there is nothing intrinsically iterative about object orientation. The two approaches work very well together, but you could do non-object orientation development iteratively, as you can also do object-oriented development non-iteratively.

So while the concepts of heavy and light methodologies, predictive and adaptive processes are useful for understanding processes, you have to be careful not to take them too far.



ACTIVITY 1.4

Fill in the right-hand column of the following table.

Comparison of traditional and object-oriented approaches to system development.

	Traditional	Object-Oriented
Approach to data and functions	Consider separately	
System development life cycle (SDLC)	Waterfall	
Notations	Entity-Relationship (ER) diagrams Data Flow Diagrams (DFDs)	
Example languages	COBOL, various 4GLs	
Database	Relational databases	

1.2.6 Analysis and Design

Analysis and design are two major components involved when developing software using the object-oriented approach. What do the terms “analysis” and “design” mean? Simply, analysis is *what* needs to be provided and design is *how* it will be provided.

During object-oriented analysis, the emphasis is on finding the objects or concepts in the problem domain. For example, in the case of the Library Information System, some of the concepts include *Book*, *Library* and *Patron*. Object-Oriented Analysis (OOA), then, is the process of defining the problem in terms of objects: real-world objects with which the system must interact and candidate software objects used to explore various solution alternatives. The natural fit of programming objects to real-world objects has a big impact here: you can define all of your real-world objects in terms of their classes, attributes and operations.

During object-oriented design, the emphasis is on defining software objects and how they collaborate to fulfill the requirements. For example, in the case of the Library Information System, a *Book* software object may have a *title* attribute and a *getChapter* method. Object-Oriented Design (OOD), then, is the process of defining the components, interfaces, objects, classes, attributes and operations that will satisfy the requirements. You typically start with the candidate objects defined during analysis, but add much more rigor to their definitions. Then, you add or change objects as needed to refine a solution. In large systems, design usually occurs at two scales: architectural design, defining the components from which the system is composed and component design, defining the classes and interfaces within a component.

You will recall that in the waterfall model there was supposed to be a clear distinction between analysis and design. In the iterative model you may be wondering where the distinction lies. It is a very fine line, and during a project or even one conversation, you may find that you are doing both.

While it is useful to keep analysis and design separate when we are splitting a large problem into small pieces, it is also helpful to consider other ideas that might not fit too neatly in at present.

1.3 THE IMPORTANCE OF DESIGN

Our lives are filled with toys, tools and technology. Some of them are well-designed and easy to use. A good knife is well-balanced, sits easily in your hand and allows you to concentrate on what you are cutting. A well-designed website gives you the information you need in a minimum amount of time and fuss. Badly designed tools or user interfaces require effort to use and consequently one frequently makes mistakes or feels frustrated. A washing machine that has five buttons and a dial with 12 settings is far too complicated for just washing clothes! An application that requires me to enter a credit card expiry date of “02/06” instead of “0206” and does not explain the error is frustrating.

Some design is visible, some is hidden. Consider a car. In a car we can see and touch the driver’s controls. Is the steering wheel easy to grip? Can you quickly find the horn in an emergency? Other design elements are revealed through use. How long does the air-conditioning take to cool the car on a hot day? How well do the brakes work in the rain? Some design elements are internal so that only an expert will be able to appreciate the design and be able to predict the future. How does the engine wear over many years? How will the car perform in a crash? Nothing in a car is accidental; a person has designed everything. So it is with software. The external aspect is the user interface. The internals are concerned with reliability, ability to change and resilience to new technology.

Donald Norman in his book *The Design of Everyday Things* challenges us to design things – coffee pots, door handles, telephone systems and refrigeration controls – with people in mind (1989, 36):

If an error is possible, someone will make it. The designer must assume that all possible errors will occur and design so as to minimize the chance of the error in the first place, or its effects once it gets made. Errors should be easy to detect, they should have minimal consequences, and, if possible, their effects should be reversible.

Always remember that users of a system are people. Good design requires an understanding of the underlying basic principles and trade-offs. Good design requires experience. Good design is not about right or wrong, but about being better in certain circumstances than others. For all these reasons learning to design well can be frustrating. However, equally, when you have created a good design it is very satisfying.

1.4 GETTING STARTED: THE INCEPTION PHASE

The inception phase is where a project begins in UP. The primary goal of the **Inception phase** is to establish the case for the viability of the proposed system. It starts with somebody having an idea. Other people then need to be brought in – people who have appropriate authority, finance, knowledge or skills. They need to work together to form a common **vision** for the project. (The term “mission statement” could also be used.) A vision gives a picture of the end result without worrying about the components and processes. Often people are worried about committing themselves to paper too early. However, remember: this is **iterative development**. So we expect to see many versions of the same thing. The vision document can easily be updated every day during a two-week period as more is learned. In forming the vision you will need to enlist the help of all the **stakeholders** involved in the project such as CEO, IT director, board of directors, etc.

A vital part of the vision for the project is the **scope**. Exactly what will and what will not be in the system? If the scope is too large then most likely the project will fail. The scope needs to be as small as possible while still meeting the core needs. The 80/20 rule, or Pareto’s principle (named after a 19th century Italian economist), explains why every feature does not need to be built. This rule states that 80 per cent of the value of a group of items is generally concentrated in 20 per cent of the items. Examples include phone bills (80 per cent of the calls are only to 20 per cent of the people) and restaurant meals (80 per cent of orders are for 20 per cent of the dishes in the menu) and in software 80 per cent of the users of word processors use only 20 per cent of all the features.

Of course, the more stakeholders you consult the bigger the scope tends to be. However, it is crucial to firmly articulate and stick with the central vision. (One useful technique is to have an appendix to your vision of all the things that are *not* being currently considered, but could be later. This means that ideas are not lost, but neither is the vision).

The scope of a system is really the sum of its parts. Some of the parts provide functionality, like accepting a customer order and others are what we call non-functional requirements. These are things like speed of the system, quality and adherence to standards (Topic 2 has more on this topic).

Functional requirements are best described with **use cases**. Use cases were invented by Ivar Jacobsen, one of the three authors of UML. A use case is a description of some interaction of the system. It is written in simple everyday language, English, or whichever language the users are most comfortable with. It can be a simple, casual narrative describing how a user's goal is satisfied.

Here is an example of a use case. Suppose we were writing a system to rent out videos.

Borrow a video – use case – simple story

The customer takes the videos he or she has selected and goes to the checkout counter. There the clerk confirms his or her membership and checks for any overdue videos. The total cost is calculated, the customer pays, and leaves with his or her videos.

Always remember that use case describes the interaction between the user and the system. It does not tell us anything about how the system works. That is done later.

Do not worry if you are still blur with use cases. Topic 2 covers use cases in more detail – determining who the users are, working out their goals, using different use case formats, etc.

As far as the inception phase of the project is concerned, what we want is an initial list of the names of the use cases. We want to know if there are 5 or 50 use cases. This, in turn, will give us some idea of how long the project will take and hence the cost.



ACTIVITY 1.5

Write a use case for each of the following:

- (a) A student registers for an OUM course.
- (b) A customer purchases an item from an online shop.

1.4.1 Inception Phase Artifacts

During the development of a system, various models, designs, documents, reports, spreadsheets, manuals and programming code are produced. We refer to these as **artifacts**. A methodology usually gives guidelines on which artifacts need to be produced, when they are produced in the development life cycle, the format (drawing, document, spreadsheet, etc.) and possibly who in the team should produce them.

Be cautious when considering the artifacts for a particular project as every artifact adds to the total cost. For each artifact:

- (a) ensure that it has a real purpose, either now or in the future; and
- (b) establish how “good” it needs to be – again think of the 80/20 rule.

Table 1.4 contains a list of artifacts that the UP specifies for the inception phase. Remember that all these artifacts can start as very simple sketches or documents. Depending on the size and nature of the project they may evolve to be more detailed and/or more formal.

Table 1.4: Inception Phase Artifacts

Artifact	Comment
Vision and business case	This can start as a simple 1–2 page document, perhaps with some diagrams. <i>Topic 2</i> discusses other material that can be included.
Use case model	Discussed further in <i>Topic 2</i> . Really what is needed in the Inception Phase is a list of the main use cases that need to be covered, plus maybe some details for the key use cases.
Supplementary specification	This is anything that is considered necessary but not included elsewhere. More details in <i>Topic 2</i> .
Glossary	Every project has special words and terms that everyone needs to be familiar with. To save needless repetition, it is useful to have all the terms gathered in one place.
Risk list and risk management plan	Every project has risks. Risks can be categorised as technical, people or political. Some are common to every project, like key people leaving and others are specific to the project. The risk list and risk management plan can be a simple document that lists the risks and how they will be reduced. Regular review of risks is vital.

Prototypes and proof-of-concepts	Prototypes can be produced for a number of reasons. It could be to demonstrate what the user interface will look like, or it could be to demonstrate that the selected technology works and that 2,000 transactions can be stored in the database in under two minutes.
Iteration plan	Describes what to do in the first elaboration iteration. In iterative development, at the end of each phase or iteration you need to plan the next phase or iteration. The plan should include – how long the iteration will be, who will participate and what they will produce.
Phase plan and software development plan	At this stage of the project, you will have some idea of the people, resources and tools that you will need for the project. This needs to be listed. As always, remember that this plan can be changed as new information is learned.
Development case	This plan indicates what sort of process you are going to follow for the project. Is this going to be a project where every document needs to be signed off by the project's sponsor, or will you be using informal reviews? This is where you state what sort of artifacts you will be producing in this project.

(Adapted from Larman (2002), pp. 37–38)

SUMMARY

- This topic sets the scene for the rest of the course. Object-orientation allows us to combine data and functions when analysing real-world and software entities.
- Approaches to application development move from **waterfall** to **iterative development**.
- We introduced two standards: the Unified Process (UP) and the Unified Modelling Language (UML). UP is a methodology for iteratively developing object-oriented systems. UML is a standardised set of notations for modelling various aspects of an object-oriented system.
- System development is both an engineering and creative venture and there are many differing opinions on how best to proceed.

- We, then, commenced the inception phase of the case study by looking at the vision, the stakeholders and some initial plans.
 - Now we are going to introduce the case study used in this course.
-

Case Study 1: Victoria's Videos

(This is an introduction to Victoria's Videos, an example that you will use in many activities and discussions in the following topics.)

Victoria's Videos is a large video shop with thousands of members and videos. It is a single shop and Victoria has no plans to expand to multiple branches.

Victoria maintains a stock of approximately 50,000 videos – each year roughly a third of the videos are sold or destroyed and new ones are bought. Every video has a classification (general G, parental guidance PG, mature audiences MA, and restricted R). Members must be over 18 to borrow R videos. Every video also has a category: romance, general, sci-fi, foreign language and children.

You have to be a member to borrow a video. There are approximately 10,000 members and the number increases by approximately 2,000 each year, but some members have not borrowed for years. When a new person requests to become a member, they must show their driver's licence or other photo ID. The minimum age is 16 years old. A member can borrow any number of videos, as long as they have no overdue videos, and it is also possible to reserve a video. There are fines for overdue videos.

The length of time that a video can be borrowed depends on the video. New releases are only lent out overnight, current releases are for three-day hire and the rest are for a week.

Members' birthdays are marked with a special letter inviting them to borrow any video of their choice for a week for free.

Every three months the shop does a stock take. Any missing videos are updated to show them as "missing".

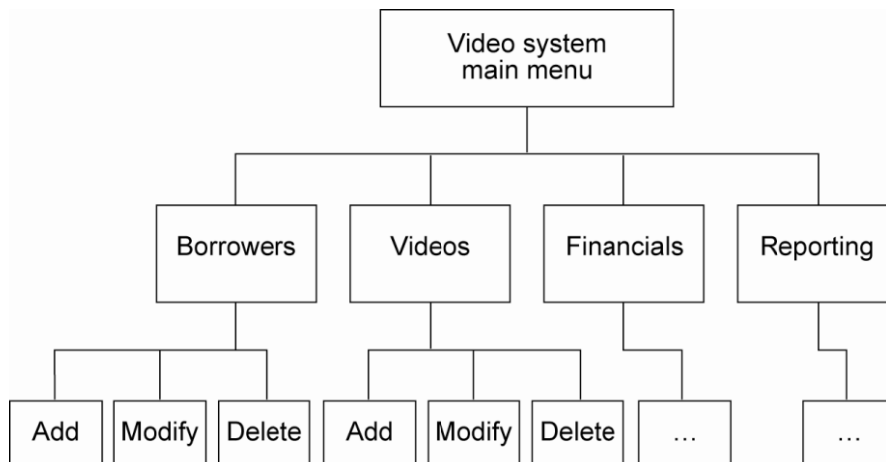
The system Victoria has three very good intelligent cash register terminals to handle all the financial transactions. These terminals handle the credit card facilities, cash reconciliation, banking and interface with the accounting system.

However, the current computer system runs on very old hardware (Windows 95) and is written in an obscure programming language that nobody understands. It is a very slow and unfriendly system that requires a fair bit of training for new staff. So Victoria has decided to invest in a completely new system with current hardware, operating system and software. Hardware and operating system decisions have not been made.

While she is doing this, she wants to add a data warehouse so that management can get various statistical reports. For example, they would like to see which types of videos are most popular or unpopular, who are the best customers, how many overdue videos there are, etc.

In addition, in the current system a barcode reader is required to scan the videos, and members need to present their ID cards to take the video out. This causes problems when members forget to bring their cards, so Victoria would like to explore other options.

The following is the functional decomposition of the targeted new system for Victoria's Video:



As there are always new staff members joining the shop, the new system has to be easy to learn and use.

Case Study 2: NextGen POS System

(This is an introduction to the NextGen POS System, an example that you will use in some of the activities and discussions in the following topics. This case study was adapted from Larman (2002), pages 29–31.

The case study is the NextGen point-of-sale (POS) system. In this case study, we shall see that there many interesting requirement and design problems to solve. This is a realistic problem as organisations really do write POS systems using object technologies. A POS system is a computerised application used (in part) to record sales and handle payments and it is typically used in a retail store. It includes hardware components such as a computer and a bar code scanner and software to run the system. It is interfaced to various service applications such as third party tax calculator and inventory control. The system must be relatively fault-tolerant. A POS system must increasingly support multiple and varied client-side terminals and interfaces. These include a Web browser, a regular PC which supports Graphical User Interface (GUI), touch screen input, wireless PDAs and so forth.

Furthermore, we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing. Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added. Therefore, we will need a mechanism to provide this flexibility and customisation. Using an iterative development strategy, we are going to proceed through requirements, object-oriented analysis, design and implementation.

KEY TERMS

Attribute	Object
Class	Polymorphism
Construction	Stakeholder
Elaboration	Subclass
Inception	Superclass
Inheritance	Transition
Instance	Unified Modelling Language (UML)
Iterative development	Variable
Message	Waterfall
Method	



REFERENCES

- Cockburn, A. (1998). *Surviving object-oriented projects: A manager's guide*. Boston, MA: Addison-Wesley.
- Constantine, L. (1995). *Constantine on peopleware*. Englewood Cliffs, NJ: Yourdon Press.
- Fowler, M. (2000) Put your process on a diet. *Software development magazine*, December. Retrieved from <<http://www.sdmagazine.com/documents/s=737/sdm0012a/0012a.htm>>.
- Krutchén, P. (1999). *The rational unified process: An introduction*. Reading, MA: Addison-Wesley.
- Larman, C. (2002). *Applying UML and patterns*. Upper Saddle River, NJ: Prentice Hall
- Norman, D. (1989). *The design of everyday things*. New York, NY: DoubleDay/Currency.
- Object Management Group. Retrieved from <http://www.omg.org>
- Taylor, D. A. (1998). *Object technology: A manager's guide* (2nd ed.). Reading, MA: Addison-Wesley.

Topic 2 ► Requirement and Use Cases

LEARNING OUTCOMES

By the end of this topic, you should be able to:

1. Differentiate between different types and levels of requirements;
2. List standard non-functional requirements;
3. Recognise that change in requirements is inevitable and outline strategies for dealing with such change;
4. Identify the actors for a system;
5. Define a use case, levels of use cases and describe how use cases fit with goals;
6. Identify the use cases for a system;
7. Draw a UML use case diagram to show the scope of a system; and
8. Draw a correct UML activity diagram for a set of use cases.

► INTRODUCTION

This topic is all about finding and then recording and analysing the requirements for a system. Badly defined requirements are a major cause of system failures. The requirements for a system cover a large spectrum, from business needs to specific technologies as well as what the system “must do”, such as calculate tax. This topic starts with a discussion of the different levels and types of requirements. We then focus on what the system “must do”, or its **functional requirements**.

Functional requirements are described by **use cases**. Use cases are the core of object-oriented analysis and design. Consequently the bulk of this topic is about writing use cases, which you were briefly introduced to in Topic 1. Use cases look simple – very simple – until you try to write some. The activities for this topic will help you practise this skill.

In addition to use cases, you will learn about three UML diagrams that are used in conjunction with use cases. Use case diagrams give a pictorial view of the actors and use cases. Activity diagrams show how some use cases interact to achieve a business task or process. Use case package diagrams to logically group the use cases to make it easier to understand the big picture.

Throughout the whole system development process we accept that requirements will change in response to various needs. The UP addresses this by using an iterative development methodology. This topic concludes by looking at the UP requirements documents and artifacts.

2.1 REQUIREMENTS ANALYSIS

As you learned in Topic 1, there are several disciplines (or workflows) within each of the four phases of the (UP). One of the earliest disciplines – and one of the most important – is requirements analysis. Obviously, before we do anything at all, we have to ask ourselves what we want the system to do.

To answer this question, it is necessary to consider:

- (a) business needs;
- (b) available resources;
- (c) possible technologies; and
- (d) social and legal implications.

The problem to be solved is discussed among team members, customers and typically users. Assumptions are expressed but may be verified or rejected using “proof of concept” prototyping techniques.

The requirements for the system are the outcome of this process and represent an agreement between the system development team members, customers and users. Note that the requirements will tell *what* the system will do, rather than *how* it will do it. Writing the requirements involves:

- (a) Defining the purpose of the system, prioritising its functionality and specifying its context (that is, who are its users and what systems does it interact with?);
- (b) Identifying external interfaces, both human and system-to-system. The technology of an existing system with which the system must interface can constrain the requirements and design;
- (c) Identifying major functionality that must be provided by the system and describing it;
- (d) Documenting assumptions upon which the requirements are based. If the assumptions change, then so might the requirements; and
- (e) Communicating with users, clients, business analysts and developers so that the system that is developed meets the clients' expectations.

The requirements may also involve “proof of concept” prototypes, which early on in the life cycle of the development process confirm or shape the direction of the solution. The prototypes can be an implemented mock up of some function in the system, or could simply be sketches of the GUI through which scenarios can be navigated and confirmed.

2.1.1 Levels and Types of Requirements

An organisation might start a new system development project for a number of reasons: new business requirements, replacing an old system, the merging of two existing systems due to acquisition and merger, etc. In most cases, systems development activities are driven by business needs. Hence, you need to understand the different levels of requirements for the system you are going to develop. These are:

- (a) Business requirements – these define the high-level processes that occur in an organisation. For example, in a banking system, what is the purpose of developing a new banking system? Is it going to replace an old system or is it going to be used in a new line of business, etc.?

- (b) System requirements – what the computer system must do for its users. These are the **functional** requirements of the system.
- (c) Internal requirements relating to technology, personnel, hardware platform requirements, etc. These are the **non-functional** requirements.

To explain the differences between functional and non-functional requirements, consider the example of building a house. Imagine that you have inherited a piece of land and that you want to build a house on it. So you contact an architect and sit down to discuss your requirements. How many rooms do you require? Do you want the dining room next to the kitchen? How big should the garden be? We can consider these as the **functional** requirements of the house. There are other requirements too – such as the quality of the bathroom taps, amount of natural light and the total budget for the project. These are **non-functional** requirements. From the system development perspective, non-functional requirements can be further divided into different categories. In the UP context, requirements are categorised according to the FURPS+ model which refers to: *f*unctional, *u*sability, *r*eliability, *p*erformance, *s*upportability and “+” for everything else (such as implementation, operations, packages, legal, etc.) as depicted in Figure 2.1.

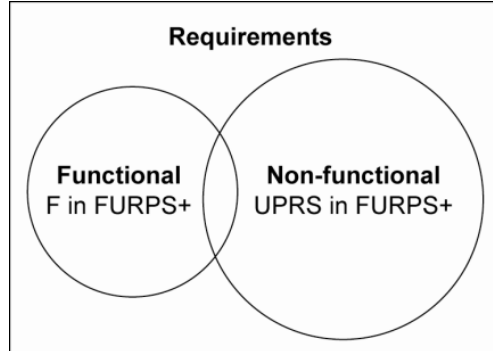


Figure 2.1: FURPS+ requirements model

Cockburn (2002) suggests that only a third of the requirements are functional. Some requirements can be regarded as both functional and non-functional.



ACTIVITY 2.1

Characterise the following requirements as functional, non-functional or both.

- (a) Customers receive a special discount on their birthdays.
- (b) Use the Java J2EE architecture.
- (c) Each POS terminal is able to process 100 sales items per minute.
- (d) Produce a report on demand of all transactions greater than \$100,000.
- (e) Each user is able to see personalised menu options.



ACTIVITY 2.2

Victoria's Videos System

In this topic, we will use Victoria's Videos system, which you were introduced to in Topic 1 as the basis for some activities.

Have another look at the case study, which was included in *Topic 1*, and list two functional requirements and two non-functional requirements for the system.

2.1.2 Challenges of Writing Requirements

Experienced systems developers will tell you that one of the main causes of project failure is poorly conducted requirements analysis. Writers of requirements face a number of challenges, not all of them technical. Communication, human and business issues also have to be taken into account. The following are possible challenges:

(a) Problem Domain Complexity

The biggest challenge is finding out about the problem domain (the environment in which the system will operate, for example, our video shop). This is especially true if the domain is a new one. Other domain-related challenges include: ascertaining the system's role within the domain, achieving immersion in the problem domain, translating what you learn

about the domain requirements specifications and system responsibilities. Obviously, the bigger and more complex the system required, the more challenging your task.

(b) **Person-to-person Communication**

You need effective communication in order to extract information from clients and users about the problem domain, to obtain information from clients and users about the system requirements, to convey your understanding of the system back to clients and users for their affirmation, to convey requirements information to developers, managers and testers, and finally to convey the need for changes to the requirements which may have been established. For successful requirements analysis, groups involved need to convey information to one another effectively.

(c) **Constant Change**

As we have already mentioned, changing requirements is an inevitable aspect of the software development life cycle. Even though requirements may be frozen at a particular point, a system and people's understanding of it will continue to evolve. This evolution may be due to improved understanding of the problem domain by analysts and developers, improved understanding of the system solution on the part of users, or simply competition, regulators, approvers, technologies or politics. Recognising that requirements will always change is a core concept of iterative development.

(d) **Writing Requirements in UP**

As you know, the goal of requirements analysis is to describe what the system should do (the requirements) and get developers and the customer to agree on that description. To achieve this, we define the system and its surroundings and the behaviour it is supposed to perform. Customers and potential users are important sources of information.

In the UP development framework, the requirements analysis is progressively presented in two key artifacts, the **Use Case Model** for the functional requirements and the **supplementary specification** for the non-functional requirements (refer to Figure 2.2).

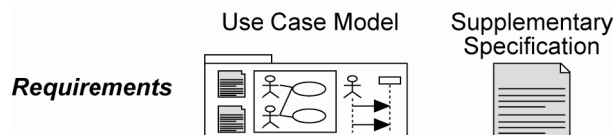


Figure 2.2: The key requirements artifacts

As noted in Topic 1, the Use Case Model is important for both the customer, who needs the model to validate that the system will become what is expected and for the developers, who need the model to get a better understanding of the requirements for the system. A Use Case Model is written so that both the customer and the developers understand it.

The Use Case Model consists of actors and use cases. **Actors** represent the entities external to the system, either users or other systems. **Use cases** represent the functional behaviour of the system. Actors help define the system and give you a clearer picture of what it is supposed to do. We will look at actors further on in the next section.

As you know from Topic 1, a use case represents events triggered by actors and describes how the system must respond. As use cases are developed according to the actors' needs, the system is more likely to be relevant to the users.

Figure 2.3 shows the main steps of the iterative process of building and delivering some functionality. Use cases live through the entire system life cycle and act as a unifying thread. The same Use Case Model is used during all subsequent workflows. This topic covers the first three steps in more detail. Step 4 is covered in later topics.

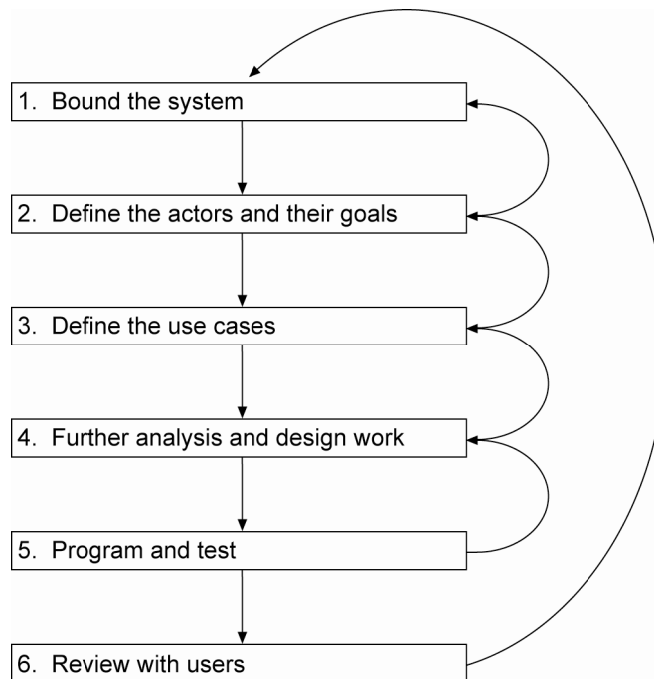


Figure 2.3: The main steps

2.2 STAKEHOLDERS, ACTORS AND ROLES

In Topic 1 we noted that stakeholders are all the people or groups, both internal and external to your organisation who will affect or be affected by your proposal. Obviously this includes the people who will use the system. In UML, the term for users is **actors** but this word can lead to confusion. We do not say a person is the actor Hamlet; we say that an actor plays the role of Hamlet.

Another way of thinking about roles is wearing different hats. One person can wear different hats, or play many different roles as they go through a day or a process. One example is creating a big document or report. Initially a person, let us call her Alice, brain dumps ideas onto the page as quickly as possible. She wants to create a rough structure. She is not interested in spelling, grammar or fonts. After a few hours of this she moves to another role – a writing role. Here she types whole sentences. She includes graphics and headings. She is concerned about correct spelling, grammar and fonts. To complete the document, she asks her colleagues Ahmad and Bernard to review it. They need to add comments to a document without affecting the original text. On paper this is scribbling in the margins of a printed copy. Lastly, Alice takes comments from all different people to produce one final document (refer to Figure 2.4).

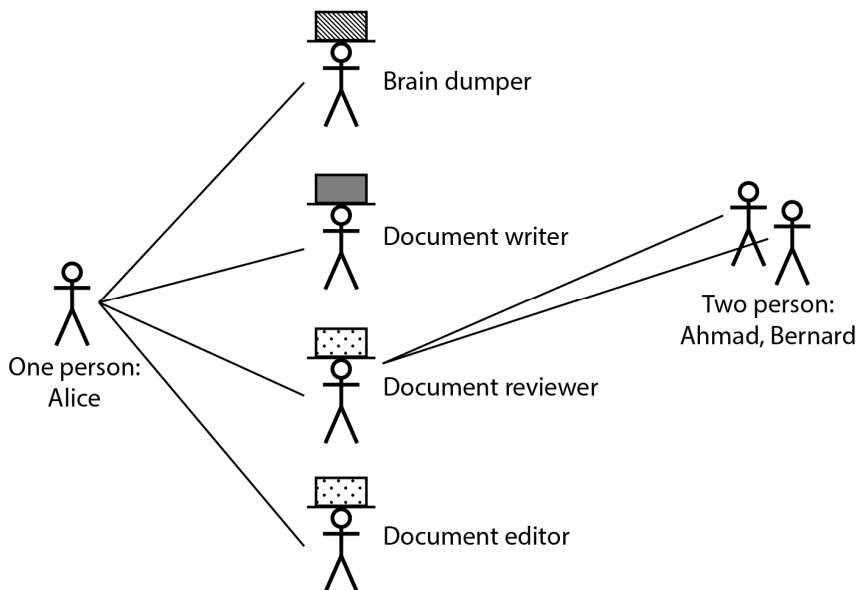


Figure 2.4: People and roles – “wearing hats”

2.2.1 Types of Roles

There are three kinds of roles or actors:

(a) **Primary**

A person in this role directly uses the system to enter, view process or extract information. Depending on the user interface, these people will use a keyboard, a mouse, buttons on a machine or talk to a voice activated system. Examples in the case study are the check out clerk and backroom stock handler.

(b) **Secondary**

A person in this role does not directly use the system, but uses or generates data for the system. A classic example is a person who receives reports from a system, particularly in another department in the organisation. This person may or may not access the system to get the report, but is a user of the information. (Typically, these roles are overlooked in the requirements analysis.) An example in the case study is the managers.

(c) **External**

Any system that receives or generates data for the system – these are external systems, for example, tax office systems, banks, accounts systems, etc.

Note that this taxonomy differs slightly from UML and in Larman (2002). In particular, in UML **users** and **external systems** are treated the same. It is true that they are both external to the system, but treating people and computers in the same way is not conducive to building **usable** systems. Furthermore, our experience shows that frequently the needs of the secondary actors are often omitted. Often the need for reporting is known, but instead of the needs being closely examined, the standard response is to simply add an end-user reporting tool. Very rarely does this simplistic approach satisfy. If senior management does not receive any benefits from a system then the project may be terminated.

2.2.2 Differences in Roles

Questions you need to ask of a person in a role are:

- (a) Do they know how to use a computer? Are they comfortable with a graphical user interface?
- (b) Do they understand the domain? (For the video shop domain, does the person know how videos are borrowed, do they know how the credit card charging system works or are they a member of the public?)

- (c) Will they receive any training in using the new system?
- (d) What is their work environment? Is there noise or sufficient light? Are there many interruptions from other people? Do they need to be able to switch between many applications? How reliable is the network?
- (e) Frequency of use. Will they use the application once a year, once a month or seventy times per day?
- (f) In the course of their work do they refer to paper forms or other data sources?
- (g) Ask them to rank one or more of the following: reliability, correctness, satisfaction, ease of use.



ACTIVITY 2.3

Here are two very different roles for two very different systems.

Clerk for a small investment company. This company has only 30 clients and there are only a few transactions per day, but the amount of money in each transaction can be large.

Teenager at a Kylie Minogue fan club website. This website has photos, lyrics and samples of Kylie's songs. There are facilities for online discussion and chats.

For each, consider what would be important to a person in that role and assign a priority (1, 2, 3 or 4) to each. You cannot make them all priority 1!

Objective	Clerk for Investment Company	Teenager at Kylie Minogue Fan Club Website
Reliability of the system, for example, the results of searches or calculations are correct		
Efficiency, for example, that a minimum number of steps is required, thus reducing human error		
Satisfaction, for example, the experience of using the system is enjoyable		
Speed, for example, the rate at which information is displayed		

2.2.3 Discovering the Roles

There are two simple ways to start discovering the roles for a system. These are brainstorming an initial list and looking at existing job titles.

(a) **Brainstorm an Initial List**

Either by yourself or with other team members, simply write down the names you can think of. (When brainstorming do not be concerned with getting the best names for the roles – that comes later.)

(b) **Look at Existing Job Titles**

This is a good place to start, but the role names may end up being quite different to the existing job titles because:

(i) A job description normally requires several roles. For example, the job title “technical writer” can be split into the roles:

- document searcher;
- document reader;
- document writer; and/or
- document reviewer.

Each of these roles would have a number of use cases.

(ii) Existing job titles are not usually good role names because they already have a lot of meaning to the users – either there is a history as to what that the job entails or the job has other components which are unrelated to the new system.

(iii) There is a many-to-many relationship between job titles and use cases or tasks. Thus, a job title can be misleading.

Warning: If you take the role analysis too far, then you will end up with a different role for every use case. This is the opposite of having one role, called “user” who does everything. Somewhere between these two extremes is a useful and workable set of roles. It sometimes takes a little while for this to come together. So start with an initial list of roles and work on identifying their use cases. Then come back and review the role model.

Table 2.1 may be useful for collating the actors.

Table 2.1: Sample Table for Collating Actors

Actor Name	Actor Type	Description	Current Job Title(s)	Importance of the Actor
This name needs to be as meaningful as possible.	Primary, secondary or external	A sentence or two to describe the actor in some detail	This is helpful in communicating to staff. There could be multiple job titles per actor, or one job title can have many actors.	Low, medium or high

Lastly assign an importance (high, medium, low) to each actor. This will determine the order in which you start looking for use cases. (Of course, as with everything, once you look more closely at it, the importance may change).

For the NextGen POS case study introduced in Topic 1 an **initial** list of actors could be as in Table 2.2.

Table 2.2: NextGen POS Initial List of Actors

Actor Name	Actor Type	Description	Current job Title(s)	Importance of the Actor
Cashier	Primary	Sells the goods to the customer and collects payment	Shop assistant Duty supervisor	High
Barcode scanner	External	** Not sure if this is part of the system, or an external actor	Not applicable	Medium
Administrator	Primary	Not much information yet	?	Medium
Inventory control	External		Not applicable, but talk to warehouse manager	Medium
Analyst	Secondary	** Further investigation required	Store manager	Medium
Accountant	Primary	** Further investigation required	Accountant	Medium
Tax calculator	External	Calculate the tax owing	Not applicable	Low
...				

You will notice that this list has questions and indicates where more work is required. This is normal.

Subsequent work on the analysis is certain to change some of this information, but it is a good starting point. Notice that the barcode scanner has been included, but the description is a question to be resolved. This is fine. Remember that iterative development is all about starting and continually refining and building on previous work.



