

# Object Oriented Programming using C#

Simon Kendal



SIMON KENDAL

---

# OBJECT ORIENTED PROGRAMMING USING C#

Object Oriented Programming using C#

2<sup>nd</sup> edition

© 2018 Simon Kendal & [bookboon.com](http://bookboon.com)

ISBN 978-87-403-2140-1

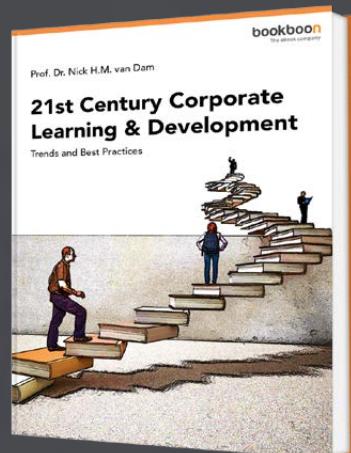
# CONTENTS

<b>Foreword</b>	<b>10</b>
<b>1 An Introduction to Object Orientated Programming</b>	<b>11</b>
1.1 A Brief History of Computing	12
1.2 Different Programming Paradigms	14
1.3 Why use the Object Orientation Paradigm?	16
1.4 Object Oriented Principles	17
1.5 What exactly is Object Oriented Programming?	22
1.6 The Benefits of the Object Oriented Programming Approach	25
1.7 Software Implementation	26
1.8 An Introduction to the .NET Framework	31
1.9 Summary	34
<b>2 The Unified Modelling Language (UML)</b>	<b>35</b>
2.1 An Introduction to UML	36
2.2 UML Class diagrams	37
2.3 UML Syntax	41

## Free eBook on Learning & Development

By the Chief Learning Officer of McKinsey

Download Now



2.4	UML Package Diagrams	55
2.5	UML Object Diagrams	62
2.6	UML Sequence Diagrams	65
2.7	Summary	67
<b>3</b>	<b>Inheritance and Method Overriding</b>	<b>68</b>
3.1	Object Families	69
3.2	Generalisation and Specialisation	70
3.3	Inheritance	71
3.4	Implementing Inheritance in C#	79
3.5	Constructors	80
3.6	Constructor Rules	81
3.7	Access Control	82
3.8	Abstract Classes	86
3.9	Overriding Methods	87
3.10	The 'Object' Class	90
3.11	Overriding ToString() defined in 'Object'	91
3.12	Summary	94
<b>4</b>	<b>Object Roles and the Importance of Polymorphism</b>	<b>95</b>
4.1	Class Types	96
4.2	Substitutability	99
4.3	Polymorphism	101
4.4	Extensibility	102
4.5	Interfaces	110
4.6	Extensibility Again	117
4.7	Distinguishing Subclasses	121
4.8	Summary	123
<b>5</b>	<b>Using Polymorphism Effectively</b>	<b>124</b>
5.1	Responsibility	125
5.2	What characterises a Polymorphic system?	131
5.3	Polymorphic Collections	135
5.4	Non-Polymorphic Components	136
5.5	A More Realistic (Complex) Polymorphic System (Part 1)	138
5.6	A More Realistic (Complex) Polymorphic System (Part 2)	141
5.7	Extending a Polymorphic System Using an Interface	149
5.8	One Final Note	157
5.9	Summary	158

<b>6</b>	<b>Overloading Methods</b>	<b>159</b>
6.1	Overloading Method Names	159
6.2	Overloading To Aid Flexibility	161
6.3	Summary	164
<b>7</b>	<b>Object Oriented Software Analysis and Design</b>	<b>165</b>
7.1	Requirements Analysis	166
7.2	The Problem	168
7.3	Listing Nouns and Verbs	169
7.4	Identifying Things Outside The Scope of The System	170
7.5	Identifying Synonyms	171
7.6	Identifying Potential Classes	173
7.7	Identifying Potential Attributes	173
7.8	Identifying Potential Methods	174
7.9	Identifying Common Characteristics	175
7.10	Refining Our Design using CRC Cards	176
7.11	Elaborating Classes	178
7.12	Summary	180
<b>8</b>	<b>Introducing the SOLID Design Principles</b>	<b>181</b>
8.1	The SOLID design principles	182
8.2	The Single Responsibility Principle	183
8.3	The Open/Closed Principle	190
8.4	The Liskov Substitution Principle	191
8.5	The Interface Segregation Principle	197
8.6	The Dependency Inversion Principle	203
8.7	Summary	217
<b>9</b>	<b>Generic Collections and Serialization</b>	<b>219</b>
9.1	An Introduction to Generic Methods	220
9.2	An Introduction to Collections	226
9.3	Different Types of Collections	227
9.4	Lists	229
9.5	HashSets	229
9.6	Dictionaries	231
9.7	A Simple List Example	232
9.8	A More Realistic Example Using Lists	236
9.9	Queues and Stacks	242
9.10	An Example Using Sets	244
9.11	Overriding Equals() and GetHashCode()	245
9.12	An Example Using Dictionaries	256

9.13	Serializing and De-serializing Collections	263
9.14	The Power of Serialization	269
9.15	Summary	276
<b>10</b>	<b>C# Development Tools</b>	<b>278</b>
10.1	Tools for Writing C# Programs	278
10.2	Microsoft Visual Studio	279
10.3	SharpDevelop	280
10.4	Automatic Documentation	281
10.5	Sandcastle Help File Builder	285
10.6	GhostDoc	286
10.7	Adding Namespace Comments	286
10.8	Summary	288
<b>11</b>	<b>Creating and Using Customised Exceptions</b>	<b>290</b>
11.1	Understanding the Importance of Exceptions	290
11.2	Kinds of Exception	293
11.3	Extending the ApplicationException Class	294
11.4	Throwing Exceptions	296
11.5	Catching Exceptions	297
11.6	Exception Hierarchies	298
11.7	A 'Finally' Block	300
11.8	Summary	301
<b>12</b>	<b>Agile Programming</b>	<b>302</b>
12.1	In the Beginning...	303
12.2	Old Software Lifecycles	304
12.3	The Changing World and the Need for Agile Methods	310
12.4	Agile Approaches	313
12.5	Scrum	314
12.6	Extreme Programming (XP)	316
12.7	Applying Object Orientated Design in Support of Agile Methods	317
12.8	Refactoring	318
12.9	Examples of Refactoring	319
12.10	Support for Refactoring	319
12.11	Unit Testing	320
12.12	Automated Unit Testing	321
12.13	Regression Testing	321
12.14	Unit Testing in Visual Studio	322
12.15	Examples of Assertions	324
12.16	Several Test Examples	325

12.17	Running Tests	332
12.18	Test Driven Development (TDD)	333
12.19	TDD Cycles	333
12.20	Claims for TDD	334
12.21	Further Help with Unit Testing	334
12.22	Summary	335
<b>13</b>	<b>Checking Your Understanding</b>	<b>336</b>
13.1	Questions Regarding UML Notation	337
15.2	Questions Regarding Object Oriented Design	339
13.3	Questions Regarding Implementation	344
13.4	Summary	348
<b>14</b>	<b>Case Study</b>	<b>349</b>
14.1	The Problem	350
14.2	Preliminary Analysis	351
14.3	Further Analysis	356
14.4	Documenting the Design using UML	361
14.5	Prototyping the Interface	366
14.6	Revising the Design to Accommodate Changing Requirements	367
14.7	Packaging the Classes	371
14.8	Programming the Message Classes	372
14.9	Programming the Client Classes	379
14.10	Creating and Handling UnknownClientException	381
14.11	Programming the Interface	383
14.12	Using Test Driven Development and Extending the System	386
14.13	Generating the Documentation	393
14.14	The Finished System	396
14.15	Running the System	397
14.16	Conclusions	399
<b>15</b>	<b>Finally...</b>	<b>400</b>
	<b>Peer review comments</b>	<b>402</b>

*'It is how we deal with adversity and how we treat others that defines who we are.'*

*To Janice and Cara: where I am weak you are strong, where I am rough you are gentle and kind!*

*Simon Kendal*

*P.s. Thank you Cara for volunteering to proof read this book. Your hard work and attention to detail are much appreciated. Any remaining errors are my own.*

# FOREWORD

This book aims help you understand the Object Oriented approach to programming and help you develop some practical skills along the way. To this end each chapter will incorporate small exercises with solutions and feedback provided.

The concepts that will be explained and skills developed are in common use among programmers using many modern Object Oriented languages and are thus transferrable from one language to another. However for practical purposes these concepts are explored and demonstrated using the C# (pronounced C sharp) programming language.

This, the second edition of this book, has been written to demonstrate:

- How to apply Polymorphism and the SOLID design principles to create effective and robust Object Oriented systems.
- Why agile software engineering methods (e.g. Scrum, XP and TDD) are now preferred over older methods.
- How Object Orientated design supports the use of modern agile software development methods.

At the end of the book one larger case study will be described – this will be used to illustrate the application of the techniques explored in the earlier chapters. This case study will culminate in the development of a complete C# program that can be downloaded with this book.

While the C# programming language is used to highlight and demonstrate the application of fundamental Object Oriented principles and modelling techniques this book is not an introduction to C# programming. The reader will be expected to have an understanding of basic programming concepts and their implementation in C# (inc. the use of loops, selection statements, performing calculations, arrays, data types and a basic understanding of file handling).

# 1 AN INTRODUCTION TO OBJECT ORIENTATED PROGRAMMING

## Introduction

This chapter will discuss different programming paradigms and the advantages of the Object Oriented approach to software development and modelling. The concepts on which Object Orientation depend (abstraction, encapsulation, inheritance and polymorphism) will be explained.

## Objectives

By the end of this chapter you will be able to...

- Explain what Object Oriented Programming is.
- Describe the benefits of the Object Oriented programming approach.
- Understand the basic concepts on which Object Oriented programming relies (abstraction, encapsulation, inheritance and polymorphism).
- Understand the reasons behind the development of the .NET framework and the role of the Common Language Runtime (CLR) engine.

All of these issues will be explored in much more detail in later chapters of this book.

This chapter consists of nine sections:

- 1) A Brief History of Computing
- 2) Different Programming Paradigms
- 3) Why use the Object Oriented Paradigm?
- 4) Object Oriented Principles
- 5) What exactly is Object Oriented Programming?
- 6) The Benefits of the Object Oriented Programming Approach
- 7) Software Implementation
- 8) An Introduction to the .NET Framework
- 9) Summary

## 1.1 A BRIEF HISTORY OF COMPUTING

Computing is constantly changing our world and our environment. In the 1960s large machines called mainframes were created to manage large volumes of data (numbers) efficiently. Bank account and payroll programs changed the way organisations worked and made parts of these organisations much more efficient. In the 1980s personal computers became common and changed the way many individuals worked. People started to own their own computers and many used word processors and spreadsheets applications (to write letters and to manage home accounts). In the 1990s email became common and the world wide web was born. These technologies revolutionised communications allowing individuals to publish information that could easily be accessed on a global scale. The ramifications of these new technologies are still not fully understood as society is adapting to opportunities of internet commerce, new social networking technologies (twitter, facebook, online gaming etc.), internet connected devices (the Internet of Things) and the challenges of internet related crime.

Just as new computing technologies are changing our world so too are new techniques and ideas changing the way we develop computer systems. In the 1950s the use of machine code (unsophisticated, complex and machine specific languages) was common.

A red recruitment advertisement for Tieto. On the left, a woman with long dark hair, wearing a white shirt and teal earrings, looks upwards with a smile. A thought bubble above her contains a white line-art icon of a crown. To the right of the woman, the text reads: "Do you want to make a difference? Join the IT company that works hard to make life easier. [www.tieto.fi/careers](http://www.tieto.fi/careers)". Below this, the Tieto slogan "Knowledge. Passion. Results." is written. The Tieto logo, consisting of the word "tieto" in a red, slanted font, is partially visible in the bottom right corner.

In the 1960s high level languages, which made programming simpler, became common. However these led to the development of large complex programs that were difficult to manage and maintain.

In the 1970s the structured programming paradigm became the accepted standard for large complex computer programs. The structured programming paradigm proposed methods to logically structure the programs developed into separate smaller, more manageable components. Furthermore methods for analysing data were proposed that allowed large databases to be created that were efficient, preventing needless duplication of data and protected us against the risks associated with data becoming out of sync. However significant problems still persisted in: a) understanding the systems we need to create and b) changing existing software as user's requirements changed.

In the 1980s 'modular' languages, such as Modula-2 and ADA were developed that became the precursor to modern Object Oriented languages.

In the 1990s the Object Oriented paradigm and component-based software development ideas were established and Object Oriented languages became the norm from 2000 onwards.

The Object Oriented paradigm is based on many of the ideas developed over the previous 30 years of abstraction, encapsulation, generalisation and polymorphism. This has led to the development of software components where the operation of the software and the data it operates on are modelled together. Proponents of the Object Oriented software development paradigm argue that this leads to the development of software components that can be re-used in different applications thus saving significant development time and reducing costs but more importantly allow better software models to be produced that make systems more maintainable and easier to understand.

From 2000 onwards new Agile methods of working were being proposed and tested, these methods rely on the development flexible and maintainable software i.e. on the advantages provided by Object Orientation, and from 2010 onwards agile methods started to be widely adopted by companies with software development teams.

Software development ideas are still evolving. Where these will lead us in the future remains to be seen.

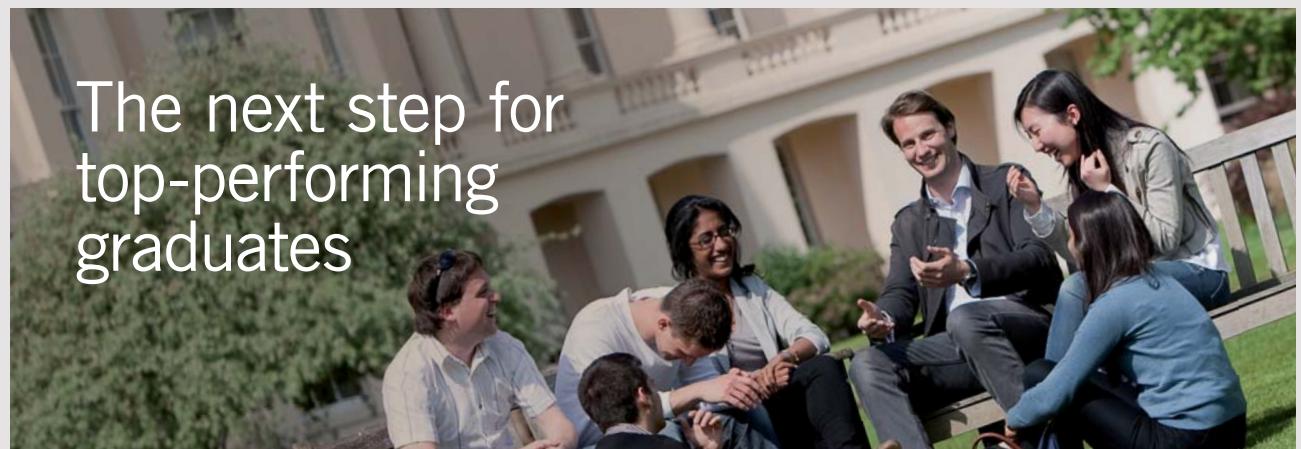
## 1.2 DIFFERENT PROGRAMMING PARADIGMS

The structured programming paradigm proposed that programs could be developed in sensible blocks that make the program more understandable and easier to maintain.

### Activity 1

Assume you undertake the following activities on a daily basis. Arrange this list into a sensible order then split this list into three blocks of related activities and give each block a heading to summarise the activities carried out in that block.

- Get out of bed
- Eat breakfast
- Park the car
- Get dressed
- Get the car out of the garage
- Drive to work
- Find out what your boss wants you to do today
- Feedback to the boss on today's results
- Do what the boss wants you to do



The next step for  
top-performing  
graduates

### Masters in Management

Designed for high-achieving graduates across all disciplines, London Business School's Masters in Management provides specific and tangible foundations for a successful career in business.

This 12-month, full-time programme is a business qualification with impact. In 2010, our MiM employment rate was 95% within 3 months of graduation\*; the majority of graduates choosing to work in consulting or financial services.

As well as a renowned qualification from a world-class business school, you also gain access to the School's network of more than 34,000 global alumni – a community that offers support and opportunities throughout your career.

For more information visit [www.london.edu/mm](http://www.london.edu/mm), email [mim@london.edu](mailto:mim@london.edu) or give us a call on [+44 \(0\)20 7000 7573](tel:+44(0)2070007573).



\* Figures taken from London Business School's Masters in Management 2010 employment report

### Feedback 1

You should have been able to organise these into groups of related activities and give each group a title that summarises those activities.

Get up:

- Get out of bed
- Get dressed
- Eat breakfast

Go to Work:

- Get the car out of the garage
- Drive to work
- Park the car

Do your job:

- Find out what your boss wants you to do today
- Do what the boss wants you to do
- Feedback to the boss on today's results

By structuring our list of instructions and considering the overall structure of the day (Get up, go to work, do your job) we can change and improve one section of the instructions without changing the other parts. For example we could improve the instructions for going to work...

- Listen to the local traffic and weather report
- Decide whether to go by bus or by car
- If going by car, get the car and drive to work.
- Else walk to the bus station and catch the bus

without worrying about any potential impact this may have on 'getting up' or 'doing your job'. In the same way structuring computer programs can make each part more understandable and make large programs easier to maintain.

The Object Oriented paradigms suggest we should model instructions in a computer program with the data they manipulate and store these as components together. One advantage of doing this is we get reusable software components.

### Activity 2

Imagine a personal address book with some data stored about your friends

Name,  
Address,  
Telephone Number.

List three things that you may do to this address book.

Next identify someone else who may use an identical address book for some purpose other than storing a list of friends.

### Feedback 2

With an address book we would want to be able to perform the following actions: find out details of a friend i.e. their telephone number, add an address to the address book and, of course, delete an address.

We can create a simple software component to store the data in the address book (i.e. list of names etc.) and the operations, things we can do with the address book (i.e. add address, find telephone number etc.).

By creating a simple software component to store and manage addresses of friends we can reuse this in another software system i.e. it could be used by a business manager to store and find details of customers. It could also become part of a library system to be used by a librarian to store and retrieve details of the users of the library.

Thus in Object Oriented programming we can create re-usable software components (in this case an address book).

The Object Oriented paradigm builds upon and extends the ideas behind the structured programming paradigm of the 1970s.

## 1.3 WHY USE THE OBJECT ORIENTATION PARADIGM?

While we can focus our attention on the actual program code we are writing, whatever development methodology is adopted, it is not the creation of the code that is generally the source of most problems. Most problems arise from:

- Poor analysis and design: the computer system we created doesn't do the right thing.
- Poor maintainability: the system is hard to understand and revise when, as is inevitable, requests for change arise.

Statistics show 70% of the cost of software is not incurred during its initial development phase but is incurred during subsequent years as the software is amended to meet the ever changing needs of the organisation for which it was developed. For this reason it is essential that software engineers do everything possible to ensure that software is easy to maintain during the years after its initial creation.

The Object Oriented programming paradigm aims to help overcome these problems by helping with the analysis and design tasks during the initial software development phase (see Chapter 7 for more details on this) and by ensuring software is robust and maintainable (see Chapter 8 and Chapters 11–14 for information on the support Object Orientation and C# provides for creating systems that are robust and maintainable).

## 1.4 OBJECT ORIENTED PRINCIPLES

Abstraction and encapsulation are fundamental principles that underlie the Object Oriented approach to software development. Abstraction allows us to consider complex ideas while ignoring irrelevant detail that would confuse us. Encapsulation allows us to focus on what something does without considering the complexities of how it works.

### Activity 3

Consider your home and imagine you were going to swap your home for a week with a new friend.

Write down three essential things you would tell them about your home and that you would want to know about their home.

Now list three irrelevant details that you would not tell your friend.

### Feedback 3

You presumably would tell them the address, give them a basic list of rooms and facilities (e.g. number of bedrooms) and tell them how to get in (i.e. which key would operate the front door and how to switch off the burglar alarm (if you have one).

You would not tell them irrelevant details (such as the colour of the walls, seats etc.) as this would overload them with useless information.

Abstraction allows us to consider the important high level details of your home, e.g. the address, without becoming bogged down in detail.

#### Activity 4

Consider your home and write down one item, such as a television, that you use on a daily basis (and briefly describe how you operate this item).

Now consider how difficult it would be to describe the internal components of this item and give full technical details of how it works.

#### Feedback 4

Describing how to operate a television is much easier than describing its internal components and explaining in detail exactly how it works. Most people do not even know all the components of the appliances they use or how they work – but this does not stop them from using these appliances every day.

You may not know the technical details, such as how the light switches are wired together and how they work internally, but you can still switch the lights on and off in your home (and in any new building you enter).

Encapsulation allows us to consider what a light switch does, and how we operate it, without needing to worry about the technical detail of how it actually works.

*Tuleva DI tai tietojenkäsittelytieteilijä,  
edunvalvojasi työelämässä on TEK.*



TEKin jäsenenä saat myös tietoa, turvaa,  
neuvontaa ja lukuisia rahanarvoisia etuja.

Opiskelijalle jäsenyys on maksuton.  
Lue lisää [www.tek.fi/opiskelijat](http://www.tek.fi/opiskelijat)

Jos sinulla on yliopistotason tutkinto  
ja olet jo työelämässä,  
lue lisää [www.tek.fi/jasenyyss](http://www.tek.fi/jasenyyss)

**Liity nyt!**

[www.tek.fi/liity](http://www.tek.fi/liity)

**TEK**  
TEKNIIKAN AKATEEMISET

Two other fundamental principles of Object Orientation are Generalization/Specialization (which allows us to make use of inheritance) and polymorphism.

Generalisation allows us to consider general categories of objects which have common properties and then define specialised sub-classes that inherit the properties of the general categories.

### **Activity 5**

Consider the people who work in a hospital and list three common occupations of people you would expect to be employed there.

Now for each of these common occupations list two or three specific categories of staff.

### **Feedback 5**

Depending upon your knowledge of the medical profession you may have listed three very general occupations (e.g. doctor, nurse, cleaner) or you may have listed more specific occupations such as radiologist, surgeon etc.

Whatever your initial list you probably would have been able to specify more specialised categories of these occupations e.g.

Doctor:

- Trainee doctor,
- Junior doctor,
- Surgeon,
- Radiologist,
- etc.

Nurse:

- Triage nurse,
- Midwife,
- Ward sister

Cleaner:

- General cleaner
- Cleaning supervisor

Now we have specified some general categories and some more specialised categories of staff we can consider the general things that are true for all doctors, all nurses etc.

### Activity 6

Make one statement about doctors that you would consider to be true for all doctors and make one statement about surgeons that would **not** be true for all doctors.

### Feedback 6

You could make a statement that all doctors have a knowledge of drugs, can diagnose medical conditions and can prescribe appropriate medication.

For surgeons you could say that they know how to use scalpels and other specialised pieces of equipment and they can perform operations.

According to our list above all surgeons are doctors and therefore still have a knowledge of medical conditions and can prescribe appropriate medication. However not all doctors are surgeons and therefore not all doctors can perform operations.

Whatever we specify as true for doctors is also true for trainee doctors, junior doctors etc. – these specialised categories (or classes) can inherit the attributes and behaviours associated with the more general class of 'doctor'.

Generalisation/Specialisation allow us to define general characteristics and operations of an object and allow us to create more specialised versions of this object. The specialised versions of this object will automatically inherit all of the characteristics of the more generalised object.

The final principle underlying Object Orientation is Polymorphism which is the ability to interact with an object as its generalized category regardless of its more specialised category.

### Activity 7

Make one statement about how a hospital manager may interact with all doctors employed at their hospital irrespective of what type of doctor they are.

### Feedback 7

You may have considered that a hospital manager could pay all doctors (presumably this will be done automatically at the end of every month) and could discipline any doctor guilty of misconduct – of course this would be true for other staff as well. More specifically a manager could check that a doctor's medical registration is still current. This would be something that management would need to do for all doctors irrespective of what their specialism is.

Furthermore if the hospital employed a new specialist doctor (e.g. a Neurologist), without knowing anything specific about this specialism, hospital management would still know that: a) these staff needed to be paid and b) their medical registration must be checked i.e. they are still doctors and need to be treated as such.

Using the same idea polymorphism allows computer systems to be extended, with new specialised objects being created, while allowing current parts of the system to interact with new objects without concern for the specific properties of the new objects. This makes it easy to extend system as we will see in a later chapter.

**#2020Resolutions**

To create a digital learning culture

**CHECK**

**bookboonglobal**

Unlock your company's full potential with Bookboon Learning. We have the highest staff usage rates in the learning industry. Find out why ►►►

## 1.5 WHAT EXACTLY IS OBJECT ORIENTED PROGRAMMING?

### Activity 8

Think of an object you possess. Describe its current state and list two or three things you can do with that object.

### Feedback 8

You probably thought about an entirely physical object such as a watch, a pen, or a car.

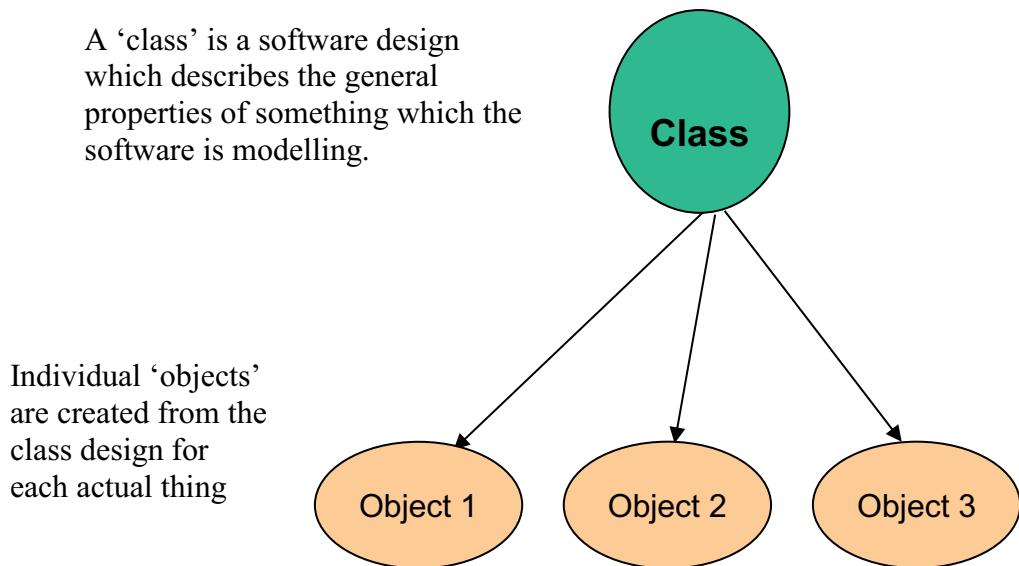
Objects have a current state. A watch has a time (represented internally by wheels and cogs or in an electronic component). A pen has a certain amount of ink in it and has its lid on or off. A car has a current speed and has a certain amount of fuel inside it.

Specific behaviour can also be associated with each object (things that you can do with it): a watch can be checked to find out the time, the time can also be set. A pen can be used to write with and a car can be started, driven and stopped.

You can also think of other non-physical things as objects: – such as a bank account. A bank account is not something that can be physically touched but intellectually we can consider a bank account to be an object. It also has a current state (the amount of money in it) and it also has behaviour associated with it (most obviously depositing and withdrawing money).

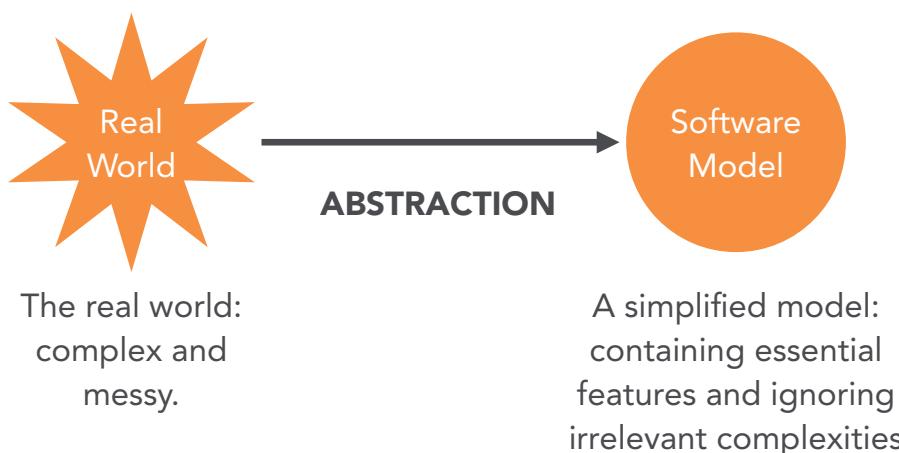
Object oriented programming is a method of programming that involves the creation of intellectual objects that model a business problem we are trying to solve (e.g. a bank account, a bank customer and a bank manager – could all be objects in a computerised banking system). With each object we model the data associated with it (i.e. its status at any particular point in time) and the behaviour associated with it (what our computer program should allow that object to do).

In creating an Object Oriented program we define the properties of a class of objects (e.g. all bank accounts) and then create individual objects from this class (e.g. your bank account).



However deciding just what classes we should create in our system is not a trivial task as the real world is complex and messy. In essence we need to create an abstract model of the real world that focuses on the essential aspects of a problem and ignores irrelevant complexities (see Chapter 7 for more advice on how to go about this). For example in the real world bank account holders sometimes need to borrow money and occasionally their money may get stolen. If we were to create a bank account system should we allow customers to borrow money? Should we acknowledge that their cash may get stolen and build in some method of them getting an immediate loan – or is this an irrelevant detail that would just add complexity to the system and provide no real benefit to the bank?

Using Object Oriented analysis and design techniques our job would be to look at the real world and come up with a simplified abstract model that we could turn into a computer system. How good our final system is will depend upon how good our software model is.



### Activity 9

Consider a computer system that will allow items to be reserved from a library. Imagine one such item that you may like to reserve and list two or three things that a librarian may want to know about this item.

### Feedback 9

You may have thought of a book you wish to reserve in which case the librarian may need to know the title of the book and its author.

For every object we create in a system we need to define the attributes of that object i.e. the things we need to know about it.

The image shows a collection of circular campaign buttons and a red lamp. The buttons include:

- A green button with the text "Reduce reuse Recycle" (partially visible).
- A yellow button with the text "WORK WITH US".
- A white button featuring a red fish.
- A pink button with the text "togetherness".
- A blue button with the text "Save water. Shower together".
- A blue button with the text "everyone deserves good design".
- A button with the IKEA logo.
- A button with flags of various countries.

In the center, a white speech bubble contains the text:

**It's only an opportunity if you act on it**

Below the speech bubble is the text:

**IKEA.SE/STUDENT**

On the right side of the image, there is a small vertical text:

© Inter IKEA Systems B.V. 2009

### Activity 10

Note: we can consider a reservation as an intellectual object (where the actual item is a physical object).

Considering this intellectual object (item reservation) list two or three actions the librarian may need to perform on this object.

### Feedback 10

The librarian may need to cancel this reservation (if you change your mind) they may also need to tell you if the item has arrived in stock for you to collect.

For each object we need to define the operations that will be performed on that object (as well as its attributes).

### Activity 11

Considering the most general category of object that can be borrowed from a library, a 'loan item', list two or three more specific subcategory of object a library can lend out.

### Feedback 11

Having defined the most general category of object (we call this a class) – something that can be borrowed – we may want to define more specialised sub-classes (e.g. books, magazines, audio/visual material). These will share the attributes defined for the general class but will have specific differences (for example there could be a charge for borrowing audio/visual items).

## 1.6 THE BENEFITS OF THE OBJECT ORIENTED PROGRAMMING APPROACH

Whether or not you develop programs in an Object Oriented way, before you write the software you must first develop a model of what that software should do and how it would work. Object Oriented modelling is based on the ideas of abstraction, encapsulation, inheritance and polymorphism.

The general proponents of the Object Oriented approach claims that this model provides:

- Better abstractions (modelling information and behaviour together).
- Better maintainability (more comprehensible, less fragile software).
- Better reusability (classes as encapsulated components that can be used in other systems).

We will look at these claims throughout this book and, in Chapter 14, will see a case study showing in detail how Object Oriented analysis works and how the resultant models can be implemented in an Object Oriented programming language (i.e. C#). The software that comes from this can be downloaded with this book.

## 1.7 SOFTWARE IMPLEMENTATION

Before a computer can complete useful tasks for us, e.g. check the spelling in our documents, software needs to be written and implemented on the machine it will run on. Software implementation involves writing program source code and preparing this so that it can run on a particular machine. Of course before the software is written it needs to be designed and at some point it needs to be tested. There are many iterative lifecycles to support the process of design, implementation and testing that involve multiple implementation phases. In a later chapter we will consider this in more detail and consider how new agile methods can help.

For now our particular concern are the three long established approaches to getting source code to execute on a particular machine...

- Compilation into machine-language object code
- Direct execution of source code by ‘interpreter’ program
- Compilation into intermediate object code which is then interpreted by run-time system

Implementing C# programs involves compiling the source code (C#) into machine-language object code. This approach has some advantages and disadvantages and it is worth comparing these three options in order to appreciate the implications for the C# developer.

## Compilation

Compilation requires a compiler to translate the source code into machine code for the relevant hardware/operating system combination.

Strictly speaking there are two stages: Compilation of program units (usually files), followed by linking (when the complete executable program is put together including the separate program units and relevant library code etc.).

The compiled program then runs as a ‘native’ application for that platform.

This is the oldest model, used by early languages like Fortran and Cobol, and many modern ones like C#. It allows fast execution speeds but requires re-compilation of the program each time the code is changed or each time we want to run this code on a machine with a different operating system.



### Shaping tomorrow's world – today

Our business is at the heart of a connected world – a world where communication is empowering people, business and society. Our networks, telecom services and multimedia solutions are shaping tomorrow. And this might just be your chance to shape your own future.

### It's a people thing

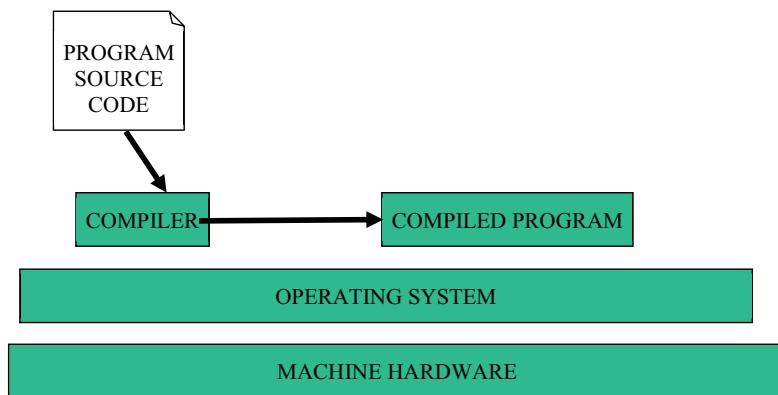
We are looking for high-caliber people who can see the opportunities, people who can bring knowledge, energy and vision to our organization. In return we offer the chance to work with cutting-edge technology, personal and professional development, and the opportunity to make a difference in a truly global company.

We are currently recruiting both new graduates and experienced professionals in four areas: **Software, Hardware, Systems and Integration & Verification**.

Are you ready to shape your future? Begin by exploring a career with Ericsson. Visit [www.ericsson.com/join-ericsson](http://www.ericsson.com/join-ericsson)



This process is indicated in the figure below:



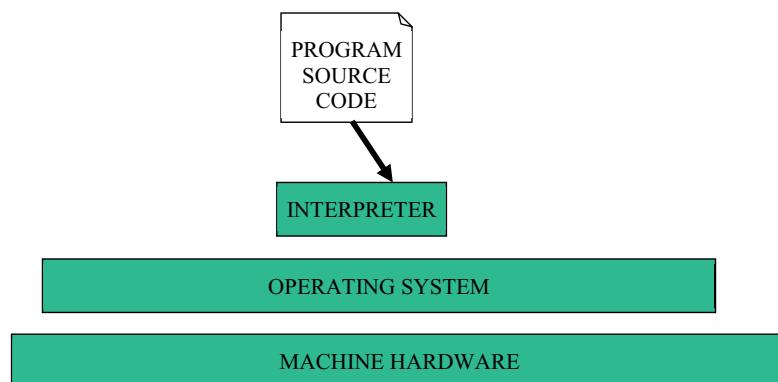
### Interpretation

An alternative method of getting code to run is to use an interpreter. With an interpreter source code is not translated into machine code. Instead an interpreter reads the source code and performs the actions it specifies.

We can say that the interpreter is like a ‘virtual machine’ whose machine language is the source code language.

No re-compilation is required after changing the code, but the interpretation process inflicts a significant impact on execution speed (see figure below).

Scripting languages such as Python tend to work in this way.



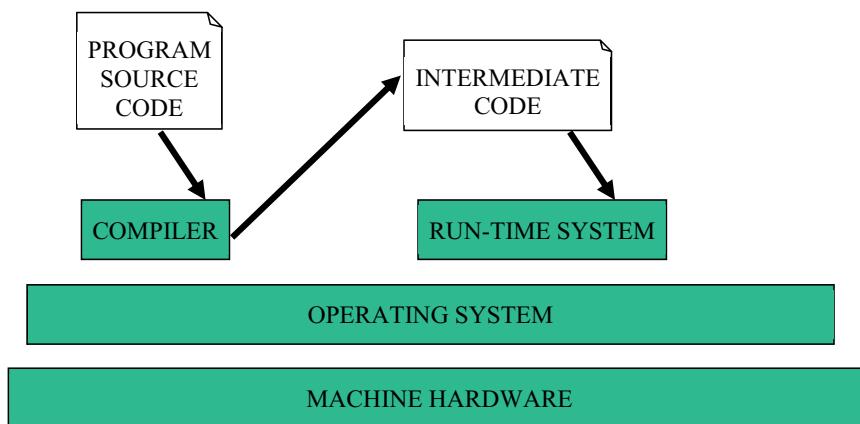
Interpreted programs can be slow but can work on any machine that has an appropriate interpreter. They do not need to be compiled for different machines.

## Intermediate Code

There is a model, a hybrid of the previous two, where intermediate code is created.