ACTIVITY 2.4

Identify possible actors for Victoria's Videos system and categorise them as primary, secondary or external.

2.3 WRITING USE CASES

We already have a good understanding of actors (as well as the roles actors will play) in the Use Case Model. Actors are entities outside of the system that will interact with the system. What is inside the system is represented by use cases in the Use Case Model. Use cases are a way to discover and document functional requirements. They describe system behaviour from the user's perspective. Use cases are written in plain language so both the users and the IT staff can understand them. Each use case must achieve something useful for the user. The next reading gives an example of a brief use case.

Use cases define a promise or contract of how a system will behave (Larman, 2002).

There are three types of format for writing use cases. The format to be considered depends on the need. The explanation for these three types of use cases format are given as follows:

- (a) **Brief use case** – it has one-paragraph summary and normally used for main success scenario. Here is an example of brief format use case for *Process Sale* (Modified from Larman, 2002):

Process Sale:

1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total.
5. Price calculated from a set of price rules.
6. System presents total with taxes calculated.
7. Cashier tells Customer the total, and asks for payment.
8. Customer pays and System handles payment.
9. System logs completed sale and sends sale and payment information to the external accounting system (for accounting and commissions) and inventory system (to update inventory).
10. System presents receipt. Customer leaves with receipt and goods (if any).

- (b) **Casual use case** – it is an informal paragraph which has multiple paragraphs that cover various scenarios. The Handle Returns shown in Reading 2.2 was the example.
- (c) **Fully dressed** – the most comprehensive that shows all the steps and variations. It has supporting sections, such as preconditions and success guarantees. Example of fully dressed use case for NextGen case study could be found at http://www3.itu.edu.tr/~buzluca/ymt/ornek_uc.pdf. However, detail discussion about fully dressed use case is beyond the syllabus.

Writing a use case is, to a certain extent, is like writing a report, and as you will know if you've ever written a report, it's much easier if you can follow some kind of format. Most companies and organizations have report templates for people to refer to. Similarly, for use cases there are templates that system analysts can refer to. One of the most popular use case templates was developed by Alistair Cockburn. The full version and a detailed description of the template are available at <http://members.aol.com/acockburn/papers/uctempla.doc>. Use case templates are not covered in this course.



ACTIVITY 2.5

1. Write the use case “borrow videos”.
2. Why do you think this is a good use case to begin with?

In Exercise 2.5 we gave you the name of a use case, but of course, normally you have to work out the use cases yourself. Deciding where one use case ends and another use case begins takes a bit of practice. When you first start writing down possible use cases do not be too concerned about getting this right. Once you get further into writing the use cases you will be able to work out the boundaries.



ACTIVITY 2.6

In Activity 2.4, you listed the actors for Victoria’s Videos. Now you need to work through the list of actors, from most important to least important, and list their use cases. Do not be too concerned about getting the names of the use cases perfect. Just write them down. Later you can change the names, delete use cases or add use cases.



ACTIVITY 2.7

Using your list from Activity 2.6, select the next most important use case of the most important actor and write a brief format of the use case.

Now select another use case and also write it in a brief format.

If all system development activities are business-driven, those writing the requirements must aim for goals that will bring value to the organisation. This means that a use case:

- (a) must deliver something of value to an actor;
- (b) typically represents a major piece of functionality that is complete from beginning to end;
- (c) represents the goal of an interaction between an actor and the system; the goal represents a meaningful and measurable objective for the actor; and
- (d) records a set of paths (scenarios) that take an actor from a trigger event (start of the use case) to the goal (**success scenarios**).

- (e) Records a set of scenarios that traverse an actor from a trigger event toward a goal but fall short of the goal (**alternate scenarios**). The format to write an alternate scenario is as below:

condition:
handling

Example of alternate scenarios for *Process Sale* are given below (Larman, 2002) based on the above format:

- 3a: Invalid item identifier found: ◀---- *condition*
1. System signals error and rejects entry ◀---- *handling*
- 3b: There are multiple of same item:
1. Cashier can enter item category identifier and the quantity

Write the condition as something that can be detected by the system or an actor. The above examples only have one *handling* for the *conditions*. There can be more than one *handling* for certain *conditions*.

Remember that an use case may have a main success scenario and some alternate scenarios. Sometimes, when we have a large and complicated system to develop, there will be lots of different scenarios. It would be impossible to discover all the possible scenarios at the very beginning. Remember, in the UP, everything is done in an iterative fashion. Hence, when you first start writing use cases for a system, it is best to ignore all the alternate scenarios and only think about everything going well – the “sunny day scenario”. Alternative scenarios can be added in future iterations. Of course, successful scenarios can also be revised and changed in later stages.



ACTIVITY 2.8

We have written up a main success scenario for the “borrow videos” use case in Activity 2.5. Try to think up a few alternate scenarios and present them in a brief format similar to the main success scenario. Compare your answer with your course mates in myVLE.

The main purpose of the UP is to help system developers to build their systems in a systematic manner. Poor system development tends to be done by developers who cannot judge what they should be doing at each stage of the whole development cycle. A common mistake is to jump into the system design before a good requirements analysis is available. Hence, in the UP framework, a black-box use cases approach is usually adopted to help system developers avoid making the same mistake twice. Let us now discuss the term. Something is a black box if we cannot see the inner workings of it. For example, from a car driver's perspective, a car is a black box. The car either moves properly or it does not. If there is a problem, then the car mechanic comes and, with a white-box view, lifts the car bonnet and looks at the engine and internals. Hence, in the black-box approach of use cases writing, we do not care about how the system will perform the tasks but just focus on what we should expect from the system. In other words, we are only focusing on the responsibilities of the system. An example of black box use case is shown below:

Black Box Style	Not Black Box Style
The system updates the student marks	<p>The system writes the new marks to a database</p> <p>The system generates a SQL UPDATE statement for the task</p>

Sometimes confusion will still arise even if we adopt the black-box approach, because different people have a different understanding of “what the system does”. For example, a cashier might think of her goal as “logging in”, but in fact this is just the mechanism that achieves the higher-level goal of identifying and authenticating herself.

When writing use cases, keep the user interface out and focus only on actor intent.



ACTIVITY 2.9

Now take the three use cases from Activity 2.5 and 2.7 and rewrite them in the essential two-column format. (Yes, writing use cases is hard work, but the more you write the easier it gets. It is not good enough to just look at the answers. You must practise). Anyway, the answer can be obtained at the end of the module.

2.3.1 Goals and Levels of Use Cases

Now you have an idea of what a use case is and, in general terms, know how it should be written. Now, let us stand back a bit and ask a fundamental question: What should the goal of the use case be? As we noted earlier, different users have different goals. For example, the general manager of the retail store is expecting the retail system in his company to streamline the whole retailing process. The cashier, on the other hand, may expect the same system to read the barcodes of items correctly and efficiently. Well, which one of these two requirements should be turned into a use case? In fact, neither of them would make a good use case. The general manager's goal is at too high a level while the cashier's goal is too low. So what level of user goals can be transformed into use cases? The general guideline is to look at **Elementary Business Processes (EBPs)**. Reading 2.1 will explain to you the concept of EBP. It also gives a very lively example of how a system analyst investigates the user goal of the cashier through a series of questions and identifies the appropriate goal level for writing a use case.

Now download **Reading 2.1** from myVLE to learn about goals and scopes of use case.

Information systems are rather abstract entities that consist of software, hardware and applications. However, even though we cannot visualise the whole information system, for the purpose of use cases, we need to define a boundary for it. It is similar to building a house, when we need to perform a land survey to fix the boundary of the site on which the house is going to be built. A clearly defined system boundary helps in the correct identification of the right actors to use cases, which in turn, helps the system analyst to obtain the goals. The four-step procedure proposed to define use cases is listed below:

- (a) define the right system boundary;
- (b) identify the primary actors;
- (c) identify the actors' goals; and
- (d) define and name the use cases.

Now download **Reading 2.2** from myVLE to learn about the four-step procedure to define use cases as listed above.

**ACTIVITY 2.10**

For Victoria's Videos, up to now we have tried to identify the actors and some brief use cases. However, all these were produced by our imagination. In a real system, the proper way to do requirements analysis is to carry out a combination of many activities such as:

- (a) Read any documentation that has been collected;
- (b) Look at any systems that currently exist. Look at screens and reports. See if you can track some real examples all the way through. For example, for an insurance system, you could see how a policy is created, and then later how a claim might be processed; and
- (c) Talk to the parties concerned. Commonly you begin by conducting some user requirements workshops. Use the initial actor analysis work as a guide to who should attend the workshop. For the first workshops it is best to have more people rather than less as you need the broadest possible range of views. Later, you can have much smaller workshops to concentrate on particular areas.

Prepare a list of questions that you would like to ask various people from Victoria's Videos.

It is important that you understand the Use Case Model as it is a crucial element in Object-Oriented System Analysis & Design (OOSAD) and it is one of the earliest tasks in the whole system development project. The future success of the project depends on getting the use case right. As it is sometimes helpful to approach a topic from a different angle, we would like you to reinforce your understanding of the Use Case Model by working through the next reading by Cockburn:

<http://alistair.cockburn.us/structuring+use+cases+with+goals>

2.3.2 Format for Writing Use Cases

There are three types of format for writing use cases. The format to be considered depends on the need. The explanation for these three types of use cases formats are given below:

- (a) **Brief use case** – it has a one paragraph summary and is normally used for main success scenarios such as the *Process Sale* example shown in Reading 2.1.
- (b) **Casual use case** – it is an informal paragraph which has multiple paragraphs that cover various scenarios. The Handle Returns shown in Reading 2.2 was an example.
- (c) **Fully dressed** – the most comprehensive format that shows all the steps and variations. It has supporting sections, such as preconditions and success guarantees.

Writing a use case is, to a certain extent, like writing a report and as you will know if you've ever written a report, it is much easier if you can follow some kind of format. Most companies and organisations have report templates for people to refer to. Similarly, for use cases there are templates that system analysts can refer to. One of the most popular use case templates was developed by Alistair Cockburn. The full version and a detailed description of the template are available at <http://alistair.cockburn.us/Basic+use+case+template>. Use case templates are not covered in this course.

The following e-book chapter from OUM's digital library (Books24x7) contains more explanation and "Soda Machine" case study on use cases. You are required to read this chapter.

Schmuller, Joseph. "Hour 6 - Introducing Use Cases". Sams Teach Yourself UML in 24 Hours. Sams. © 1999. Books24x7. <http://common.books24x7.com/toc.aspx?bookid=414>

We will revisit the "Soda Machine" case study again in Topic 4 and Topic 5

2.3.3 Use Case Diagrams

"A picture paints a thousand words": sometimes a visual illustration can represent complicated ideas in a simple and easy way. A **use case diagram** can provide a static view of a system to allow us to have an overview of the system and the relationship between the use cases as well as the actors of the system.

Figure 2.5 is the simple UML notation to depict a use case.

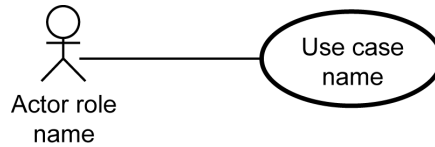


Figure 2.5: Use case diagram notation

A collection of use case diagrams will form a context for use case diagrams that shows the scope of the system and the relationship between the actors and use cases. It serves as a diagrammatic summary of the behaviour of the system. According to Larman (2002), use case diagrams are only secondary in use case writing. System developers should not spend too much time drawing use case diagrams. Writing use case text should be the primary focus.

Now download **Reading 2.3** from myVLE to learn about use case diagram in detail.



ACTIVITY 2.11

By using the example of use case diagram shown in Figure 2.5 of Reading 2.3 as a guide, draw a use case diagram for Victoria's Videos.

2.3.4 Activity Diagrams

For many applications, it is extremely important to see how all the use cases fit together to achieve something overall. This is particularly true when there are many different people involved in an overall activity. For example, one of the main challenges to businesses that sell products via the Internet is to work out the whole fulfilment side. Taking the order and the money is the easy bit! UML has a notation called activity diagrams. These diagrams, in many aspects are quite similar to flow diagrams. They can be used at different stages in the development, from requirements through to detailed programming.

The following diagram shows how the different actors invoke their own use cases to achieve the overall task. The diagram has "swim lanes" for each of the different actors.

Each system is different so you have to decide whether or not activity diagrams are useful or not.

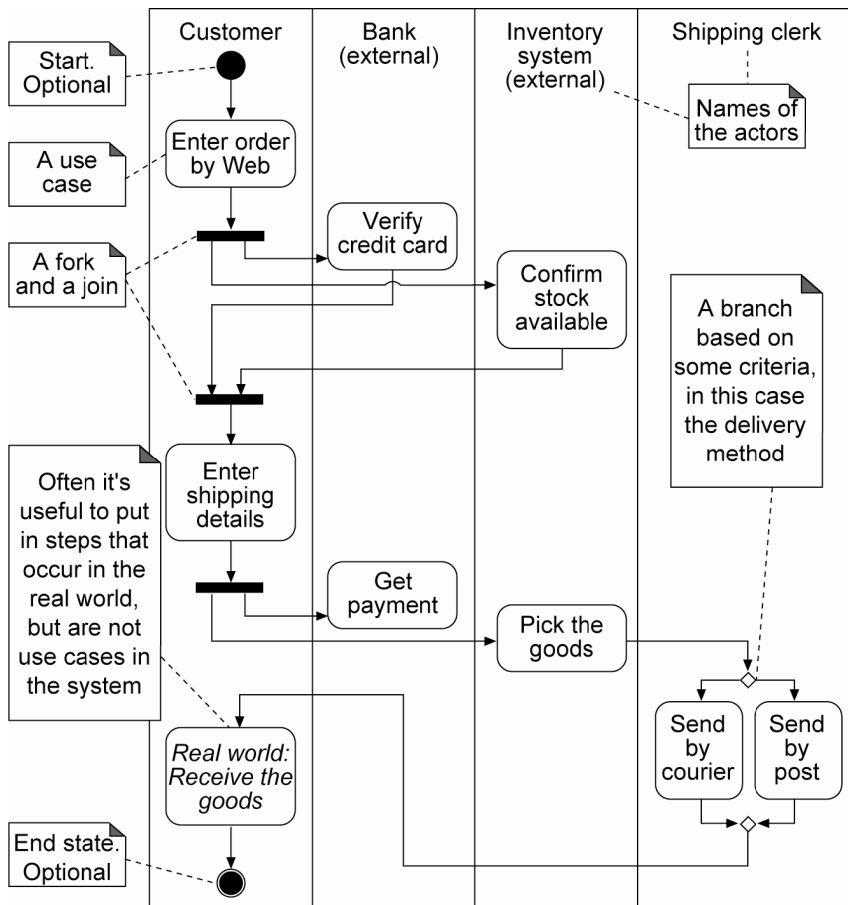


Figure 2.6: An example of an activity diagram



ACTIVITY 2.12

Do you think an activity diagram would help explain how a *subset* of the use cases for Victoria's Videos system fit together to form a whole? If so, then draw it. Note: We are not trying to draw *all* the use cases of the whole system, but only some of them that make up a process that makes sense to the users as a whole.

2.3.5 Potential Problems with Use Cases

Writing effective use cases is not a simple matter and there are a number of pitfalls to be avoided. Here are ten, identified by Lilly (1999):

- (a) The system boundary is undefined or inconstant;
- (b) The use cases are written from the system's point of view;
- (c) The actor names are inconsistent;
- (d) There are too many use cases;
- (e) The actor-to-use case relationships resemble a spider's web;
- (f) The use-case specifications are too long;
- (g) The use-case specifications are confusing;
- (h) The use case does not correctly describe functional entitlement;
- (i) The customer does not understand the use cases; and
- (j) The use cases are never finished.

In writing this list, Lilly (1999) is not arguing against use cases, but trying to alert newcomers to the potential problems and suggesting ways to avoid them. She advises:

- (a) Explicitly define the system's boundary. This makes clear what is inside the system and what is outside the system;
- (b) Make use of a standardised template for documenting your use-case specifications. If the team uses a template, it will help communications between team members as well as help newcomers get started;
- (c) When writing use cases, focus on the goals of the actors. This will help you find use cases from the users' perspective and keep a focus on the primary function of the system; and
- (d) Do not make use-case specification synonymous with user-interface design. You can use low-fidelity representation of user interfaces, but since user interface design is particularly subject to design-specific change, it is best not to include it as part of the requirements if you want to get them signed off earlier rather than later.

To help catch potential problems, Lilly suggests that you review your Use Case Model, the diagrams and specifications in incremental steps with the development team, clients and users. Start by reviewing your use-case diagram before you have defined use case details, check that the system boundary has been clearly defined, that obvious actors have not been missed and that the use cases focus on the users' goals.



ACTIVITY 2.13

Differentiate between use cases and requirement statements. Check your answer at <<http://ootips.org/use-cases-vs-requirements.html>>

2.3.6 How to Deal with Hundreds of Use Cases

The case study that you are working on is purposely chosen to be a small application. So you may wonder how this process “scales up” to cater for larger systems which may have 200+ use cases. The key is to organise the use cases so the team can understand them. One specific technique is to organise the use cases into groups or **packages**.

A UML package is a mechanism to group anything together. So a use case package is a collection of use cases, actors and even other use case packages to structure the use case model by dividing it into smaller parts. Thus, instead of trying to understand 200+ use cases, the team needs to understand perhaps only 20 use case packages. Individual team members, then, need to understand the details of a few use case packages.

Deciding on the packages for a system requires a little experience and there are generally several valid approaches for one system. Some possibilities for packaging the use case are listed below:

- (a) By functional areas. For example, put everything to do with money in one package, and inventory in another.
- (b) By actor. For example, put all the cashier's use cases in one package.
- (c) By “style” of use case. For example, put all the simple setup or maintenance style use cases in one package.
- (d) By importance or iteration. For example, all the use cases that are going to be built first could be in one package.

- (e) By physical implementation. For example, all the use cases that are delivered via the Web are in one package, and use cases accessed via the head office computer are in another package.

Generally a combination of multiple approaches is best. Deciding on the use case packages is usually a collaborative effort between the users, the requirements analysts, the technical architect, perhaps the database designer and the project manager.

The UML notation for a package is a folder. However,, as emphasised earlier, the text of the use case is more important than the diagrams. So often a use case package is simply a short description of the use case, followed by a list of the use cases that comprise the package. Optionally, include a use case diagram of the use cases (refer to Figure 2.7).

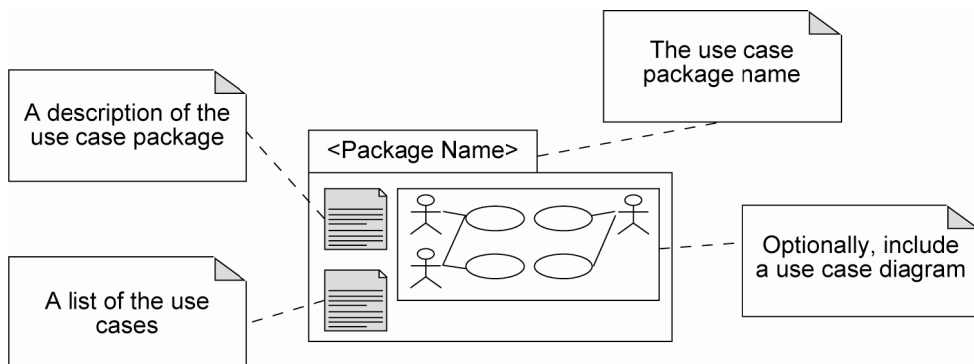


Figure 2.7: Use case package

2.4 CAPTURING REQUIREMENTS ITERATIVELY

As you learned in *Topic 1*, all the activities in the unified process including requirements analysis, will be done iteratively. The whole process of preparing and writing this use case may take several iterations. The working of this use case has spanned both the inception and elaboration phases of the UP. It will probably not be completed until we reach the construction phase. The distribution of work for requirements analysis and use cases writing throughout these iterations and phases in the UP very much depends on the conditions and needs of the system development itself as well as the working style of the system development team. However, as a general guideline, most of the requirements analysis work will be done during the iterations in the elaboration phase.

Particularly with large systems, the key point is to identify all the *architecturally significant use cases* as early as possible and leave less important use cases for later. Architecturally significant here means all the essential functionalities of the system. Use cases are vital and central to the Unified Process (Larman, 2002).

During the requirements analysis, use cases are the main product. However, as we noted earlier, use cases cater for the functional requirements of the system. As you can recall, there are also non-functional requirements that need to be identified. Alongside the use case, there are three more artifacts in the requirements workflow – vision, supplementary specification and glossary. These three artifacts capture other system requirements that cannot be captured in the use cases. The supplementary specification captures other non-functional requirements in the URPS+ category such as documentation, packaging, supportability, licensing requirements, etc. The vision is probably the highest-level document of the project, providing a broad picture of the purpose and business needs of the project and how the proposed system should look. It sets forth the ultimate goals of the project from the organisation's business perspective. The glossary mainly serves as a data dictionary for the whole system development project. It clearly defines all the technical terms in the context of the system development in order to avoid problems in communication and ambiguity that can possibly arise during the whole development process.

The detailed discussion about vision, supplementary specification and glossary is not in the scope of this course.

SUMMARY

- Every system development project starts with requirements analysis. Getting the right requirements is critical to the success of system development especially for large and complex systems and in a rapidly changing business environment.
- Unlike the traditional waterfall system development life cycle (in which requirements analysis is done and completed before the design and implementation phases) the UP model has the requirements workflow (discipline) extended throughout the whole system development process. In other words, the requirements analysis is done iteratively.
- Different types of requirements are categorised as functional or non-functional according to the FURPS+ model. In the UP context, requirements writing is based on the use case model.

- The use case model includes the identification of actors, their goals and the writing of use cases. The functional requirements are captured in the use cases. Sometimes, use case diagrams can be used to help to understand the overall picture of the system, the flow of events and the relationship between actors and use cases.
- Other non-functional requirements are presented in other artifacts such as the vision, supplementary specification and glossary. These artifacts together with the use cases form the artifacts in the requirements workflow.

KEY TERMS

Actor	Use case
Functional requirement	Use case activity diagram
Non-functional requirements	Use case diagram
Role	Use case package



REFERENCES

- Constantine, L. L., & Lockwood, L. A. D. (1999). *Software for use*. Reading, MA: Addison-Wesley. (Gives a detailed treatment of using use cases to design the user interface. See also their website <www.foruse.com>)
- Cockburn, A. (2002). *Writing effective use cases*. Boston, MA: Addison Wesley.
- Eriksson, H. E., & Penkus, M. (2000). *Business modelling with UML – Business patterns at work*. New York, NY: Wiley. (This covers some extensions to use case modelling as presented in this topic.)
- Larman, C. (2002). *Applying UML and patterns*. Upper Saddle River, NJ: Prentice Hall.
- Lilly, S. (1999). Retrieved from: <http://www.api.adm.br/GRS/referencias/HowToAvoidUseCasePitfalls.pdf>

Robertson, S., & Robertson, J. (1999). *Managing requirements. Mastering the requirements process*. Addison-Wesley. (See also their website for a requirements template, <<http://www.atlsysguild.com/GuildSite/Robs/Template.html>>)

Zachman, J. The Zachman Framework. Retrieved from: <http://www.zachmaninternational.com> (Select “Framework” from the left menu. The Zachman Framework provides a complete view of requirements from an enterprise perspective.)

Topic 3 ► Object-Oriented Modelling

LEARNING OUTCOMES

By the end of this topic, you should be able to:

1. Define the elaboration steps;
2. Apply system sequence diagrams;
3. Explain the purpose of domain modelling;
4. Explain what a conceptual class is;
5. List techniques for finding conceptual classes;
6. Define the terms association and attribute;
7. Draw a domain class diagram for a given domain, using correct UML class diagram notation; and
8. Discuss the issues of applying UML to different perspectives and models and explain what is meant by representation gap.

► INTRODUCTION

From Topic 2 we have defined our system requirements to some degree. Hopefully we are clear about the scope (that is, what is included) and have some use cases and maybe a use case diagram which has been written for parts of the system.

Up to now, there has been nothing object-oriented in the material. We could write use cases and then continue in a non-object-oriented manner. In this topic, we start the object-oriented modelling that will continue for several topics. While the material is presented to you sequentially, in reality the object-oriented modelling happens in parallel with the use case writing.

This topic is all about domain modelling. Why do we do this? This is because we need to confirm our understanding of the domain and the scope of the problem. In some cases the software developer or system analyst already understands the domain, but in many cases there is some learning to do. In certain specialist fields, such as finance, IT designers are expected to know a great deal about the finance domain. While the theory may look quite straightforward, you do need to practise by working through the activities and the tutorials. By the end of this topic you will have a domain model for Victoria's Videos.

3.1 FROM INCEPTION TO ELABORATION

This topic, which focuses on domain modelling, provides your first encounter with object-oriented modelling. Before examining this in detail, let us check where we are now in the UP. Remember that the UP has two aspects: phases and disciplines (or workflows). In *Topic 2*, you worked through one important discipline – **requirements**. Using the use case model, you saw how to produce a number of artifacts in the requirements workflow, including use cases (which probably incorporate some use case diagrams), the vision, supplementary specification and a glossary. Once again, we would like to emphasise that these artifacts are not produced in one go: the UP is founded on the premise that all tasks are carried out in phases and iterations. Most of the tasks in the requirements workflow are accomplished during the inception and elaboration phases. (Of course, within these two phases, other tasks are also carried out.) So after the inception phase, it is essential to check what has been already done and what we are going to do in the elaboration phase.

During the inception phase, we should have established the business case for the system and defined the project scope. To accomplish this we must have identified all external entities with which the system will interact (actors and their roles) and defined the nature of this interaction at a high level. This involves identifying all use cases and describing (briefly) a few significant ones. The business case includes success criteria, risk assessment and an estimate of the resources needed as well as a phase plan showing dates of major milestones. The outcome of the inception phase should include:

- (a) Vision document: a general vision of the project's requirements, key features, and main constraints;
- (b) Initial use case model (10–20 per cent complete);

- (c) Initial project glossary;
- (d) Initial business case, which includes business context, success criteria (revenue projection, market recognition and so on), and financial forecast;
- (e) Initial risk assessment;
- (f) Project plan showing phases and iterations;
- (g) Business model, if necessary; and
- (h) One or several simple prototypes.

The inception phase should not last too long, generally one to two weeks, so many of the things in the above list will be started but they will not be finished. For example, a serious architectural prototype could easily take a month to choose and to install the relevant technologies (hardware and software) and then to build it. The artifacts produced should be brief and incomplete. The first major project milestone is at the end of the inception phase. The evaluation criteria for the inception phase are:

- (a) Stakeholders agree on scope definition and cost or schedule estimates;
- (b) Requirements understanding as presented in the primary use cases;
- (c) Credibility of the cost or schedule estimates, priorities, risks and development process;
- (d) Depth and breadth of any architectural prototype that was developed; and
- (e) Actual expenditures versus planned expenditures.

If the inception phase milestone is not met, the scope of the project may be redefined, or it may even be cancelled. If it passes the milestone, the project team enters the elaboration phase where a more in-depth requirements investigation is carried out and implementation of the core architecture is initiated. The purpose of the elaboration phase is to analyse the problem domain, establish a sound architectural foundation, develop the project plan and eliminate the highest risk elements of the project. Architectural decisions have to be made with an understanding of the whole system: its scope, major functionality and non-functional requirements such as performance requirements, etc.

In terms of project management, the elaboration phase is the most critical of the four phases. At the end of this phase, the hard “engineering” is considered complete and the project team should make a very important decision – whether or not to commit to the construction and transition phases. For most projects, this also corresponds to the transition from a low cost, low-risk operation to a high-cost, high-risk operation with substantial investment. While the process must always accommodate changes, the elaboration phase activities ensure that the architecture, requirements and plans are stable enough and the risks are sufficiently mitigated, so you can determine the cost and schedule for the completion of the development.

In the elaboration phase, an executable architecture prototype is built in one or more iterations, depending on the scope, size, risk and novelty of the project. This effort should at least address the major use cases identified in the inception phase, which typically exposing the major technical risks of the project. While an evolutionary prototype of a production-quality component is always the goal, this does not exclude the development of one or more exploratory, throwaway prototypes to mitigate specific risks such as design or requirements tradeoffs, component feasibility study, or demonstrations to investors, customers and end-users. The outcome of the elaboration phase is:

- (a) Use case model (at least 80 per cent complete): all use cases and actors have been identified, and most use case descriptions have been developed;
- (b) Supplementary requirements: these capture the non-functional requirements and any requirements that are not associated with a specific use case;
- (c) System sequence diagrams that describe the events and their order: these are generated by the actors and the system or inter-system events;
- (d) Domain model: a visualisation of things in the domain of interest;
- (e) Design model (partially complete): a set of diagrams that describes the logical design of the system;
- (f) Software architecture document: this summarises the key architectural issues and their resolution in the design;

- (g) Data model (partially complete): this includes the database schema and the mapping strategies between object and non-object representations;
- (h) Test model (partially complete): this describes what will be tested;
- (i) Executable architectural prototype;
- (j) Revised risk list and a revised business case;
- (k) Development plan for the overall project, including the coarse-grained project plan, showing iterations and evaluation criteria for each iteration;
- (l) Updated development case specifying the process to be used; and
- (m) Preliminary user manual (optional).

The targeted system will be designed and built by the system development team of the organisation. However, it may happen that there is a commercially available software package that fits the requirements of the targeted system (or requires only minor adaptation). In this case, the organisation may choose not to build its own system but to purchase the available commercial package, which, in most cases is more economical. The decision of whether to buy or to build the system has to be made during the elaboration phase before we move to the next phases, which are very costly and more risky.

3.2 USE CASE REVIEW – SYSTEM SEQUENCE DIAGRAMS

As you will remember, the use cases mainly describe “what” the system will do rather than “how” it will do it. In effect, as you will see when you work through the reading for this section, the system is treated as a black box with which the actors interact. Before progressing to the logical design of the system, it is useful to further clarify its behaviour using a system sequence diagram (SSD), a technique that enables you to review your use cases. SSDs illustrate events and operations sequentially, starting with the external actor’s input to the system. They can be seen as timeline drawings of expanded use cases, the earliest event being placed at the top of the diagram. Figure 3.1 is an example of a simple SSD for one use case. According to Larman (2002), SSDs are a part of Use-Case Model.

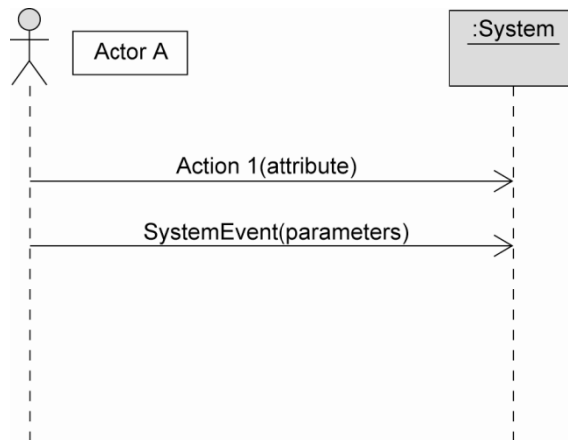


Figure 3.1: System sequence diagram



ACTIVITY 3.1

Refer to the use cases you wrote for Victoria's Videos in Topic 2: for Exercise 2.5 you wrote the "borrow videos" use case, and for Exercise 2.7 you wrote another use case. Draw an SSD for each of those use cases. From actually doing the SSD you may find that your use cases are not quite right. A common mistake is that it is not clear from the text of the use case what is happening. For example, you may have a phrase like "update details". Well, who is updating the details – the user or the system? So, if you find something wrong with your use case, then fix it.

3.3 DOMAIN MODEL

If you refer back to the UP phases and disciplines plan in Topic 1, you will see that there is a "business modelling" discipline with a "domain model" artifact. For the rest of this topic, we will focus on the discussion of how to perform business modelling and the production of a domain model (or as it is referred to in some of the literature, a *conceptual model*). A domain model is a representation of things in the real world.

In fact, domain modelling is a simplified version of business modelling. As depicted in Figure 3.2, more detailed business modelling can be done in an alternate way with more detailed processes.

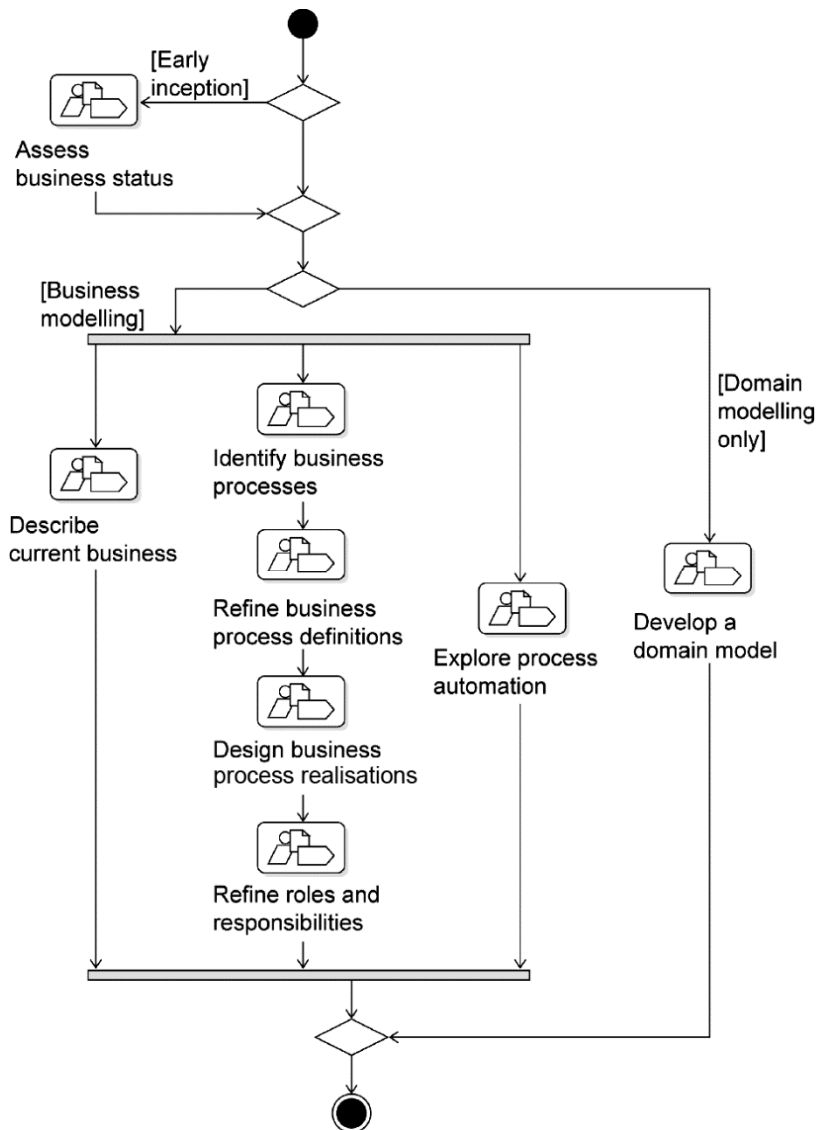




Figure 3.2: Business modelling

Before we go any further, let us discuss the various kinds of models you meet in UP, as they are liable to become confusing unless you are clear about the way they fit together.