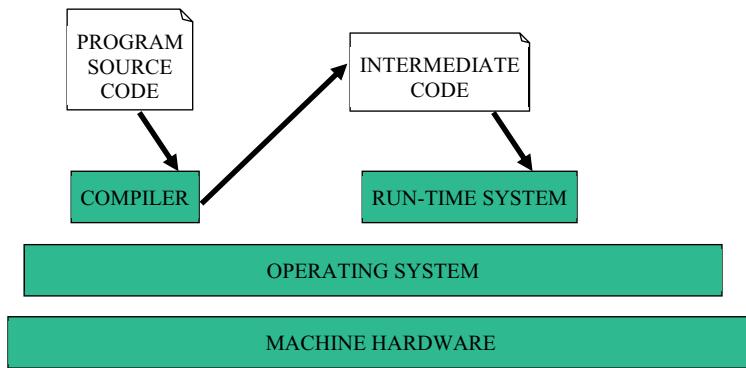
Compilation takes place to convert the source code into a more efficient intermediate representation which can be executed by a ‘run-time system’ (again a sort of ‘virtual machine’) more quickly than direct interpretation of the source code. However, the use of an intermediate code which is then executed by run-time system software allows the compilation process to be independent of the operating system/hardware platform, i.e. the same intermediate code should run on different platforms so long as an appropriate run-time system is available for each platform.

This approach is long-established (e.g. in Pascal from the early 1970s) and is how Java works. Java is a modern Object Oriented Language which is an alternative to C# and yet shares many similar features from the programmers point of view.



## Running Java Programs

To run Java programs we must first generate intermediate code (called bytecode) using a compiler available as part of the Java Development Kit (JDK). Thus a Java compiler does not create .exe files i.e. code that could run directly on a specific machine but instead generates .class files.



A version of the Java Runtime Environment (JRE), which incorporates a Java Virtual machine (VM), is required to execute the bytecode and the Java library packages. Thus a JRE must be present on any machine which is to run Java programs.

The Java bytecode is standard and platform independent and as JRE's have been created for most computing devices (including PCs, laptops, mobile phones, internet devices etc.) this makes Java programs highly portable and by compiling the code to an intermediate language Java strives to attain the fast implementation speeds obtained by fully compiled systems. Once complied Java code can run on any machine with a JRE irrespective of its underlying operating system without needing to be recompiled.

**Löydä koulutuksesi!**

Studentum.fi auttaa sinua löytämään  
itsellesi sopivan opiskelupaikan  
koulutusviidakosta. Etsi, vertaile ja  
löydä oma koulutuksesi!

**Studentum.fi**  
Löydä koulutuksesi!

## Running C# Programs

As we will see in section 1.8, C# programs, like Java programs, are also converted into an intermediate language but a C# compiler then goes further and links these with the necessary dll files to generate programs that can be directly executed on the target machine. C# programs are therefore fully compiled programs (i.e. .exe files are generated). While these are fast and efficient, the source code must be recompiled each time we wish to run the program on a machine with a different operating system.

C# it is part of the .NET framework which aims to deliver additional benefits for the programmer.

Some programmers mistakenly believe that C# was written to create programmes only for the Windows operating system. This is not true, as we will see when we consider the .NET framework.

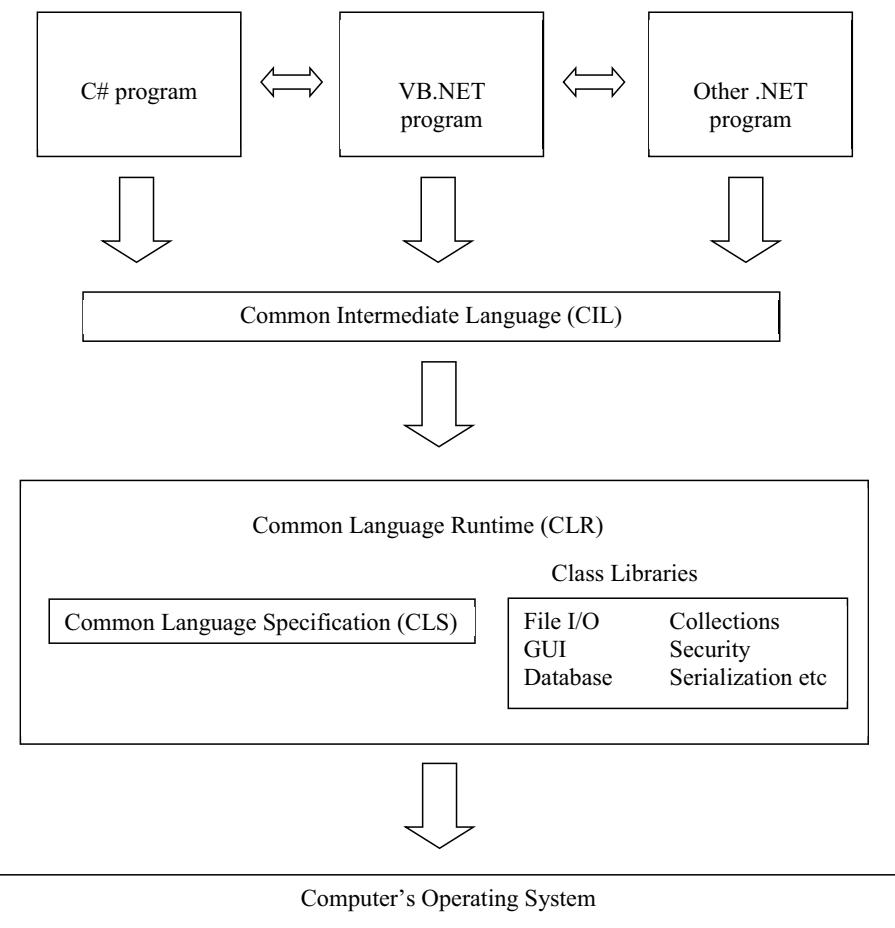
## 1.8 AN INTRODUCTION TO THE .NET FRAMEWORK

The .NET framework was designed to make life simpler for programmers by providing a common platform that can be used when writing programs in several different programming languages (C# and Visual Basic among others).

The .NET platform provides common features that can be used by programmers of all .NET languages. These include:

- A Common Language Runtime (CLR) engine that allows all .NET programs to run and use common features based on a Common Language Specification (CLS). The CLR also shields the programmer from having to deal with issues that are specific to individual operating systems.
- A comprehensive class library providing a wealth of built functionality (GUI, File I/O, Database, Threading, Serialization, Security, Web applications, Networking etc.).
- A Common Intermediate Language (CIL). All .NET source code is compiled into a Common Intermediate Language this provides the ability to integrate code written in any .NET language making use of common exception handling facilities.

This relationship can be shown graphically as below:



C# programs are thus partly compiled into a Common Intermediate Language. This is then linked with a CLR engine to produce a '.exe' file.

Taken all together the .NET framework does not mean that all of the .NET languages are identical, but they do have access to an identical library of methods that can be used by all .NET Languages. Thus learning to program one .NET language makes it easier to learn and program another .NET language.

.NET programs can only run where a CLR engine has been created for the underlying operating system. Microsoft Windows comes with a CLR engine and can therefore run .NET programs but very old operating systems may need a CLR engine installed before .NET programs can be run.

To enable .NET programs to be platform independent Microsoft published an open source specification for a CLR engine (called a Virtual Execution System) this included the definition of the C# language and a Common Language Infrastructure (CLI).

By using these definitions, CLR engines have been created for other operating systems (e.g. Mac OS and Linux) and, by using these, .NET programs can run on different platforms... not just Microsoft Windows. However before programs can be run on these different platforms they do need to be recompiled for each platform thus .NET programs, including C# programs, are not as portable as Java programs.

While CLR engines are perhaps not as widely available as JREs, they do exist for other platforms. One example of an open source CLR is Mono ([www.mono-project.com](http://www.mono-project.com)). Mono is an open source, cross-platform, implementation of C# and the CLR that is compatible with Microsoft's .NET framework. One part of the Mono project was called is MonoTouch but is now Xamarin ([www.xamarin.com](http://www.xamarin.com)). This allows you to create C# and .NET apps for iPhone and iPod Touch, while taking advantage of iPhone APIs.

As well as compiling our C# programs it is possible to create a software installation routine that will download via the web a .NET runtime environment and automatically install this along with our software (assuming a .NET runtime environment is not already installed).

The advertisement features a large central image of a woman in a dark blazer leaning over two young children (a boy and a girl) who are looking at a laptop screen together. To the left, the e-Learning for Kids logo is displayed. The background is a yellow and orange abstract design with swirling patterns. In the bottom right corner, there is a green oval containing text about the organization's achievements. At the very bottom, there is a paragraph providing information about e-Learning for Kids.

**About e-Learning for Kids** Established in 2004, e-Learning for Kids is a global nonprofit foundation dedicated to fun and free learning on the Internet for children ages 5 - 12 with courses in math, science, language arts, computers, health and environmental skills. Since 2005, more than 15 million children in over 190 countries have benefitted from eLessons provided by EFK! An all-volunteer staff consists of education and e-learning experts and business professionals from around the world committed to making difference. eLearning for Kids is actively seeking funding, volunteers, sponsors and courseware developers; get involved! For more information, please visit [www.e-learningforkids.org](http://www.e-learningforkids.org).

- The number 1 MOOC for Primary Education
- Free Digital Learning for Children 5-12
- 15 Million Children Reached

We will be writing and running code written in C# throughout this book and in doing so we will be making use of a compiler, the Common Language Runtime (CLR) engine and some of the more common class libraries. However the prime aim of this book is to teach generic Object Oriented programming and modelling principles that are common across a range of Object Oriented languages – not just .NET languages.

While we will illustrate Object Oriented principles in this book with C# code we will not concern ourselves further with the intricacies of how the CLR engine works, details regarding how .NET programs are compiled nor the detailed operation of the Common Intermediate Language (CIL).

However in order that the examples illustrated in this book can be demonstrated as practical worked examples we will introduce two modern Interactive Development Environments (IDEs), and some other tools specifically designed for the creation of .NET programs (see Chapter 10).

## 1.9 SUMMARY

Object Oriented programming involves the creation of classes by modelling the real world. This allows more specialised classes to be created that inherit the behaviour of the generalised classes.

Polymorphic behaviour means that systems can be changed, as business needs change, by adding new specialised classes and these classes can be accessed by the rest of the system without any regard to their specialised behaviour and without changing other parts of the current system.

We will return to each of the concepts introduced here throughout the book and hopefully, by the end, you will have a good understanding of these concepts and understand how to apply them using C#.

# 2 THE UNIFIED MODELLING LANGUAGE (UML)

## Introduction

This chapter will introduce you to the roles of the Unified Modelling Language (UML) and explain the purpose of four of the most common diagrams (class diagrams, object diagrams, sequence diagrams and package diagrams). These diagrams are used extensively when describing software designed according to the Object Oriented programming approach. Throughout this book particular emphasis will be placed on class diagrams as these are the most used part of the UML notation.

## Objectives

By the end of this chapter you will be able to...

- Explain what UML is and explain the role of four of the most common diagrams.
- Draw class diagrams, object diagrams, sequence diagrams and package diagrams.
- Use the correct notation to represent different class relationships: Association, Dependency, Inheritance and Implementation.

The material covered in this chapter will be expanded on throughout later chapters of the book and the skills developed here will be used in later exercises (particularly regarding class diagrams).

This chapter consists of six sections:

- 1) An introduction to UML
- 2) UML Class Diagrams
- 3) UML Syntax
- 4) UML Package Diagrams
- 5) UML Object diagrams
- 6) UML Sequence Diagrams

## 2.1 AN INTRODUCTION TO UML

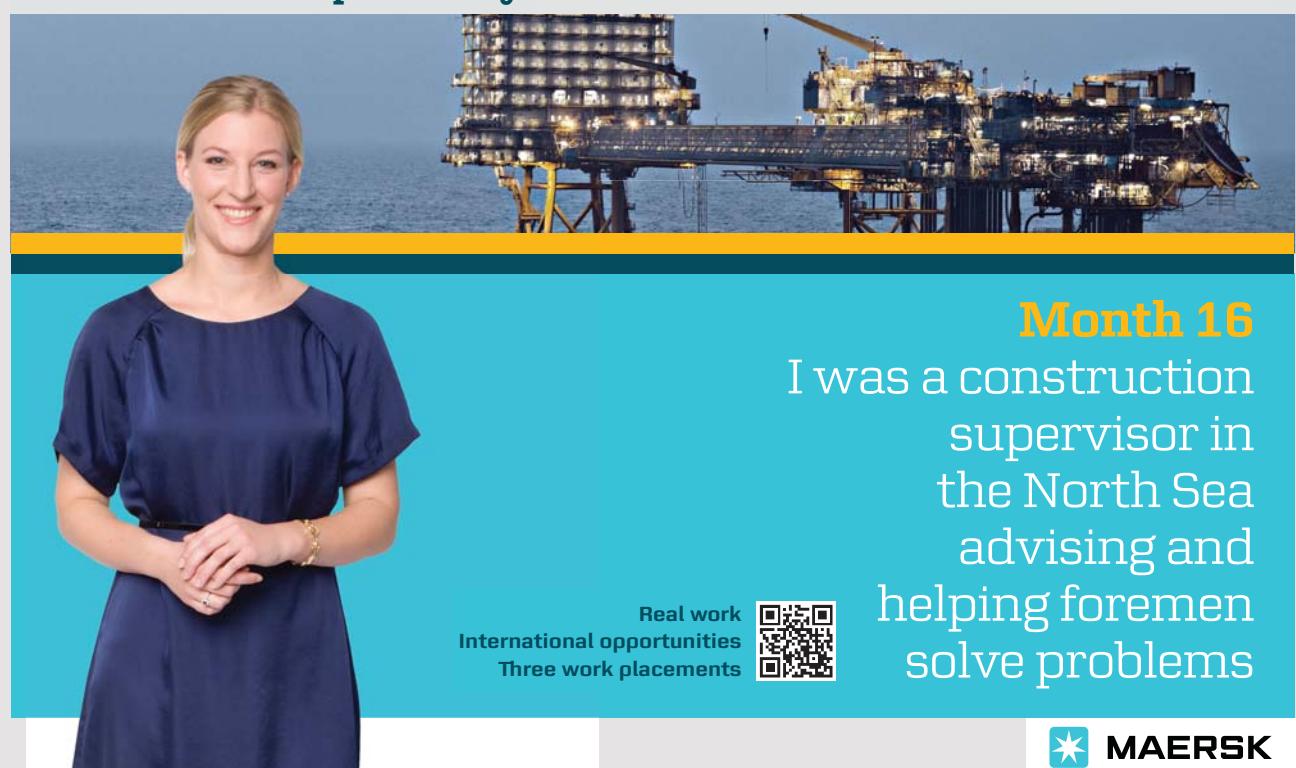
The Unified Modelling Language (UML) is sometimes described as though it was a methodology. It is not!

A methodology is a system of processes in order to achieve a particular outcome e.g. an organised sequence of activities in order to gather user requirements. UML does not describe the procedures a programmer should follow – hence it is not a methodology. It is, on the other hand, a precise diagramming notation that will allow program designs to be represented and discussed. As it is graphical in nature it becomes easy to visualise, understand and discuss the information presented in the diagram. However, as the diagrams represent technical information they must be precise and clear in order for them to work. Therefore, there is a precise notation that must be followed.

As UML is not a methodology, it is left to the user to follow whatever processes they deem appropriate in order to generate the designs described by the diagrams. UML does not constrain this – it merely allows those designs to be expressed in an easy to use, but precise, graphical notation.

I joined MITAS because  
I wanted **real responsibility**

The Graduate Programme  
for Engineers and Geoscientists  
[www.discovermitas.com](http://www.discovermitas.com)



Month 16

I was a construction supervisor in the North Sea advising and helping foremen solve problems

Real work  
International opportunities  
Three work placements

QR code

MAERSK

A process will be explained, in Chapter 7, that will help you to generate good UML designs and this is demonstrated in the case study, see Chapter 14. Developing good designs is a skill that requires practise. For now we will just concentrate on the UML notation not these processes.

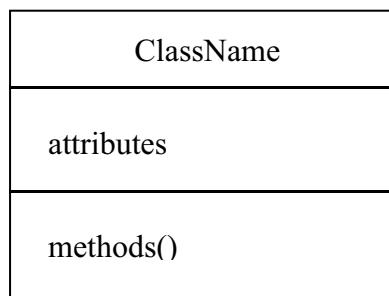
## 2.2 UML CLASS DIAGRAMS

Classes are the basic components of any Object Oriented software system and UML class diagrams provide an easy way to represent these. As well as showing individual classes, in detail, class diagrams show multiple classes and how they are related to each other. Thus a class diagram shows the architecture of a system.

A class consists of:

- A unique name (conventionally starting with an uppercase letter).
- A list of attributes (int, double, boolean, String etc.).
- A list of methods.

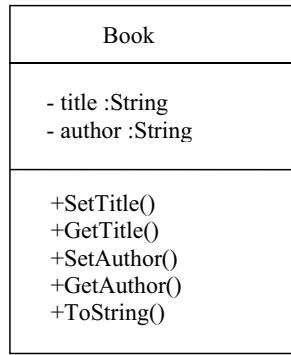
This is shown in a simple box structure...



For attributes and methods, visibility modifiers are shown (+ for public access, – for private access). Attributes are normally kept private and methods are normally made public.

Accessor methods are created to provide access to private attributes when required. Thus, a public method SetTitle() can be created to change the value of a private attribute ‘title’.

Thus, a class Book, with String attributes of title and author, and the following methods SetTitle(), GetTitle(), SetAuthor(), GetAuthor() and ToString() would be shown as...



Note: String shown above is not a primitive data type but is itself a class. Hence it starts with a capital letter.

### A Note on Naming Conventions

Some programmers use words beginning in capitals to denote class names and words beginning in lowercase to represent attributes or methods (thus `ToString()` would be shown as `toString()`). This is a common convention when designing and writing programs in Java (another common Object Oriented language). However, it is not a convention followed by C# programmers – where method names usually start in Uppercase. Method names can be distinguished from class names by the use of `()`. This in the example above.

- ‘Book’ is a class
- ‘title’ is an attribute and
- ‘SetTitle()’ is a method.

UML diagrams are not language specific thus a software design, communicated via UML diagrams, can be implemented in a range of Object Oriented languages.

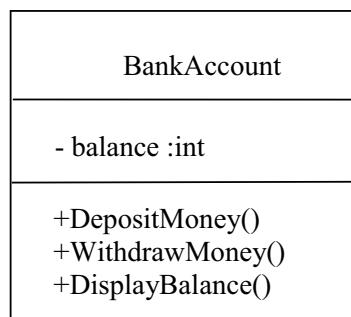
Furthermore, traditional accessor methods, i.e. getters and setters, are not required in C# programs as they are replaced by ‘properties’. Properties are, in effect, hidden accessor methods thus the getter and setter methods shown above (`GetTitle()`, `SetTitle()` etc.) are not required in a C# program. In C# an attribute would be defined called ‘title’ and a property would be defined as ‘Title’. This would allow us to set the ‘title’ directly by using the associated property ‘`Title =...;`’.

The UML diagrams shown in this book will use the naming convention common among C# programmers...for the simple reason that we will be writing sample code in C# to demonstrate the Object Oriented principles discussed here. Though initially, we will show conventional accessor methods these will be replaced with properties when coding.

### Activity 1

Draw a diagram to represent a class called 'BankAccount' with the attribute balance (of type int) and methods DepositMoney(), WithdrawMoney() and DisplayBalance(). Show appropriate visibility modifiers.

### Feedback 1



The diagram above shows this information.



FACTCARDS

Are you working in academia, research or science? And have you ever thought about working and moving to the Netherlands?

Arriving (33)

Living (50)

Working (101)

Studying (51)

Research (50)

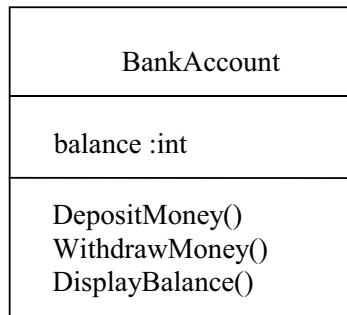
Factcards.nl offers all the **information** that you need if you wish to proceed your **career** in the **Netherlands**.

The information is ordered in the categories arriving, living, studying, working and research in the Netherlands and it is freely and easily accessible from your smartphone or desktop.

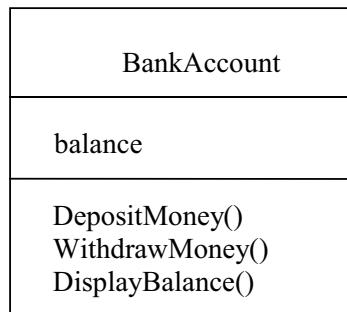
[VISIT FACTCARDS.NL](http://VISIT FACTCARDS.NL)

UML allows us to suppress any information we do not wish to highlight in our diagrams – this allows us to suppress irrelevant detail and bring to the reader's attention just to the information we wish to focus on. Therefore the following are all valid class diagrams...

Firstly with the access modifiers not shown...



Secondly with the access modifiers and the data types not shown...



And finally with the attributes and methods not shown...



i.e. there is a class called 'BankAccount' but the details of this are not being shown.

Of course virtually all C# programs will be made up of many classes and classes will relate to each other – some classes will make use of other classes. These relationships are shown by arrows. Different type of arrow indicate different relationships (including inheritance and aggregation relationships).

In addition to this class diagrams can make use of keywords, notes and comments.

As we will see in examples that follow, a class diagram can show the following information:

- Classes
  - attributes
  - operations
  - visibility
- Relationships
  - navigability
  - multiplicity
  - dependency
  - aggregation
  - composition
- Generalization/specialization
  - inheritance
  - interfaces
- Keywords
- Notes and Comments

## 2.3 UML SYNTAX

As UML diagrams convey precise information there is a precise syntax that should be followed.

Attributes should be shown as: *visibility name : type multiplicity*

Where visibility is one of:

- ‘+’ public
- ‘-’ private
- ‘#’ protected
- ‘~’ package

and Multiplicity is one of:

- ‘n’ exactly n
- ‘\*’ zero or more
- ‘m..n’ between m and n

The following are examples of attributes correctly specified using UML:

**- custRef : int [1]**

A private attribute custRef is a single int value.

This would often be shown as – **custRef : int** however with no multiplicity shown we cannot safely assume a multiplicity of one was intended by the author.

**# itemCodes : String [1..\*]**

A protected attribute itemCodes is one or more String values

**validCard : boolean**

An attribute validCard, of unspecified visibility, has unspecified multiplicity

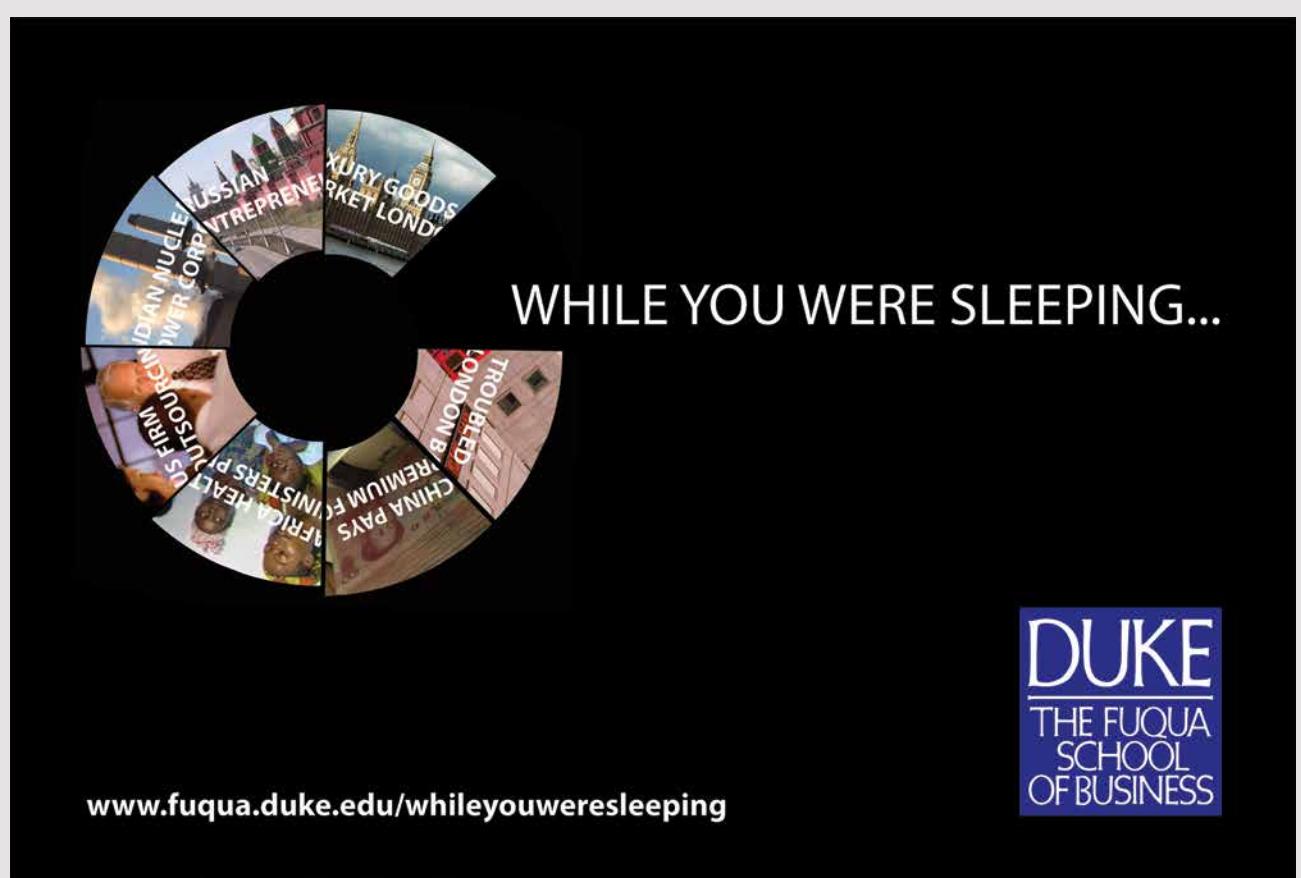
Operations also have a precise syntax and should be shown as:

*visibility name (par1 : type1, par2 : type2): returntype*

where each parameter is shown (in parenthesis) and then the return type is specified.

An example would be:

**+ AddName (newName : String) : boolean**



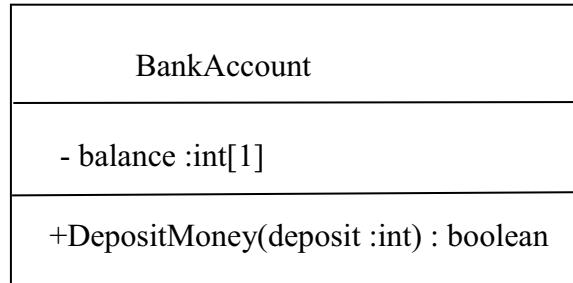
This denotes a public method 'AddName' which takes one parameter 'newName' of type String and returns a boolean value.

### Activity 2

Draw a diagram to represent a class called 'BankAccount' with a private attribute balance (this being a single integer) and a public method DepositMoney() which takes an integer parameter, 'deposit' and returns a boolean value. Fully specify all of this information on a UML class diagram.

### Feedback 2

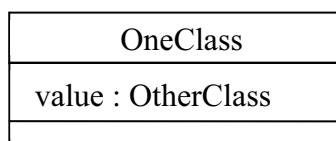
The diagram below shows this information



### Denoting Relationships

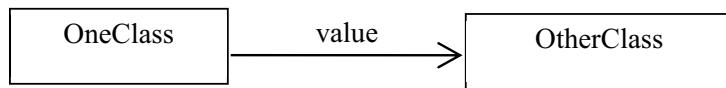
As well as denoting individual classes, Class diagrams denote relationships between classes. One such relationships is called an 'Association'. We will learn a lot more about associations in Chapter 7 where we look at how to model an Object Oriented system. Here we are just learning the UML notation we will be using later.

In a class attributes will be defined. These could be primitive data types (int, boolean etc.) or complex objects as defined by other classes.



Thus the figure above shows a class 'OneClass' that has an attribute 'value'. This value is not a primitive data type but is an object of type defined by 'OtherClass'.

We could denote exactly the same information by the diagram below.

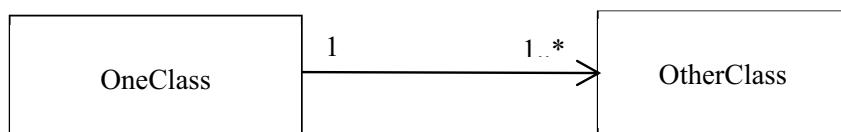


We use an association when we want to give two related classes, and their relationship, prominence on a class diagram

The ‘source’ class points to the ‘target’ class.

Strictly we could use an association when a class we define has a String instance variable – but we would not do this because the String class is part of the C# platform and ‘taken for granted’ like an attribute of a primitive type. This would generally be true of all library classes unless we are drawing the diagram specifically to explain some aspect of the library class for the benefit of someone unfamiliar with its purpose and functionality.

Additionally we can show multiplicity at both ends of an association:



This implies that ‘OneClass’ maintains a collection of objects of type ‘OtherClass’. Collections are an important part of the C# library that we will look at the use of collections in Chapter 9.

### Activity 3

Draw a diagram to represent a class called ‘Catalogue’ and a class called ‘ItemForSale’ as defined below:

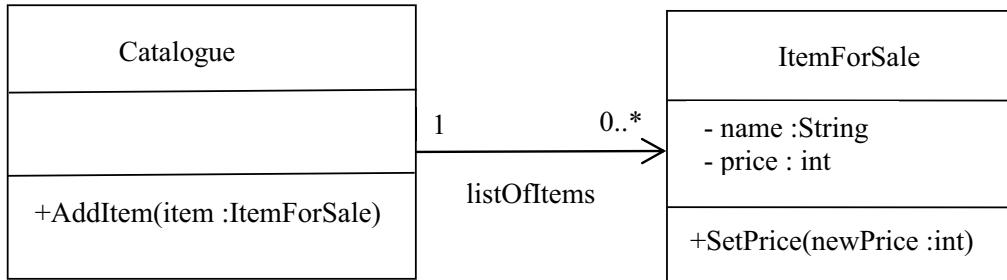
ItemForSale has an attribute ‘name’ of type String and an attribute ‘price’ of type int. It also has a method SetPrice() which takes an integer parameter ‘newPrice’.

‘Catalogue’ has an attribute ‘listOfItems’ (i.e. the items currently held in the catalogue). As zero or more items can be stored in the catalogue ‘listOfItems’ will need to be an array or collection. ‘Catalogue’ also has one method AddItem() which takes an ‘item’ as a parameter (of type ItemForSale) and adds this item to the ‘listOfItems’.

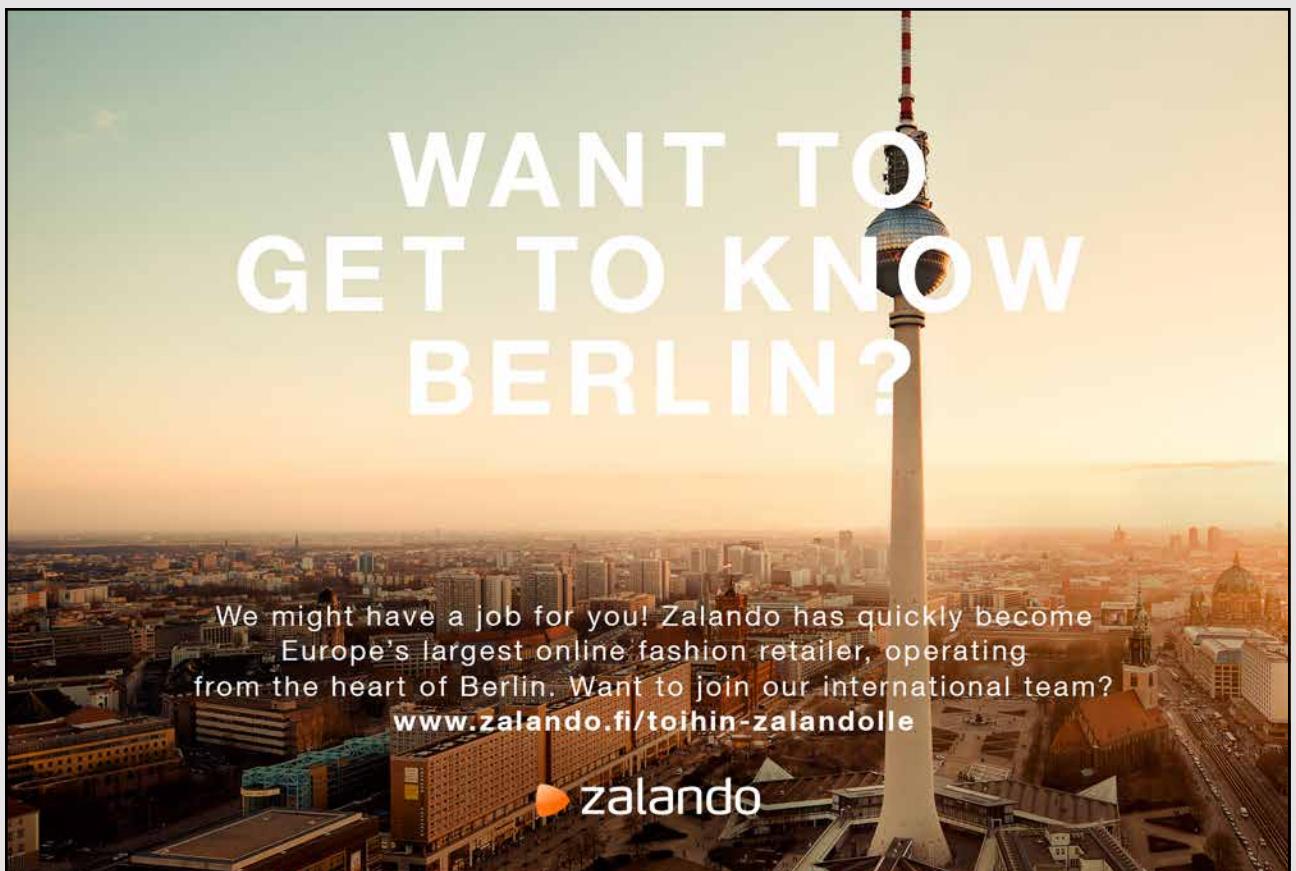
Draw this on a class diagram showing appropriate visibility modifiers for attributes and methods.

### Feedback 3

The diagram below shows this information



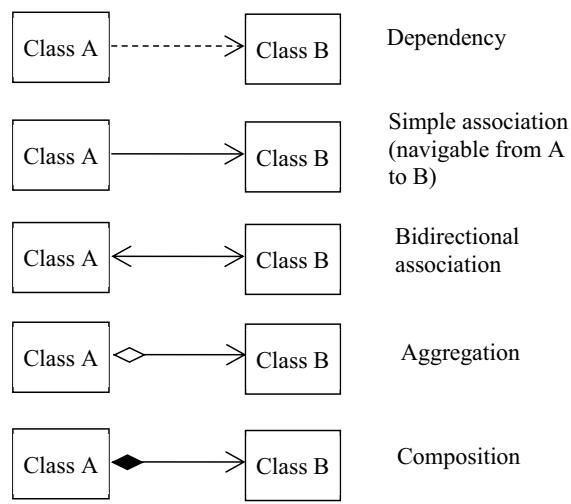
Note: According to the naming convention followed here all class names begin in uppercase, attribute names begin in lowercase, method names begin in uppercase and use () to distinguish them from class names. Also note that the class ItemForSale describes a single item (not multiple items). 'listOfItems' however maintains a list of zero or more individual objects.



## Types of Association

There are various different types of association denoted by different arrows:

- Dependency,
- Simple association
- Bidirectional association
- Aggregation
- Composition



## Dependency



- Dependency is the most unspecific relationship between classes (not strictly an ‘association’)
- Class A in some way uses facilities defined by Class B
- Changes to Class B may affect Class A

Typical use of dependency lines would be where Class A has a method which is passed a parameter object of Class B, or uses a local variable of that class, or calls ‘static’ methods in Class B.

Example: A Print() method may require a printer object as a parameter. Each time the Print() method is invoked a different printer object could be passed as a parameter and thus the printout will appear in a different place. Thus while the class containing the Print() method requires a printer object to work it does not need to be permanently associated with one specific printer.

## Simple Association

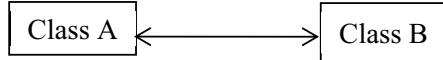


- In an association Class A ‘uses’ objects of Class B
- Typically Class A has an attribute of Class B
- Navigability is from A to B:  
i.e. A Class A object can access the Class B object(s) with which it is associated.  
The reverse is not true – the Class B object doesn’t ‘know about’ the Class A object

A simple association typically corresponds to an instance variable in Class A of the target class B type.

Example: the Catalogue above needs access to 0 or more ItemForSale so items can be added to or removed from a Catalogue. An ItemForSale does not need to access a Catalogue in order to set its price or perform some other method associated with the item itself.

## Bidirectional Association



- Bidirectional Association is when Classes A and B have a two-way association
- Each refers to the other class
- Navigability A to B and B to A:
  - A Class A object can access the Class B object(s) with which it is associated
  - Object(s) of Class B ‘belong to’ Class A
  - Implies reference from A to B
  - Also, a Class B object can access the Class A object(s) with which it is associated

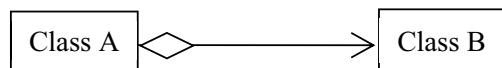
A bidirectional association is complicated because each object must have a reference to the other object(s) and generally, bidirectional associations are much less common than unidirectional ones.

An example of a bidirectional association may be between a ‘Degree’ and ‘Student’. i.e. given a Degree we may wish to know which Students are studying on that Degree. Alternatively, starting with a student we may wish to know the Degree they are studying.



As many students study the same Degree at the same time, but students usually only study one Degree there is still a one to many relationship here (of course we could model a situation where we record degrees being studied and previous degrees passed – in this case, as a student may have passed more than one degree, we would have a many to many relationship).

## Aggregation



Aggregation denotes a situation where Object(s) of Class B ‘belong to’ Class. This implies reference from A to B.