In UP, we adopt the modelling methodology to perform most of the tasks in the whole system development process. There are a number of different models that we need to build at different phases and iterations. We can categorise all these different models into three levels, conceptual, logical and physical.

Table 3.1: Hierarchy of Models in UP

Conceptual model 	<p>This model is a representation of the real world (in the context of system development, the business domain where the resulting system will be operated) from the user's conception. For example, in the Victoria's Videos case, a conceptual model is one that will describe the entities as well as their relation in the video check out system as perceived by, for example, a member of the video shop. The entities within the video check-out domain may well include:</p> <ul style="list-style-type: none"> (a) members (b) video CDs (c) video titles (d) video categories, etc. <p>The conception model has the highest level of abstraction in the modelling hierarchy in that the model closely reflects the way the users perceive their own operations.</p>
Logical model 	<p>This model shows what a system must <i>do</i> or <i>have</i>, without regard for how it is to be done, built and represented. This includes the <i>requirements model</i> discussed in Topic 2. Logical models are <i>implementation-independent</i> in that the final system can be implemented in any programming platform, be it C++ or Java and even a manual system.</p>
Physical model	<p>This is the final design model showing how things will be done or built and depicting all platform details, data storage and other implementation details.</p>

Business modelling discipline comes before the requirements discipline. However as you have seen, in practice, we do not keep to this sequence. For example, we have already had quite a detailed discussion of the requirements discipline without mentioning a single word about business modelling. In fact, it is recommended that business modelling should start in the first iteration of the elaboration phase after some preliminary requirements artifacts have been produced as shown in Table 3.2.

Table 3.2: Business Modelling

Discipline	Artifact	Incep. I1	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	s		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

s – start, r – refine

Source: Larman (2002), p. 24

This is because in UP practice, we work through the disciplines iteratively. They are not started and completed in sequence. Within each iteration, there may be several disciplines in progress (of course with different degrees of completion) at the same time. To facilitate the development of the domain model (mainly in the identification of domain objects, which we will discuss in the coming section), use cases (at least preliminary ones) are developed before the domain model. As you will see later in the discussion, the domain model and use case model are interrelated.

Now that you have a clear understanding of the whole modelling hierarchy of the UP, you will see why it is important to have a business modelling process on top of the other UP models. As we are all aware, information systems no longer merely support businesses. Increasingly, they form an integral part of a business, hence the business itself defines the requirements of the system we are going to develop. A business model (the domain model) is an abstraction of how a business functions. It is a somewhat simplified view of the complex reality of the business. A business model enables the system developers to eliminate irrelevant details and focus on more important aspects of the business.

As we said above, the use case model and domain model are interrelated. The use case model (like most of its artifacts) is basically textual in nature: it describes the details of the functional (and non-functional) requirements of the system in words. However, as has often been stated, ideas presented in visual rather than textual terms are often easier to understand (a picture paints a thousand words). So a visual business (domain) model may enhance the understanding of business operations and help to refine the development of the requirements model and the design model in the later stages of the UP.

Sometimes, a large-scale system development project goes hand-in-hand with major changes in the business processes (which we call Business Process Reengineering, BPR). In that case, a business model not only serves the purpose of system analysis and design but may also be treated as a blueprint for those undertaking the reengineering (in this case, we may need to go through the detailed business modelling processes as depicted in Figure 3.2). It provides a common language and platform for both communities, as well as shows how to create and maintain direct traceability between business and software models.

There are many different ways to do business modelling. In the context of the UP, we usually make use of the UML class diagram notations (which results in the so-called domain model or conceptual model in some other object-oriented literature). Bear in mind that UML is just a tool for object-oriented analysis and design. The class diagram in UML can be used for other purposes in UP such as the “design class diagram” in the design model which we will discuss later. Here, in the context of business modelling, a domain model is a class diagram drawn from the conceptual perspective (which is different from the specification perspective in the design class diagram at the design modelling stage).

In a domain model, we have three types of information:

- (a) **Domain object** (or conceptual class) – which identifies a business entity or concept, for example, shop, video CD, member, etc.;
- (b) **Associations between domain objects** – which define relevant relationships, those that capture business information that needs to be preserved and their multiplicity, for example, a shop has many video CDs, a member borrows many video CDs; and
- (c) **Domain object attributes** – which are logical data values of an object, for example, each member may have a “member’s number” which is an attribute of the object “member”.

It is important to note that analysis class diagram is a visualization of things in the real world domain of interest and not the software components such as Java class.

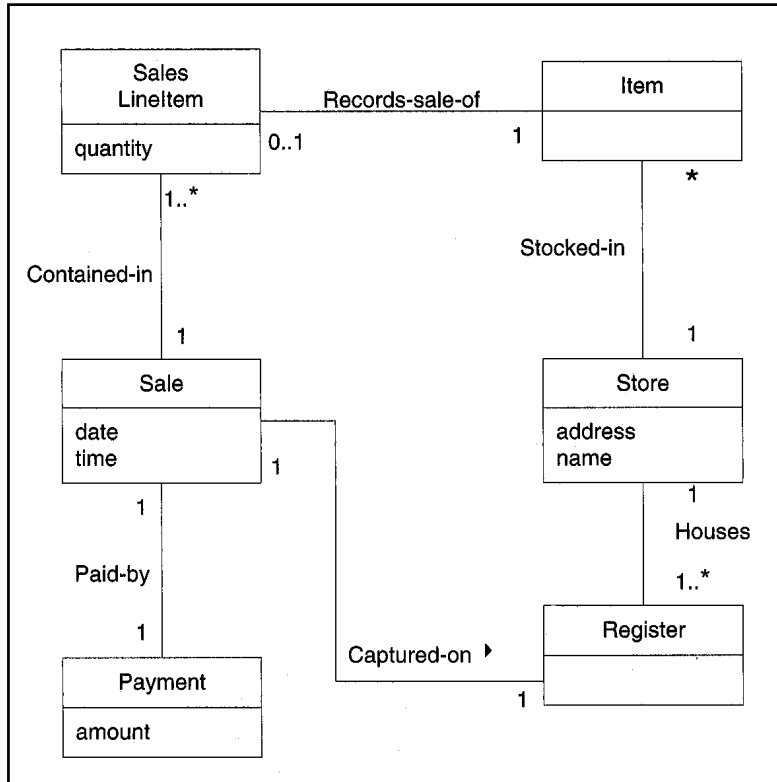
Remember...

Analysis class diagram which is produced during the analysis phase is a visualization of things in the real world domain of interest and **NOT** the software components such as Java classes. On the other hand, the relationship between the software classes, which depicts the details of the classes are done through the **design class diagram** which takes place during the design phase. Design class diagram is beyond of the syllabus.



ACTIVITY 3.2

Before we go into the details of the domain model, let us make sure we are clear about its function. Refer to the diagram below:



Source: Larman (2002)

Do not be concerned with the exact meanings of the notation – the lines, words, arrows, etc. – as this is all explained later in this topic. For the moment, just try to read the diagram with the following words, starting in the top right-hand corner: *Items are Stocked-In a Store*. A Store has an *address* (for example, 15 Jalan Buntong, Ipoh) and a *name* (for example, Kedai Perabot Ali). Each *Store houses*, or has, some *Registers*. Every *Sale* is *captured-on*, or recorded on, a *Register*.

Continue with this description to cover all the domain objects and relationships as depicted in the figure above. See if it gives you a better understanding of the “payment sale” process of the NextGen POS case.

3.4 CONCEPTUAL CLASSES

The first thing we have to do in building a domain model is to identify conceptual classes (or domain classes, as termed by some of the object-oriented literature). A conceptual class represents something – physical or conceptual – in a business. There are different kinds of conceptual classes, as you will see in a moment. However, first, we need to revise what we mean by the words *class* and *object*. You will recall from *Topic 1* that a class is a template for a set of objects which all look the same. So a manufacturing plant for a car is like a class, and all the cars that it makes are the objects. Of course, every car has its own license plate and belongs to a different owner (refer to Figure 3.3).

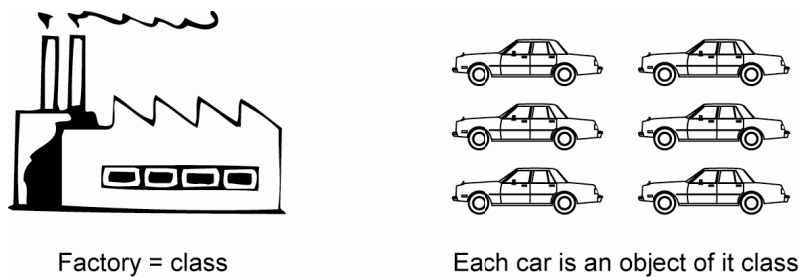


Figure 3.3: Class and object

As the relationship between a class and its objects is so clear, we often get a bit lazy with our language. If the head of engineering for a car factory says: “The painting on the car needs to be done better”, he or she is probably talking about the process that goes on in the factory rather than a specific car. So, too, in software modelling where we swap between using the words class and object.

The entity “objects” fall into a number of categories, based on the real-world objects that they represent. Here are three kinds of conceptual classes that we will encounter in business modelling:

- (a) **Concrete objects** are tangible, that is, they have a physical presence. Concrete objects are the most easily understood by analysts and users. In the video shop example, *video CD* is a concrete object;
- (b) **Conceptual objects** are intangible and often far more difficult to understand. For example, school is a conceptual object. You may argue that a school tangibly exists – it has buildings. However, the concept of a school is not just the buildings. Imagine that all the buildings in a school collapse in an earthquake. We cannot say that the school does not exist anymore. It can be rebuilt. Similarly, *video title* is also a conceptual object; and

- (c) **Event and state objects** are highly abstract in nature. They are related in that typically when an event of any significance occurs, some or more objects will switch to a different state. An example of an event and state object in the video shop is *borrow video*. We can see that borrow video events will typically change the status (for example, borrow video list) of the object.

Conceptual class is an important concept in the object-oriented paradigm. In the traditional software development process, complicated software programmes were broken down into processes or functions (in computer programming terms). For example, see the figure of the functional decomposition for the video shop system in *Topic 1*.

In the object-oriented software development approach, complicated software programmes are broken down into objects (which, in most cases, resemble real-world objects). Hence, the division of the business or problem domain into conceptual classes is the first step in object-oriented analysis and design. However, please note that the conceptual classes or objects we are talking about in domain modelling are different from the software object classes we will talk about in the later stages of the UP.

According to Larman (2002), a conceptual class may be considered in the following terms:

- (a) **Symbol** – words or images that represents a conceptual class
- (b) **Intension** – the definition of a conceptual class
- (c) **Extension** – the set of examples to which the conceptual class applies

For example, in a credit transaction, the conceptual class can have the symbol *Borrow*. The intension of *Borrow* could be “represents the event of borrowing transaction which has a date and transaction ID”. The extension of *Borrow* is all the examples of borrow.

3.4.1 Identification of Conceptual Classes

Conceptual classes in a domain model can serve as an inspiration for the design of the software classes in the design model. In this respect, the identification of the conceptual classes in the domain model is of vital importance to the success of the system development. In a complicated business domain, there may be hundreds of entities that can potentially be treated as conceptual classes. **REMEMBER:** Whether they are included in the domain model very much

depends on the users and how they view their business environment. We have to choose the right conceptual classes so that a meaningful domain model can be created.

There are a number of ways that we can identify conceptual classes systematically. One method is to come up with a conceptual class category list. We have already categorised conceptual classes into three broad categories, namely concrete objects, conceptual objects and event and state objects. These three categories are further expanded into finer categories according to Table 3.3.

Table 3.3: Conceptual Class Category List

Conceptual Class Category	Examples
Physical or tangible object	Register
Specifications	ProductSpecification
Places	Store
Transactions	Sale, Payment
Transaction line items	SalesLineItem
Roles of people	Cashier
Containers of other things	Store, Bin
Things in the container	Item
Other computer systems external to the system	CreditPaymentAuthorisationSystem
Abstract Noun Concepts	Hunger
Organisation	SalesDepartment
Events	Sales, Payment, Meeting
Processes	SellingAProduct
Rules or Policy	RefundPolicy
Catalogues	ProductCatalogue
Records of finance, work, contracts, legal matters	Receipt, Ledger, EmploymentContract
Financial Instruments and services	LineOfCredit
Manuals, documents, reference papers, books	DailyPriceChangeList

Source: Adapted from Larman (2002)

Even though the category list as presented is drawn from some particular domains (in the Table 3.3 case, the store domain), it more or less covers many common categories in other business domains.



ACTIVITY 3.3

Using Table 3.3 as a guide, produce a conceptual class categorisation for Victoria's Videos.

Another technique for identifying conceptual classes is the use of the noun phrase identification. After you have worked through Reading 3.5, you will see that one of the arguments for developing use cases before domain modelling is that they are an excellent source of noun phrases for conceptual class identification. However, note the danger of ambiguity in the natural language of use cases translating to the conceptual classes if the transformation of one to the other is made too mechanically.

Now let us see one example in which we going to use noun phrase identification to identify conceptual classes.

Example

Below is a description of a Web-based ticket reservation system. All the noun phrases that are candidates for conceptual classes are highlighted in bold.

An Internet-based **ticket reservation company** specialises in selling **tickets** for **entertainment events** to **customers** who visit the company's **website**. The company acquires tickets for events from the usual **ticket agencies** on a regular basis or on demand when **stocks** become low.

Before they can use the site, customers must register their **contact details** or enter sufficient information to identify themselves if they have registered before. They may then view **publicity material** and **ticket availability information** for a number of different events and add tickets for chosen events to their order.

When they have finished, they enter their **credit card** details and, if all is well, their tickets will be dispatched through the post. The company keeps a **contact address** and a **billing (credit card) address** (which may be the same) for each registered customer.

Concentrate just on the customer side – do not model the supply of tickets from “the usual ticket agencies”.

3.4.2 Issues in Identifying Conceptual Classes

There is no unique conceptual class list for a particular business domain. Different users and object-oriented systems analysts may produce different lists. Whether or not to include a certain entity object in the conceptual class list or not depends on the context of the use cases and sometimes on the stages and iterations in the UP. For example, should the object “borrow video list” be included in the video shop domain model? This is a debatable question as the “borrow video list” of a member is generated upon request (probably for some member who has forgotten how many videos they have borrowed). It is not a part of the normal video borrowing process; it is only required, say, in the “request for borrow video list” use case.

There is also the issue of the proper naming of the conceptual classes so that the use of the domain model as an effective communication tool in the system development project can be highlighted. There are two basic principles that we need to follow in naming conceptual classes:

- (a) They must be distinct; and
- (b) There must be a way of telling them apart.

We will discuss “attributes” of conceptual classes in the coming section. Sometimes, there may be confusion between the conceptual class itself and its attribute. For example, should “Popularity” be a separate conceptual class to “Video” or just its attribute? Use the following hints to resolve this confusion:

“A conceptual class is not considered something a number or text but something that has legal entity, an organisation and occupies space”

3.4.3 Specification Conceptual Classes

There is a conceptual class category called “specification” or “description” in the extended conceptual class category list in Table 5.1. This category of conceptual class requires a bit of elaboration. The following example will help you to grasp the idea.

In the video shop example, suppose we have the video *Star Trek* that has once been in stock. The video has a video title, video classification and video category, etc., that go with it. After the video has been borrowed by a member and he or she has lost it, the video will be deleted from the system. If the management needs to obtain some information about the video (for example, what classification it is) there is no way they can obtain that information. To solve this

problem, we need a “VideoSpecification” conceptual class that records information about videos so that whenever information about a specific video is needed (no matter whether it is still in stock or not), it can be retrieved from the specification. The following suggests when it is appropriate to use specification conceptual classes:

- (a) There is a need to describe an item or service, independent of the current existence of these items or services.
- (b) Deleting instances of things that they describe (for Example: Item) results in a loss of information that need to be maintained.
- (c) It reduces duplicated information.



ACTIVITY 3.4

Update domain model shown in **Activity 3.2** to include a description conceptual class.

3.5 ASSOCIATIONS

Now we have successfully identified the conceptual classes in the business domain. Say, in the video shop system, we have identified “Member”, “Video”, “FrontDesk”, “MemberRecord” as the conceptual classes (please note that this is a highly simplified domain model and may not be the real domain model for the video shop case). What are we going to do with all these objects (or conceptual classes)? An obvious answer is to find out how they are related. An association is an object-oriented term for relationship. People in the real world have relationships with other people, with things and with places. So we can say a person has a mother, is married to their spouse, likes to drive their Porsche, lives in a house, etc. These are the associations of the object “people” to other objects in the real world. We notice that most of the relationships between the object “people” and the other objects are described by verbs such as “has”, “married to”, “likes to”, “lives in”, etc.

In UML, an association is a relationship expressing the interaction between instances of two conceptual classes, represented by the verb that describes what they do to each other, and/or by the nouns for the roles that each play in the life of the other.

Figure 3.4 shows the relationship between two conceptual classes “Member” and “Video”.

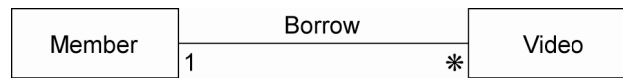


Figure 3.4: UML notation for association

The line between the conceptual class “Member” and “Video” represents the association between the two classes with the label “Borrow” that indicates the association name (relationship). The “1” and “*” represent multiplicity, which we will discuss in detail later.

3.5.1 Identifying Associations

Just as we identify conceptual classes, we need to identify all the essential associations between conceptual classes that will result in a meaningful domain model. There are a number of ways to do this. The following is a simple and straightforward way in which we use the video shop system as an example.

Begin with a list of all the classes on the left and the UML diagram of all the classes drawn (without association) on the right, as shown in Figure 3.5. Pick the first and second classes in the list, that is, “Member” and “Video” and check whether they have a relationship (either from the use case or discussion with users). If there is a relationship (in this case “Member Borrow Video”), add an association to the UML diagram. At the same time, a line is drawn linking the “Member” and “Video” items in the list, indicating that the relationship of these two classes has been checked.

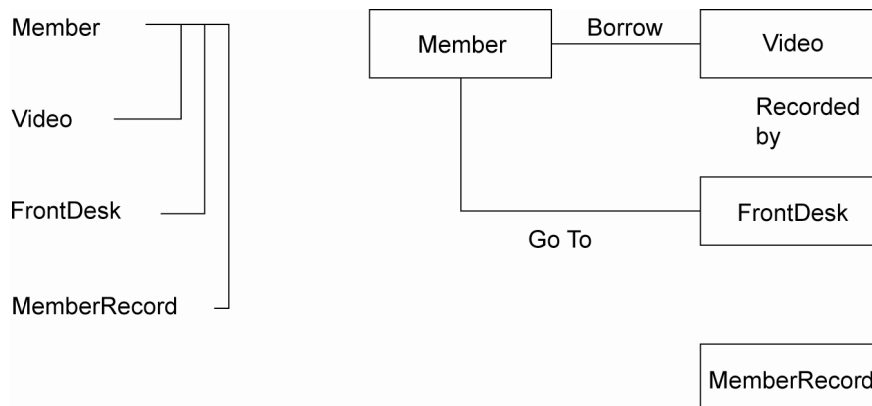


Figure 3.5: Association identification step 1

Then we move down the list to check the relationship between “Member” and “FrontDesk” and the association “Go To” is added. Again, the two items in the list are also linked with a line.

The process is repeated with the other items down the list, that is, the “Member”-“MemberRecord” pair. If a relationship exists, an association is added in the UML diagram. For the items in the list, even though there is no relationship between two items, such as “Member” and “MemberRecord”, a link is also drawn to join the two items indicating that the relationship between the two has been considered. When “Member” and all the other items down the list have been linked, it indicates that the relationship between “Member” and all the other classes has been considered (regardless of whether or not they have an association in the UML diagram). We can move on to the next step.

In the next step, the relationship between the class “Video” and the other classes down the list is considered. Similarly, items are linked after their relationship has been checked and an association is added between any two classes in the UML diagram if a relationship exists (refer to Figure 3.6).

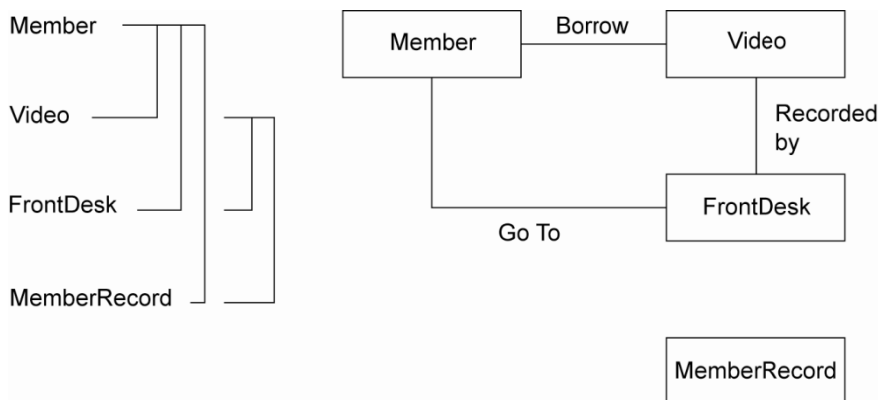


Figure 3.6: Association identification step 2

This step is repeated with class pairs down the list until all class pairs have been considered. The final model will look like Figure 3.7.

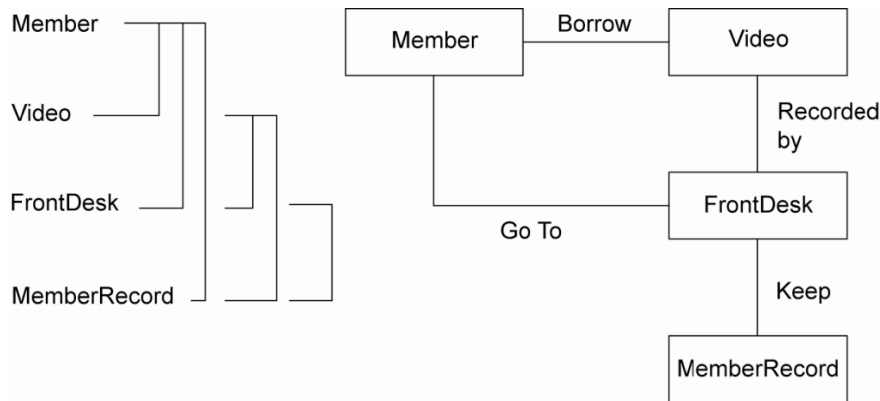


Figure 3.7: Association identification final step

The simple method for association identification is, in essence, an exhaustive searching method that will check all the possible relationships between class pairs. The advantage of this method is that the possibility of missing an association in the model is very low. However, for large models with a large number of conceptual classes, the number of class pairs that need to be checked will be very large ($n*[n-1]$ for n classes). The effort involved may not be justified when some of the associations are not significant or meaningful to the model. Hence, in the textbook, an alternative method making use of a so-called “common association list” is suggested. This method is based on the “need-to-know” association principle to avoid too many resulting associations, which would lead to confusion in the model.

“Common association list” by Larman (2002) contains common categories that are usually worth considering in determining the relationships between the classes. The common association list is given as follows for the airline reservation system (Table 3.4).

Table 3.4: Common Association List (As Proposed by Larman (2002))

Category	Examples
A is a physical part of B	Wing-Airplane
A is a logical part of B	FlightLeg-FlightRoute
A is physically contained in/on B	Passenger-Airplane
A is logically contained in/on B	Flight-FlightSchedule
A is description for B	FlightDescription-Flight
A is a line item of a transaction or report B	MaintenanceJob-MaintenanceLog
A is known/logged/recorded/reported/captured in B	Reservation-FlightManifest
A is member of B	Pilot-Airplane
A is an organizational subunit of B	Maintenance-Airplane
A uses or manages B	Pilot-Airplane
A communicates with B	ReservationAgent-Passenger
A is related to a transaction B	Passenger-Ticket
A is transaction related to another transaction B	Reservation-Cancellation
A is next to B	City-City
A is owned by B	Plane-Airplane
A is an event related to B	Departure-Flight

3.5.2 Multiplicity

As you identify and draw each association, you will initially draw it as a simple line joining two boxes in the UML diagram. Once you have established the association, for example, “Member Borrow Video”, there is another question to ask: How many videos will the member borrow? This question relates to the multiplicity of the association.

The multiplicity of an association is the number of instances of each class that can participate in an occurrence of the association. The following table lists some possible multiplicity values for associations (refer to Table 3.5). Multiplicity is an important piece of information for the later system design stage. It communicates important domain constraints that will affect the software design, especially in database design in the implementation stage.

Table 3.5: Multiplicity Values

Multiplicities	Meaning
1 1	one-to-one
1 *	one-to-many
1 1..*	one-to-one or more
1 0,1	one-to-zero or one-to-one

Figure 3.8 depicts the domain model of the video shop system with multiplicity added.

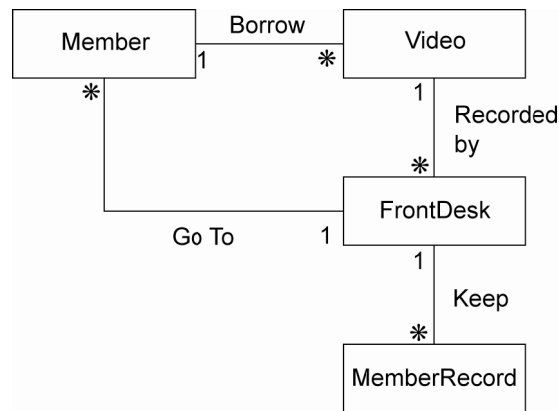


Figure 3.8: Associations with multiplicity

3.5.3 Other Issues of Association

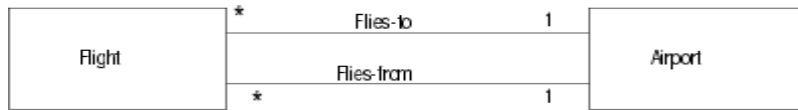
There are other issues concerning association that are helpful in making the domain model more comprehensible and serve the purpose of communicating domain knowledge more effectively. These include:

(a) **Naming Convention for Associations**

The association should start with a capital letter such as Send-by or SendBy. It is a common convention to read an association from left to right or from top to bottom.

(b) **Multiple Associations between Two Classes**

There can be multiple relationships between objects in the domain model as shown below. Here, Flies-to and Flies-From are distinctly separate relationships.



(c) **Associations and Implementation**

Associations in the analysis class diagram is not about data flows, or object connections in a software solution but it shows the relationship in a purely conceptual sense of the real world. Also, some of the associations in the analysis class diagram may not be implemented during the implementation (i.e. coding). You may also discover new associations need to be implemented but were missed in the analysis class diagram.

(d) **“Need-to-know” vs Comprehensive Associations**

When identifying associations in the analysis class diagram, pay attention on the followings (Larman, 2002):

- (i) Focus on the associations for which knowledge of the relationship need to be preserved for some duration (“*need-to-know*” associations).
- (ii) Avoid showing redundant or derivable associations

A strict used of “need-to-know” approach highlighted above to maintain the associations in a analysis class diagram will provide a minimal information on the analysis class diagram. It means it may not convey a full understanding of the problem to the user. According to Larman (2002), in the terms of the association, a good model of analysis class diagram can be constructed somewhere between a minimal need-to-know model and one that illustrates every conceivable relationship. In other words, use the following:

Emphasize need-to-know associations, but add choice comprehension-only associations to enrich critical understanding of the model (Larman, 2002).

In principal, use the following association guidelines when identifying associations between classes (Larman, 2002):

- (a) Focus on the “need-to-know” associations;
- (b) It is more important to identify the conceptual classes than to identify associations;
- (c) Too many associations will lead to confusion and marginal benefit; and
- (d) Avoid showing redundant or derivable associations.

**ACTIVITY 3.5**

Re-draw Figure 3.8 to add in a “VideoSpecification” conceptual class to the video shop domain model.

3.6 ATTRIBUTES

Attributes are the pieces of data we use to identify or describe things. For example, a person has a name, date of birth and eye colour. Attributes usually correspond to nouns followed by possessive phrases, such as “the colour of that car”. In this case, colour is the attribute of car. Attributes are usually simple data types or primitive data types such as integer, float, string, date, time, etc. Later in Topic 5 you will see that an attribute can also designate a class.

For the video shop system, the following are possible attributes to the conceptual classes identified:

- (a) Member – member’s number;
- (b) Video – video ID, classification;
- (c) FrontDesk – front desk register number; and
- (d) MemberRecord – number of video items borrowed, due dates of items borrowed.

In UML, attributes are shown in the lower compartment of the rectangular box representing the conceptual class. Hence the completed domain model, as represented in UML for the video shop system is as shown in Figure 3.9.

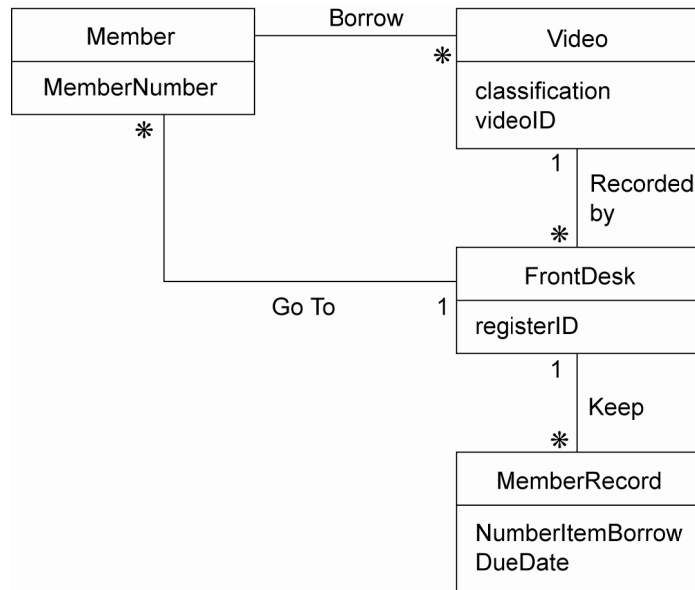


Figure 3.9: Conceptual model of video shop system

Sometimes, it is difficult to judge whether an attribute is really an attribute or whether we should treat it as another class. For example, a date is usually treated as an attribute of a class such as a DueDate. However, now consider a weather forecasting system in the observatory that keeps daily records of atmospheric conditions and produces analyses for different weeks, months and years. Each date may be described by many other variables, for example, maximum, minimum and average temperature, hours of sunshine, total rainfall, average wind speed, etc. These analyses might also require separate attributes for the day of the week, month and year. In this case, we might then choose to treat “Date” as a separate conceptual class with the other variables as its attributes.

When there is a situation where it is difficult to judge whether to include an attribute, use the following criteria to eliminate unnecessary and incorrect attributes:

- If the independent existence of an entity is important, rather than just its value, then it is a separate class;
- An entity that has features of its own within the given application is a separate class;
- If the value of an attribute depends on a particular context, then consider restating the attribute as a qualifier;

- (d) A name is a class attribute when it does not depend on context, especially when it need not be unique;
- (e) Do not list object identifiers that are used purely for unambiguously referencing an object;
- (f) If an attribute describes the internal state of a class that is invisible outside the class then eliminate it from the analysis; and
- (g) Omit minor attributes that are unlikely to affect most operations.

The following tips by Larman (2002) is very useful when creating analysis class diagram:

- (a) Use Conceptual Class Category (from Table 3.3) or noun phrase identification to identify the conceptual classes.
- (b) Draw them in the analysis class diagram.
- (c) Add the associations necessary to show the relationship between the classes.
- (d) Add the attributes necessary to fulfill the information requirements.



ACTIVITY 3.6

Similar to Activity 3.5, add in the conceptual class “VideoSpecification” and its corresponding attributes to Figure 3.9.

Congratulations! You have now built a domain model. It probably is not perfect yet, but it is sufficiently good to use.

3.7 UML NOTATION, MODELS AND METHODS: MULTIPLE PERSPECTIVES

We have made our domain model. The domain model is the first model in UP for which we used UML notation seriously (of course we have used some UML notation in our use cases but not very extensively). We have also had our first encounter with objects and classes in the development of the domain model. As was pointed out in the previous section, in the UP, there are several levels of modelling with different levels of abstraction. However, regardless of what level

of abstraction of the modelling stage we are in, we basically rely on the same set of UML notations to visualise and communicate the models. Hence, it is very important to make it clear that we do not mix up models with UML notations. For example, the same rectangular box in UML can be used to represent classes in different models, be it conceptual classes in the domain model or software classes in the design model.