While aggregation implies that objects of Class B belong to objects of Class A it also implies that object of Class B retain an existence independent of Class A. Some designers believe there is no real distinction between aggregation and simple association.

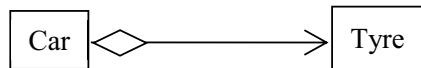
The advertisement features a man in a suit looking at a car made of paper cutouts of various documents, including a CV and a house plan. The background includes the Skoda logo and the slogan "SIMPLY CLEVER".

**We will turn your CV into an opportunity of a lifetime**

Do you like cars? Would you like to be a part of a successful brand?  
We will appreciate and reward both your enthusiasm and talent.  
Send us your CV. You will be surprised where it can take you.

Send us your CV on  
[www.employerforlife.com](http://www.employerforlife.com)

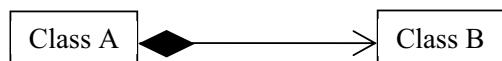
An example of aggregation would be between a class Car and a class Tyre



We think of the tyres as belonging to the car they are on, but at the garage they may be removed and placed on a rack to be repaired. Their existence isn't dependent on the existence of a car with which they are associated.

Some designers believe that aggregation can be replaced as a simple association as this change makes no difference to the programmer. While it is worth being aware of this arrow type we will not be making use of this relationship in this book.

## Composition



Composition is similar to aggregation but implies a much stronger belonging relationship i.e. Object(s) of Class B are 'part of' a Class A object. Again it implies reference from A to B.

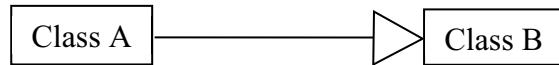
Composition is a much 'stronger' relationship than aggregation: in this case Class B objects are an integral part of Class A and in general objects of Class B never exist other than as part of Class A, i.e. they have the same 'lifetime'.

An example of composition would be between Points, Lines and Shapes as elements of a Picture. These objects can only exist as part of a picture, and if the picture is deleted they are also deleted.

As well as denoting associations, class diagrams can denote:

- Inheritance,
- Interfaces,
- Keywords and
- Notes

## Inheritance



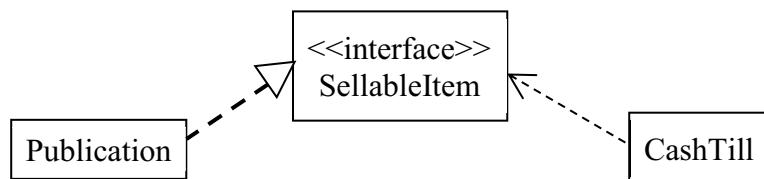
Aside from associations, the other main modelling relationship is inheritance: Class A ‘inherits’ both the interface and implementation of Class B, though it may override implementation details and supplement both.

Inheritance is very important and we will look at this in detail in Chapter 3.

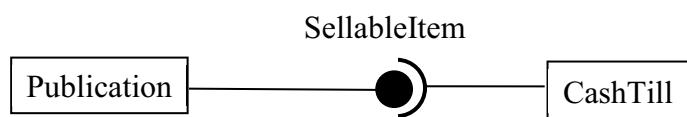
## Interfaces

Interfaces are similar to inheritance however with interfaces only the interface is inherited. The methods defined by the interface must be implemented in every class that implements the interface.

Interfaces can be represented using the <<interface>> keyword:



There is also a shorthand for this



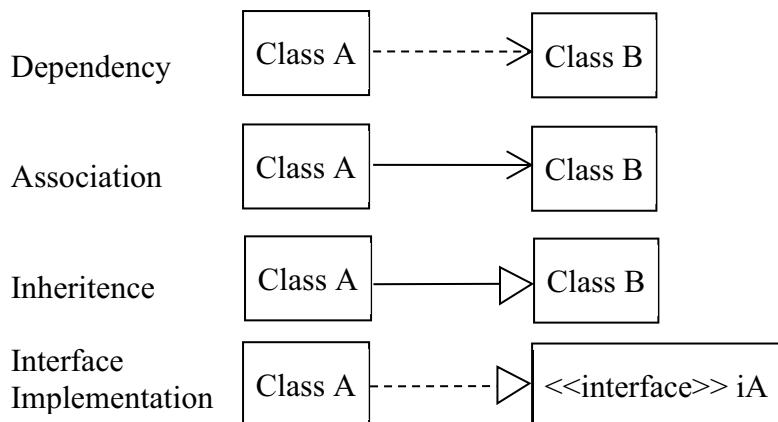
In both cases these examples denote that the SellableItem interface is **required by** CashTill and **implemented by** Publication.

Note: the dotted-line version of the inheritance line/arrow which shows that Publication ‘implements’ or ‘realizes’ the SellableItem interface.

The “ball and socket” notation is new in UML 2 – it is a good visual way of representing how interfaces connect classes together.

We will look at the application of interfaces in more detail in Chapter 4.

Indeed throughout most of this book we will make extensive use of the following relationships:



## Turning a challenge into a learning curve. Just another day at the office for a high performer.

### Accenture Boot Camp – your toughest test yet

Choose Accenture for a career where the variety of opportunities and challenges allows you to make a difference every day. A place where you can develop your potential and grow professionally, working alongside talented colleagues. The only place where you can learn from our unrivalled experience, while helping our global clients achieve high performance. If this is your idea of a typical working day, then Accenture is the place to be.

It all starts at Boot Camp. It's 48 hours that will stimulate your mind and enhance your career prospects. You'll spend time with other students, top Accenture Consultants and special guests. An inspirational two days

packed with intellectual challenges and activities designed to let you discover what it really means to be a high performer in business. We can't tell you everything about Boot Camp, but expect a fast-paced, exhilarating

and intense learning experience. It could be your toughest test yet, which is exactly what will make it your biggest opportunity.

Find out more and apply online.

Visit [accenture.com/bootcamp](http://accenture.com/bootcamp)

- Consulting • Technology • Outsourcing

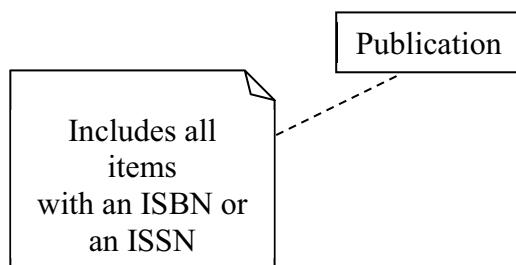
**accenture**  
High performance. Delivered.

## Keywords

UML defines keywords to refine the meaning of the graphical symbols. We have seen <>interface<> and we will also make use of <>abstract<> but there are many more.

An abstract class may alternatively be denoted by showing its name in *italics* though this is perhaps less obvious to a casual reader.

## Notes



Finally we can add notes to comment on a diagram element. This gives us a 'catch all' facility for adding information not conveyed by the graphical notation.

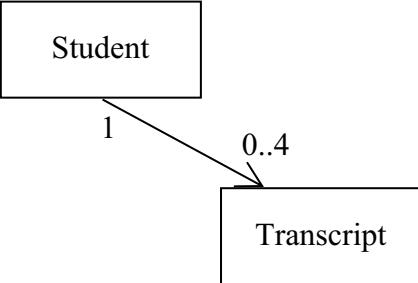
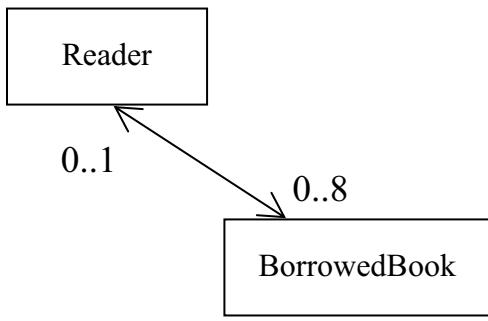
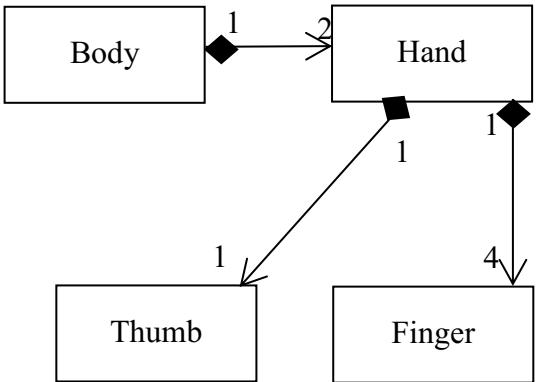
### Activity 4

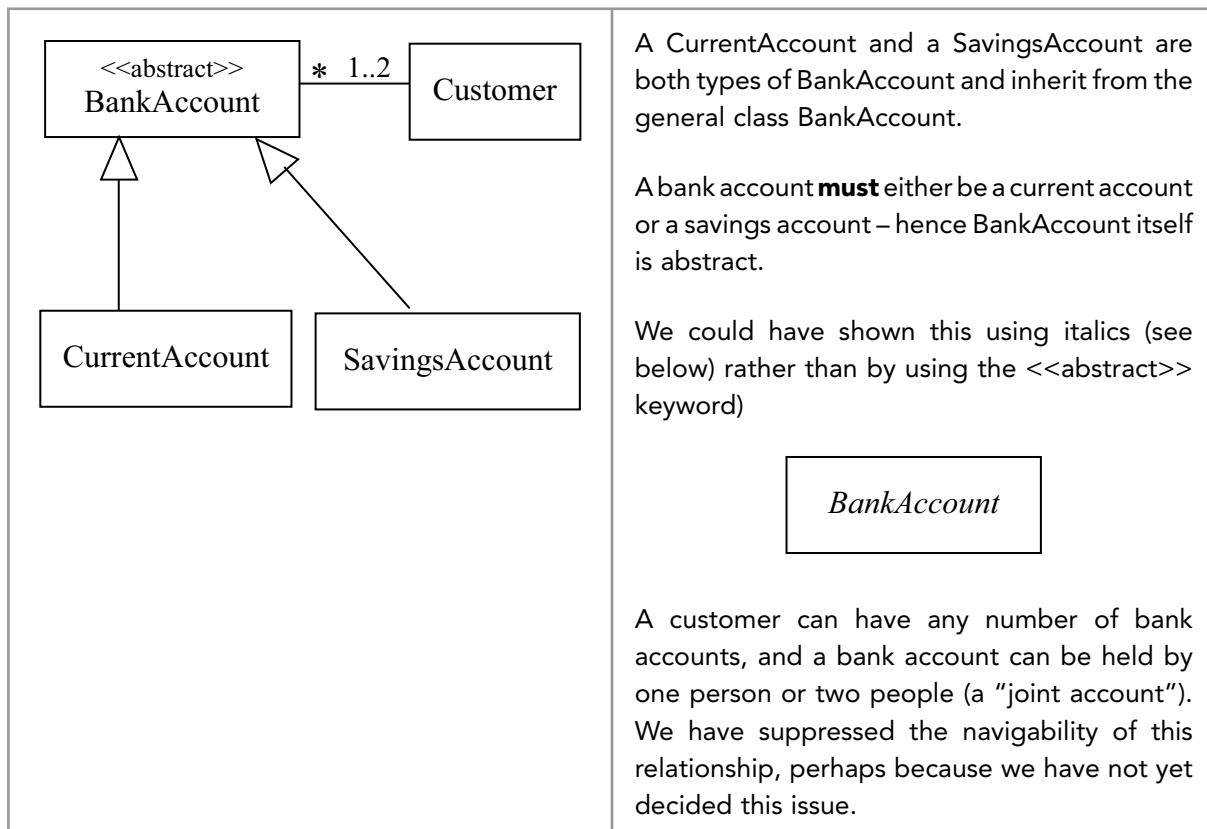
From your own experience, try to develop a model which illustrates the use of the following elements of UML Class Diagram notation:

- simple association
- bidirectional association
- aggregation (tricky!)
- composition
- association multiplicity
- generalization (inheritance)
- interfaces
- notes

For this exercise concentrate on the relationships between classes rather than the details of the class members. If possible explain and discuss your model with other students or friends.

To help you get started some small examples are given below:

 <pre> classDiagram     class Student     class Transcript     Student "1" --&gt; "0..4" Transcript   </pre>	<p>In a University administration system we might produce a transcript of results for each year the student has studied (including a possible placement year).</p> <p>This association relationship is naturally unidirectional – given a student we might want to find their transcript(s), but it seems unlikely that we would have a transcript and need to find the student to whom it belonged.</p>
 <pre> classDiagram     class Reader     class BorrowedBook     Reader "0..1" &lt;--&gt; "0..8" BorrowedBook   </pre>	<p>In a library a reader can borrow up to eight books. A particular book can be borrowed by at most one reader.</p> <p>We might want a bidirectional relationship as shown here because, in addition to being able to identify all the books which a particular reader has borrowed, we might want to find the reader who has borrowed a particular book (for example to recall it in the event of a reservation).</p>
 <pre> classDiagram     class Body     class Hand     class Finger     class Thumb     Body "1" --&gt; "2" Hand     Hand "1" --&gt; "4" Finger     Finger "1" --&gt; "1" Thumb   </pre>	<p>This might be part of the model for some kind of educational virtual anatomy program.</p> <p>Composition – the “strong” relationship which shows that one object is (and has to be) part of another seems appropriate here.</p> <p>The multiplicities would not always work for real people though – they might have lost a finger due to an accident or disease, or have an extra one because of a genetic anomaly (polydactyly).</p> <p>But what if we were modelling the “materials” in a medical school anatomy lab? A hand might not always be part of a body! Perhaps the “weaker” aggregation relationship would reflect this better.</p>



# Brain power



By 2020, wind could provide one-tenth of our planet's electricity needs. Already today, SKF's innovative know-how is crucial to running a large proportion of the world's wind turbines.

Up to 25 % of the generating costs relate to maintenance. These can be reduced dramatically thanks to our systems for on-line condition monitoring and automatic lubrication. We help make it more economical to create cleaner, cheaper energy out of thin air.

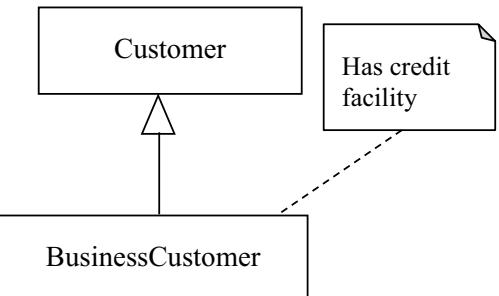
By sharing our experience, expertise, and creativity, industries can boost performance beyond expectations.

Therefore we need the best employees who can meet this challenge!

The Power of Knowledge Engineering

Plug into The Power of Knowledge Engineering.  
Visit us at [www.skf.com/knowledge](http://www.skf.com/knowledge)

**SKF**

 <pre> classDiagram     class Customer     class BusinessCustomer      Customer &lt; -- BusinessCustomer     note over BusinessCustomer: Has credit facility   </pre>	<p>A business customer is a type of customer and can inherit the attributes of a customer.</p> <p>A note is used here to add some information which cannot be conveyed with standard UML notation and may not be obvious simply from the names and relationships of the classes depicted.</p>

#### Feedback 4

There is no specific feedback for this activity.

## 2.4 UML PACKAGE DIAGRAMS

While class diagrams are the most commonly used diagram of those defined in UML notation, and we will make significant use of these throughout this book, there are other diagrams that denote different types of information. Here we will touch upon three of these:

- Package Diagrams
- Object Diagrams and
- Sequence Diagrams

World maps, country maps and city maps all show spatial information, just on different scales and with differing levels of detail. Large Object Oriented systems can be made up of hundreds, or potentially thousands, of classes and thus if the class diagram was the only way to represent the architecture of a large system it would become overly large and complex.

Thus, just as we need world maps, we need package diagrams to show the general architecture of a large system. Even modest systems can be broken down into a few basic components (i.e. packages). We will see an example of packages in use in Chapter 14. For now we will just look at the package diagramming notation.

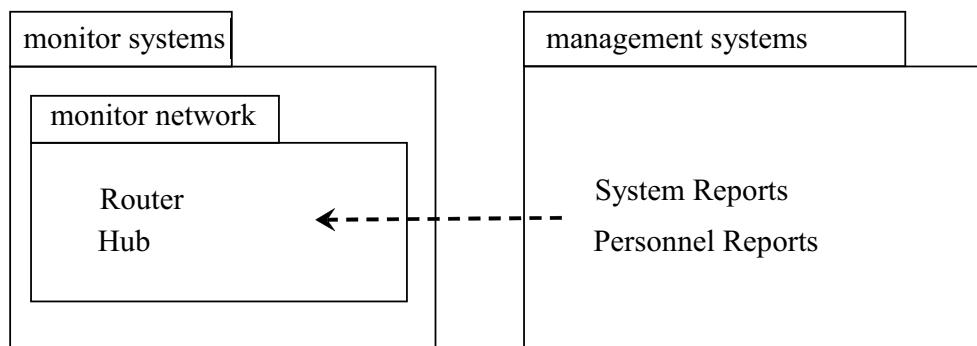
Packages diagrams allow us to provide a level of organisation and encapsulation above that of individual classes. Packages are implemented in C# by creating subfolders and defining a ‘namespace’. When writing a large system in C# we use this to segment a large system into smaller more manageable sub-systems. We denote these sub-systems using package diagrams during the design stage.

A large C# development should be split into suitable packages at the design stage. UML provides a ‘Package Diagram’ to represent the relationships between classes and packages.

We can depict:

- Classes within packages
- Nesting of packages (i.e. packages within packages)
- Dependencies between packages

In the diagram below we see two packages: ‘monitor systems’ and ‘management systems’. These depict part of a large system for a multinational corporation to manage and maintain their operations including their computer systems and personnel.



Looking at this more closely we can see that inside the ‘monitor systems’ package is another called ‘monitor network’. This package contains at least two classes, ‘Router’ and ‘Hub’ though presumably it contains many other related classes.

The package ‘management systems’ contains two (or more) classes: ‘System Reports’ and ‘Personnel Reports’.

Furthermore we can see that the classes inside the package ‘management systems’ in some way use the classes inside ‘monitor network’. Presumably it is the ‘System Reports’ class that makes use of these as it needs to know about the status of the network.

Note: the normal UML principle of suppression applies here – these packages may contain other packages and each package may contain dozens of classes but we simply choose not to show them.

In the class diagram below we have an alternative way of indicating that ‘Router’ is a class inside the ‘monitor network’ package, which is in turn inside ‘monitor systems’ package.



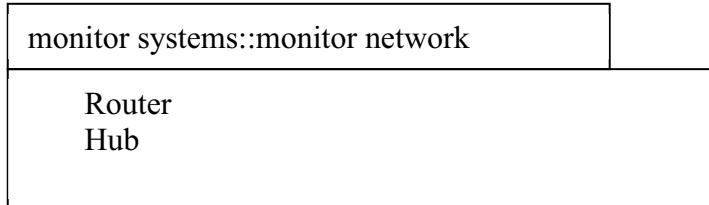
### |||| We have ambitions. Also for you.

SimCorp is a global leader in financial software. At SimCorp, you will be part of a large network of competent and skilled colleagues who all aspire to reach common goals with dedication and team spirit. We invest in our employees to ensure that you can meet your ambitions on a personal as well as on a professional level. SimCorp employs the best qualified people within economics, finance and IT, and the majority of our colleagues have a university or business degree within these fields.

**Ambitious?** Look for opportunities at [www.simcorp.com/careers](http://www.simcorp.com/careers)

monitor systems::monitor network::Router

And again, below, a format which shows both classes more concisely.



These different representations will be useful in different circumstances depending on what a package diagram is aiming to convey.

## Package Naming

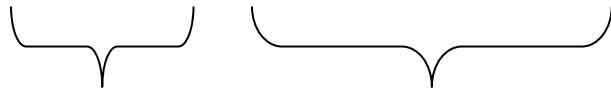
By convention, package names are normally in lowercase. We will follow this convention as it helps to distinguish between packages and classes.

For local, individual projects, packages could be named according to personal preference, e.g.

```
mysystem
mysystem.interface
mysystem.engine
mysystem.engine.util
mysystem.database
```

However, packages are often distributed and to enable this packages need globally unique names, thus a naming convention has been adopted based on URLs

### uk.co.ebay.www.department.project.package



Part based on organisation URL (e.g. www.ebay.co.uk) reversed, though this does **not** specifically imply you can download the code there.

Part distinguishing the particular project and component or subsystem which this package contains.

Note: on a package diagram each element is not separated by a ‘.’ but by ‘::’.

#### Activity 5

You and a flatmate decide to go shopping together. For speed, split the following shopping list into two halves – items to be collected by you and items to be collected by your flatmate.

Apples, Furniture polish, Pears, Carrots, Toilet Rolls, Potatoes, Floor cleaner, Matches, Grapes

#### Feedback 5

To make your shopping efficient you probably organised your list into two lists of items that are located in the same parts of the shop:

List 1	List 2
Apples,	Furniture polish,
Pears,	Floor cleaner
Grapes	Matches
Carrots,	Toilet Rolls,
Potatoes	

### Activity 6

You run a team of three programmers and are required to write a program in C# to monitor and control a network system. The system will be made up of seven classes as described below. Organise these classes into three packages. Each programmer will then be responsible for the code in one package. Give the packages any name you feel appropriate.

Main	This class starts the system
Monitor	This class monitors the network for performance and breaches in security
Interface	This is a visual interface for entire system
Reconfigure	This allows the network to be reconfigured
RecordStats	This stores data regarding the network in a database
RemoteControl	This allows some remote control over the system via telephone
PrintReports	This uses the data stored in the database to print management reports for the organisations management.



What do you want to do?

No matter what you want out of your future career, an employer with a broad range of operations in a load of countries will always be the ticket. Working within the Volvo Group means more than 100,000 friends and colleagues in more than 185 countries all over the world. We offer graduates great career opportunities – check out the Career section at our web site [www.volvologroup.com](http://www.volvologroup.com). We look forward to getting to know you!

**VOLVO**

AB Volvo (publ)  
[www.volvologroup.com](http://www.volvologroup.com)

VOLVO TRUCKS | RENAULT TRUCKS | MACK TRUCKS | VOLVO BUSES | VOLVO CONSTRUCTION EQUIPMENT | VOLVO PENTA | VOLVO AERO | VOLVO IT  
VOLVO FINANCIAL SERVICES | VOLVO 3P | VOLVO POWERTRAIN | VOLVO PARTS | VOLVO TECHNOLOGY | VOLVO LOGISTICS | BUSINESS AREA ASIA

### Feedback 6

When organising a project into packages there is not always 'one correct answer' but if you organise your classes into appropriate packages (with classes that have related functionality) you improve the encapsulation of the system and improve the efficiency of your programmers. A suggested solution to activity 6 is given below.

```
interface
    Main
    Interface
    RemoteControl

network
    Monitor
    Reconfigure

database
    RecordStats
    PrintReports
```

### Activity 7

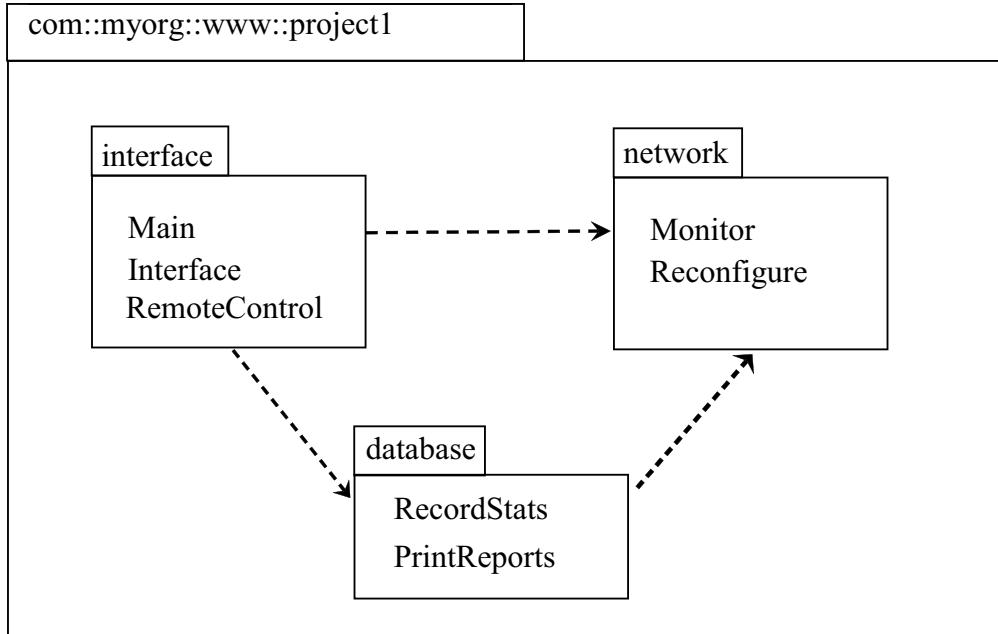
Assume the URL of your organisation is 'www.myorg.com' and the three packages and seven classes shown below are all part of 'project1'. Draw a package diagram to convey this information.

```
interface
    Main
    Interface
    RemoteControl

network
    Monitor
    Reconfigure

database
    RecordStats
    PrintReports
```

### Feedback 7



Note: Dependency arrows have been drawn to highlight relationships between packages. When more thought has been put into determining these relationships they may turn out to be associations (a much stronger relationship than a mere dependency).

## 2.5 UML OBJECT DIAGRAMS

Class diagrams and package diagrams allow us to visualise and discuss the architecture of a system. However, at times we wish to discuss the data a system processes. Object diagrams allow us to visual one instance of time and the data that a system may contain in that moment.

Object diagrams look superficially similar to class diagrams however, the boxes represent specific instances of objects.

Boxes are titled with:

**objectName : ClassName**

As each box describes a particular object at a specific moment in time, the box contains attributes and their values (at that moment in time).

attribute = value

These diagrams are useful for illustrating particular ‘snapshot’ scenarios during design.

The object diagram below shows several objects that may exist at a moment in time for a library catalogue system. The system contains two classes:

Book, which stores the details of a book and

Library, which maintains a collection of books, with books being added, searched for or removed as required.

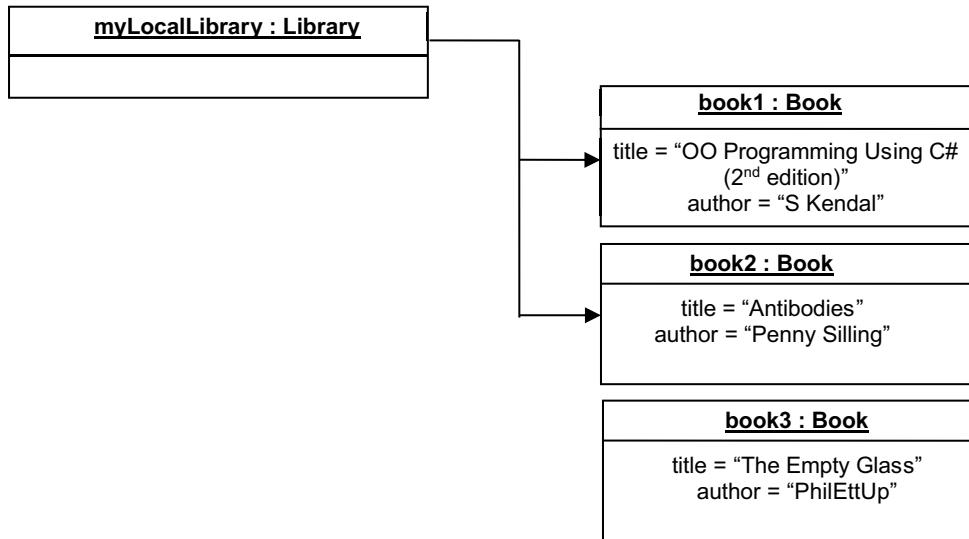
## TURN TO THE EXPERTS FOR SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develop acquisition and retention strategies.

Learn more at [linkedin.com/company/subscrybe/](https://www.linkedin.com/company/subscrybe/) or contact Managing Director Morten Suhr Hansen at [mtsuhr@subscrybe.dk](mailto:mtsuhr@subscrybe.dk)

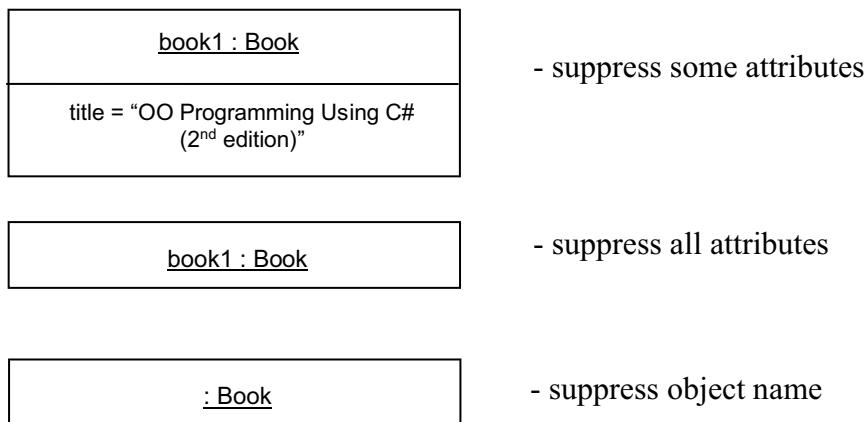
**SUBSCRYBE** - to the future



Looking at this diagram we can see that at a particular moment in time, while three books have been created only two have been added to the library. Thus if we were to search the library for ‘The Empty Glass’ we would not expect the book to be found.

As with class diagrams, elements can be freely suppressed on object diagrams.

For example, all of these are legal:



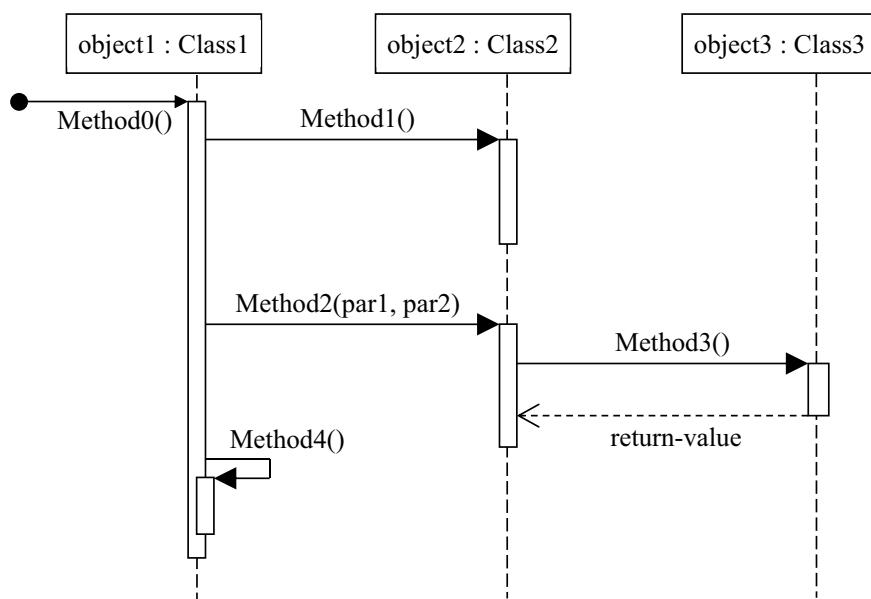
## 2.6 UML SEQUENCE DIAGRAMS

Sequence diagrams are entirely different from class diagrams or object diagrams.

Class diagrams describe the architecture of a system and object diagrams describe the state of a system at one moment in time.

Sequence diagrams describe how the system works over a period of time. Sequence diagrams are ‘dynamic’ rather than ‘static’ representations of the system. They show the sequence of method invocations within and between objects over a period of time. They are useful for understanding how objects collaborate in a particular scenario.

See the example below:

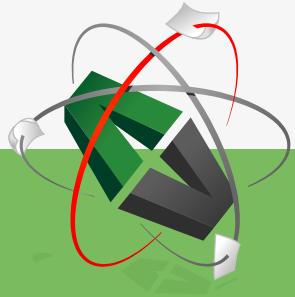


We have three objects in this scenario. Time runs from top to bottom and the vertical dashed lines (lifelines) indicate the objects' continued existence through time.

This diagram shows the following actions taking place:

- Firstly a method call (often referred to in Object Oriented terminology as a message) to Method0() comes to object1 from somewhere – this could be another class outside the diagram.
- object1 begins executing its Method0() (as indicated by the vertical bar (called an activation bar) which starts at this point).
- object1.Method0() invokes object2.Method1() – the activation bar indicates that this executes for a period then returns control to Method0().
- Subsequently object1.Method0() invokes object2.Method2() passing two parameters.
- Method2() subsequently invokes object3.Method3(). When Method3() ends it passes a return value back to Method2().
- Method2() completes and returns control to object1.Method0().
- Finally Method0() calls another method of the same object, Method4().

This e-book  
*is made with*  
**SetaPDF**



PDF components for PHP developers

[www.setasign.com](http://www.setasign.com)

## Selection and Iteration

The logic of a scenario often depends on selection ('if') and iteration (loops). There is a notation ('interaction frames') which allow if statements and loops to be represented in sequence diagrams however these tend to make the diagrams cluttered. Sequence diagrams are generally best used for illustrating particular cases, with the full refinement reserved for the implementation code.

Fowler ("UML Distilled", 3rd edition) gives a brief treatment of these constructs.

## 2.7 SUMMARY

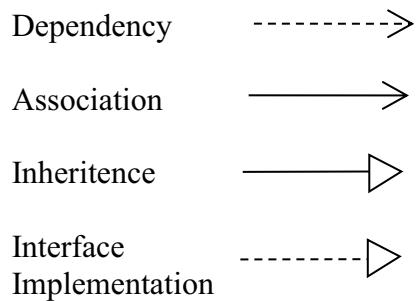
UML is not a methodology but a precise graphical notation.

Class diagrams and package diagrams are good for describing the architecture of a system.

Object diagrams describe the data within an application at one moment in time and sequence diagrams describe how a system works over a period of time.

UML gives different meaning to different arrows therefore one must be careful to use the notation precisely as specified.

Four essential relationships that we will make significant use of are:



With any UML diagram suppression is encouraged – thus the author of a diagram can suppress any details they wish in order to convey essential information to the reader.

# 3 INHERITANCE AND METHOD OVERRIDING

## Introduction

This chapter will start the develop an understanding of core Object Oriented concepts introduced in Chapter 1 and develop the skills needed to apply this knowledge. In particular the essential concepts of inheritance, method overriding and the appropriate use of the keyword ‘base’ will be explored.

## Objectives

By the end of this chapter you will be able to...

- Appreciate the importance of an Inheritance hierarchy.
- Understand how to use Abstract classes to factor out common characteristics
- Override methods (including those in the ‘Object’ class).
- Explain how to use ‘base’ to invoke methods that are in the process of being overridden.
- Document an inheritance hierarchy using UML.
- Implement inheritance and method overriding in C# programs.

All of the material covered in this chapter will be developed and expanded on in later chapters of this book. While this chapter will focus on understanding the application and documentation of an inheritance hierarchy, Chapter 7 will focus on developing the analytical skills required to define your own inheritance hierarchies.

This chapter consists of twelve sections:

- 1) Object Families
- 2) Generalisation and Specialisation
- 3) Inheritance
- 4) Implementing Inheritance in C#
- 5) Constructors
- 6) Constructor Rules
- 7) Access Control
- 8) Abstract Classes
- 9) Overriding Methods

- 10) The ‘Object’ Class
- 11) Overriding ToString() defined in ‘Object’
- 12) Summary

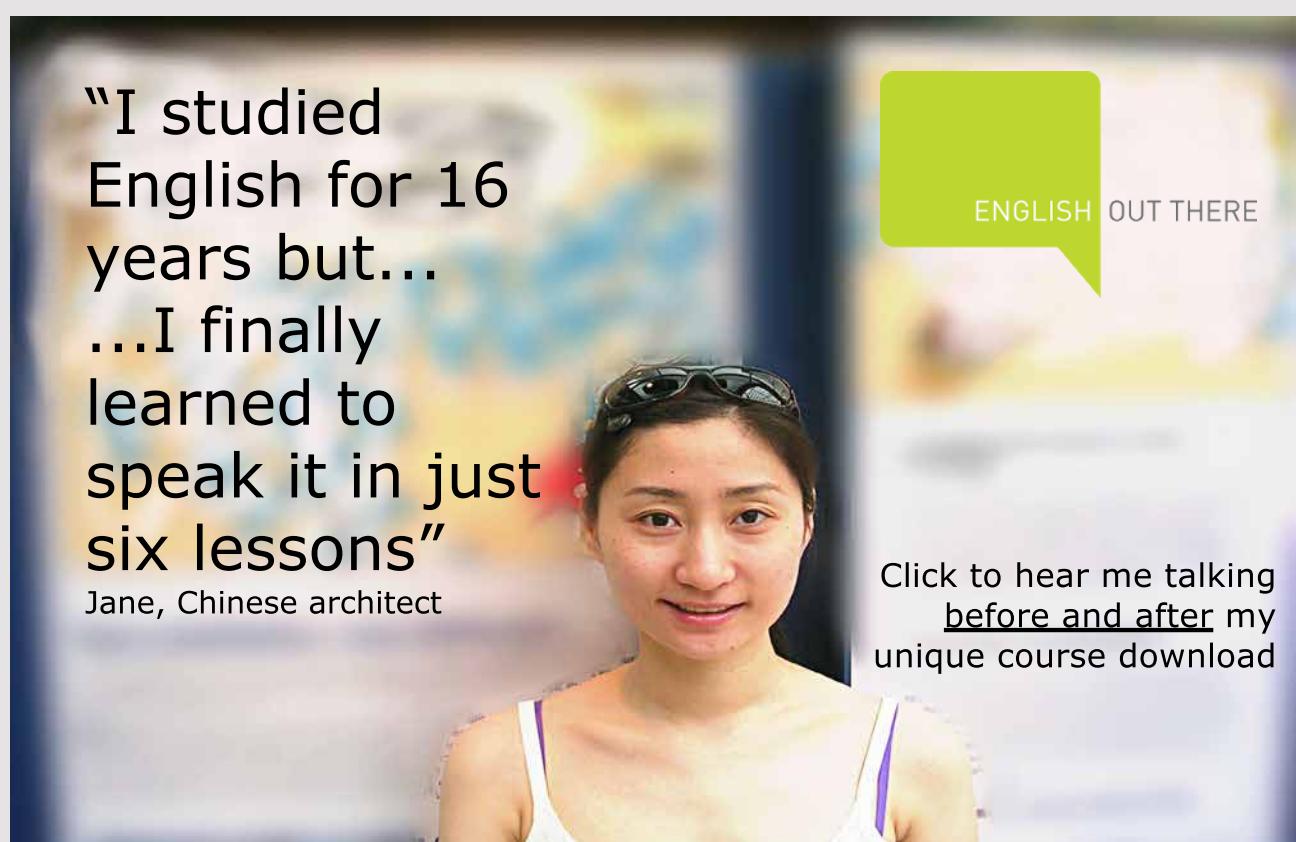
### 3.1 OBJECT FAMILIES

Many kinds of things in the world fall into related groups of ‘families’. ‘Inheritance’ is the idea ‘passing down’ characteristics from parent to child, and plays an important part in Object Oriented design and programming.