One advantage of using the same set of notations throughout the whole development process is to reduce the so-called semantic gap (or “representation gap”). Semantic gap means the gap between our mental model of the domain and its representation in software. In a broader sense, it refers to the lack of shared conceptual understanding on the aspects of business and system that need to be modelled in a way that allows both people and technology to work together in harmony and adapt to change. With the object-oriented approach and the use of UML, we use the same representation, notation, and most importantly the same style of thinking, from analysis all the way through to final implementation and maintenance of the system. This not only makes it easier and cheaper to accomplish all system development tasks, it also allows us to better educate users. By reducing the complexity of the whole thing through a reduction of the number of difficult concepts, we can bring knowledge within reach of a larger group of people within the organisation and eventually get more cooperation and enthusiasm for and user ownership of the project.

SUMMARY

- In this topic, you had your first encounter with objects and classes. You modelled a business domain with highly abstract conceptual classes. You learned how use cases which is used to aid conceptual class identification. You also identified the associations between conceptual classes and found their intrinsic properties by identifying their attributes.
- The idea of using the same set of standard notations, the UML, in different models and throughout the different stages in the UP was emphasised. Using this framework will reduce the representation gap between our mental model of the domain and the software representation and result in a much better system development environment.

KEY TERMS

Associations

Attributes

Conceptual classes

Conceptual model

Domain model

Logical model

Multiplicity

Physical model



REFERENCES

Larman, C. (2002). *Applying UML and patterns*. Upper Saddle River, NJ: Prentice Hall.

Eriksson, H. E., & Penkus, M. (2000). *Business modelling with UML – Business patterns at work*. New York, NY: Wiley.

Topic ► More Use-Cases

4

LEARNING OUTCOMES

By the end of this topic, you should be able to:

1. Develop sub-use cases with the include and extend relationships;
2. Describe the purpose of a UML state diagram at the conceptual modelling level;
3. Draw simple state diagrams for conceptual classes;
4. Define CRUD and apply it;
5. Describe and apply business rules in requirements analysis;
6. Describe the purpose of robustness analysis; and
7. Draw a set of robustness analysis diagrams for a system.

► INTRODUCTION

This topic consolidates the work done on use cases and helps you learn more about the models. While it might seem a bit confusing, in practice either there are different people working on different aspects of a project or, in the case of a small-scale project, the same people are working simultaneously on use cases and the other models.

Use cases are the core to the whole process, so it is worth spending the time needed to “get it right”. However, knowing what is “right” is often the hard part. As you will see in this topic, not all writers and practitioners agree on what is right. You need to work this out for every project.

In the sections on use cases we look at how we can split use cases into sub-use cases. How far you take this is a bit controversial – as you will see.

A big question when working with use cases is when are you finished? In other words, when have you found all the use cases? One very useful technique to help answer this question is to draw some state diagrams. In a nutshell, we take the main entities from the domain class model and look at the life cycle of each of them – when they are created, updated and perhaps deleted.

During the development of the use cases, in one way or another, we need to find out how the operations and business of the company or organisation are running. Every company or organisation has its own policies and rules for doing things, which can be summarised in a set of business rules. We have probably found bits and pieces of business rules during the process of writing use cases. There is no formal way of presenting business rules (in terms of UML or other development tools). However, as you may realise later, a set of good business rules is important to each stage in the whole system development project, so it is again worth dedicating some effort to it.

Lastly, we look at a technique called robustness diagrams. These diagrams help us to move from analysis to design.

4.1 RELATING USE CASES

As you remember, we first discussed use cases in Topic 2. Use cases describe the interaction of an actor (either a person or an external system) with the system under discussion. Use cases are written in plain language so that both the users and technical staff can read them. The use cases can be written informally or using more formal templates. The initial version of a use case describes the main flow of events – how things should occur in a normal situation, assuming there are no problems. Later, as the work on the system progresses, the use cases get fleshed out to contain alternate flows and exception handling.

There are many different approaches and styles to writing use cases. Reasons for these differences include personal preferences, different styles of application (for example, a website selling books is different from a stock market trading system), and different scales or styles of development (for example, two people on a three-month project work very differently than 50 people on a two-year project).

The next reading is a review of use cases, written by their inventor, Ivar Jacobsen. He starts with a short history of their development.

http://download.boulder.ibm.com/ibmdl/pub/software/dw/rationaledge/mar03/usecases_TheRationalEdge_mar2003.pdf

As you can see from this reading, one aspect of use cases that has been evolving since 1986 is how we can relate one use case with other use cases. Relating use cases to each other allows us to see commonalities and differences between them.

There are three kinds of relationships between use cases, namely:

- (a) **Include** – typically where a set of use cases each includes some common functionality that is expressed in one shared sub-use case;
- (b) **Extend** – where the functionality of a use case is extended by referencing other use cases; and
- (c) **Generalise** – where several use cases share something similar.

Before we discuss these relationships in more detail, remember that we primarily write use cases for people to read. Use cases are not read by a compiler (a software programme that turns a computer programme into executable computer code) or by any other kind of tool. So before we add relationships to our use cases, we must be sure that adding complexity will help the (human) reader to better understand them.

One more word of caution. As you saw in the reading, Jacobsen has changed his views on how best to relate use cases. Use cases are not mathematically defined, so the relationships between them are not rigorously defined either. You will probably find that when you read the details on the different types of relationships, it will all seem quite simple, but that when you try to apply this knowledge, it can be quite hard. Everyone, both novices and skilled practitioners, face the same problem. The solution is to remember that use cases are for **human** use.

http://download.boulder.ibm.com/ibmdl/pub/software/dw/rationaledge/mar03/usecases_TheRationalEdge_mar2003.pdf

Read pages 4–5, that is, the sections “*Use cases and the unified modelling language*” and “*Use caution when formalizing use cases*”.

Note: The rest of this reading, starting from the heading “*Tomorrow: potential next steps*”, is more advanced and theoretical.

4.1.1 The Include Relationship

In the section of the reading that you have just worked through, Jacobsen describes the “include relationship”. When you are writing your use cases, you may find that sections of some use cases occur in multiple use cases. In order to capture this similarity and to simplify the complexity of long use cases, you can put the common section in a sub-use case and then reference this sub-use case from the other relevant use cases. If you have experience in traditional *procedural languages programming*, you should be familiar with the concept of using a subroutine to reduce programming complexity.

As an example, let us consider Victoria’s Videos. In Topic 2 we identified the use case Maintain Members (Activity 2.6). The next table is a first draft for that use case.

Table 4.1: Maintain Members Use Case

Maintain Members

Actor Intention	System Responsibility
Identify the borrower	Display personal details (name, address, date of birth, date, joined, etc.) of the borrower, plus a summary of current borrowings
Optionally, the user can modify the personal details of the member	Store the changes
Optionally, the user can delete the member	Check that borrower has nothing borrowed Delete the member

Now compare this use case with Borrow Videos (from the answer for Exercise 2.9). Is there anything in common? Yes. The first line for each is the same: “Identify the borrower”. As we are looking at use cases in essential format, this might not seem like very much. However, before we can build this system, we will have to decide how the borrower will be identified. Will the user have a card to be scanned, or will she give us her name or an identification number? Will she need to quote a password? So, to cater for all these later concerns, we can create a sub-use case that we will call “Get Borrower”. Now both the Borrow Videos and Identify Members use cases will «include» the Get Borrower sub-use case. (Note the UML notation of placing the special characters « » around the word “include”.)

So now how do our use cases look? In long hand we could have (please refer to Table 4.2).

Table 4.2: Inclusion a Sub-use Case

Maintain Members

Actor Intention	System Responsibility
Include use case <u>Get Borrower</u>	...

These days the use of underlined text in a website means a hyperlink to something else. This convention is used for use cases to indicate a sub-use case. So a short way of writing is simply to underline the use case name, as follows (please refer to Table 4.3).

Table 4.3: Short Way of Writing the Inclusion of a Sub-use Case

Maintain Members

Actor Intention	System Responsibility
<u>Get Borrower</u>	...

The Get Borrower use case is then as follows (please refer to Table 4.4).

Table 4.4: A Sub-use Case

Get Borrower

Level:	Subfunction
Actor Intention	System Responsibility
Identify the borrower	Display personal details (name, address, date of birth, date joined, etc.) of the borrower ...

You will notice the headings at the start of the use case that has level clause. It tells us that this is a sub-use case, as it only gives us a subfunction. That is, this use case makes no sense by itself. It needs to be called by another use case.

Use Case Diagram Include Notation

To show an include relationship on the use case, UML specifies a dotted line, with the arrow pointing to the included use case (please refer to Figure 4.1).

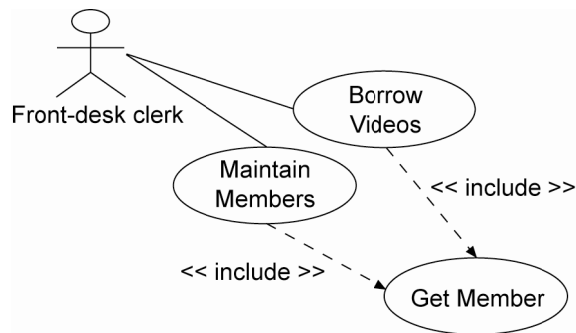


Figure 4.1: Depicting the include relationship



ACTIVITY 4.1

Let us again think back to the situation of Victoria's Videos. You already have a use case to borrow videos. But now Victoria tells you that she would also like to sell old videos that she does not need.

Update your use cases to reflect this change. If possible, incorporate an include relationship to simplify your use cases.

Update your use case diagram (from Activity 2.11) with any new use cases.

The include relationship is the most useful relationship between use cases, and you can build very sophisticated models using only this relationship. Indeed, Cockburn (2002, p. 207), writes:

As a first rule of thumb, always use the includes relationship between use cases. People who follow this rule report that they and their readers have less confusion with their writing than people who mix includes and extends and [generalizes].

Cockburn holds this view because his emphasis is on the text of the use cases, and their role in communicating between people. By contrast, people who put more emphasis on the use case diagram, particularly when done with a CASE tool, like the added complexity of having other relationships.

The next sections discuss the extend and generalise relationships.

4.1.2 The Extend Relationship

The extend relationship is another type of relationship between use cases. In the previous section we looked at the include relationship, which you can think of as “doing something common to several use cases”. An informal definition of the extend relationship is “doing something extra”.

Let us look at an example of one way in which an extend relationship could be used. In Victoria’s Videos we have this requirement (from the description for the Victoria Videos Case Study in Topic 1):

Members’ birthdays are marked with a special letter inviting them to borrow any video of their choice for a week for free.

So we obviously need a use case to generate the birthday letters – that is, a straightforward use case. However, how do we handle the situation when a person comes to the shop with their birthday letter wanting to borrow a video for free? Clearly this is an extension to the use case Borrow Videos.

So how would these use cases look?

Table 4.5: An Extend Relationship

Handle a Birthday Letter

Level:	Subfunction	
Trigger:	Customer has a birthday letter	
Extension point:	Calculate total fee due in Borrow Videos	
Actor Intention		System Responsibility
Indicate that the borrower has a birthday letter		Confirm birthday borrowing has not already been used this year Record use of birthday letter Adjust total due

Once again you can see we are at the subfunction level. The trigger clause tells us when it gets started, namely when a customer has a birthday letter. The extension point tells us that the use case Handle a Birthday Letter gets started from the use case Borrow Videos when it reaches the part “calculate total fee due”. This is called an extension point.

Lastly, to keep the use cases tidy, we can document the Borrow Videos use case to show that there is an extension. In other words, we make it clear to someone reading this use case that, as part of “calculate total fee due”, there is something more happening.

Table 4.6: Documenting a Use Case to Show an Extension

Borrow Videos

Extension point:	Calculate total fee due	
Actor Intention		System Responsibility
Get borrower		Calculate the total fees due for any past fines and the new borrowings Pass the total amount due to the Cash Register Terminal
Identify the videos		

Note that apart from adding the extension point at the start of the Borrow Videos use case, we have not changed the text of it at all. This is an important aspect of the extend relationship. Borrow Videos was a complete and whole use case before we added the birthday letters, so it does not need to change.

Use Case Diagram Extend Notation

To show an extend relationship on the use case, UML specifies a dotted line, with the arrow pointing to the base use case (please refer to Figure 4.2).

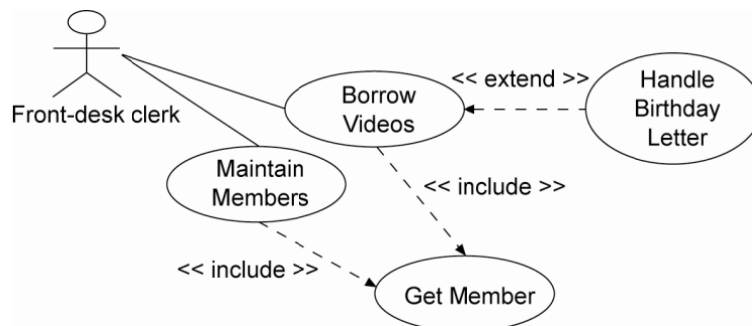


Figure 4.2: Depicting the extend relationship

A very common use of the extend relationship is for some action that can be taken at any time. A common example is the ability to print the information on the screen at any time.

So, for example, if we had the Maintain Members use case, then at any time the front desk clerk should be able to print out a summary of the member.



ACTIVITY 4.2

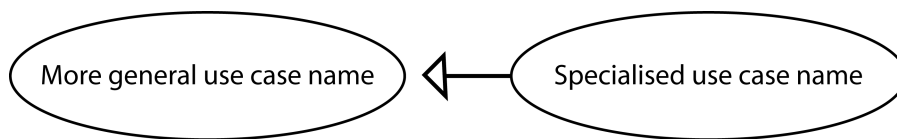
As discussed in the preceding section, a print function is an example of the extend relationship. Write the Maintain Members use case including anything required to allow the user to print. Also draw the use case diagram.

The differences between extend and include relationships are given below:

Extend	Include
<ul style="list-style-type: none"> – Use case with extension (sub) use case – The main use case still complete without sub use case – Just an additional activity 	<ul style="list-style-type: none"> – Use case with common use case – Main use case need this sub use case to complete the process or activity

4.1.3 The Generalise Relationship

This relationship is rarely used, and even more rarely used properly. We include it here simply as a warning to those of you who may come across it in other material. This is what Cockburn (2002, p. 241) says about it. Use case generalisation notation:



In general, the problem with the generalize relation is that the professional community has not yet reached an understanding of what it means to subtype and specialize behaviour, that is, what properties and options are implied. Since use cases are descriptions of behaviour, there can be no standard understanding of what it means to specialize them.

Case Study

The following e-book chapter from OUM's digital library (Books24x7) contains a case study of constructing use case diagram. You are required to read this chapter.

*Schmuller, Joseph. "Hour 7 - Working with Use Case Diagrams". Sams Teach Yourself UML in 24 Hours. Sams. © 1999. Books24x7.
<<http://common.books24x7.com/toc.aspx?bookid=414>>*

4.2 FINDING ALL USE CASES

So how do you know when you've found all the use cases?

In the previous online reading, Jacobsen gives a rule of thumb for the number of use cases. He says, "a large system supporting one business process might have no more than 20 use cases". This number does not include the sub-use cases, nor the generic type use cases, such as logon or logoff. Think about Victoria's Videos, where there is really only one small business function, that is, renting out videos. We have a lot fewer than 20 use cases, so this rule seems to hold. However, in practice, with large systems, it is sometime hard to say what constitutes one business process.

Another way of looking at how many use cases there should be is to note the rate of discovery of use cases. Typically, an experienced practitioner will identify about 70 per cent of the use cases pretty quickly. The next section covers some of the techniques employed when working with users to help make that process as smooth as possible.

However, what about the remaining 30 per cent of the use cases? Some of them will become obvious once you start elaborating the initial 70 per cent. There are also two techniques that are very useful. Both require that we work with the conceptual class model. (This is why we are teaching you use case writing and domain modelling in parallel.)

The first technique is to draw state diagrams for the major conceptual classes. On these diagrams we then write the use case names. In so doing, we often discover missing use cases.

The second technique is to check that you have use cases to create, read, update and delete information for every conceptual class.

4.2.1 State Diagrams

A state diagram shows the life cycle of an object: what events it experiences, its transitions and the states it is in between these events. Like many UML diagrams, the state diagram provides a basic notation that can be used for a variety of purposes. Our purpose at this stage is to draw a picture of the life cycle of the conceptual classes that the system is all about. Usually in a system, there are only a few conceptual classes that really require state diagrams. It is not wrong to draw state diagrams for all the conceptual classes, but it is probably a waste of time, because doing all of this drawing will not show us anything new. So the key is to quickly work out which ones you need to do.

The following table gives some examples of systems and a list of their conceptual classes. The third column suggests the conceptual classes that probably require a state diagram. As you can see, the words in the third column are really the core of each system.

Table 4.7: Examples of Systems and Conceptual Casses that Require State Diagrams

System	Conceptual Classes	Conceptual Classes that Require a State Diagram
Order entry	Order, customer, product, receipt, shop, etc.	Order, customer, product
Asset management	Asset, purchase, sale, location, etc.	Asset
Prisoner management	Prisoner, bed, sentence, medical report, court orders, etc.	Prisoner
Student enrolment	Student, course, university, lecturer, etc.	Student, course



ACTIVITY 4.3

For Victoria's Videos, decide which conceptual class(es) require a state diagram.

As an example, let us consider a prison system.

Suppose we have just started work and we have identified three use cases:

- (a) **Admission** – Handles the prisoner being transferred from the police or courts to the prison, and being admitted.

- (b) **Transfer** – Handles transferring a prisoner to a half-way house.
- (c) **Release** – Handles the way in which a prisoner is released back into the community.

In parallel with identifying and writing use cases, we are also working on the conceptual class model. Quickly we realise that Prisoner is a key conceptual class, so we start to draw its state diagram.

What does a state diagram look like? Here is an example (please refer to Figure 4.3).

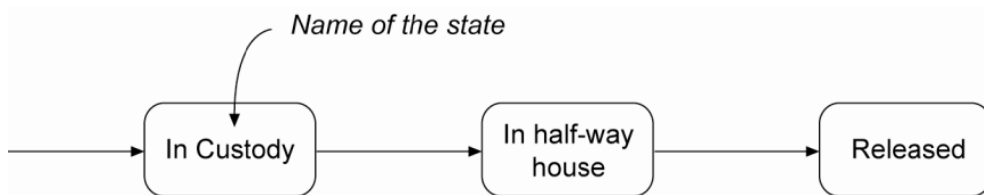


Figure 4.3: State diagram for Prisoner – version 1

Note: Half-way house is a place to stop midway on a journey.

We have boxes with rounded corners to denote each state. Each state must be given a meaningful name. Notice that the name of the last state ends in “ed” – this is a very common ending for English state names.

What this diagram shows is that there are three states that a prisoner can be in.

Next, we look at how our use case model supports these states. So what we do is show the use cases on the state diagram (please refer to Figure 4.4). So our diagram is now this:

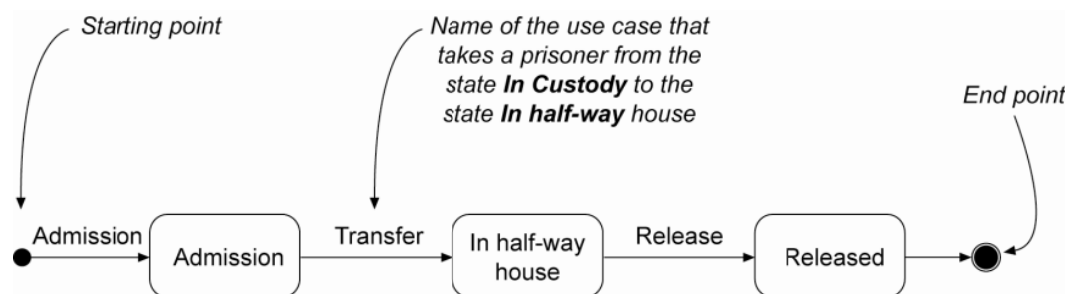


Figure 4.4: State diagram for Prisoner – version 2

Notice, too, the starting and end state symbols. These denote the first and last states, respectively.

So now we could show this diagram to the users to confirm that we are correct in our understanding. Very likely a person who understands prison systems would say:

“Well, that is all fine, but what if a prisoner escapes?”

After some discussion with the user, we could redraw the diagram like this (Figure 4.5):

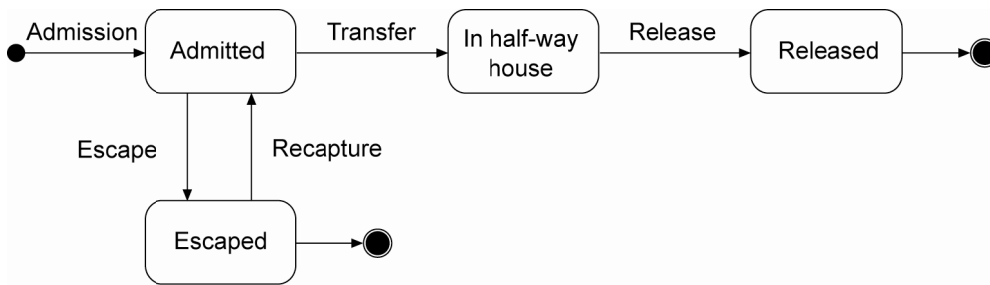


Figure 4.5: State diagram for Prisoner – version 3

This new version has a new state called Escaped (notice that it ends in “ed”), and we have identified two new use cases, namely Escape and Re-capture. This is the crucial point of drawing state diagrams – they help us to quickly find missing use cases.

Notice in this last diagram that we now recognise that some escaped prisoners never get recaptured.



ACTIVITY 4.4

For Victoria’s Videos, draw two separate state diagrams, one for Video Tape, and one for Member.

Did you identify any missing use cases?

4.2.2 Showing Super States

Super states are a technique for simplifying state diagrams. After drawing “State diagram for Prisoner – version 3”, we could show it to another user who might say:

“That is fine, but prisoners can also escape from being in the half-way house. Of course if we capture them, then they go back into custody.”

So the diagram could look like this:

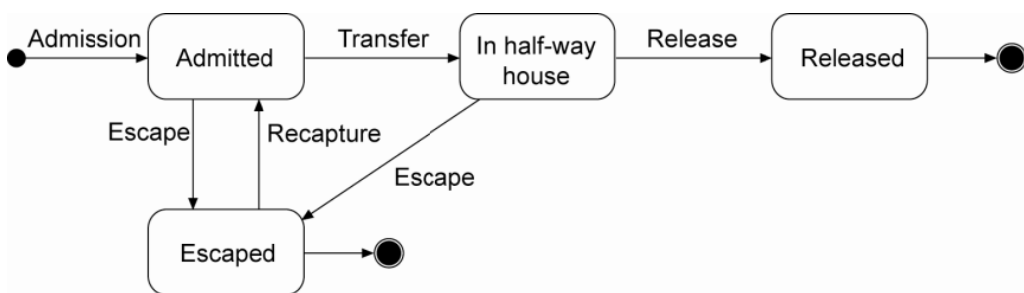


Figure 4.6: State diagram for Prisoner – version 4

You will notice that there are two transitions, or use cases, called Escape. Now since these refer to the same use case, we can simplify the diagram by introducing the concept of a super state (please refer to Figure 4.6).

In the next version you can see that there is a super state called Admitted. So what we are now saying is that if a prisoner is In Custody or In half-way house, and they Escape, then they will be in the state Escaped. Once they are in the state Escaped, if they are re-captured, they go to the state In Custody (please refer to Figure 4.7).

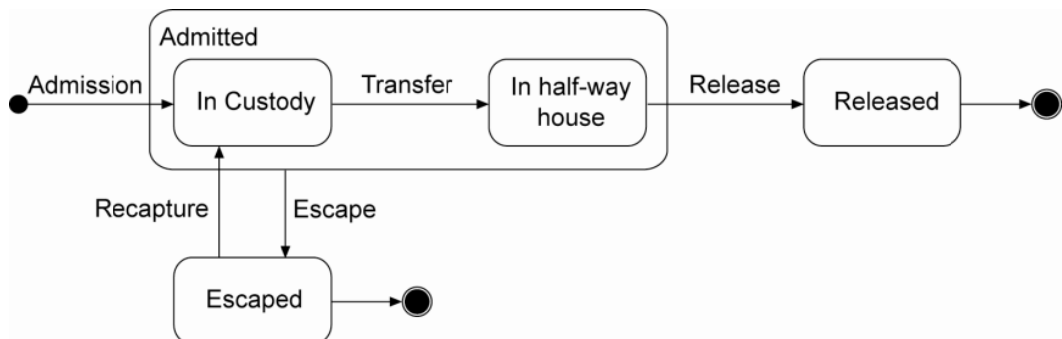


Figure 4.7: State diagram for Prisoner – version 5

4.2.3 Create + Read + Update + Delete = CRUD

Now we look at the second technique that helps us to find missing use cases.

The acronym “CRUD” stands for Create, Read, Update, and Delete. We need to examine every class in the conceptual class diagram to ensure that either we have all of these operations, or we can decide not to include them. For instance, the delete operation is often omitted.

In Victoria’s Videos, there is the conceptual class Member. In an informal review of the conceptual class model and use cases, this conversation could easily occur:

Sarojini Do we have a use case to create members?
(*reviewer*):

Tiveesha Yes, it is called “Create Members”.
(*designer*):

Sarojini: Do we have any use cases that read members?

Tiveesha: Yes, we have a few. The main ones are Maintain Members and Borrow Videos.

Sarojini: Does the system support updating members?

Tiveesha: Yes, again that’s the Maintain Members use case.

Sarojini: And lastly, what about delete?

Tiveesha: Same answer again – because in this system updating and deleting are pretty straightforward we have combined them into one use case.

Sarojini: Good idea. Now, moving on to another conceptual class ...

In this conversation, Sarojini has confirmed that Tiveesha has considered all the CRUD operations for the conceptual class Member. Her next set of questions would be about another conceptual class, such as Video.



ACTIVITY 4.5

Continue Sarojini's review and look at the Video Specification class.

What do you find?

4.3 BUSINESS RULES

We are now approaching the end of requirements analysis. (Bear in mind that “approaching the end” here means the end of the discussion of requirements analysis. In practice, since we are doing things in the iterative manner in UP, all the tasks in requirements analysis will span the different phases in the UP.) Before we move on to the discussion of design, there is a very important topic that we cannot miss: business rules.

Business rules are probably the most basic part of requirements analysis and should be done concurrently with the use case modelling. The following reading explores the relationship between business rules and use cases more fully:

Gottesdiener, E. (1999). “Capturing business rules”, *Software Development Magazine*, 7(12), at
 <http://www.ebgconsulting.com/pubs/articles/businessrulesrule_gottesdiener.pdf>

In fact, besides their role in the use cases, business rules are also linked to other artifacts in the software development process. The following reading discusses such links, in particular to user interface designs, data cleansing and validation.

Ambler, S (2000) “Object-oriented business rules” at:
 <<http://www.sdmagazine.com/documents/s=826/sdm0006j/>>

In the last section of the reading, Ambler mentions the Object Constraint Language (OCL) as a means to express business rules. Other experts disagree. For example, Gottesdiener says that business rules should be written in a language that the users understand. We also like to adopt this approach in writing business rules. The following are some options that we suggest can best express business rules:

- (a) list all the rules in one list, and then refer to the list from everywhere else – use cases, screen mockups, class diagrams, etc.;
- (b) simply locate the rules in the use cases, as we have done in this course; and

- (c) keep the use cases simple, but add a section to your use case template to explicitly refer to the business rules that each use case implements.

So, as you can see, there is a range of possibilities. Additional factors to consider are the style of system (for example, does it involve lots of calculations), the size of the application, the tools that are being used, etc.

So what are the business rules in Victoria's Videos? Here are some examples:

- (a) Only members can borrow videos;
- (b) When a new person requests to become a member, they must show their driver's license or other photo ID;
- (c) The minimum age is 16 years old;
- (d) A member can borrow any number of videos, as long as they have no overdue videos;
- (e) There are fines for overdue videos;
- (f) The length of time that a video can be borrowed for depends on the video. New releases are only lent out overnight, current releases are for three-day hire and the rest are for a week;
- (g) Members can reserve a video;
- (h) Every video has a classification, (general G, parental guidance PG, mature audiences MA, and restricted R). Members must be over 18 to borrow R videos. Every video also has a category: romance, general, sci-fi, foreign language, and children;
- (i) When a member has a birthday, they are sent a special letter inviting them to borrow any video of their choice for a week for free;
- (j) Every three months the shop does a stock take; and
- (k) Any missing videos are updated to be shown as missing.

You should recognise most of this text as it is pretty close to the initial description that you saw from Victoria Videos Case Study in Topic 1. When describing a system, one very common method is to list and describe the business rules. The skill of a requirements analyst is to capture these rules and make them into something that the rest of the design and programming team can work with. Use cases are a key way of weaving the business rules together to create a system.



ACTIVITY 4.6

For each of the example business rules we have just given for Victoria's Videos, decide if the rule has been captured in any use case(s). If so, name them; or, you might note that a business rule has not yet been captured in any use case.

The previous exercise should have helped you to appreciate the fact that quite a strong link exists between use cases and business rules, but that each form is quite different.

4.4 MORE MODELLING

So where are we up to? We are almost done with our discussion of the requirements analysis. However, at this stage of the UP, our requirements are not complete, so there are likely to be some sections of the system that are more complete than others. We have a conceptual class diagram, and a set of use cases with the business rules. Some of the other work described in Topic 3 may also be done, such as SSDs. We also have some state diagrams for the important conceptual classes. It should be time, therefore, for us to begin some "design" work.

In terms of Victoria's Videos, what do we have?

From Topic 2 and this topic, our use case model is being developed. We have a list of actors and use cases. We have a use case diagram of the whole. Some of the use cases have been written out in detail, while others only have a name. We have an activity diagram to show how some key use cases fit together.

From Topic 3 we have a conceptual class model and from this topic as well we have drawn some state diagrams for the conceptual classes Member and Video Tape.

So now we want to move on to the design model. However, there is a very large gap between the conceptual domain classes and the design classes. To help us bridge this gap, there is a technique called robustness analysis (its also sometimes called "use case analysis"). This technique can help us to produce a preliminary design. It is particularly useful for beginners, but even experienced designers, when faced with a difficult problem, often use this technique.

Robustness diagrams help us to quickly identify design classes and methods. Robustness diagrams are much quicker to do than sequence diagrams because they are a lot less detailed. Robustness diagrams help us cross the divide from analysis to design(please refer to Figure 4.8).

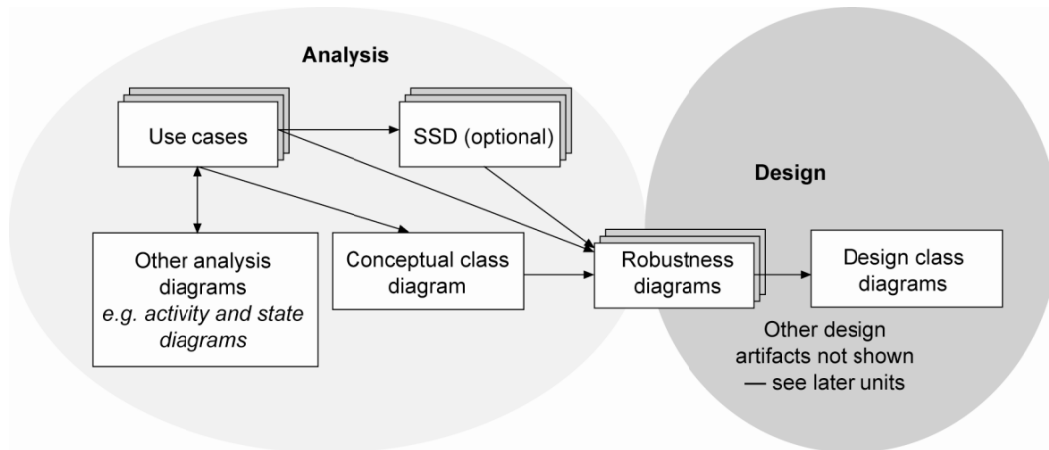


Figure 4.8: Robustness diagrams help bridge the gap between analysis and design phases

Robustness diagrams are not part of formal UML so most CASE tools do not support them. However, the spirit of this type of diagram is that it is really a “throw-away”, that is, it is the act of drawing it that helps you far more than the end product. In fact, as the semantics of robustness diagrams are quite loose, it is quite possible to have many different versions of the same thing.

4.4.1 Robustness Diagrams

This section takes you through drawing a robustness diagram for one use case. Robustness analysis involves analysing the narrative text of use cases and identifying a first-guess set of objects that will participate in each use case. These objects can be classified into three types, namely

- (a) Boundary;
- (b) Entity; and
- (c) Controller.

Each has its own symbol (please refer to Figure 4.9).

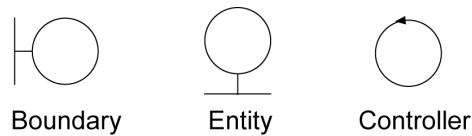


Figure 4.9: Symbols for boundary, entity and controller

Boundary objects represent all connections between the internal objects of the system and the outside world. The most common sort of boundary object is part of a user interface, such as a window or dialog box. Bar code readers are also represented by boundary objects.

For example, in Victoria's Videos we would have objects to control the bar code readers.

Entity objects are the objects that represent data that have to be remembered by the system, either on a more permanent basis beyond the execution of the use case (such data might be stored in a database table) or for the execution of a use case. The conceptual domain model should be the source of many of your entity objects. Typically, you will find new entities that are not on your domain class diagram. This is not a flaw with your conceptual model but a sign that the robustness analysis is helping you.

In Victoria's Videos the objects that we have already identified as conceptual classes will probably all be represented as entity objects.

Controller objects represent anything else you find you need to make the whole diagram make sense. Things like business rules or processes that are captured by a use case will be shown as controller objects.

In Victoria's Videos all the logic that makes up the use case would be in a controller object. In addition, we could put some of the calculations in their own controller object.

We will study an example, but first we need to note the following rules, which the robustness diagram must obey:

- (a) Actors only talk to boundary objects;
- (b) Boundary objects only talk to controller objects;

- (c) Controller objects talk to boundary objects, entity objects or other controllers; and
- (d) Entity objects talk only to controllers.

If you find that you cannot model the diagram in this way, then perhaps you need to add a new object. The set of objects you end up with is a first cut at the set of design classes.

Here is an example of a valid robustness diagram (please refer to Figure 4.10).

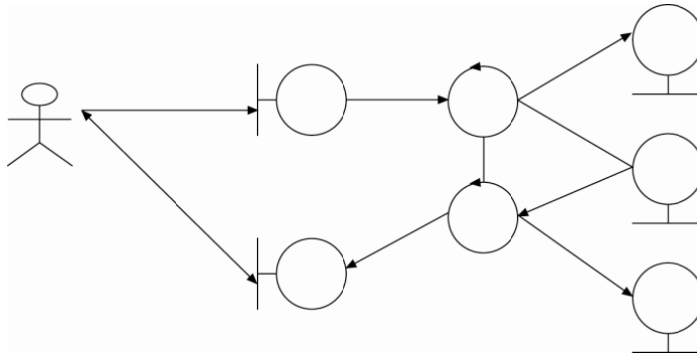


Figure 4.10: An example of a valid robustness diagram

Of course this diagram needs some words to label all the entities so we know what it is about. The directions of the arrowheads are not particularly important, so you can draw them whichever way makes more sense to you.

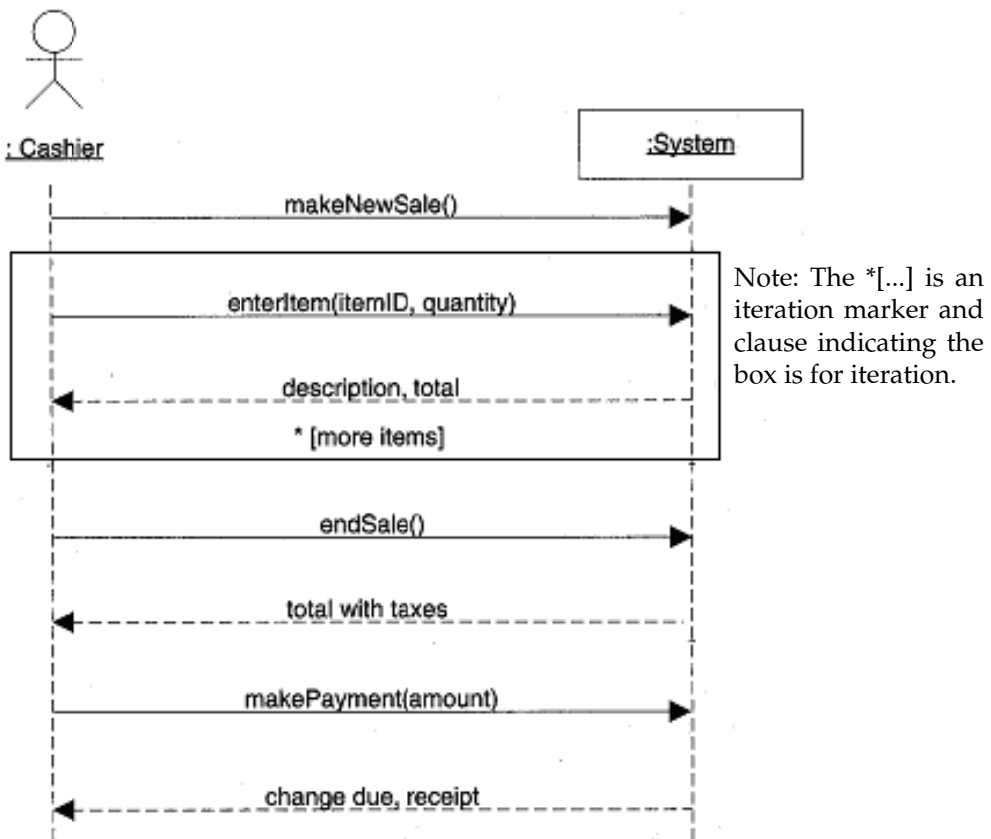
For those who are already familiar with sequence diagrams, note that we are not concerned with detailed messages between the objects – that comes later when we do the sequence diagrams themselves.

So how do you start? Well, you go back to your use cases. Take the text, and, if you did one, its SSD. Start reading through the use case. Ignore any error conditions – in other words, take a “sunny day” approach. Later you can consider how major or likely errors could be handled.

Let us now spend a bit of time working through an extended example of how robustness diagramming can work in practice.

Example

Refer to the figure below which shows the SSD for the Process Sale use case (Larman (2002)):



Source: Larman (2002), page 119

So let us draw a robustness diagram for this. The first line is “Make New Sale”, from the cashier to the system. This is represented by a boundary object. From our rules above, we know that a boundary object can only talk to a controller object, so draw one. It may look like the following.

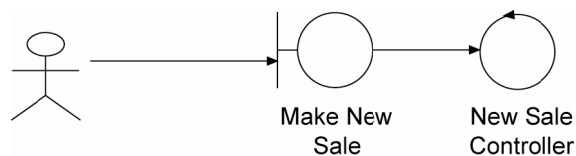


Figure 4.11: Version 1

The next line of the SSD is “Enter Item”. Clearly, this requires another boundary object called Enter Item.

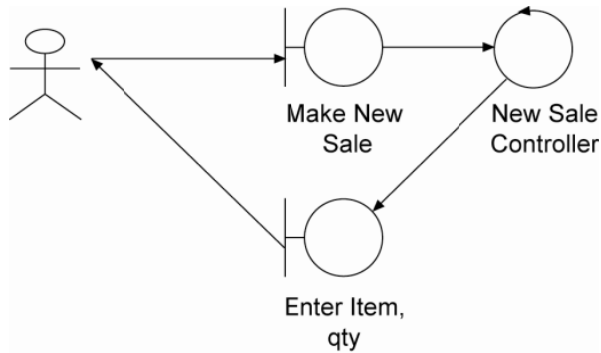


Figure 4.12: Version 2

The next line of the SSD is displaying the description and price. Here we need the entities Item and Product Specification. The rules say that only controllers can talk to entities, so we draw lines from the New Sale Controller to the Item and Product Specification objects.

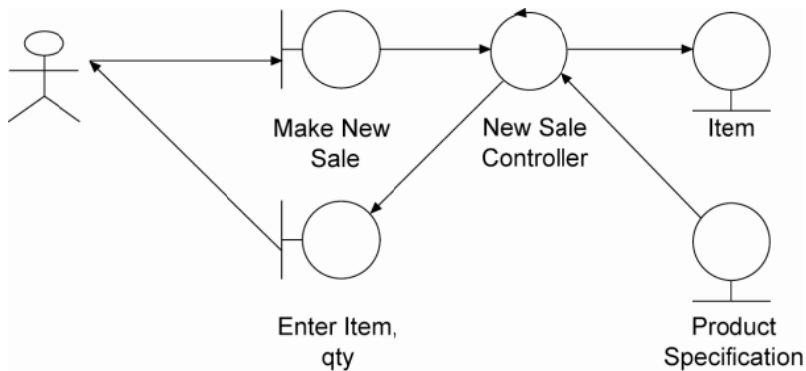


Figure 4.13: Version 3

To display the prices and descriptions, add a new boundary:

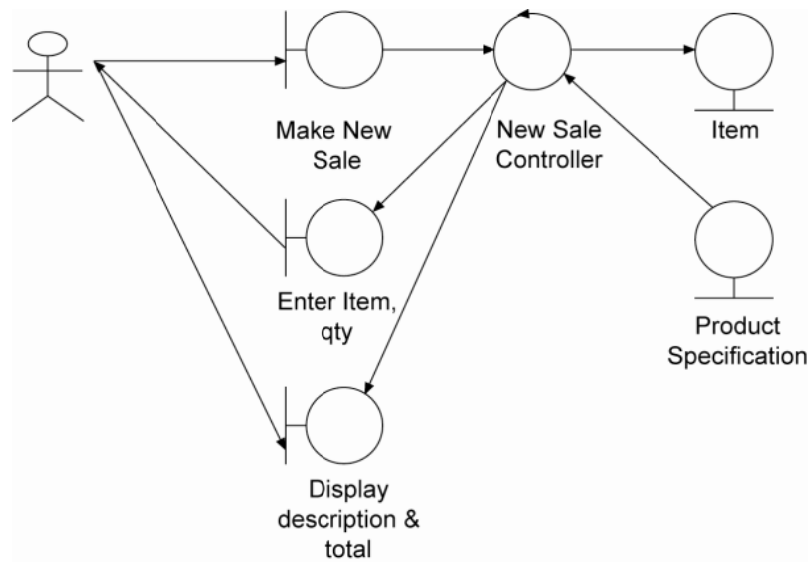


Figure 4.14: Version 4

This keeps on going, but we do not need to show repetition or looping. This is because we are really only interested in finding out what the entities are, rather than in the detailed timing of how they are used – that happens in later topics.

The next interesting event is when the user indicates that he is “done”. Then, we go and calculate the tax. Here we probably need some more details from the Product Specification (for example, many countries have different levels of sales tax on different items). Also, we might need some information on tax rates. So let us put in an entity object called Tax Info with the note “investigate further”. You may recall from Topic 3 that no such conceptual class existed, but Tax Info could be a candidate for being in the class diagram. Once again, the act of drawing a diagram, in this case a robustness diagram, indicates where there are holes in our design.

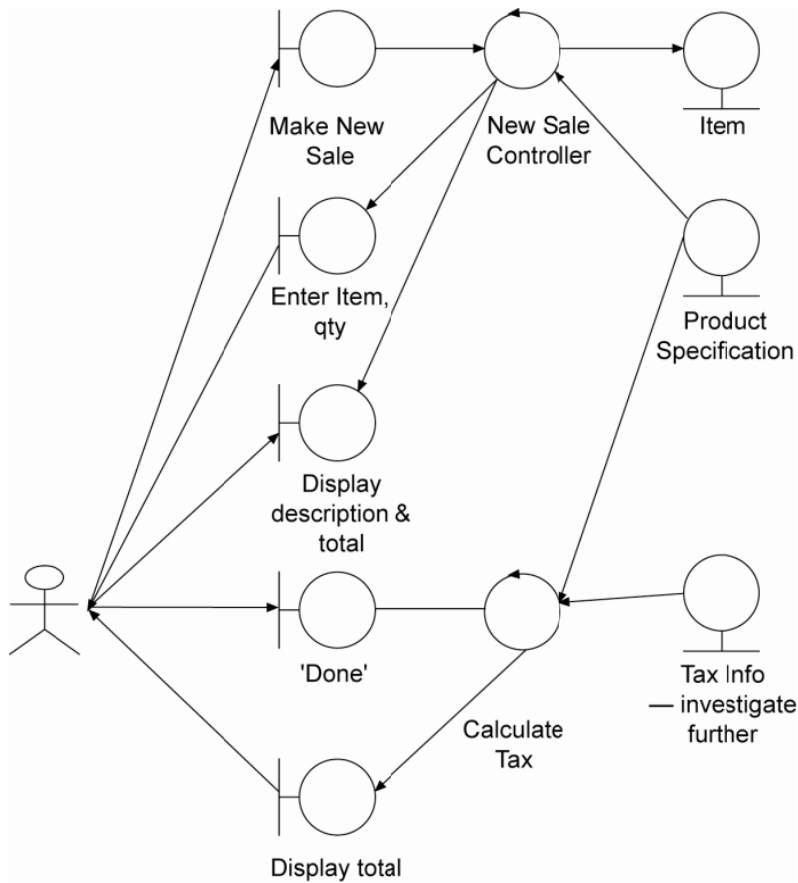


Figure 4.15: Version 5

Next, we need to store all the information collected. Looking at our conceptual class diagram, we see that there are two classes involved here, namely Sale and Sales Line Item.

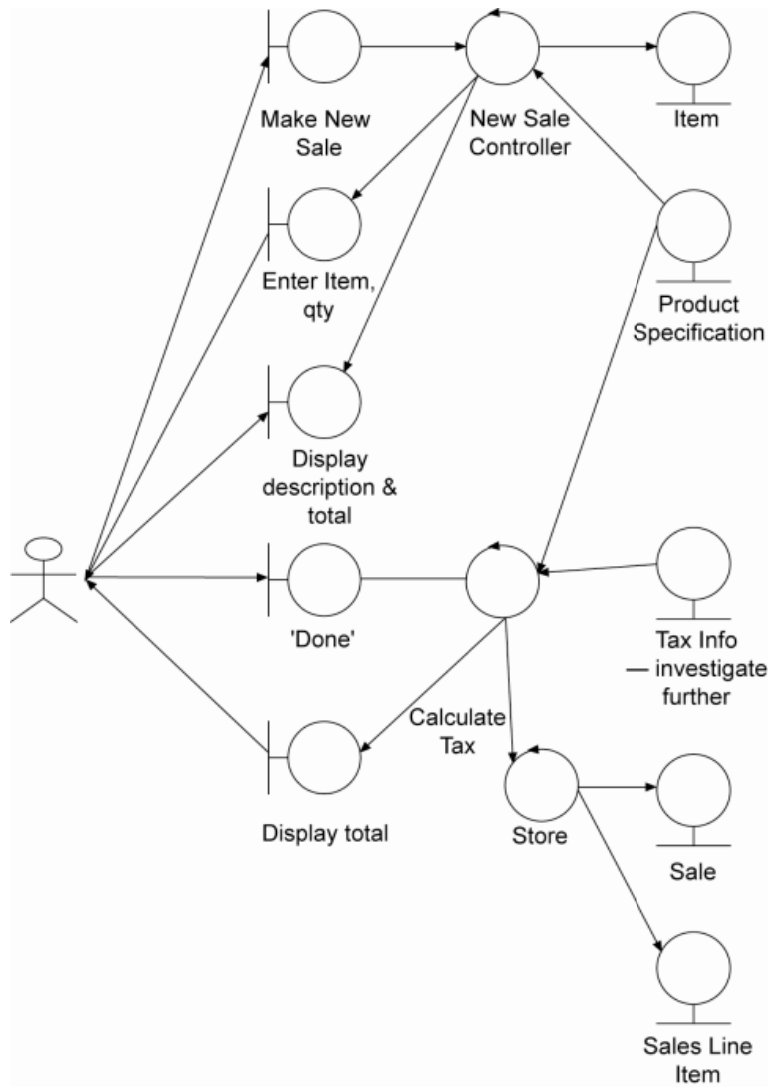


Figure 4.16: Version 6

Lastly we deal with the payment.

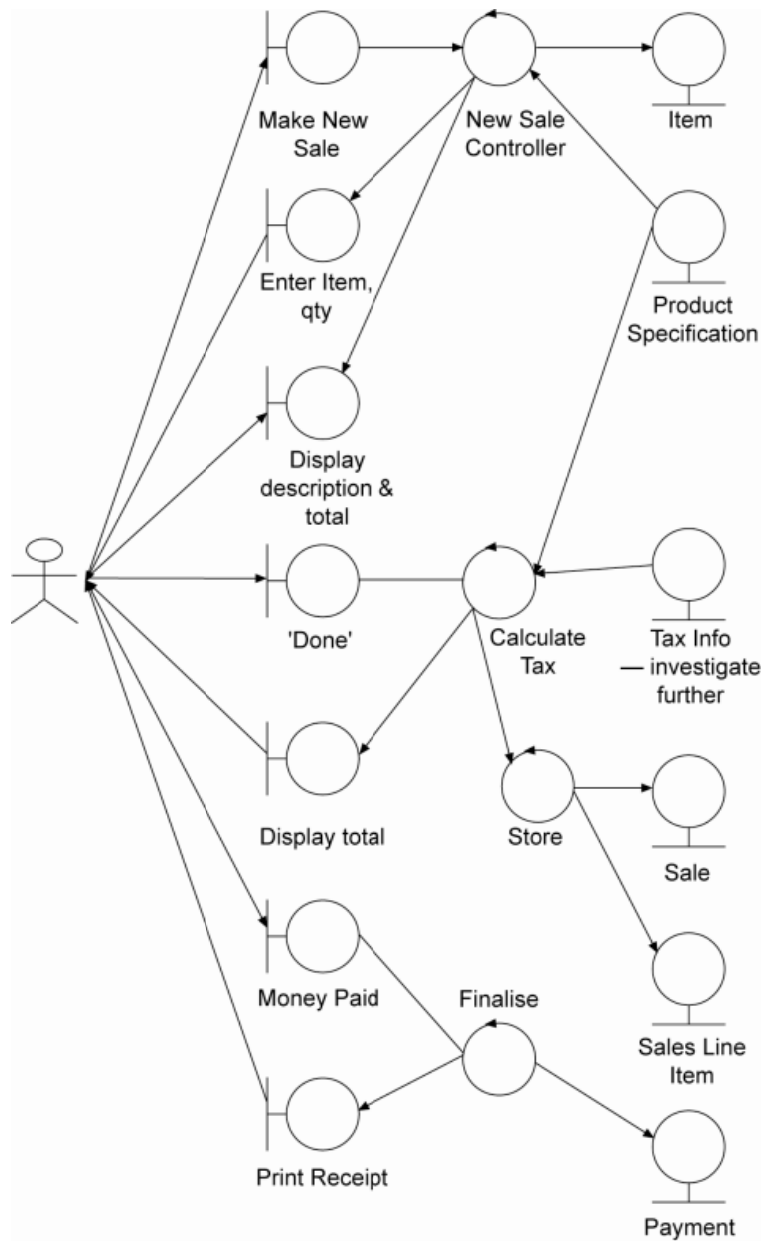


Figure 4.17: Version 7

The following reading will reinforce your understanding of robustness analysis and the way to draw robustness diagrams for your use cases. The reading also indicates some of the common errors in robustness analysis that you should avoid.

“Successful robustness analysis” at <<http://www.sdmagazine.com/documents/s=733/sdm0103c/0103c.htm>>



ACTIVITY 4.7

Assuming you had the following use case for Borrow Videos, draw a robustness diagram to handle the following conditions:

- (a) Use Case – Video Shop – Borrow Videos
- (b) the clerk scans the borrower’s card
- (c) system displays the borrower’s password
- (d) borrower verbally gives password
- (e) clerk scans the videos
- (f) system displays cost
- (g) borrower pays
- (h) the system records info.

SUMMARY

- Use cases are the core of requirements analysis which lay the foundation of the whole development project by guiding us to “do the right things”. We started the discussion of use cases in Topic 2. In this topic, we elaborated on this by discussing how to relate use cases using sub-use cases, including “extend” and “include”.
- One question that always puzzles inexperienced system developers is “how many use cases is enough for this system?” In most cases, there is no definite answer. However, there are techniques that can help us to identify missed use cases. One of these is drawing “state diagrams” of the important conceptual classes in the domain model. Another is to look at the CRUD operations of the conceptual classes. These methods can, in one way or another, help to identify a more complete set of use cases for the project.

- Before we wrap up our discussion on requirements analysis, we need to understand the rules of logic behind the business the targeted system is serving. Otherwise, we will definitely end up with a system that cannot function properly. The business rules of the company or organisation have to be studied in detail. Business rules not only can help us do a better requirements analysis, they will also be referred to by other models later in the design, implementation and testing stages.
- We also took a look at a tool that can help to bridge the gap between the requirements and design – robustness analysis. Drawing robustness diagrams can help us to identify a first-guess set of objects in the use cases.

KEY TERMS

.....

<<extend>>

State diagram

<<include>>

Use cases

Robustness diagram

Use cases diagram



REFERENCES

- Ambler, S. (2000). Object-oriented business rules. Retrieved from: <http://www.sdmagazine.com/documents/s=826/sdm0006j/>
- Cockburn, A. (2002). *Writing effective use cases*. Reading, MA: Addison-Wesley.
- Gottesdiener, E. (1999). Capturing business rules. *Software Development Magazine*. Retrieved from: http://www.ebgconsulting.com/pubs/articles/businessrulesrule_gottesdiener.pdf
- Jacobson, I., Christerson, M., Jonsson, P., & Overgaard, G. (1992). *Object-oriented software engineering, a use case driven approach*. Reading, MA: Addison-Wesley.
- Larman, C. (2002). *Applying UML and patterns*. Upper Saddle River, NJ: Prentice Hall.

Rosenberg, D. and Scott, K. (1999). *Use case driven object modelling with UML: A practical approach*. Reading, MA: Addison-Wesley.

Von Halle, B. (2002). *Business rules applied: Building better systems using the business rules approach*. New York NY: Wiley.

Topic 5 ► Dynamic Modelling

LEARNING OUTCOMES

By the end of this topic, you should be able to:

1. Explain the role of design modelling;
2. Explain the purpose of a UML collaboration diagram;
3. Draw correct UML collaboration diagrams for a system;
4. Describe the purpose of a UML sequence diagram;
5. Draw correct UML sequence diagrams for a system;
6. Draw a design class diagram;
7. Define and use inheritance on a class diagram;
8. Define the term polymorphism;
9. Explain the abstract and concrete classes;
10. Describe the design principles of coupling and cohesion; and
11. List other design principles.

► INTRODUCTION

In this topic we carry on from Topic 4 and go into the design stage. You will really get to “see” how an object-oriented system hangs together in terms of software classes. In particular, you will see how the objects pass messages to other objects, and by doing that, you will get an overview of how the system works.

There are two interaction diagrams in UML. Both show how messages are passed between objects. This topic describes both in detail.

By working through the robustness diagrams in Topic 4 and the interaction diagrams in Topic 5 you will learn more about how the software classes will actually work. Thus, the result of the interaction diagrams is to take what has been learned and start work on the design class diagram.

This topic will also discuss the very important object-oriented concepts of inheritance and polymorphism.

The topic concludes with a discussion on the principles of good object-oriented design.

5.1 THE DESIGN MODEL

At this point in the course we are almost done with our analysis of the system. We should have a complete domain model and most of the use cases should have been completed. In other words, we should have a complete idea of “what the system will do”. Now, it is time to begin the design of the system – which, as you may remember, entails describing and documenting how the system will work and ensuring that it works properly. In this respect, we need to come up with a good design based on the artifacts that we identified in the analysis stage. At the end of Topic 4, we introduced the robustness diagram – an important tool that will lead us from the analysis stage into the design stage. In this topic, we look at this process in more detail. First of all, let us take a look at what we are going to do and the artifacts that will be produced in object-oriented design.

Figure 5.1 depicts the artifacts and the relation between them in the design model. As you can see, the robustness diagrams act as an interface between analysis and design.

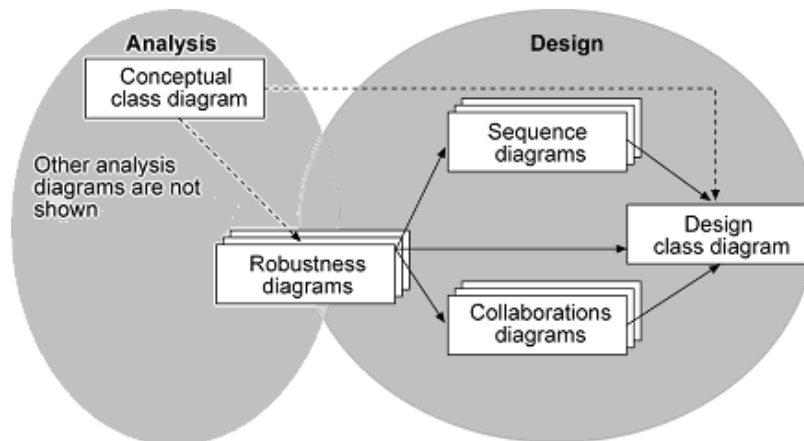


Figure 5.1: The key design diagrams

In the design model, we will produce two sets of diagrams:

- (a) **Design class diagram** – which defines classes and the relationships between them. It represents the static aspect of the design model; and
- (b) **Interaction diagrams** – which define class or object interactions. They represent the dynamic aspect of the design model. There are two types of interaction diagrams, namely the sequence diagrams and the collaboration diagrams.

Notice that in Figure 5.1, while there is one conceptual class diagram and one design class diagram, there are several robustness, sequence and collaboration diagrams. Why is this? Recall from Topic 4 that we draw one robustness diagram for one use case. Similarly, we take each robustness diagram and then draw either a sequence and/or a collaboration diagram for each.

In the UP environment, we have always emphasised that the development of requirements and design artifacts is an iterative process. This approach is also applied in the identification of the software classes in the design model. The development of the two sets of diagrams in the design model also proceeds in this way. They will be defined in parallel and iteratively. Therefore, do not be alarmed if you have a “gut feeling” that your first cut of software classes is not complete. As you perform steps further along in the design and further clarify details of the design, these missing classes will become apparent.

The following steps outline how you would go about identifying the first cut of software classes for the design model:

- (a) Use the requirements model, use cases or domain model and find the nouns. These will most likely be classes;
- (b) Use the technique of robustness analysis to identify additional classes;
- (c) Suggest responsibilities;
- (d) Identify attributes; and
- (e) Identify operations.

We will go into the details of how to proceed with these steps in the coming sections.

5.1.1 Classes versus Objects

Let us first revise the key concepts of objects and messages that have been introduced in Topic 1. This will give us a clearer picture of how design classes can be expressed in the design class model. Let us re-cap the differences between classes and objects. A class is a collection of objects with common structures, behaviours, relationships and common semantics.

Table 5.1: Difference between Classes and Objects

Definition	Real World Examples	How Many?
A class is a definition or specification or a template.	A plan for building a house A template for a CBOP3103 study unit	One
An object is a physical instance created in accordance with the class definition.	A house is built according to the building plan. Study topics are produced formatted according to the template.	As many as required As many as required

In UML, a class is drawn as a rectangle with three compartments: one for the class name and any modifiers, one for attributes and one for operations.

- (a) The class name is a textual string used to identify a class (for example, “Customer”). Each name has a unique meaning in its context.

Classes should be named using the vocabulary of the domain.

Naming standards should be created (for example, all classes are singular nouns starting with a capital letter).

- (b) The structure of a class is represented by its attributes.

An attribute has a name and type separated by a colon “:”.

Attributes may be found by examining class definitions, the problem requirements and by applying domain knowledge. We have learnt in detail about attributes in Topic 3.

- (c) Operations are the behaviour of a class or object that acts upon the attributes in the class or object. Operations can have input (*in*), output (*out*) and input/output (*inout*) arguments.

All arguments have a name and type (separated by a colon) and are prefixed by the term describing what type of argument they are.

Note that we do not recommend the use of input/output (*inout*) arguments.

In programming, a procedure that implements the operation is called a method.

In general, there are three types of operations that could exist in a class diagram namely **constructor operation**, **query operation** and **update operation**. Constructor operation is used to create new objects for a class by giving initial values for the attributes. The name of the constructor operation is the same with the name of the class. Query operation is used to access only the object's state and this operation does not change the state of a object (e.g. getName, getPay, RetrieveID, checkPassword, etc). Finally, update operation is used to update or change the value of attribute(s) of an object. This update operation will change the state of the object (e.g. setName, updateAge, etc).

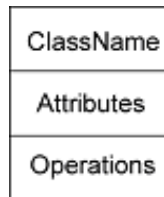


Figure 5.2: Class notation

Figure 5.3 shows the design class notation of the class Member in the video shop example.

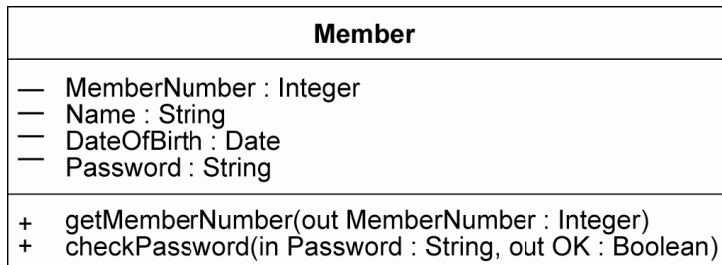


Figure 5.3: Class notation for the member class

The other details within the class notation will be discussed in the coming sections.

To specify an object, we also use a rectangle, but the name is underlined. In the following figure, the first box represents the class Member. The second box is for an object. In this case, the object is named *Jill*. Usually objects are not named – they are just anonymous objects. The third box represents an anonymous object of the class Member.

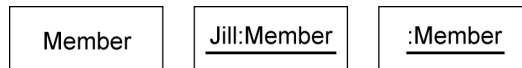


Figure 5.4: Examples of a class, an object and an anonymous object

5.1.2 Objects and Messages

Objects communicate with each other by sending and receiving messages. Sometimes a message has data with it, whereas other times it does not.

We can first think about how people communicate. Here are some examples:

- (a) A host for a dinner party announces, “Dinner is ready.”
- (b) An office worker returns to their desk after the lunch break and her coworker says to her, “While you were out I answered your phone – Please ring Jason Yip on 6343 3472.”
- (c) Two people have met at a bar, and at the end of the evening one says, “What is your phone number?” and the other replies, “3472 3424.”
- (d) A mother gives her child a note, “Here is a note I want you to give to your teacher.”

Table 5.2 shows how each of these messages could be expressed in UML.

Table 5.2: Example Messages Expressed in UML

Words	Style	UML Equivalent
"Dinner is ready."	Single command	DinnerIsReady()
"Please ring Jason Yip on 6343 3472."	Message with data	RingPerson("Jason Yip", "6343 3472")
"What is your phone number?" "3472 3424."	Question with an answer expected	GetPhoneNumber():phoneNumber
"Here is a note I want you to give to your teacher."	Give an object to an object.	GiveToTeacher(Reason:Note)

The table above summarises the examples just given, and in the last column gives you a sample of how the UML message could be written.

Look at the second row. The details "Jason Yip" and "6343 3472" within the parentheses are called **parameters**.

Now look at the third row – ":phoneNumber". The colon tells us that an answer is expected, and the word phoneNumber tells us what sort of an answer is expected.

Lastly, the fourth row is an example of an object, in this case a note for the teacher, being passed as part of a message.

The basic notation to show the message passing is quite straightforward. However, like much of UML, there are many small complexities that only occasionally arise. This course sticks to the basics.

5.2 INTERACTION DIAGRAMS

Remember that two sets of artifacts (class diagrams and interaction diagrams) will be produced based on the artifacts in the analysis stage and the robustness diagrams. These two sets of diagrams will be produced in parallel and iteratively. We will look at the interaction diagrams first.

Interaction diagrams are used to define the class or object interactions within the system. They show the dynamic aspect of the system, for example, the flow of messages. There are two types of interaction diagrams: sequence and collaboration. Both types are used to describe the behaviour of objects to fulfil a single use case.

The two types of interaction diagrams each have their own strengths and weaknesses, and so are used slightly differently. **Collaboration diagrams** (some textbooks use the term communication diagrams) are a bit more intuitive and are useful for learning UML. However, they are less used in industry. **Sequence diagrams** are more complex and show much more detail.

Interaction diagrams are a way to show how one use case will be realised in terms of the software objects. You will notice that, just as with the robustness diagrams, some of the class names are the same or very similar to the names of the conceptual classes. This is expected. However, this time the name refers to the software representation of that thing. In addition, you will find that there are other classes that only exist in software and not in real life.

5.2.1 Collaboration Diagram Notation

Collaboration diagrams depict an interaction among elements of a system and their relationships organised in time and space. These diagrams contain the following elements:

- (a) Classes, denoted as class rectangles, to represent the objects involved in the interaction;
- (b) Association roles, which represent roles that links may play within the interaction; and
- (c) Messages, denoted as labelled arrows, to represent messages sent between objects. The arrow points from the object that sends the message to the receiving object. Optionally, parameters are passed as part of the message; and also optionally, some information, or another object is passed back.

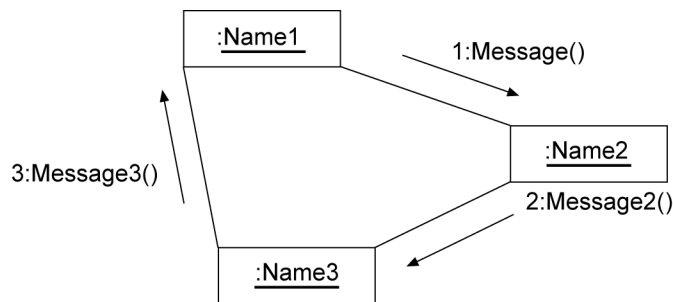


Figure 5.5: Example of a collaboration diagram

Collaboration is the specification of how a classifier, such as a use case or operation, is realised by a set of classes and associations playing specific roles and used in a specific way. The collaboration defines an interaction.

A collaboration diagram shows the messages the objects send each other. A message is represented as an arrow near the association line between two objects. The arrow points to the receiving object. A label near the arrow shows what the message is. The message tells the receiving object to execute one of its operations. A pair of parentheses end the message. Inside the parentheses are the parameters (if any) that the operation works on.

Table 5.3 summarises the details of collaboration diagrams.

Table 5.3: Details of Collaboration Diagrams

Link	<p>A connection path between two objects that allows communication between them.</p> <p>These are the association links on a class diagram. (Note: in Visual Paradigm you need to put a link between objects before you can send a message along the link.)</p>
Messages	The actual message from one object to another. Note that there may be several messages between the same pair of objects.
Messages to “self”	Objects often send themselves messages.
Creation of instances	Objects are continually being created. For example, new sale results in a new transaction record.
Message numbering sequence	Sequence numbers are needed to show the order in which the messages are sent.

Conditional messages	Rarely used. Only send a certain message if a condition is met.
Mutually exclusive conditional paths	A method of numbering to allow two separate message sequences to be followed, depending on the result of a conditional test.
Iteration or looping	To specify that a particular message is sent multiple times.
Iteration over a collection	Very commonly used. Send the same message to a collection of objects, such as a list.
Messages to a class object	This is something we have not yet explicitly mentioned, and is a subtle point. In addition to methods on objects, there is sometimes a need to specify a method that operates at a class level. The primary reason for this is the “create” message. If an object does not yet exist then how does it get created? The answer is that every class has the class method called <i>create</i> . Class methods are also called static methods .

The following e-book chapter from OUM’s digital library (Books24x7) contains more explanation and example on collaboration diagrams. You are required to read this chapter.