While you may already familiar with constructors, and access control (public/private), there are particular issues in relating these to inheritance we need to consider.

Additionally we need to consider the use of Abstract classes and method overriding as these are important concepts in the context of inheritance.

Finally we will look at the ‘Object’ class which has a special role in relation to all other classes in C#.



## 3.2 GENERALISATION AND SPECIALISATION

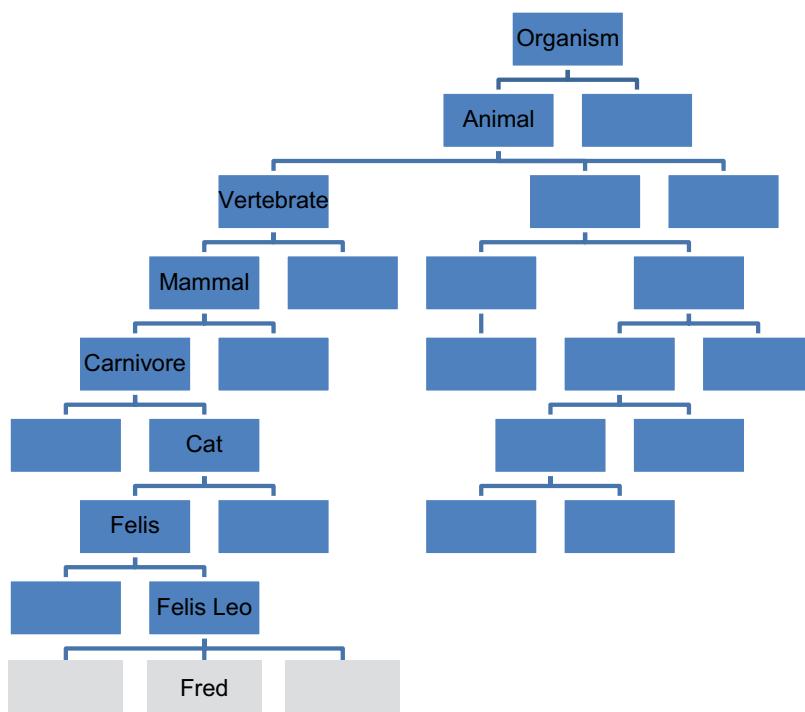
Classes are a generalized form from which objects with differing details can be created. Objects are thus ‘instances’ of their class. For example Student 051234567 is an instance of class Student. More concisely, 051234567 **is a** Student. Constructors are special methods that create an object from the class definition.

Classes themselves can often be organised by a similar kind of relationship.

One hierarchy, that we all have some familiarity with, is that which describes the animal kingdom:

- Kingdom (e.g. animal)
- Phylum (e.g. vertebrate)
- Class (e.g. mammal)
- Order (e.g. carnivore)
- Family (e.g. cat)
- Genus (e.g. felis)
- Species (e.g. felis leo)

We can represent this hierarchy graphically...



Of course to draw the complete diagram would take more time and space than we have available.

Here we can see one specific animal shown here: 'Fred'. Fred is not a class of animal but an actual animal.

Fred **is a** felis leo **is a** felis **is a** cat **is a** carnivore

Carnivores eat meat so Fred has the characteristic 'eats meat'.

Fred **is a** felis leo **is a** felis **is a** cat **is a** carnivore **is a** mammal **is a** vertebrate

Vertebrates have a backbone so Fred has the characteristic 'has a backbone'.

The 'is a' relationship links an individual to a hierarchy of characteristics. This sort of relationship applies to many real world entities, e.g. BonusSuperSaver **is a** SavingsAccount **is a** BankAccount.

### 3.3 INHERITANCE

We specify the general characteristics high up in the hierarchy and more specific characteristics lower down. An important principle in Object Oriented – we call this **generalization** and **specialization**.

All the characteristics from classes above in a class/object in the hierarchy are automatically featured in it – we call this **inheritance**.

Consider books and magazines – both are specific types of publication.

We can show classes to represent these on a UML class diagram. In doing so we can see some of the instance variables and methods these classes may have.

Book	Magazine
title author price copies	title price orderQty currIssue copies
SellCopy() OrderCopies()	SellCopy() AdjustQty() RecNewIssue()

Attributes ‘title’, ‘author’ and ‘price’ are obvious. Less obvious is ‘copies’ this is how many are currently in stock.

For books, OrderCopies() takes a parameter specifying how many extra copies are added to stock.

For magazines, orderQty is the number of copies received of each new issue and currIssue is the date/period of the current issue (e.g. “January 2011”, “Fri 6 Jan”, “Spring 2011” etc.). When a new issue is received the old issues are discarded and orderQty copies are placed in stock. Therefore RecNewIssue() sets currIssue to the date of new issue and restores copies to orderQty. AdjustQty() modifies orderQty to alter how many copies of subsequent issues will be stocked.

### Activity 1

Look at the ‘Book’ and ‘Magazine’ classes defined above and identify the commonalities and differences between two classes.

The advertisement features a background image of three diverse professionals (two men and one woman) smiling and looking at a tablet or document together. The We Thrive.net logo is in the top left corner. The main headline reads "How to retain your top staff" in large white text, with a subtext "FIND OUT NOW FOR FREE" below it. To the right, a sidebar titled "DO YOU WANT TO KNOW:" lists three questions with icons: a brain icon for "What your staff really want?", a checkmark icon for "The top issues troubling them?", and a stopwatch icon for "How to make staff assessments work for you & them, painlessly?". At the bottom right is a green button with the text "Get your free trial" and the tagline "Because happy staff get more done" below it.

### Feedback 1

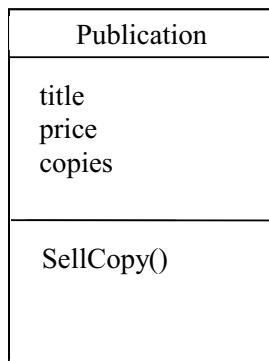
These classes have three instance variables in common: title, price, copies.  
They also have in common the method SellCopy().

The differences are as follows...

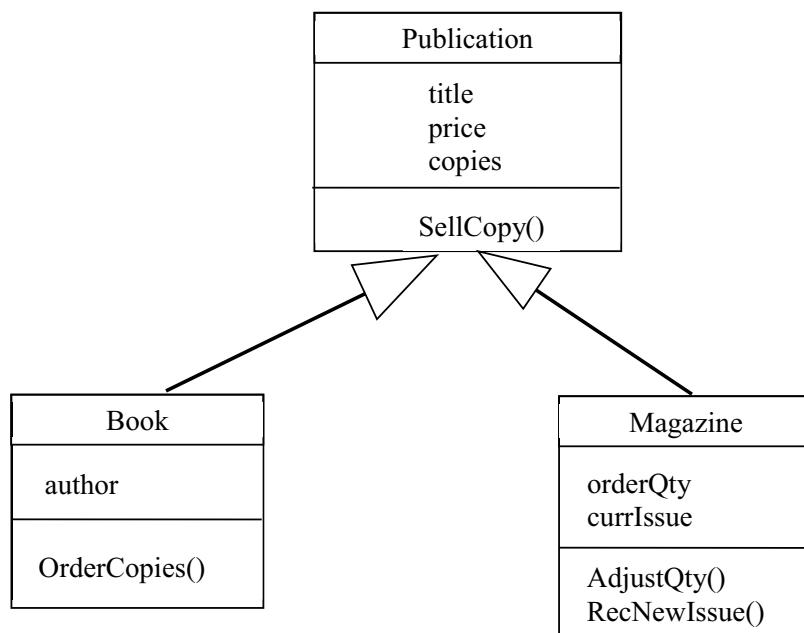
Book additionally has author, and OrderCopies().

Magazine additionally has orderQty, currIssue, AdjustQty() and RecNewIssue().

We can separate out ('factor out') these common members of the classes into a superclass called Publication. In C# a superclass is often called a 'base class'.



The differences will need to be specified as additional members for the 'subclasses' Book and Magazine. This is shown in the figure below.



In a UML Class Diagram the solid line with a hollow-centred arrow denotes inheritance.

**Different arrowheads, and different lines (solid or dashed), mean different things in UML (see Chapter 2) so, if we want Software Engineers and programmers to understand our designs, it is important that we use correct arrows.**

A subclass has the generalized superclass (or base class) characteristics + additional specialized characteristics. Thus the Book class has four instance variables (title, price, copies and author) it also has two methods (SellCopy() and OrderCopies()).

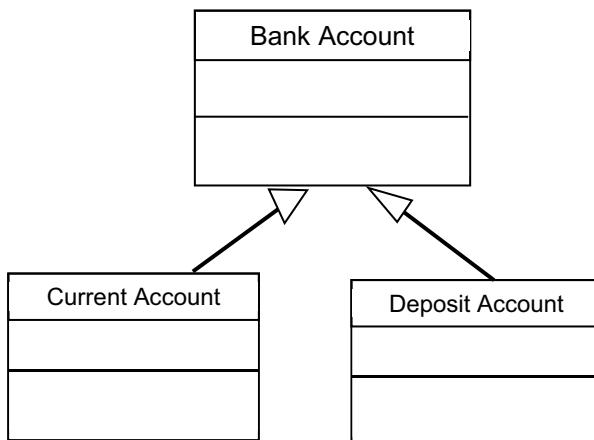
The inherited characteristics **are not** listed in subclasses. The arrow shows they are acquired from the superclass.

### Activity 2

Arrange the following classes into a suitable hierarchy and draw these on a class diagram...

- A current account
- A deposit account
- A bank account
- Simon's deposit account

### Feedback 2



The most general class goes at the top of the inheritance hierarchy with the other classes then inheriting the attributes and methods of this class.

Simon's deposit account should not be shown on a class diagram as this is a specific instance of a class i.e. it is an object.

### Activity 3

Arrange the following classes into a suitable hierarchy and draw these on a class diagram...

- A building
- A house
- A car

The advertisement features a background photograph of a person running on a path at sunset. In the foreground, there is a graphic of a target with three concentric circles. A yellow call-to-action button in the bottom right corner contains the text "READ MORE & PRE-ORDER TODAY" and "WWW.GAITEYE.COM". A hand cursor icon is positioned over the bottom right corner of the button.

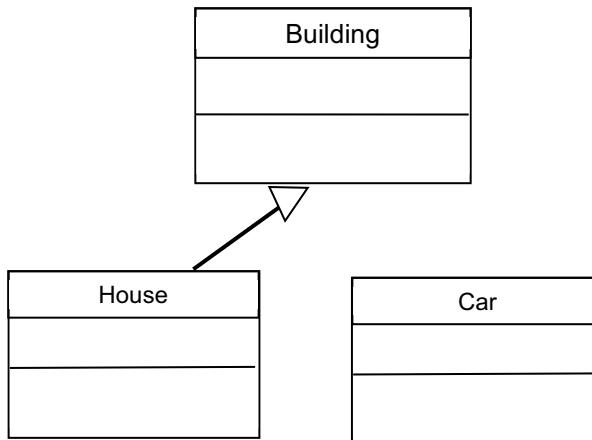
**gaiteye®**  
*Challenge the way we run*

**EXPERIENCE THE POWER OF FULL ENGAGEMENT...**

**RUN FASTER.  
RUN LONGER..  
RUN EASIER...**

READ MORE & PRE-ORDER TODAY  
[WWW.GAITEYE.COM](http://WWW.GAITEYE.COM)

### Feedback 3



A house is a type of building and can therefore inherit the attributes of building however this is not true of a car. We cannot place two classes in an inheritance hierarchy unless we can use the term **is a**. A car is not a building and cannot inherit the attributes of a building.

Note: class names, as always, begin with an uppercase letter.

### Activity 4

Describe the following using a suitable class diagram showing **any** sensible relationship...

A building for rent

This will have a method to determine the rent

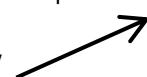
A house for rent

This will inherit the determine rent method

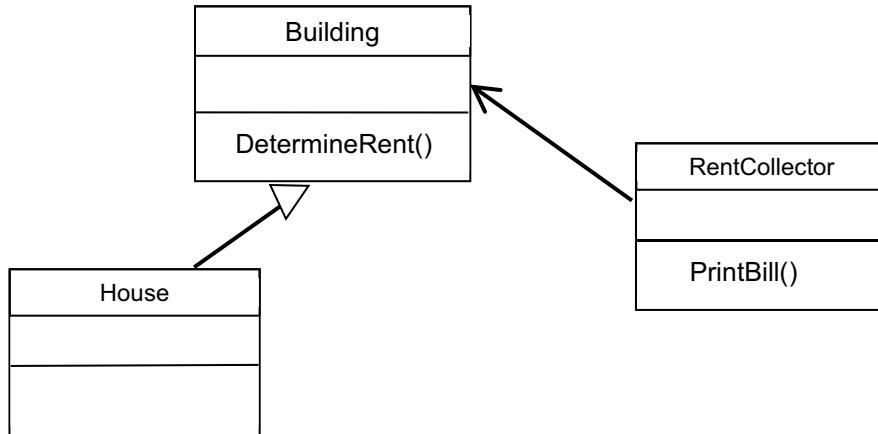
A rent collector (person)

This person will use the determine rent method to print out a bill

HINT: You may wish to use the following arrow



#### Feedback 4



Note: RentCollector does not inherit from Building as a RentCollector is a person not a type of Building. However there is a relationship (an association) between RentCollector and Building ie. a RentCollector needs to determine the rent for a Building in order to print out the bill.

#### Activity 5

Look at the feedback from Activity 4 and determine if a RentCollector can print out a bill for the rent due on a house (or can they just print a bill for buildings?).

#### Feedback 5

Firstly to print out a bill a RentCollector would need to know the rent due. There is no method 'DetermineRent()' defined for a house – but this does not mean it does not exist.

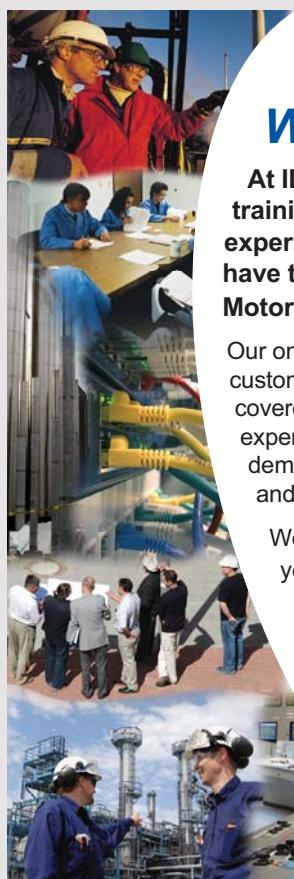
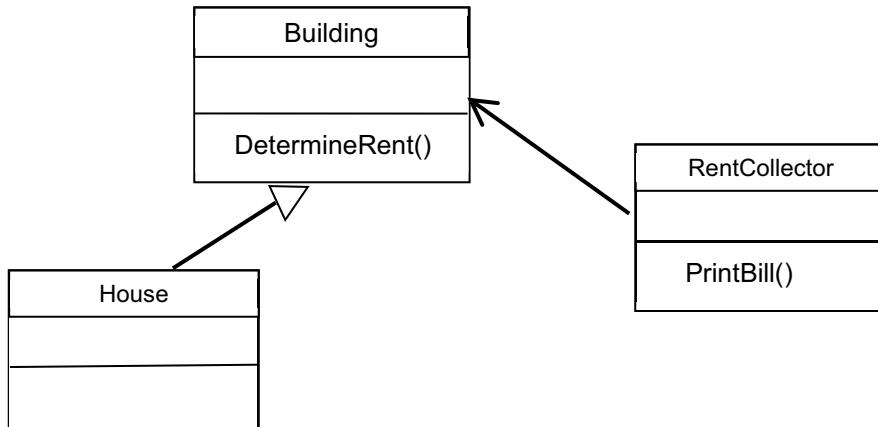
It must exist as House inherits the properties and methods of Building!

We only show methods in subclasses if they are either additional methods or methods that have been overridden (we will cover overriding very shortly).

A rent collector requires a building but a House **is a** type of Building. So, while no association is shown between the RentCollector and House, a Rentcollector could print a bill for a house. Wherever a Building object is required we could substitute a House object as this is a type of Building. This is an example of polymorphism and we will see other examples of this in Chapter 4.

### Activity 6

Modify this UML diagram to show that DetermineRent() is overridden (i.e. replaced) in House.



## Technical training on ***WHAT*** you need, ***WHEN*** you need it

At IDC Technologies we can tailor our technical and engineering training workshops to suit your needs. We have extensive experience in training technical and engineering staff and have trained people in organisations such as General Motors, Shell, Siemens, BHP and Honeywell to name a few.

Our onsite training is cost effective, convenient and completely customisable to the technical and engineering areas you want covered. Our workshops are all comprehensive hands-on learning experiences with ample time given to practical sessions and demonstrations. We communicate well to ensure that workshop content and timing match the knowledge, skills, and abilities of the participants.

We run onsite training all year round and hold the workshops on your premises or a venue of your choice for your convenience.

For a no obligation proposal, contact us today  
at [training@idc-online.com](mailto:training@idc-online.com) or visit our website  
for more information: [www.idc-online.com/onsite/](http://www.idc-online.com/onsite/)

Phone: +61 8 9321 1702  
Email: [training@idc-online.com](mailto:training@idc-online.com)  
Website: [www.idc-online.com](http://www.idc-online.com)

OIL & GAS  
ENGINEERING

ELECTRONICS

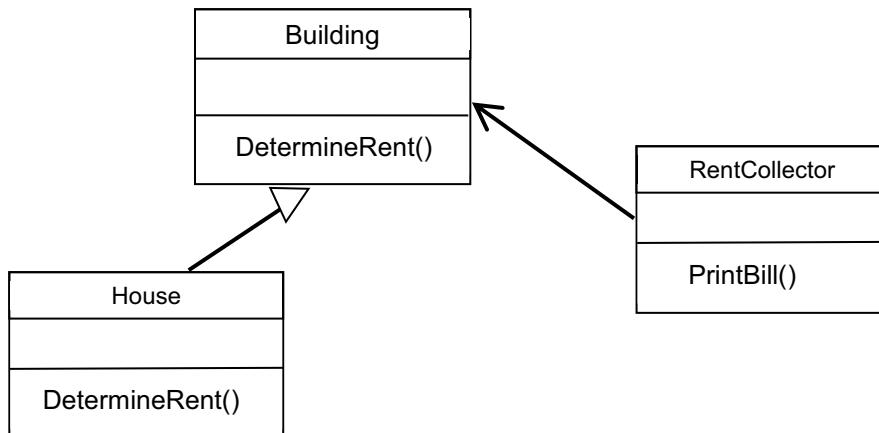
AUTOMATION &  
PROCESS CONTROL

MECHANICAL  
ENGINEERING

INDUSTRIAL  
DATA COMMS

ELECTRICAL  
POWER



**Feedback 6**

By showing `DetermineRent()` in `House` we are showing that this method is overriding (replacing) the one defined in the base class (`Building`).