Schmuller, Joseph. “Hour 10 – Working with Collaboration Diagrams”. Sams Teach Yourself UML in 24 Hours. Sams. © 1999. Books24x7.
<<http://common.books24x7.com/toc.aspx?bookid=414>>

In addition to the notation in the reading, there is also the concept of a multi object. This is also known as a collection object (such as a list). So a message which is shown as being sent to a multi object, is really a message sent to the collector of that object. The collector then finds the correct object.

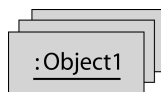


Figure 5.6: A multi object for a collaboration diagram



ACTIVITY 5.1

Look at the following collaboration diagram. It shows the use case of a thirsty drinker coming into a restaurant and requesting a drink from a Bot. Think of a Bot as a robot that interacts with the other parts of the restaurant.

Your task is to translate this diagram into a dialogue between the various components.

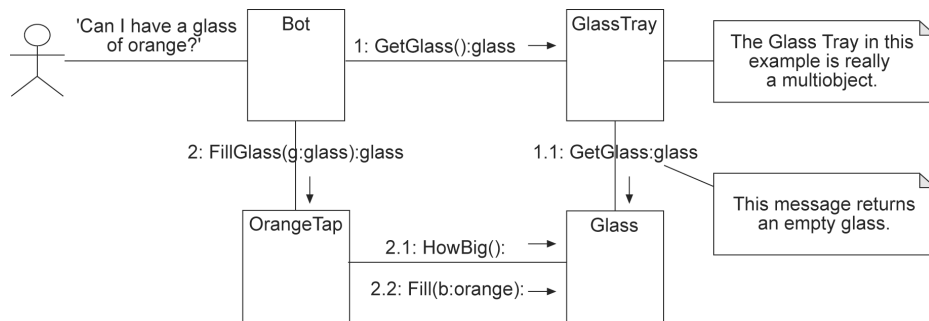


Figure 5.7: A thirsty drinker walks into a restaurant

5.2.2 Sequence Diagram Notation

Sequence diagrams depict an interaction among elements of a system organised in time sequence. These diagrams contain the following elements:

- Class roles, denoted as class rectangles, represent roles that objects may play within the interaction;
- Lifelines, denoted as dashed lines, represent the existence of an object over a period of time;
- Activations, denoted as thin rectangles, represent the time during which an object is performing an operation; and
- Messages, denoted as labelled horizontal arrows between lifelines, represent communication between objects.

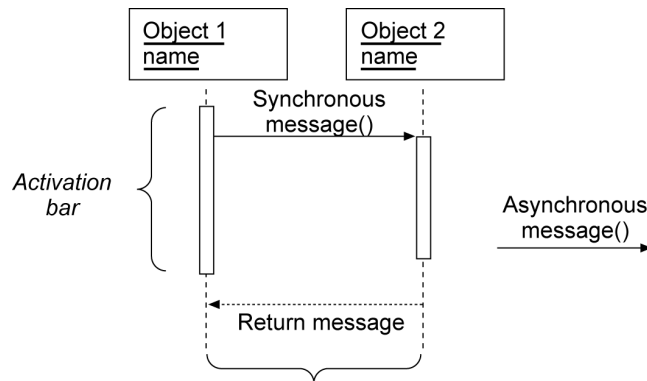


Figure 5.8: Sequence diagram notation

Typically a sequence diagram shows one use case. However, to adequately show how errors are handled, often extra diagrams are required.

Table 5.3 summarises the details of sequence diagrams.

Table 5.3: Details of Sequence Diagrams

Links	Not Shown
Messages	A message is a simple line from one object to another.
Focus of control and activation boxes	This is extremely useful when you are new to sequence diagrams. It allows you to very easily see which objects are active or waiting to get an answer for a message that they have sent.
Illustrating returns	Again, very useful for beginners. Explicitly show when the message is answered.
Messages to “self” or “this”	A simple line from an object to itself. Nested activation boxes, as shown in the text, are not really necessary.
Creation of instances	Very important
Object lifelines and object destruction	Sometimes useful
Conditional messages Mutually exclusive conditional messages	As for collaboration diagrams
Iteration for a single message Iteration for a series of messages Iteration over a collection (multi object)	As for collaboration diagrams
Messages to class objects	As for collaboration diagrams

The following e-book chapter from OUM's digital library (Books24x7) contains more explanation and example on sequence diagrams. You are required to read this chapter.

Schmuller, Joseph. "Hour 9 - Working with Sequence Diagrams". Sams Teach Yourself UML in 24 Hours. Sams. © 1999. Books24x7.

<<http://common.books24x7.com/toc.aspx?bookid=414>>



ACTIVITY 5.2

Take the collaboration diagram in Exercise 5.1 and translate it into a sequence diagram.

5.2.3 Designing for Responsibilities

You have seen what interaction diagrams (either collaborative or sequence diagrams) look like. Now the question is, how do we come up with them? Obviously, the design model has to be based on the artifacts that we have in the requirements analysis. We have a set of use cases and a robustness diagram for all, or at least the important use cases. We will start to produce the interaction diagrams based on these artifacts.

As we work through the use cases or robustness diagram, and we are trying to decide which object should fulfil a particular function, the word *responsibility* is used.

Booch, Jacobson and Rumbaugh (1999) defined a responsibility as "a contract or obligation of a type or class". A class is responsible for knowing things and doing things.

In terms of its actions, an instance of a class may do something itself, may cause another class to do something by initiating an action or controlling or coordinating its actions. These are its types of "doing" responsibilities that must be defined appropriately for classes in a system.

A class's responsibilities might be: knowing about its own data, knowing about a related object or knowing about things it can figure out.

Design is about assigning responsibilities to appropriate classes.

Doing responsibilities of an object include:

- (a) Doing something itself, such as creating an object or doing a calculation;
- (b) Initiating action in other objects; and
- (c) Controlling and coordinating activities in other objects.

Knowing responsibilities of an object include:

- (a) Knowing about private encapsulated data (for example, the glass in Exercise 5.1 knew its size);
- (b) Knowing about related objects; and
- (c) Knowing about things it can derive or calculate.

Example: Process a Sale

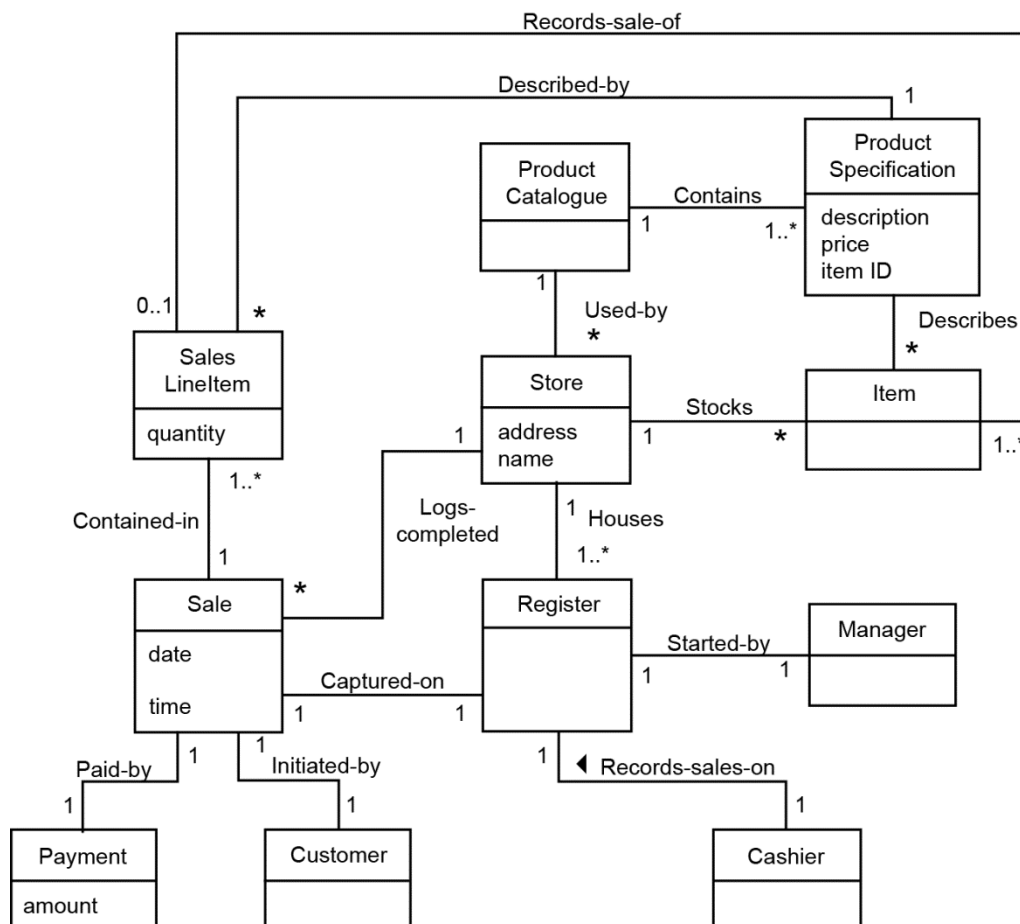
Now we are ready to start work on creating a collaboration diagram for the use case Process a Sale. In Topic 4 we drew a robustness diagram for this use case (have another look at Figure 4.17). There are some important differences between a robustness diagram and a collaboration diagram.

The first important difference between robustness and collaboration diagrams is that the former are not concerned with the actual sequence of events. So, the order that messages are sent will often change between the robustness diagrams and the collaboration diagrams.

The second important difference between robustness and collaboration diagrams is that robustness diagrams show many user interface objects. However, we do not want to complicate collaboration diagrams by showing all these. Indeed, very often they have not yet been defined, so we do not know whether there will be list boxes, plain text or voice-activated inputs. The interface objects of the robustness diagram assist us in seeing exactly what information enters and leaves the system. The interaction diagram goes one step further and shows how the information is processed within the system. So, in other words, we can translate all the robustness diagram interface objects by a single object that we will simply call GUI (“graphical user interface”) which generally interfaces devices between human users and the software in the real system).

Next, we want to encapsulate all the intelligence of the use case into a single object that we will call Register. Alternatively, you could call it the Process Sale Use Case Controller. On a real project, once several use cases have been done, these use case controllers can be re-examined and perhaps combined or tackled in another way.

Where to begin? We start with the GUI telling the Register it is ready to create a new sale. This is a simple create operation on the class Sale. Next the GUI sends the Register the id or barcode of the item and the quantity. Then, we need to find out what the item is, as per our robustness diagram. Let us just check that this is correct. So look at the conceptual class model on page below:



Source: Larman (2002)

What do you notice about the **itemId**? Is this attribute on the Item class? No, it is not – it is on ProductSpecification. However, how do we get to ProductSpecification? From the domain model, we know that it is from the ProductCatalogue. We are trying to get a whole ProductSpecification object back, so note the syntax of message 5 from Figure 5.9. Next the ProductCatalogue object sends a message to the ProductSpecification collection object, denoted with the special notation. This collection object finds the correct ProductSpecification and sends it back.

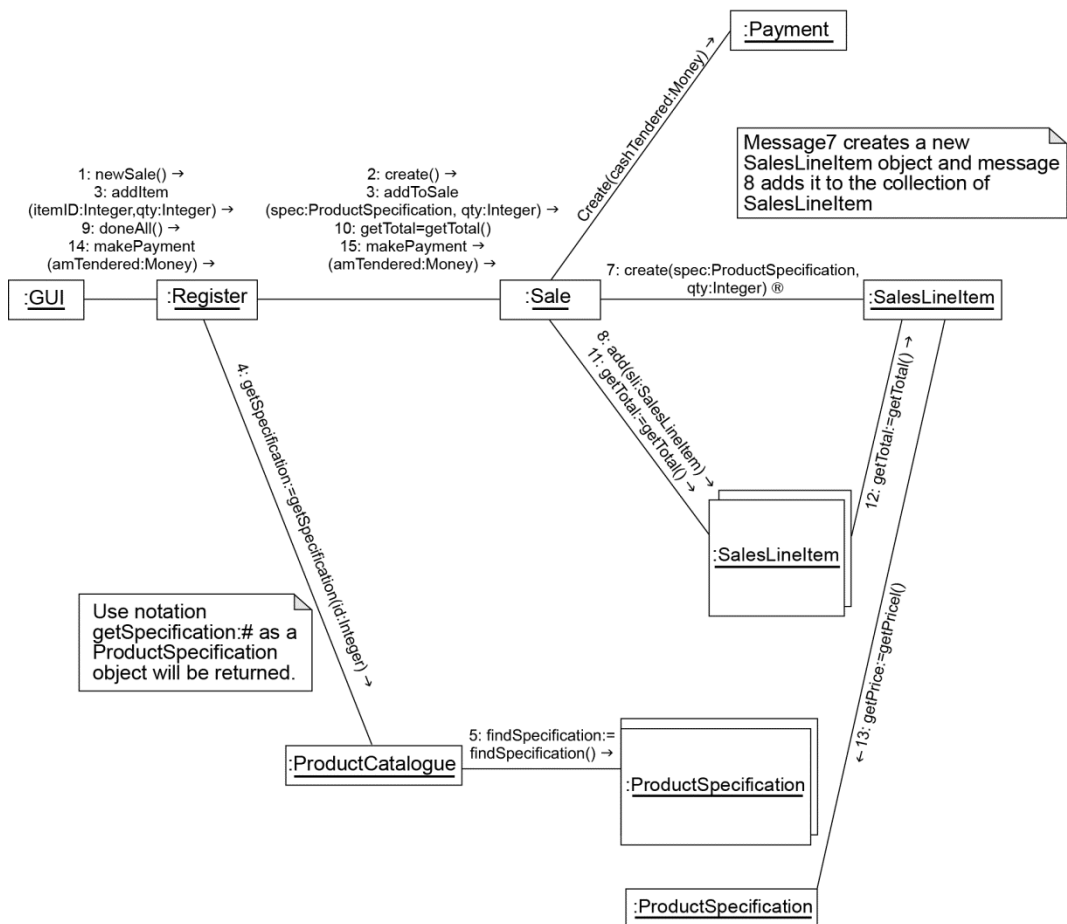


Figure 5.9: Collaboration diagram for process a sale

(Note: At this point we have assumed that all these objects are sitting around just waiting for us to talk to them. In reality, of course, they would be in a database and we would have to do some work to get them out. Getting data in and out of databases is not covered in this module.)

So now we have a `ProductSpecification` object. Should the `Register` object create the `SalesLineItem`? No, as that is a detail that is best delegated to another object. So, as we correctly identified in the robustness diagram, albeit in a different order, the `ProductSpecification` is passed to the `Sale` object, which it then uses to create the `SalesLineItem` in message 7. Message 8 then adds the `SalesLineItem` to the `SalesLineItem` collection object. Message 9 is the GUI telling us that there are no more items, so we need to calculate the total. Once again the `Register` delegates this task to the `Sale` object which then, in turn, asks each `SalesLineItem` for its total. Each `SalesLineItem` asks its `ProductSpecification` for its price.

Lastly, the payment is made (assume it is cash only and that correct money is given) and the `Payment` object is created.

So now we have one collaboration diagram for the use case `Process a Sale`. Of course, we could also show this as a sequence diagram.



ACTIVITY 5.3

Draw a sequence diagram for `Process a Sale`, based on the understanding of the collaboration diagram that has been given in Figure 5.9.



ACTIVITY 5.4

Draw a collaboration diagram and a sequence diagram for the use case `Borrow Videos` in the `Victoria's Video` case.

5.3 DESIGN CLASS DIAGRAM

We have gone through the steps of producing interaction diagrams (either collaboration diagrams or sequence diagrams), which show the dynamic look of the design model. Inside these interaction diagrams, we have basically identified the software classes based on the artifacts from the analysis stage such as use cases and robustness diagrams. However, interaction diagrams show how these software classes interact in terms of the sequence of the messages that have passed back and forth in between these classes. In parallel to these interaction diagrams, we have also a static picture of the relationship between these classes, which depicts the details of the classes – the design class diagram.

A design class diagram describes classes that will exist in software. As we work through each use case and draw interaction diagrams, we figure out what these classes are. There are different ways to come up with a design class diagram. One way is to base it on the interaction diagrams we have. As an example, we can produce our first version of a design class diagram of the video sale simply by:

- (a) Drawing the classes from the interaction diagrams;
- (b) If a message is passed from class A to class B, then draw a line between them; and
- (c) Adding the messages received by a class as one of its operations.

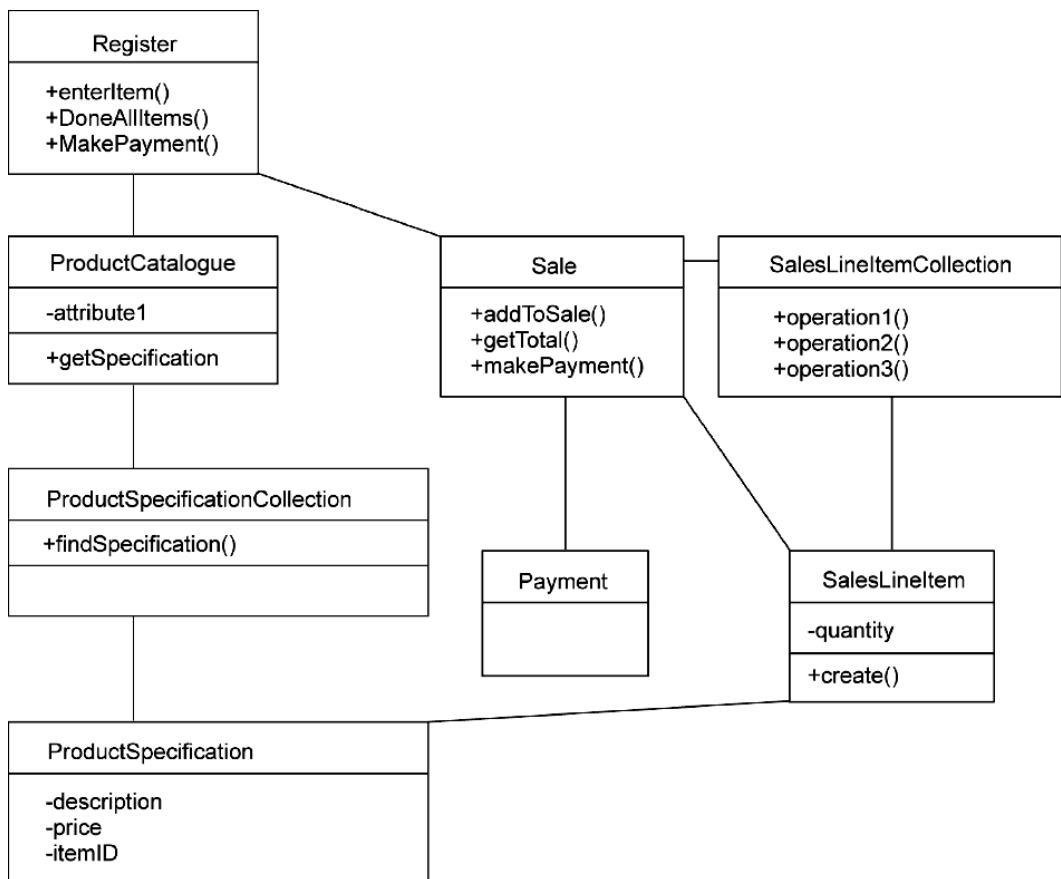


Figure 5.10: A first-cut attempt at the design class diagram

Notice how in this diagram, the names of the classes are not underlined – this means these are classes rather than objects.

As you have learned in the previous section, there are more details of classes that are needed that we have not yet worked on. We will now discuss these individually.

5.3.1 Visibility

The visibility of an attribute or method within a class determines what other objects can access it. The options are:

- (a) **Public visibility** – denoted by “+” in UML (or by an icon in the tool) means that objects of any class can use the attribute or operation.
- (b) **Protected visibility** – denoted by “#”: in UML (or by an icon in the tool) means that only objects of that class or its subclasses can use the attribute or operation. Subclasses are discussed later in this topic.
- (c) **Private visibility** – denoted by “-” in UML (or by an icon in the tool) means that objects of only that class can use the attribute or operation.

It is a desirable design practice to keep attributes of your class private, providing access to them through “accessor” methods. Also, only those methods that must be accessed by other objects should be made public or protected.

5.3.2 Full Attribute Specification

The format of an attribute specification is:

[visibility] name [:type] [= initial value]

- (a) Every part is optional, except the name;
- (b) Visibility was discussed above;
- (c) Type is either a basic type, such as integer or string, or it can be another class. A classic example is the type “money”; and
- (d) Initial value is the value that will be set for every new object of this class.

5.3.3 Association

An association is a connection between classes and is shown as a line connecting the related classes. If there are no arrows on the line, it is assumed to be bi-directional. An association represents that one method or methods of Class B can report about the relationship of Class A to Class B and vice versa. The association can be bi-directional or uni-directional. An arrowhead on the end of the line is used to represent uni-directional navigability. In Figure 5.11, the class User knows about the class Password, but not the other way around.

In the domain model, when we drew the associations on the conceptual class diagram, we were not concerned with how the objects were accessed. It was enough to say that there was an association. When designing the class diagram, we are concerned with how the objects can be accessed. Thus, we may specify the direction of access, as shown in Figure 5.11.



Figure 5.11: An association with one-way navigation

As with the domain model, the multiplicity of an association between classes indicates the number of instances of each class that can participate in an occurrence of the association. For example, a video shop member can borrow one or any number of videos, which is indicated as follows.

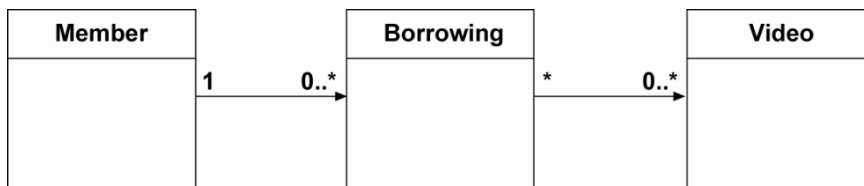


Figure 5.12: Multiplicity of associations between classes

Figure 5.13 depicts the final version of the design class diagram of the Process a Sale use case.

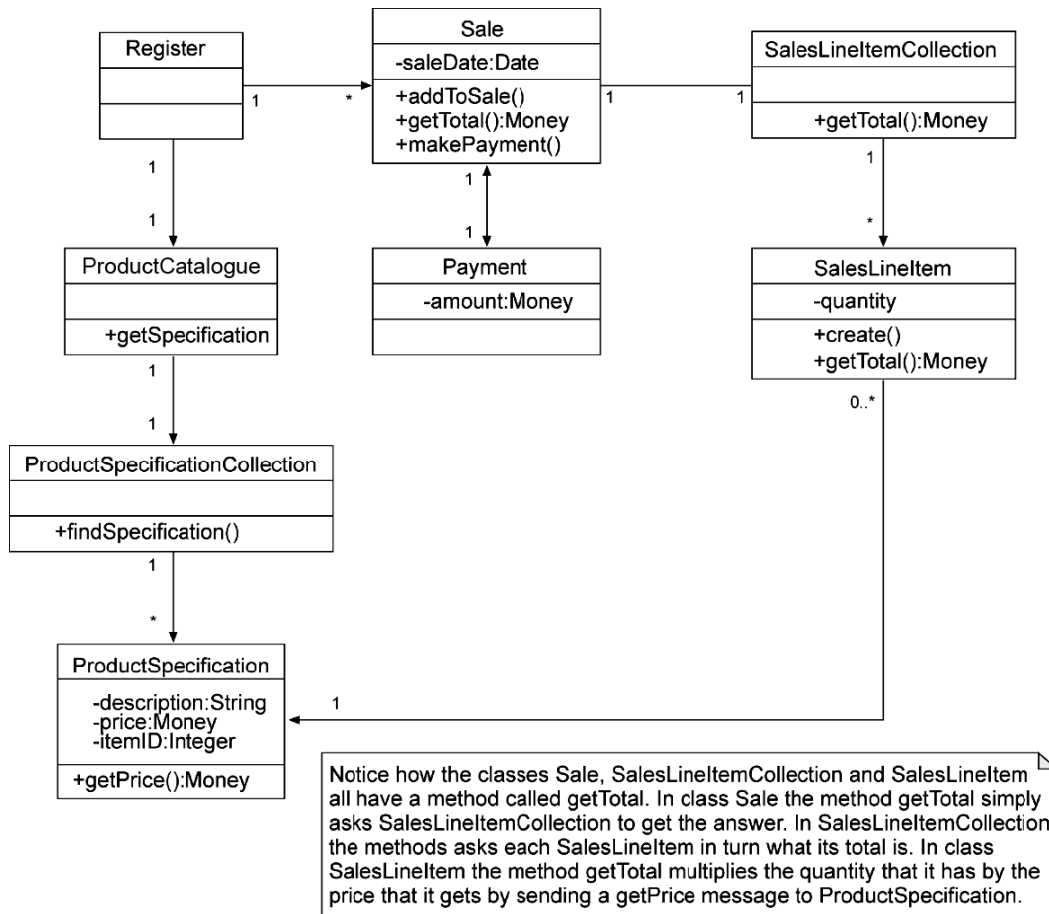


Figure 5.13: A second version of the design class diagram

5.3.4 Composition ("Uses a" Relationship)

A composition relationship, also known as a composite aggregation, is a "stronger" form of aggregation where the part is created and destroyed with the whole. It indicates that one class belongs to the other. Example of composition relationship: A rectangle is made up of several points. If the rectangle is destroyed, so are the points.

A composition relationship is indicated in the UML with a filled diamond and a line as follows:



5.3.5 Aggregation ("Has a" Relationship)

In an aggregation relationship, the part may be independent of the whole but the whole requires the part. In other words, aggregation is similar to composition, but is a less rigorous way of grouping things. Example of aggregation relationship: an order is made up of several products, but a product continues to exist even if the order is destroyed. An aggregation relationship is indicated in the UML with an unfilled diamond and a line as follows:

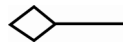
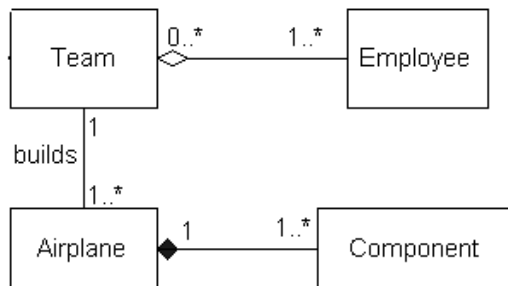


Figure below shows a design class diagram which has association, composition and aggregation relationships. The multiplicity of an association between classes are also shown in the figure below:



Source: Modified from www.agilemodeling.com/style/classDiagram.htm

Notice that with both types of aggregation and composition relationships, the diamond is located on the side of the line pointing to the class which represents the “whole” in the relationship. Aggregation is a special form of association that is tighter than a normal association. Aggregation means that there is a whole-part relationship. Furthermore, the parts usually cannot exist without the whole. Aggregation causes much confusion. As Martin Fowler writes in *UML Distilled*, 2nd edition, page 85:

“One of my biggest bêtes noires [a thing someone particularly dislikes or fears] in modelling is aggregation. It’s easy to explain glibly: Aggregation is the part-of relationship. It’s like saying that a car has an engine and wheel as its parts. This sounds good, but the difficult thing is considering what the difference is between aggregation and association.”

You will probably notice that the design class diagram is very similar to the conceptual class diagram that we produced in the domain model. However, there are some differences. The following table summarises these key differences.

Table 5.3: Key Differences between Conceptual and Design Class Diagrams

Concept	Conceptual	Design
Classes	Yes – represent real world entities	Yes – represent software classes. Some of these will map 1:1 to the real world classes.
Relationships	“Has-a” Navigation is irrelevant	Add direction to the “has-a” relationships Pass messages to classes.
Inheritance (discussed later)	“Is-a” relationship	Inherit code
Cardinalities	Yes	Yes
Attributes	Show key ones, types are optional	Add more attributes. Add types.
Methods	Usually very few	Add many more, with full parameters.

**ACTIVITY 5.5**

Draw a design class diagram for the Borrow Videos use case in Victoria’s Videos.

5.4 INHERITANCE

Inheritance is a vital aspect of object-orientation and we have briefly discussed this concept in Topic 1. Inheritance is a relationship between objects. It is known as the “is-a” relationship. This contrasts with an association relationship, which is the “has-a” relationship. Inheritance relationships can be shown on both conceptual class diagrams and design class diagrams.

Let us look at an example. Imagine you were writing a system to handle vehicle registrations for the Transport Department. This system has to handle many different types of vehicles, namely cars, motorbikes and buses. All of these have some things in common and some things that are different. If we were to draw the classes we might have the following (please Refer to Figure 5.14):

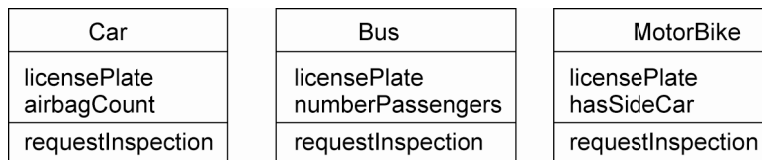


Figure 5.14: Three classes with similarities

So what do you notice? They all have the attribute licencePlate and the operation requestInspection. So we make abstractions of these out into a new superclass called *Vehicle*, and create three subclasses (please refer to Figure 5.15).

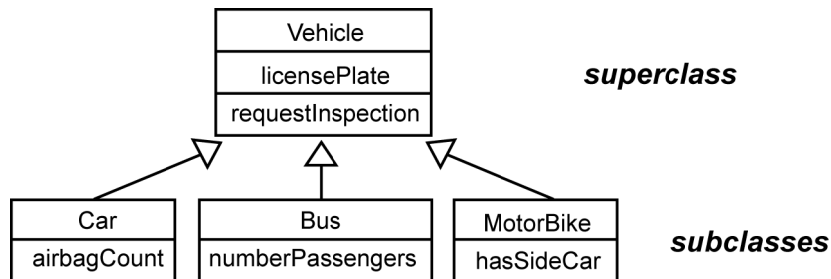


Figure 5.15: An inheritance hierarchy

Note that the arrow is drawn pointing *from* the subclass to the superclass. (Initially you may find this arrow counter-intuitive). The arrows can be drawn in various ways, as shown in figure 5.16 – the choice is yours.

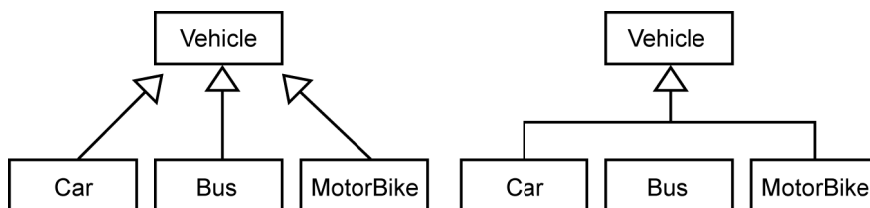


Figure 5.16: Two different arrow styles for an inheritance hierarchy

So we have the general concept of a vehicle and its specialisations. We say that a Car *is-a* vehicle, a Bus *is-a* vehicle and MotorBike *is-a* vehicle.

The superclass holds the attributes and methods that are common to all of its subclasses. Each of the subclasses then can optionally have additional attributes and operations.

As Larman (2002) writes, identifying a superclass and subclasses is of value in a conceptual class model because their presence allows us to understand concepts in more general, refined and abstract terms. It leads to economy of expression, improved comprehension and a reduction in repeated information. Further, when specifying the design classes, inheritance is a powerful tool to simplify the design by expressing commonalities succinctly.

When a subclass inherits from a superclass it inherits *everything* that the superclass has. A subclass inherits all the attributes, operations and relationships of the superclass. Here is an example:

Study the following diagram (please refer to Figure 5.17).

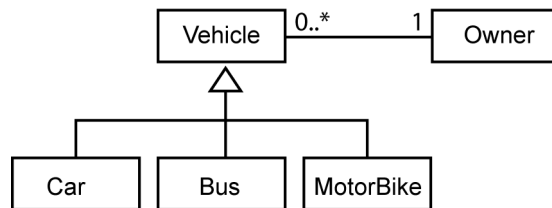


Figure 5.17: Inheritance means inheriting everything

Can we say a Bus *has-a* Owner? The answer is yes, because all relationships are also inherited.

Can an owner own two Cars and a Bus? Yes.

Does every Bus have an owner? Yes.

Can objects change class? This is an important question and a very good illustration of the difference between conceptual and design class diagrams. Let us ask the question, can a bus become a motorbike? Clearly in the real world this is nonsense. However, in another example, it is not so clear.

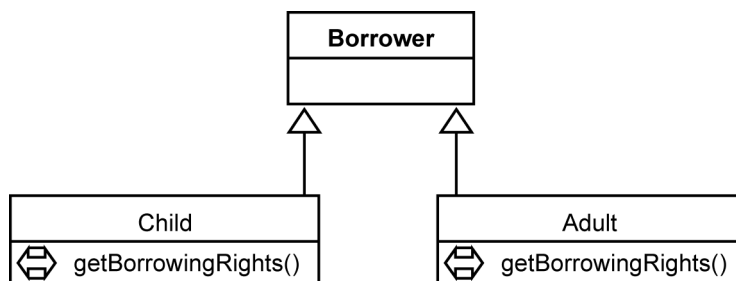


Figure 5.18: A good use of inheritance for a conceptual class diagram

Say in Victoria's Videos we drew a conceptual class with a superclass of Borrower, and two subclasses Child and Adult (please refer to Figure 5.18). There are different rules that apply to each so this makes sense. However, when we ask the question "Does a child become an adult?" Clearly the answer is yes. If this were the design class diagram we would be in trouble, as *objects cannot change class*. Yes, we could always write some code to delete the Child object, and create an Adult object, but that is a very messy approach. For a design class diagram it is far better not to use inheritance, but to use a plain association (please refer to Figure 5.19).

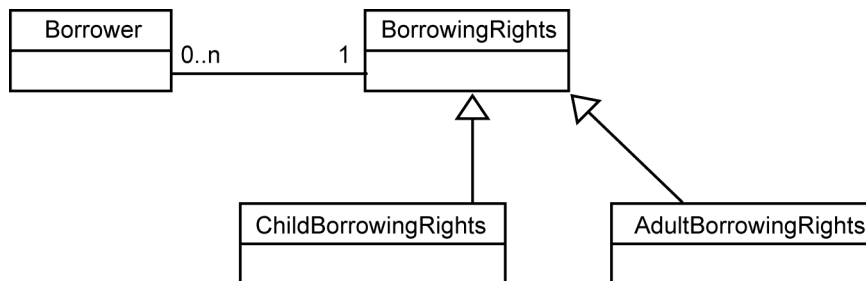


Figure 5.19: A good use of inheritance for a design class diagram



ACTIVITY 5.6

In the NextGen POS system, payments can be made by cash, credit card or cheque. For a credit card payment over \$200, the system must record an authorisation number. To accept a cheque some form of identification (such as drivers licence or passport) must be recorded.

Update the class diagram in Figure 5.13 to reflect this requirement.



ACTIVITY 5.7

In Victoria's Videos there are different categories of videos, namely romance, general, sci-fi, foreign language and children. Show how this information could be reflected on both the conceptual and design class diagrams.

5.4.1 Polymorphism

Polymorphism is very much related to inheritance. Consider the following example (please refer to Figure 5.20):

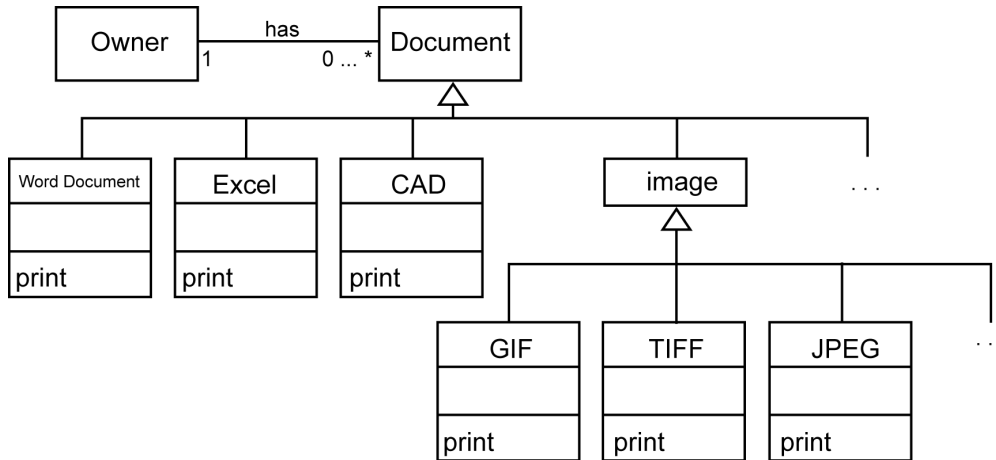


Figure 5.20: Document inheritance tree

The inheritance tree for a number of document types is shown in Figure 6.3. Notice that each of the document types in the tree has a print method. So when an owner object wants to print a document, it simply sends a “print” message to the document. The important point is that the owner object does not need to know what type of document it is before it can issue the print command. This concept is known as **polymorphism** – taking on many forms. Note that polymorphism does *not* imply that objects change class. It means that the exact code of the method will change according to the class of the object that receives the message.

5.4.2 Concrete and Abstract Classes

When discussing inheritance, we cannot avoid discussing abstract and concrete classes. An abstract class is a class that never has any objects. Consider this: would it make sense to have an object Fruit? Clearly not, as the class Fruit is an abstraction of a fruit, and not a fruit itself. In this case, Fruit is an abstract class and we cannot create an object for this class.

A concrete class is class that has objects. Up until this topic, all the classes we have discussed were concrete classes.

Returning to abstract classes, if we cannot create an object from an abstract class, why then may we have abstract classes in a class diagram? Actually, each abstract class represents some idea, although it will not be totally complete as some details and definitions will be left out to the concrete classes that inherit from the abstract class. The name of the abstract class should reflect that it is a concept rather than something specific. So “Account” versus “Checking Account”, or “Vehicle” versus “Truck”.

Two points to note:

- (a) Concrete classes should always be the leaves; and
- (b) Abstract classes should not inherit from concrete classes, as it makes no sense to go from general to specific and back to general as shown in Figure 5.21:

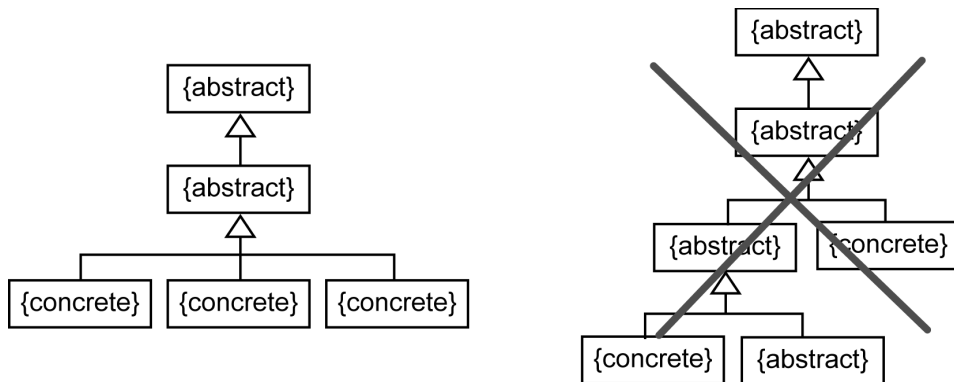


Figure 5.21: Abstract and Concrete classes – allowed and not allowed

There are two ways of specifying abstract and concrete classes in UML. One way is to put the name of the class in italics; the other way is to put the word abstract in curly brackets as shown in the figure below.

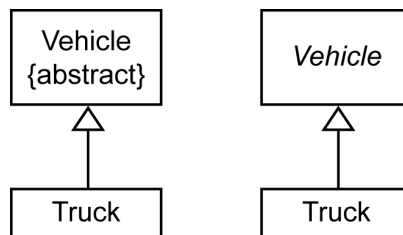


Figure 5.22: Notation for abstract classes

5.5 DESIGN PRINCIPLES

Up to this point we have discussed how to convert our analysis model into a design model. Based on the use cases, conceptual class diagrams and robustness diagrams, we have come up with both the dynamic view and the static view of the design model. For the dynamic view, the UML representations are the sequence diagrams or collaboration diagrams. For the static view of the design model, the UML representation is the design class diagrams. To make the design model work better, there are some design class principles that we need to follow.

This section looks at some principles that a good object-oriented design should embody namely cohesion and coupling. This section is about how you can evaluate a design. In your work to date, sometimes you may have seen that there were choices in how you approached something. For example, should you split a class into two separate classes, or which class should implement a certain method? In this section, we provide some terminology for you to discuss a design in terms of various pros and cons. As with all design endeavours – be it interior design, clothing, town planning and so on – trade-offs need to be made.

As you saw in previous topics, the design classes initially come straight from the conceptual model, and then are added to, changed or merged as a solution develops. As the design develops, it must be constantly evaluated against the following object-oriented design principles.

5.5.1 Coupling

Coupling comes from the structured methods of Edward Yourdon, a developer of object-oriented design models. Coupling represents the strength of association between two classes. Two classes are coupled if one of them uses, refers to the other, or in some way has knowledge of the other.

Looking back at Figure 5.12, we can say that Borrowing and Video are coupled, while Member and Video are not. To confirm this, we could read through the code of the methods of the Member and Video classes to confirm that neither sends any messages to the other.

If a class changes, then all coupled classes will (probably) change. So strong coupling complicates a system in that it is harder to change and understand one class if it is highly interrelated with other classes. Further, tightly coupled classes are difficult to re-use in later systems. Loose coupling seeks to make a system easier to understand, maintain and change. You can classify coupling into different types – the four most basic forms are (see Richter (1999), p. 133)

- (a) Identity coupling;
- (b) Representational coupling;
- (c) Subclass coupling; and
- (d) Inheritance coupling.

Identity coupling measures the level of connectivity of a design. This is shown as an association on a class diagram. Effectively an object “knows” about another object. A one-way association is less coupled than a two-way association.

Representational coupling is a measure of how one object accesses the data of another object.

Accessing the public method foo in the class X, as in Figure 5.23, is a very low-level approach and so has a high degree of representational coupling.

X
+foo : Integer secret : Integer
+getFoo() : Integer

Figure 5.23: Method getFoo is an example of a “getter” – it gets the value of an attribute

On the other hand, using the method getFoo, which would simply give us the value of attribute foo, is a low level of representational coupling. The latter version is preferred as the low-level implantation details of class X are hidden. This is encapsulation.

Subclass coupling occurs when a client class directly references a subclass rather than its superclass. This is to be avoided wherever possible (please refer to Figure 5.24).

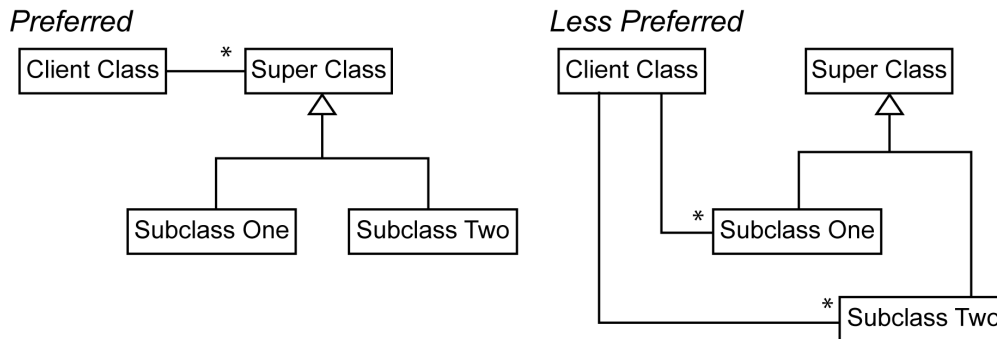


Figure 5.24: Subclass coupling – preferred and less preferred approaches

Inheritance coupling: a subclass is related to its superclass by inheritance coupling. Recall that a subclass inherits everything from its superclass and that this cannot be changed at run-time. In section 5.4, we warned against the possible over-use of inheritance and recommended that association should be considered. See Figure 5.19 for an example.

5.5.2 Cohesion

Cohesion also comes from the structured methods of Edward Yourdon and is a measure of how focused the functionality of a class is. A class has high cohesion if it only represents one idea.

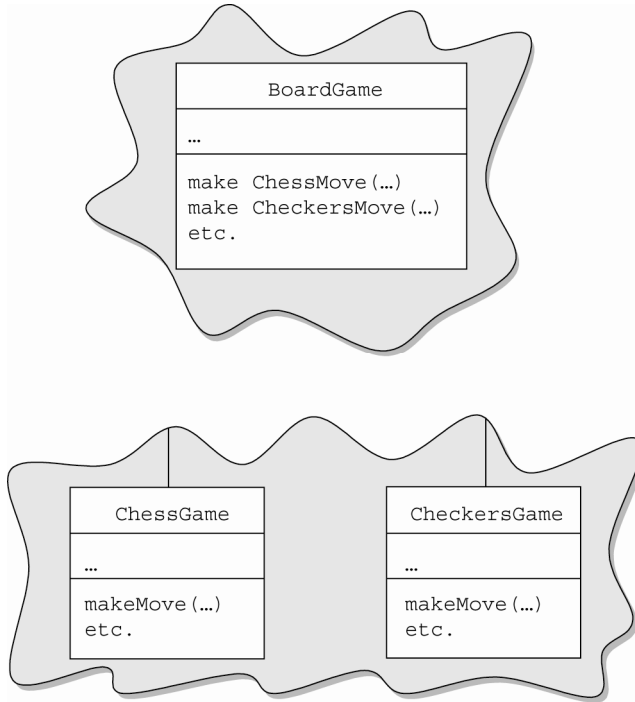
The class `VideoSpecification` defines a particular title of a movie. It tells us something about that movie (its name, the actors, year it was made, ratings and so on). The class `Video` is all about one particular tape of a video specification. Thus `Video` and `VideoSpecification` are separate classes. If they were merged into one, the resulting class would not have high cohesion.

If a class's behaviour is multi-functional, or only part of a function, then it has low cohesion and this is not desirable. A highly cohesive class is easier to understand not only in itself but also in its relationship to other classes. If you know exactly what a class will do (its single function) then you understand exactly how other classes are going to use it. In fact, it facilitates the design of new classes that are going to use the functionally cohesive class.

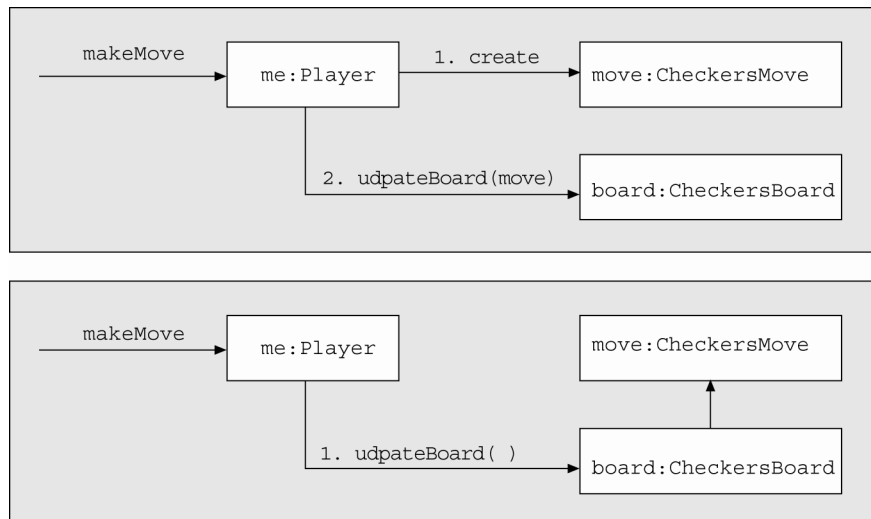


ACTIVITY 5.8

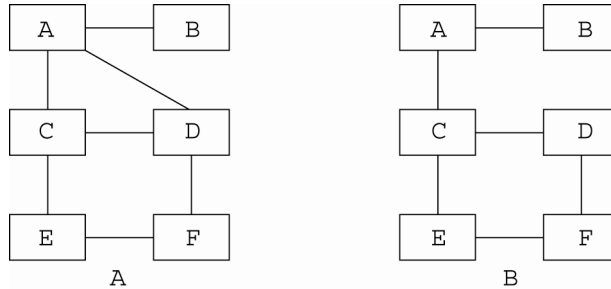
Which of the following two class diagram excerpts shows good cohesion?



2. Which of the following two diagrams shows better coupling?



3. Which of the following diagrams show better coupling?



SUMMARY

- In this topic, we entered the design phase of the system. With all (or most) of the artifacts and documents in the analysis phase being complete, we begin to see how the system can be built and how the software classes and objects can work together to achieve the tasks that we identified in the analysis.
- The design model consists of two broad types of artifacts, namely the interaction diagrams and the design class diagram. The interaction diagrams will give us the dynamic look of the system that indicates the sequence of exchange of messages between the classes. The design class diagram will depict the relationship between the classes and give more details of the structure of the classes and the operations that they can perform.
- We have also studied a very important property of object-oriented design – the inheritance of classes besides polymorphism, abstract and concrete classes.
- At the same time, the principles of good object-oriented design have been introduced.
- In this course, we have learnt object-oriented approach in software development.
- We also focused on Unified Process and UML.

KEY TERMS

Class method	Interaction diagram
Cohesion	Parameters
Collaboration diagrams	Polymorphism
Coupling	Sequence diagrams
Encapsulation	Static method



REFERENCES

- Booch, G. (1991). *Object-oriented design with applications*, Redwood City, CA: Benjamin Cummings.
- Booch, G., Rumbaugh, J., & Jacobson, J. (1999). *The unified modelling language*. Reading, MA: Addison-Wesley. (This is a great reference for UML, providing more detail than you will ever need.)
- Fowler, M., & Scott, K. (2000). *UML Distilled* (2nd ed.). Reading, MA: Addison-Wesley. (This is a short and practical overview to using UML in actual practice.)
- Larman, C. (2002). *Applying UML and patterns*. Upper Saddle River, NJ: Prentice Hall.
- Richter, C. (1999). *Designing flexible object-oriented systems with UML*. Indianapolis, IN: Macmillan Technical Publishing.
- Scott, K. (2001). *UML explained*. Boston, MA: Addison-Wesley. (This book presents the basics of UML.)
- Shalloway, A., & Trott, J. R. (2002). *Design patterns explained: A new perspective on object-oriented design*. Boston, MA: Addison Wesley.

Case Study

Now we have come to the end of the book. In order to make sure that you have understood all the important concepts and UML diagrams discussed in this book, let us discuss a case study. The case study is taken from an e-book chapter available at the OUM's digital library (Books24x7) as indicated in the table below. In going through this case study, follow the task sequence indicated in the table below.

Task	Reference
Introduction to the case study	Schmuller, Joseph. "Hour 16 – Introducing the Case Study". Sams Teach Yourself UML in 24 Hours. Sams. © 1999. Books24x7. < http://common.books24x7.com/toc.aspx?bookid=414 >
Performing domain analysis	Schmuller, Joseph. "Hour 17 – Performing a Domain Analysis". Sams Teach Yourself UML in 24 Hours. Sams. © 1999. Books24x7. < http://common.books24x7.com/toc.aspx?bookid=414 >
Developing use cases	Schmuller, Joseph. "Hour 19 – Developing the Use Cases". Sams Teach Yourself UML in 24 Hours. Sams. © 1999. Books24x7. < http://common.books24x7.com/toc.aspx?bookid=414 >
Developing the interaction diagrams	Schmuller, Joseph. "Hour 20 – Getting into Interactions and State Changes". Sams Teach Yourself UML in 24 Hours. Sams. © 1999. Books24x7. < http://common.books24x7.com

Answers

TOPIC 1: INTRODUCTION TO OBJECT-ORIENTED SOFTWARE DEVELOPMENT

Activity 1.1

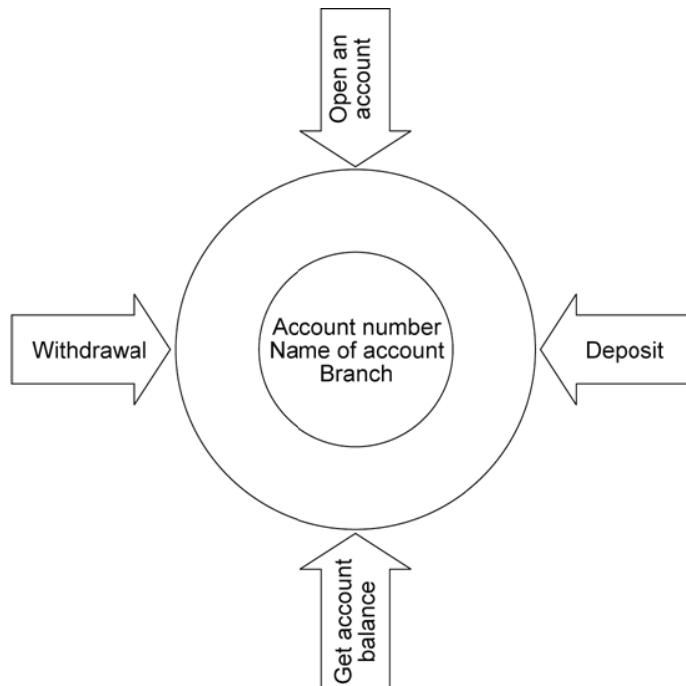
Examples of IT for support:

- mobile phones (for phone calls, text messages, phone books and WAP applications), leave management system.
- PDA (Personal Data Assistant) like a Palm Pilot.

Examples for IT as an “enabling business”:

- Data Mining Applications
- Expert Systems, etc.

Activity 1.2



- (a) messages
- (b) class
- (c) instance

Activity 1.3

- (a) There are several possible answers, including:
 - Risk mitigation – by working on the riskiest aspects of a project we can either address any problems or stop the project early rather than waiting until the end.
 - Provide early and tangible results to users, thus getting greater cooperation.
 - Instil the habit of delivering artifacts regularly. This means the project maintains momentum and does not become mired in fussy details or features that are not absolutely necessary.
- (b) Here are three challenges:
 - An iterative project can be harder to predict, control and manage.
 - Version control is absolutely fundamental as every document and piece of code can be changed several times.
 - Once a system is implemented, you must both support the current system as well as continue development.

Activity 1.4

	Traditional	Object-oriented
Approach to data and functions	Consider separately	Consider together
System development life cycle (SDLC)	Waterfall	Iterative
Notations	Entity-relationship (ER) diagrams Data Flow Diagrams (DFDs)	UML diagrams
Example languages	COBOL, various 4GLs	Java, C++
Database	Relational databases	Usually still use relational databases

Activity 1.5

- **A Student Registers for an OUM Course**

The student receives a registration package from the Registry by mail. He or she looks up the courses list provided in the Prospectus or Courses Supplement. He or she decides which courses he or she would like to register for in the coming semester. He or she fills in the registration form and mails it back to the Registry. He or she receives a payment slip from the Registry. He or she goes to the designated bank to pay the tuition fee for the courses he or she has registered for. He or she receives instructions from the Registry to collect the course materials and the study schedules for the courses.

- **A Customer Purchases an Item from an Online Shop**

The customer turns on his PC and connects to the Internet. The customer opens an Internet browser and keys in the URL of the online shop. The customer looks up the online catalogue for the item he or she would like to purchase. The customer clicks on the purchase icon of the item. The item is added to the online shopping cart. The customer clicks on the online shopping cart and checks whether the item has been added. The customer clicks on the checkout icon. On the payment page, the customer keys in his or her purchase information (credit card number, expiry date, e-mail, address) and clicks the confirm icon.

TOPIC 2: REQUIREMENT AND USE CASES

Activity 2.1

- (a) functional
- (b) non-functional
- (c) functional
- (d) non-functional
- (e) could be regarded as either

Activity 2.2

Functional requirements:

- register members;
- track overdue videos;
- produce a daily upload file for the general ledger; and
- print letters for members on their birthdays

Non-functional requirements:

- speed of the system should allow a video borrowing transaction to be completed within one minute;
- the barcode reader should be at least 99.5 per cent accurate; and
- a front desk clerk should be able to become familiar with all the functions of the system within one week.

Activity 2.3

Objective	Clerk	Teenager
Reliability	1	4
Efficiency	2	3
Satisfaction	4	1
Speed	3	2

Activity 2.4

Customer (or member) – primary or secondary, depending on whether the customer directly interacts with the system. Suggest to the client, Victoria, that for the first version of the system, members are secondary and the front desk clerk interacts with the system. Later versions could allow customers to borrow their own videos.

Front desk clerk – primary – This actor deals with the members – joining, borrowing videos, asking questions.

- | | |
|------------------------|--|
| Back office clerk | – primary – This actor deals with all the back office functions, such as sending data to the general ledger system. (Will probably find later that this “actor” is actually several actors.) |
| Stock taker | – primary or secondary, depending on implementation. Require more information on what a stock take entails. |
| Management | – secondary, as their reporting needs are met from the data warehouse. Need sufficient information to run the business. |
| Security manager | – primary – maintains the user names to control access to the system. |
| Data warehouse | – external – needs a daily update of data. |
| Cash register terminal | – external – handles all financial transactions. |

In addition, every primary user needs to be able to logon, logoff and change their password.

Activity 2.5

Borrow a video – brief format for an initial version of the use case

This use case starts when the member finishes selecting their videos and brings them to the front desk. The front desk clerk asks to see their membership card. The clerk then enters the member’s number into the system. The front desk clerk then scans the barcodes of the member’s videos. The system calculates the price according to whether the videos are new releases, current releases, or other, and the price is displayed on the cash register terminal. The customer pays by credit card or cash, and then leaves the shop with his or her videos.

The most important actor is the front desk clerk because this actor serves the members. After all, if there were no members, then the video shop would go out of business. Similarly, if nobody borrowed any videos, then it would be time to pack up and go home.

Activity 2.6

The following list has the main use cases that you might have identified. Note that there are other use cases, which we will find during later analysis. You might have already thought of some of these yourself. Obviously the more use cases you find earlier, the easier the whole process is.

Actor	Use Cases	Notes or Issues
Front desk clerk	Borrow videos	This use case interacts with the cash register
	Return videos	
	Create members	Need to decide if still using membership cards
	Enter a reservation	
	Members	This includes both changing and deleting
	Search for titles	
Back office clerk	Generate birthday letters	
Stock taker	Do stock take	Not very well understood at this stage
Internal	Generate data for the data warehouse	
Security manager	Maintain valid users	
Everyone	Logon	
	Logoff	
	Change password	
	Get online help	

Activity 2.7**1. Return Videos**

This use case is used when a member returns videos. The front desk clerk scans the videos and the system is updated to show the videos as returned.

(As you can see, some use cases are very simple.)

2. Generate Birthday Letters

The back office clerk performs this use case approximately once a week. The system displays the range of birthday dates that was last produced. The clerk then selects another date range, usually another week. The system produces the letters. The clerk puts the letters in envelopes and posts them.

Activity 2.8

Alternate flows in borrow videos

- The member's number might be invalid. Ask clerk to re-enter the number.
- Member might have overdue videos. They are not allowed to borrow any more videos, so end the use case.
- Member might have fines to pay for videos that were returned late. They must pay the fines as well as the fees for the new videos.

Activity 2.9

Borrow videos

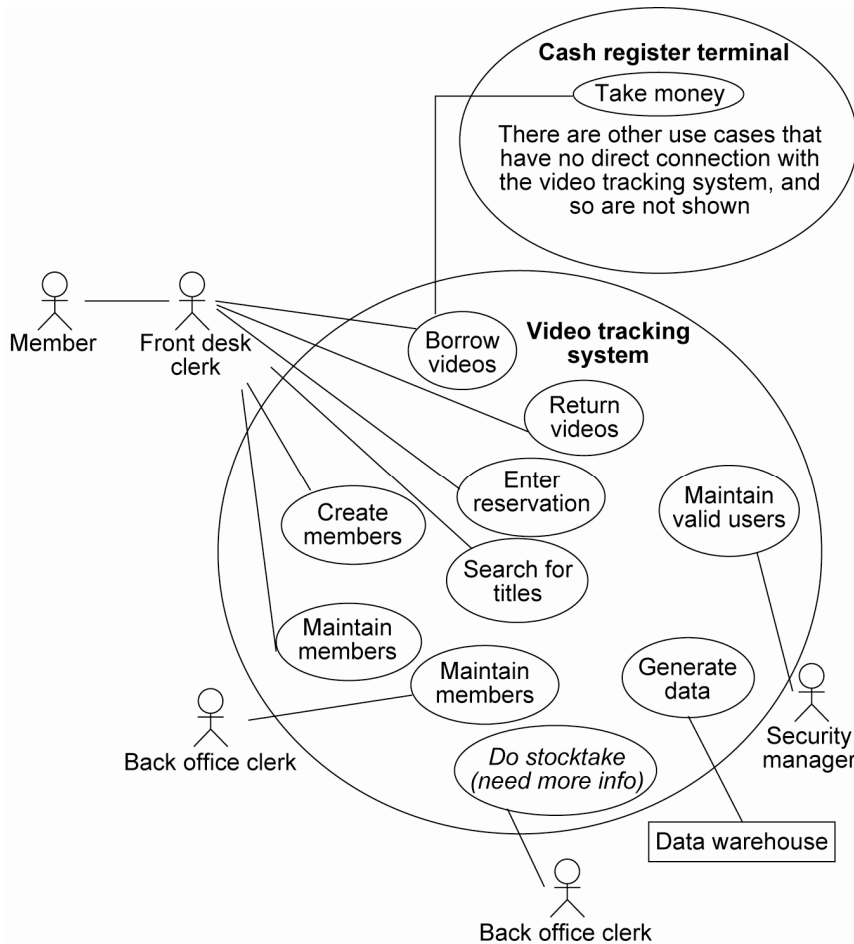
Actor Intention	System Responsibility
Identify the borrower	Confirm borrower exists
Identify the videos	Calculate the total fees due for any past fines and the new borrowings
	Pass the total amount due to the cash register terminal
Identify the videos	Record videos as returned

Activity 2.10

Here are just a few examples. They are many, many more:

- Is it true that members will not use the computer system? That is, will everything be done through the clerk?
- How are the members going to identify themselves?
- Tell me about stock takes. What are the issues? What are you trying to achieve?
- Who will support and maintain the system?
- Have you thought about transferring the data from the old system to the new system?
- Is any auditing required?

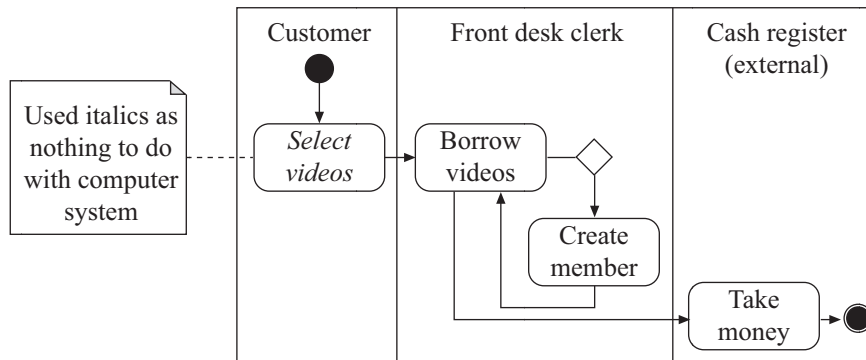
Activity 2.11



Points to note about this diagram:

- See how the member is shown on the left-hand side. While they have no direct interaction with the system, it is useful to see where they fit in.
- While the cash register terminal is an external system from the point of view of video tracking, sometimes it is useful to look inside it and see its use cases. So, in this example, we can see that there is a use case called “take money”. Please remember that this is not a dataflow diagram so we are not saying anything about the data flowing between the two systems.
- Note how this diagram quickly gets very busy with lots of lines. This is why the use cases that everyone uses, namely logon, logoff, etc., are not shown. Later in the unit we will show you a way for dealing with large systems with hundreds of use cases.

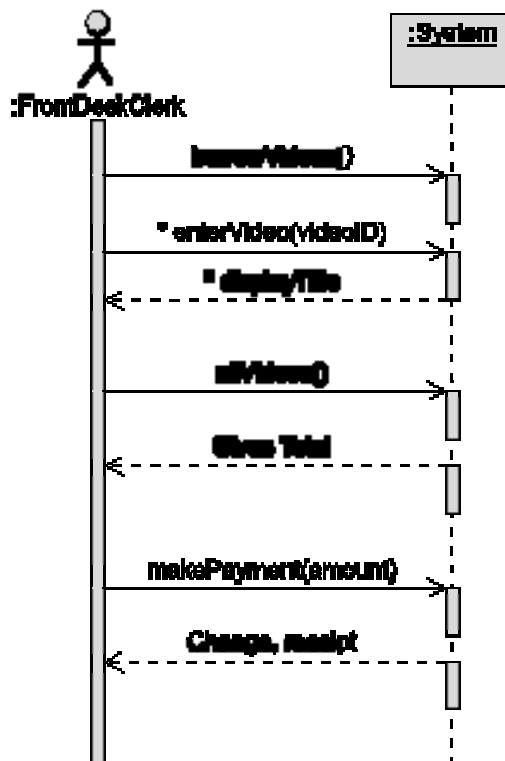
Activity 2.12



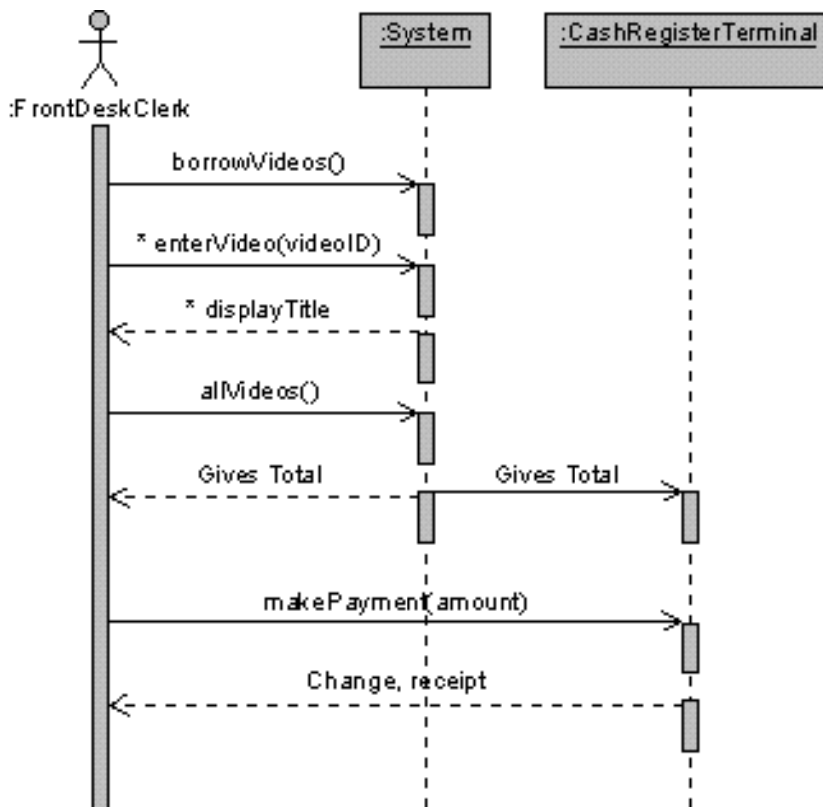
TOPIC 3: OBJECT-ORIENTED MODELLING

Activity 3.1

There are many ways you could draw this. In particular, the big question is, what is the “system”. Here are two examples.



Version 1 View the “system” as a whole



Version 2 View the “system” as just the video tracking part

In Version 1 we see the “system” as a whole. In version 2 we see the “system” as just being the video tracking part. Which is correct? Both. However it is important that you understand the difference. If there is any confusion, then you should attach a note to your diagram so that people (such as tutors and assignment markers) can understand what you are showing.

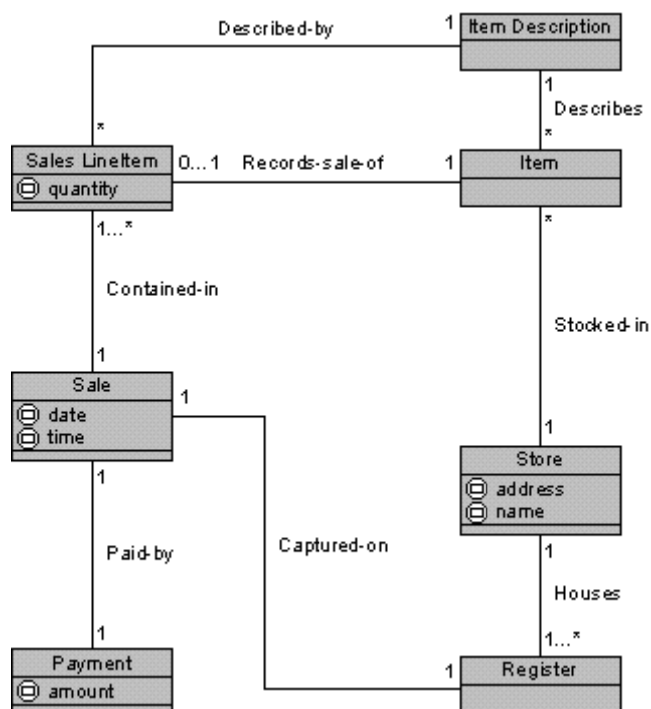
In Version 2 the important issue is how the **CashRegisterTerminal** gets told the total amount. Does it come from the Video Tracking System, or from the user? Version 2 clearly shows that the Video Tracking System tells the terminal.

The preceding diagrams were drawn using the Sequence Diagram tool in Visual Paradigm. Note that the **enterVideo** and **displayTitle** messages have a “*” prefix. The “*” represents iteration. You may have noticed in your textbook, a rectangle is placed in the diagram to enclose a group of messages to indicate that the messages are iterated together. However, the Sequence Diagram tool in Visual Paradigm does not allow the placing of rectangles in the diagram.

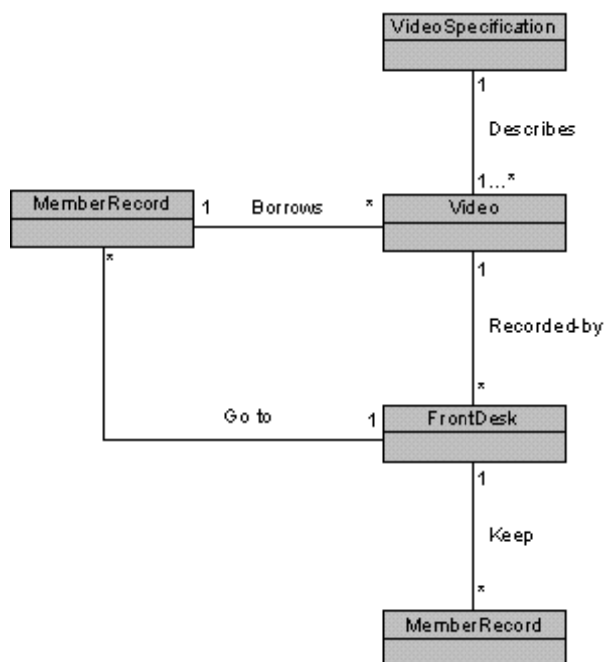
Activity 3.3

Conceptual Class Category	Example
Physical or tangible objects	<i>Video CD</i>
Specifications, designs, or descriptions of things	<i>Video title</i>
Places	<i>Video shop</i> <i>Front desk</i>
Transactions	<i>Borrow a video</i> <i>Return a video</i>
Transaction line items	<i>Video CD</i>
Roles of people	<i>Member</i> <i>Front desk clerk,</i> <i>Back room clerk,</i> <i>Accountant</i> <i>– all the roles you identified would go here</i>
Abstract noun concepts	<i>Boring</i> <i>Exciting</i>
Organizations	<i>Companies that make videotapes</i> <i>Banks</i>
Events	<i>A new video is released</i>
Processes	<i>Borrow a video</i> <i>Return a video</i> <i>Do a stock take</i> <i>– many use cases fit here</i>
Rules and policies	<i>Borrowing policy</i> <i>Joining policy</i> <i>Overdue policy</i>
Catalogue	<i>Video catalogue</i>
Manuals, documents, reference, paper, books	<i>Member records</i> <i>New releases</i> <i>New employee training</i>

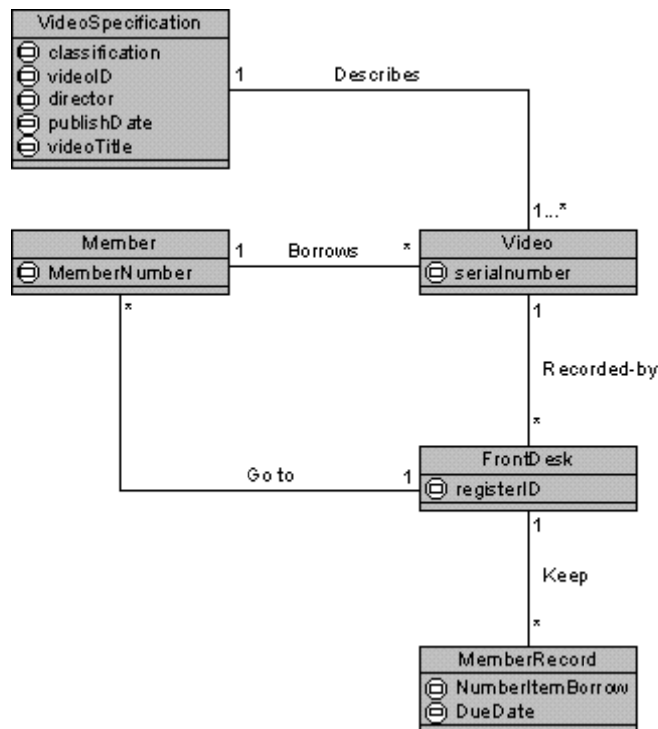
Activity 3.4



Activity 3.5



Activity 3.6



TOPIC 4: MORE USE-CASES

Activity 4.1

Perhaps you can already envisage the answer, but let us work through it slowly. We need a use case to sell old videos. So here is our first version.

Actor Intention	System Responsibility
Identify the old videos	Get the prices Pass the total amount due to the Cash Register Terminal

What do you notice about the last line? It is the same as the last line of the Borrow Videos use case.

So we could create a new sub-use case called Process Payment.

Level:	Subfunction
Actor Intention	System Responsibility
Get payment	Pass the total amount due to the Cash Register Terminal

Then, updating the other two use cases gives us the following:

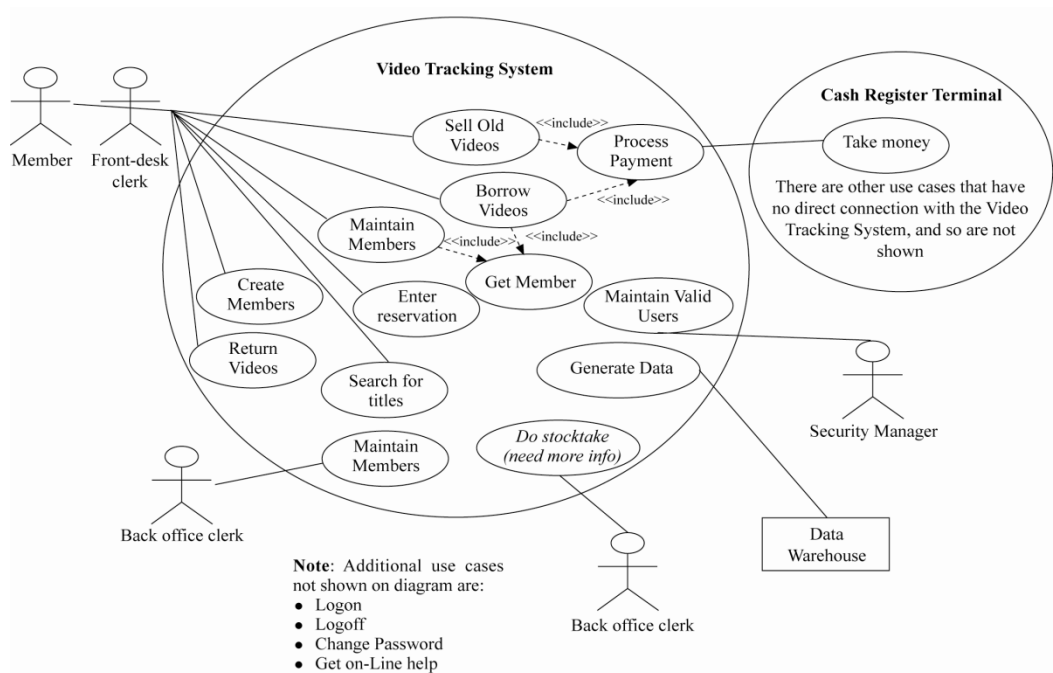
Borrow Videos

Actor Intention	System Responsibility
<u>Get Borrower</u> (note underline) Identify the videos	Display details of current borrowings Calculate the total fees due for any past fines and the new borrowings <u>Process Payment</u> (note underline)

Sell Old Videos

Actor Intention	System Responsibility
Identify the old videos	Get the prices <u>Process Payment</u> (note underline)

The use case diagram from Activity 2.11 is now as follows.



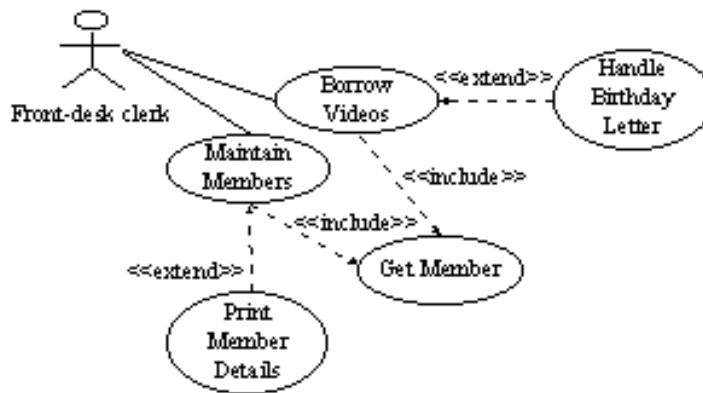
Activity 4.2

Maintain Members

Extension Point:	At any point once the member's details are displayed	
Actor Intention	System Responsibility	
<u>Get Borrower</u>	Display personal details (name, address, date of birth, date joined, etc.), plus a summary of current borrowings (for example, total number borrowed, total number of overdues, current number borrowed, current number overdues)	
Optionally, the user can modify the personal details of the member	Store the changes	
Optionally, the user can delete the member	Check that the borrower has nothing borrowed Delete the member	

Print Borrower Details

Level:	Subfunction
Trigger:	User wishes to print the details of a member
Extension Point:	Anytime in <u>Maintain Members</u>
Actor Intention	System Responsibility
Indicate that a printout is required	Print the details



Activity 4.3

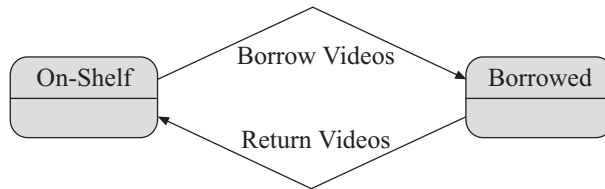
The most obvious entity that we wish to track is each Video Tape.

The other entity for which a state diagram could be useful is the class Member. The problem discussion did not mention that if a member has too many overdue, then she is not allowed to borrow, but this is the sort of fact that should be verified with individual users if a state diagram is used.

As we stated, it is not wrong to draw state diagrams for other classes, it is simply that they will be very simple and you will not learn from them. However, of course, the best way to find this out is to draw them.

Activity 4.4

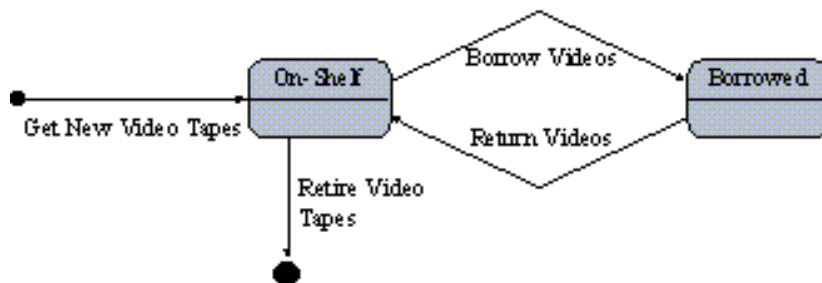
The first version for the state diagram for Video Tape might look like this:



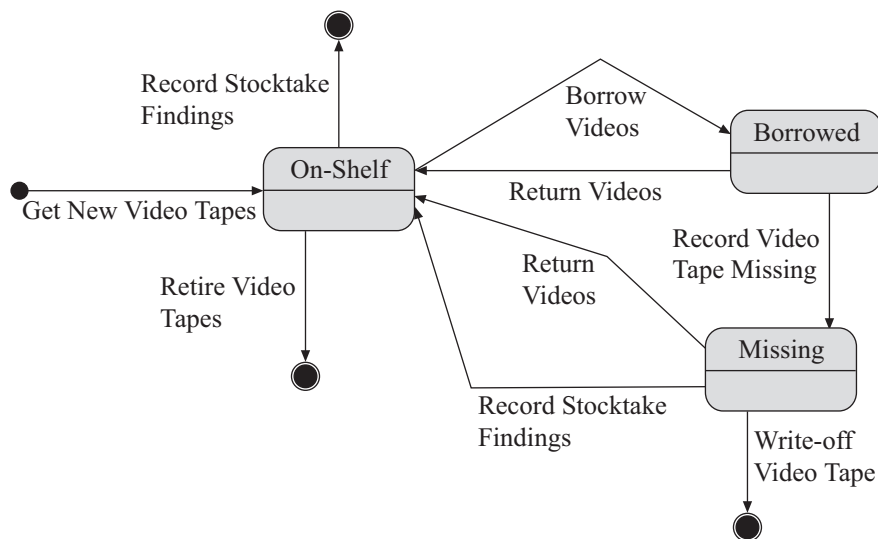
Then, you could think about how a video tape comes into existence. Clearly we are missing a use case here – let us call it Get New Video Tapes.

And what about the final state? An initial answer could be to create another new use case, this time called Retire Video Tapes. Clearly we have uncovered new functionality and would need to go back to the users to discuss this. For example, perhaps they try to sell some old video tapes.

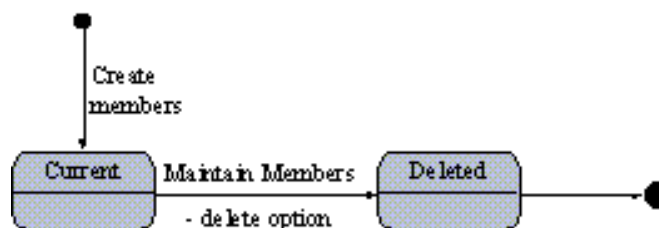
So now the diagram looks like this:



Lastly, let us think about lost or missing video tapes. Once again, we would need to discuss this with Victoria, and the answer could be the following.



Now to the state diagram for the member:



Activity 4.5

If you think about all the use cases we have discussed so far, you should realise that the only use case that might have something to do with the Video Specification class is the use case that we identified in the answer to Activity 4.4, namely Get New Video Tapes.

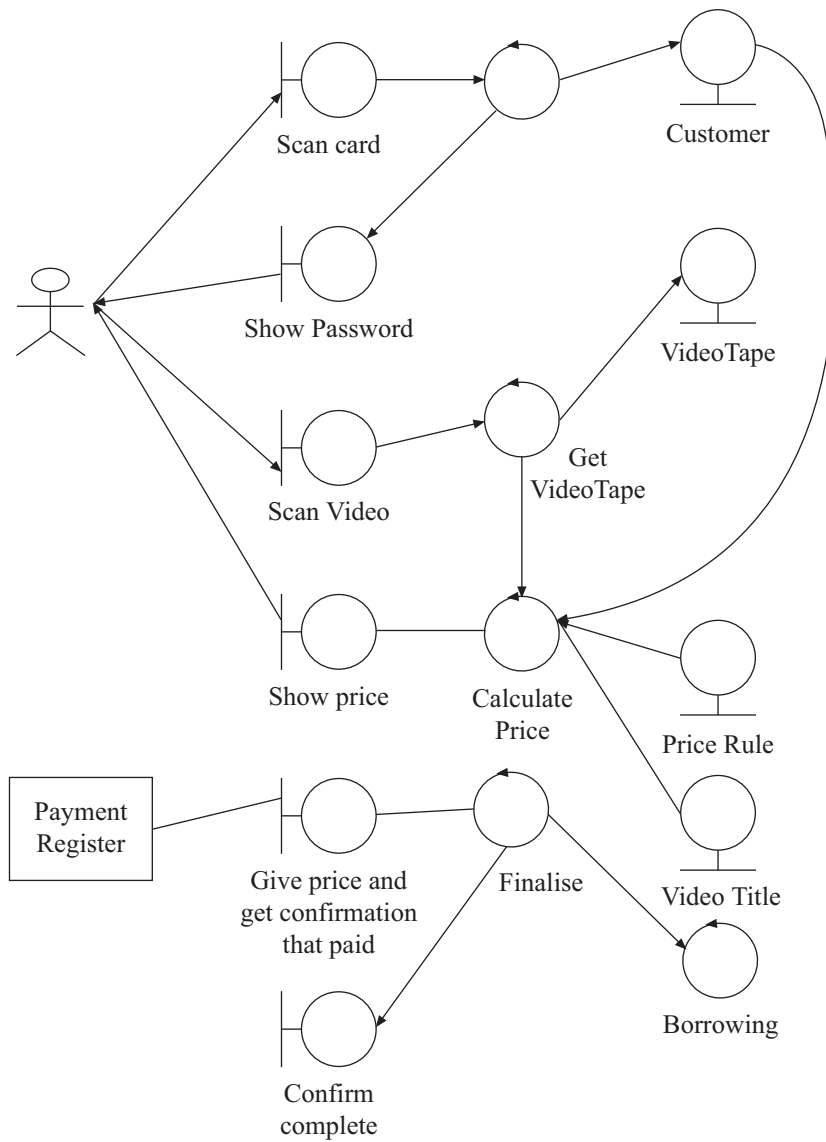
So the next step would be write that use case, and then continue with the review.

Operation on Video Specification	How Handled
Create	new use case: Create Video Title
Read	new use case: Maintain Video Title use case: Get New Video Tapes
Update	new use case: Maintain Video Title
Delete	new use case: Maintain Video Title

Activity 4.6

Business Rule	Use Cases
Only members can borrow videos.	Borrow Videos
When a new person requests to become a member, they must show ID.	Handled by the Clerk, not the system
The minimum age is 16 years old.	Handled by the Clerk, not the system
A member can borrow any number of videos, as long as they have no overdue videos.	Should be in Borrow Videos
There are fines for overdue videos.	Borrow Videos Also look at Return Videos
The length of time that a video can be borrowed for depends on the video. New releases are only lent out overnight, current releases are for three-day hire, and the rest are for a week.	The Borrow Videos use case needs to handle this
Members can reserve a video.	Reserve Videos
Every video has a classification, (general G, parental guidance PG, mature audiences MA, and restricted R). Members must be over 18 to borrow R videos. Every video also has a category: romance, general, sci-fi, foreign language and children.	The Classification and Category are attributes of the Video Specification, so these are maintained by the use case Maintain Video Title, identified in Activity 4.5.
When a member has a birthday they are sent a special letter inviting them to borrow any video of their choice for a week for free.	Generate Birthday Letters
Every three months the shop does a stock take.	Do stock take. Users have to remember to do it.
Any missing videos are updated to be shown as missing.	Record Video Tape Missing. This is the use case that we identified in Activity 4.4

Activity 4.7



TOPIC 5: DYNAMIC MODELLING

Activity 5.1

Thirsty drinker: (to the BarBot) Can I have a glass of beer?

BarBot: (to the GlassTray) Get me a (clean) Glass?

GlassTray: (gets glass from GlassTray and passes it back to the BarBot)

BarBot: (passing empty Glass to BeerTap) Fill it up.

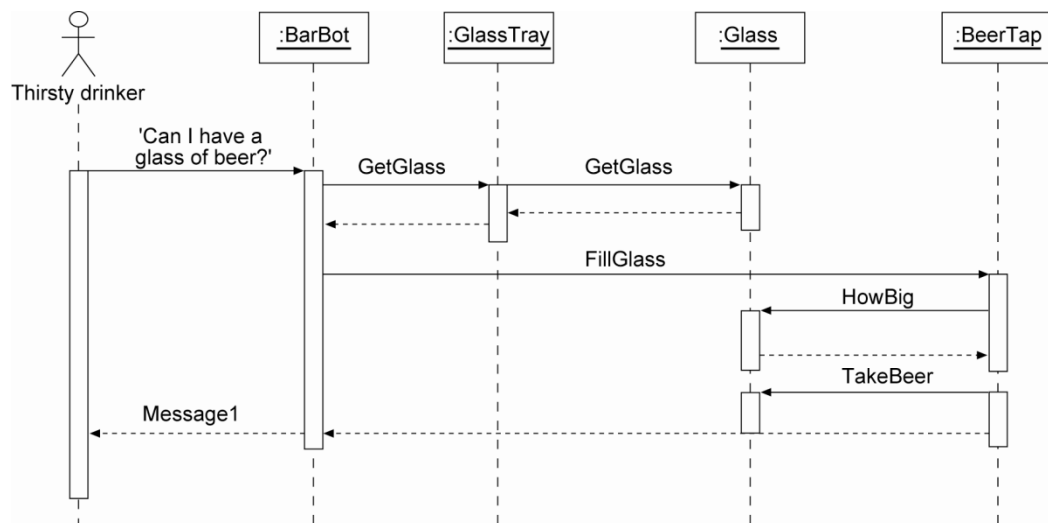
BeerTap: (to the Glass) How big are you?

Glass: (to the BarBot) 285 ml.

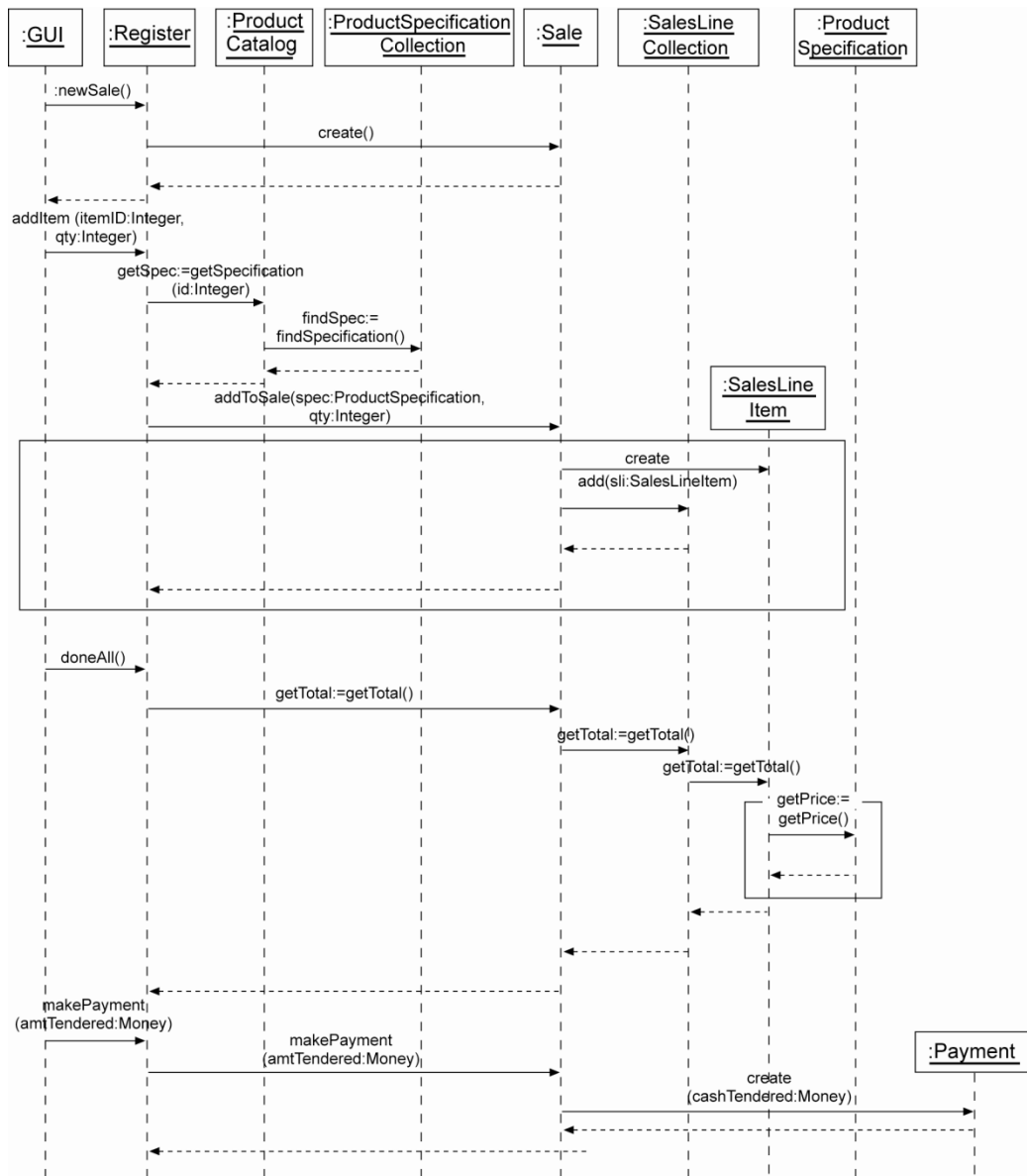
BeerTap: (dispenses 285ml of beer into glass, and returns the filled Glass to the BarBot)

BarBot: (passes full Glass to the drinker)

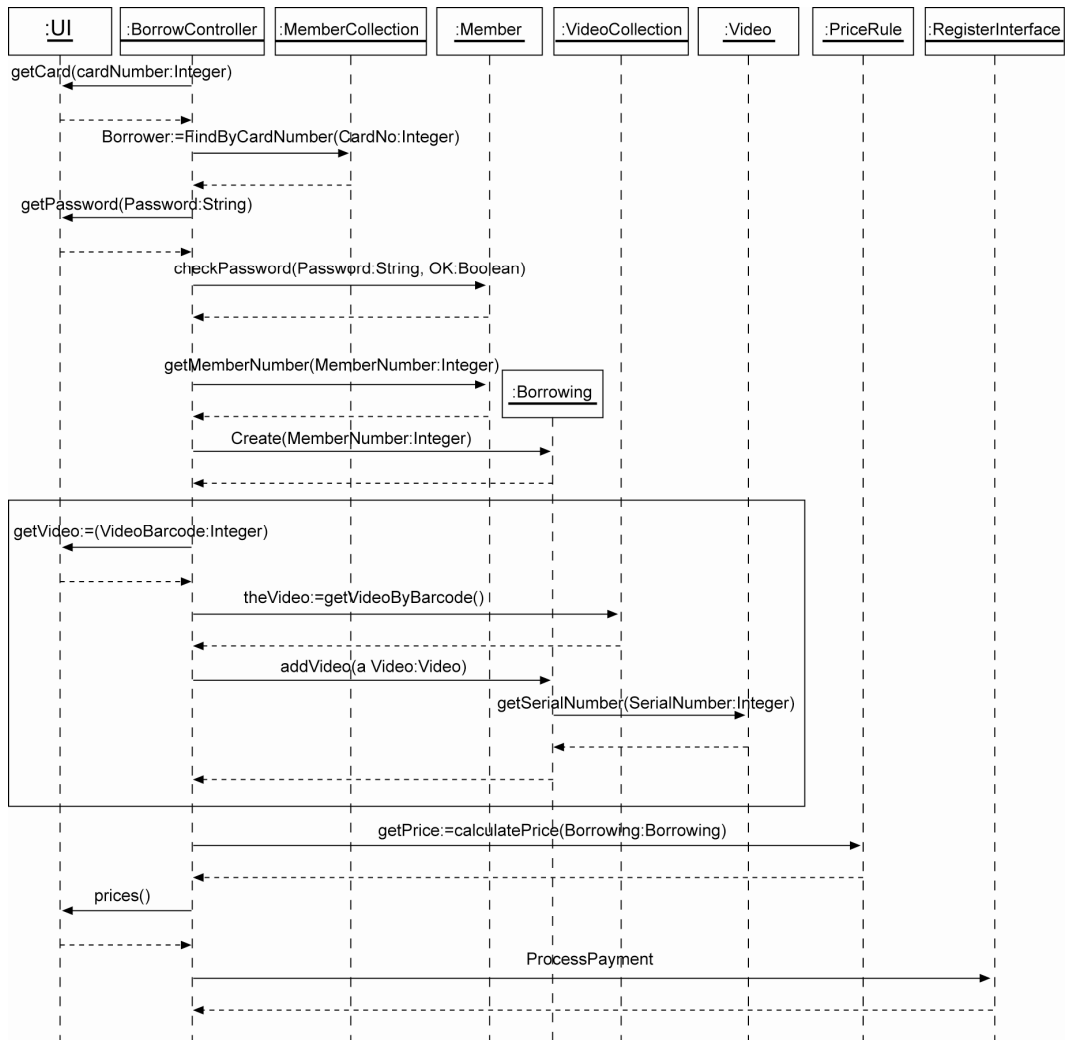
Activity 5.2



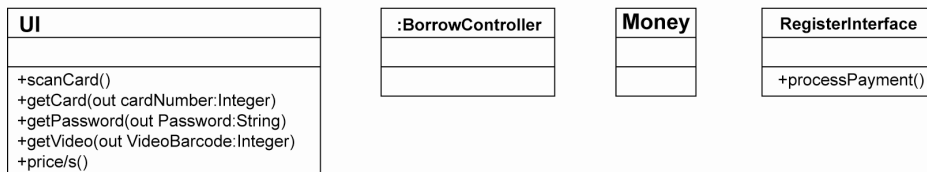
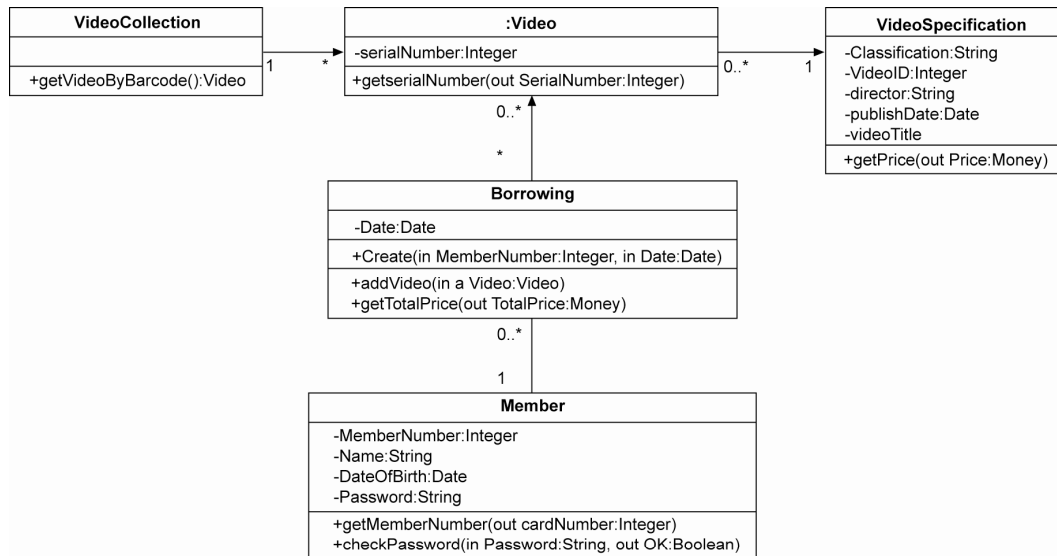
Activity 5.3



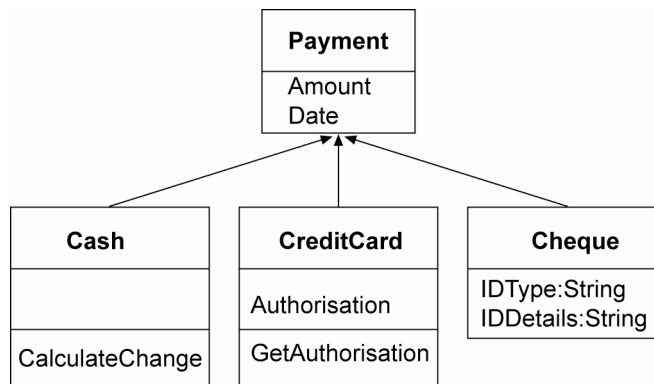
Activity 5.4



Activity 5.5



Activity 5.6



Activity 5.7

This is a bit of a trick question, as thus far we have no information or different processing requirements for different categories. So, while conceptually it may be quite important for the owner of the business, from the processing requirements side nothing extra is required. Probably the second design class diagram is initially a better version, but once design is complete, and no further requirements are found, then use the first design class diagram.

Activity 5.8

1. The second diagram; having the functionality for Chess and Checkers in the same class (BoardGame) is not functionally cohesive.
2. The second diagram; The first diagram has the Player class creating a CheckersMove class for the CheckersBoard class. The second diagram lets the CheckersBoard class deal with the CheckersMove class itself.
3. Figure B has less coupling because every class has at most one association with another class.

MODULE FEEDBACK

<i>MAKLUM BALAS MODUL</i>

If you have any comment or feedback, you are welcome to:

1. E-mail your comment or feedback to modulefeedback@oum.edu.my

OR

2. Fill in the Print Module online evaluation form available on myVLE.

Thank you.

Centre for Instructional Design and Technology
(*Pusat Reka Bentuk Pengajaran dan Teknologi*)

Tel No.: 03-27732578

Fax No.: 03-26978702



www.oum.edu.my