Interestingly the .NET CLR engine will use the most correct `DetermineRent()` method depending upon which type of object the method is invoked on. Thus `RentCollector` will invoke the method defined in `House` if printing a bill for a house but will use the method defined in `Building` for any other type of building. This is automatic – the code in the `RentCollector` class does not distinguish between different types of `Building`.

Overriding will be discussed in more detail later in this chapter.

### 3.4 IMPLEMENTING INHERITANCE IN C#

No special features are required to create a superclass. Thus any class can be a superclass unless specifically prevented.

A subclass specifies it is inheriting features from a superclass using the `:` symbol. For example....

```
class MySubclass : MySuperclass
{
    // additional instance variables and
    // additional methods
}
```

### 3.5 CONSTRUCTORS

Constructors are methods that create objects from a class. Each class (whether sub or super) should encapsulate its own initialization in a constructor, usually relating to setting the initial state of its instance variables. Constructors are methods that are given the same name as the class.

A constructor for a superclass (or base class) should deal with general initialization.

Each subclass can have its own constructor for specialised initialization but it must often invoke the behaviour of the base constructor. It does this using the keyword **base**.

```
class MySubClass : MySuperClass
{
    public MySubClass (sub-parameters) : base(super-parameters)
    {
        // other initialization
    }
}
```

Usually some of the parameters passed to MySubClass will be initializer values for superclass instance variables and these will simply be passed on to the superclass constructor as parameters. In other words, *super-parameters* will be some (or all) of *sub-parameters*.

Shown below are two constructors, one for the Publication class and one for Book. The Book constructor requires four parameters three of which are immediately passed on to the base constructor to initialize its instance variables.

```
// a constructor for the Publication class
public Publication(String title, double price, int copies)
{
    this.title = title;
    // etc
}
```

```
// a constructor for the Book class
public Book(String title, double price, int copies, String author)
    : base(title, price, copies)
{
    this.author = author;
}
```

Thus, in creating a Book object we first create a Publication object. The constructor for Book does this by calling the constructor for Publication and passing on the three parameters required by the Publication constructor...title, price and copies.

The constructor for Book then finishes initialising the book by setting its author.

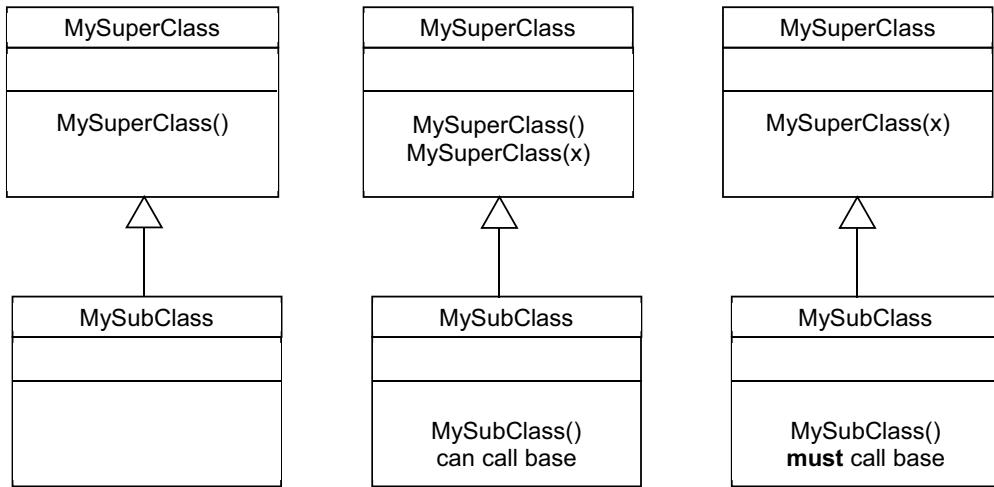
### 3.6 CONSTRUCTOR RULES

Rules exist that govern the invocation of a superconstructor.

If the superclass has a parameterless (or default) constructor this will be called automatically if no explicit call to base is made in the subclass constructor (though an explicit call is still better style for reasons of clarity).

However, if the superclass has no parameterless constructor but does have a parameterized one, this **must** be called explicitly using : base.

To illustrate this...



On the left above: it is legal (though bad practice) to have a subclass with no constructor because the superclass has a parameterless constructor.

In the centre: If the subclass constructor doesn't call the base constructor then the parameterless superclass constructor will be called.

On the right: Because the superclass has no parameterless constructor, the subclass **must** have a constructor, it **must** call the super constructor using the keyword `base` and it **must** pass on the required parameter. This is simply because a (super) class with only a parameterized constructor can only be initialized by providing the required parameter(s).

### 3.7 ACCESS CONTROL

To enforce encapsulation we normally make instance variables **private** and provide accessor/mutator methods as necessary (or in C# we use properties).

The `SellCopy()` method of `Publication` needs to alter the value of the variable 'copies'. It can do this even if 'copies' is a private variable. However Book and Magazine both need to alter 'copies'.

There are three ways we can do this in C#...

- 1) Make 'copies' 'protected' rather than 'private' – this makes it visible to subclasses.
- 2) Create accessor and mutator methods.
- 3) Create 'properties' in C#. These are effectively accessor methods but make the coding simpler.

Generally, we should keep variables private and create accessors/mutators methods rather than compromise encapsulation, though **protected** may be useful to allow subclasses to use methods (e.g. accessors and mutators) which we would not want generally available to other classes. In C# it is simpler, and hence normal practise, to create properties which effectively do the same job as accessor methods.

We will demonstrate the use of accessor methods and properties here.

Firstly, using accessor methods: In the superclass Publication we define ‘copies’ as a private variable but create two methods that can set and access the value ‘copies’. As these accessor methods are public or protected, they can be used within a subclass when access to ‘copies’ is required.

In the superclass Publication we would therefore have...

```
private int copies;
public int GetCopies ()
{
    return copies;
}

public void SetCopies(int newValue)
{
    copies = newValue;
}
```

These methods allow the superclass to control access to private instance variables.

As currently written, they don’t actually impose any restrictions, but suppose for example we wanted to make sure ‘copies’ is not set to a negative value?

- If ‘copies’ is **private**, we can put the validation (i.e. an if statement) within the SetCopies() method and know for sure that the rule can never be compromised.
- If ‘copies’ is partially exposed as **protected**, we would have to look at every occasion where a subclass method changed the instance variable and do the validation at each separate place.

We might even consider making these *methods* **protected** rather than **public** themselves so their use is restricted to subclasses only and other classes cannot interfere with the value of ‘copies’ at all.

Making use of these methods in the subclasses Book and Magazine we have...

```
// in Book
public void OrderCopies(int orderQty)
{
    SetCopies(GetCopies() + orderQty);
}
```

```
// and in Magazine
public void RecNewIssue(String newIssue)
{
    SetCopies(orderQty);
    currIssue = newIssue;
}
```

These statements are equivalent to

in Book

copies = number of copies already in stock + orderQty;

and in Magazine

```
copies = orderQty;
```

In C# ‘properties’ can be defined. These are really accessor methods and make this code simpler. Here the word ‘property’ is used with a meaning particular to C# and is not the same as a ‘property’ of a class.

In the code below two variables are defined, ‘price’ and ‘copies’ and a property is defined for each...the properties have the same name as the variable but start with an uppercase letter.

```
private double price;
public double Price
{
    get { return price; }
    set { price = value; }
}

private int copies;

public int Copies
{
    get { return copies; }
    set { copies = value; }
}
```

Thus when we refer to ‘copies’ we are referring to a private variable that cannot be accessed outside of the class. When we refer to ‘Copies’ with a capital C we are referring to the public property...as this is public we can use this to obtain or change the value of ‘copies’ from any class.

In the code above the properties have been defined such that they will both get and set the value of their respective variables...with no restrictions. We could change this code to impose restrictions or to remove either the ‘get’ or ‘set’ method.

By using ‘Copies = orderQty’ we are effectively invoking a setter method but we are doing this by using the property. This is effectively the same as using the setter method shown earlier to set a new value... ‘SetCopies(orderQty);’

Thus using the properties we could replace the methods shown above with those shown below.

```
// in Book
public void OrderCopies(int orderQty)
{
    Copies = Copies + orderQty;
}
// and in Magazine

public void RecNewIssue(String newIssue)
{
    Copies = orderQty;
    currIssue = newIssue;
}
```

Using ‘properties’ is normal in C#. In other Object Oriented languages the use of accessor methods does exactly the same job.

### 3.8 ABSTRACT CLASSES

The idea of a Publication which is not a Book, or a Magazine, or some other specific type of publication is meaningless (just like the idea of an adult who is neither of working age or retired). Thus while we are happy to create Book or Magazine objects we may want to prevent the creation of objects of type Publication.

If we want to deal with a new type of Publication which is genuinely neither Book nor Magazine – e.g. a Newspaper – it would naturally become another new subclass of Publication.

As Publication will never be instantiated (i.e. we will never create objects of this type) the only purpose of the class is to gather together the generalized features of its subclasses in one place for them to inherit.

We can enforce the fact that Publication is non-instantiable by declaring it ‘abstract’:

```
public abstract class Publication
{
    // etc.
```

### 3.9 OVERRIDING METHODS

A subclass inherits the methods of its superclass and must therefore always provide at least that set of methods, and often more. However, the implementation of a method can be changed in a subclass.

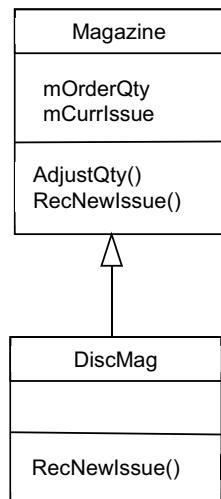
This is overriding the method.

To do this we write a new version of the method in the subclass which replaces the inherited one.

The new method should essentially perform the same functionality as the method that it is replacing. However, by changing the functionality we can improve the method and make its function more appropriate to a specific subclass.

For example, imagine a special category of magazine which has a disc attached to each copy – we can call this a DiscMag and we would create a subclass of Magazine to deal with DiscMags. When a new issue of a DiscMag arrives not only do we want to update the current stock but we want to check that the discs are correctly attached. Therefore we want some additional functionality in the RecNewIssue() method to remind us to do this. We achieve this by redefining RecNewIssue() in the DiscMag subclass.

Note: when a new issue of Magazine arrives, as these don't have a disc we want to invoke the original RecNewIssue() method defined in the Magazine class.



- The definition of **RecNewIssue()** in **DiscMag** overrides the inherited one.
- Magazine is not affected – it retains its original definition of **RecNewIssue()**
- By showing **RecNewIssue()** in **DiscMag** we are stating that the inherited method is being overridden (i.e. replaced). Remember, we do not show inherited methods in subclasses.

When we call the **RecNewIssue()** method on a **DiscMag** object the CLR engine automatically selects the new overriding version – the caller doesn't need to specify this, or even know that it is an overriden method at all. When we call the **RecNewIssue()** method on a **Magazine** it is the method in the superclass that is invoked.

## Implementing DiscMag

To implement **DiscMag** we must create a subclass of **Magazine**. No additional instance variables or methods are required (though it is possible to create some if there was a need). The constructor for **DiscMag** can simply pass all of its parameters directly on to the superclass and a version of **RecNewIssue()** is defined in **DiscMag** to override the one inherited from **Magazine** (see code below).

```
public class DiscMag : Magazine
{
    public DiscMag(String title, double price, int copies, int
                   orderQty, String currIssue)
        : base(title, price, copies, orderQty, currIssue)
    {
    }

    public override void RecNewIssue(String newIssue)
    {
        base.RecNewIssue(newIssue);
        Console.WriteLine("Check discs are attached");
    }
}
```

Note: The use of **base.RecNewIssue()** to call a method of the superclass, thus re-using the existing functionality as part of the replacement, just as we do with constructors. Additionally, it then displays the required message for the user.

One final change is required. Before a method can be overridden permission for this must be granted by the author of the superclass. Using the keyword **virtual** when defining methods basically grants permission for them to be overridden.

Thus for the code above to work the RecNewIssue() method in the magazine must be made virtual. See below...

```
// in Magazine
public virtual void RecNewIssue(String newIssue)
{
    Copies = orderQty;
    currIssue = newIssue;
}
```

This mechanism gives us the ability to allow, or prevent, the methods we create from being overridden in subclasses.

## Operations

Formally, ‘RecNewIssue()’ is an operation. This one operation is implemented by two different methods, one in Magazine and the overriding one in DiscMag. This distinction is an important part of ‘polymorphism’ which we will meet in Chapter 4.

### 3.10 THE ‘OBJECT’ CLASS

In C# all classes are (direct or indirect) subclasses of a class called ‘Object’. Object is the ‘root’ of the inheritance hierarchy in C#. Thus, this class exists in every C# program ever created.

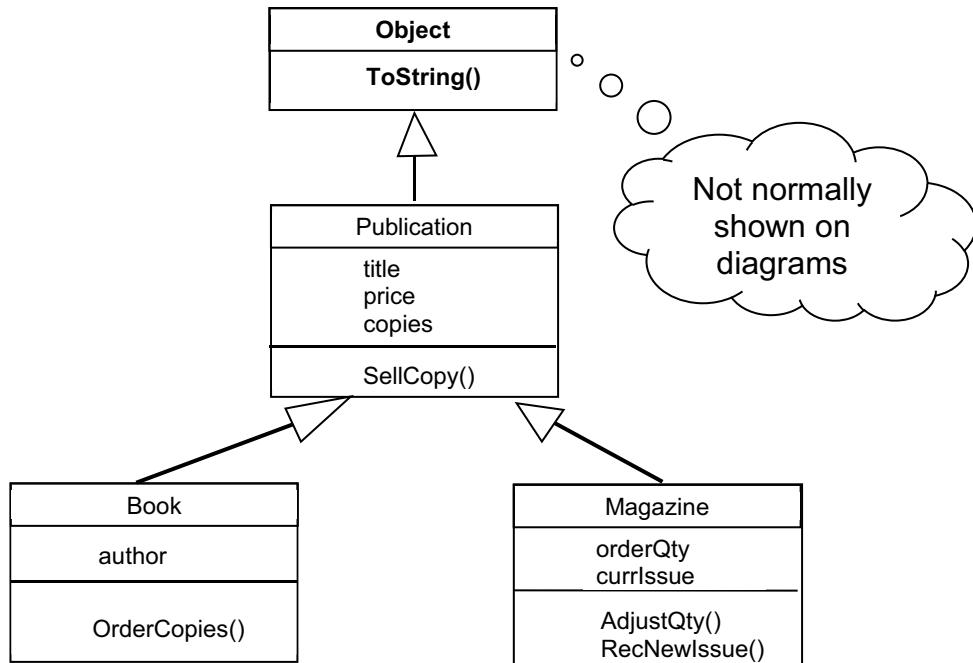
If a class is not declared to inherit from another then it implicitly inherits from Object.

‘Object’ defines no instance variables but several methods. Generally these methods will be overridden by new classes to make them useful. An example is the **ToString()** method.

Thus, when we define our own classes, by default they are direct subclasses of Object.

If our classes are organised into a hierarchy then the topmost superclass in the hierarchy is a direct subclass of object, and all others are indirect subclasses.

Thus directly, or indirectly, all classes created in C# inherit the `ToString()` method defined in the `Object` class.



### 3.11 OVERRIDING TOSTRING() DEFINED IN 'OBJECT'

The `Object` class defines a `ToString()` method, one of several useful methods.

`ToString()` has the signature

```
public String ToString()
```

The purpose of the `ToString()` method is to return a string value that represents the current object. The version of `ToString()` defined by `Object` produces an output like: "Book". This is the class name from which an object is instantiated. However to be generally useful we need to override this to give a more meaningful string.

### In Publication

```
public override String ToString()
{
    return title;
}
```

### In Book

```
public override String ToString()
{
    return base.ToString() + " by " + author;
}
```

### In Magazine

```
public override String ToString()
{
    return base.ToString() + " (" + currIssue + ")";
}
```

In the code above `ToString()` originally defined in `Object` has been completely replaced, ie. overridden, so that `Publication.ToString()` returns the title of the publication.

The `ToString()` method has been overridden again in `Book` such that `Book.ToString()` returns title (via the base classes' `ToString()` method) and author i.e. this overridden version uses the version defined in `Publication`. Thus if `Publication.ToString()` was rewritten to return the title and ISBN number then `Book.ToString()` would automatically return the title, ISBN number and author.

`Magazine.ToString()` returns title (via the base class `ToString()` method) and issue.

We will do not need to further override the `ToString()` method in `DiscMag` because the version it inherits from `Magazine` is OK.

We could get the `ToString()` method to provide an even more complete description of an object by using more of the data (i.e. more, or even all, of the instance variable values). The design judgement here is that these will be the most generally useful printable representation of objects of these classes. In this case title and author (for a book) or title and current issue (for a magazine) serve well to uniquely identify a particular publication.

Perhaps for a `Newspaper`, we would override `ToString()` to return the title of the newspaper and the date it was printed.

### 3.12 SUMMARY

Inheritance allows us to factor out common attributes and behaviour. We model the commonalities in a superclass (sometimes called a base class in C#).

Subclasses are used to model specialized attributes and behaviour.

Code in a superclass is inherited by all subclasses. If we amend or improve code for a superclass it impacts on all subclasses. This reduces the code we need to write in our programs.

Special rules apply to constructors for subclasses.

A superclass can be declared **abstract** to prevent it being instantiated (i.e. objects created).

We can ‘override’ inherited methods so a subclass implements an operation differently from its superclass.

In C#, all classes descend from the base class ‘Object’

‘Object’ defines some universal operations which can usefully be overridden in our own classes.

# 4 OBJECT ROLES AND THE IMPORTANCE OF POLYMORPHISM

## Introduction

Through the use of worked examples this chapter will explain the concept of polymorphism and the impact this has on Object Oriented software design. This is such an important and useful concept that we will explore this further in later chapters.

## Objectives

By the end of this chapter you will be able to...

- Understand how polymorphism allows us to handle related classes in a generalized way
- Employ polymorphism in C# programs
- Understand the implications of polymorphism with overridden methods
- Define interfaces to extend polymorphism beyond inheritance hierarchies
- Appreciate the scope of extensibility which polymorphism provides

This chapter consists of eight sections:

- 1) Class Types
- 2) Substitutability
- 3) Polymorphism
- 4) Extensibility
- 5) Interfaces
- 6) Extensibility Again
- 7) Distinguishing Subclasses
- 8) Summary

## 4.1 CLASS TYPES

Within hierarchical classification of animals

Pinky is a pig (species sus scrofa)

Pinky is (also, more generally) a mammal

Pinky is (also, even more generally) an animal

We can specify the type of thing an organism is at different levels of detail:

higher level = less specific

lower level = more specific

If you were asked to give someone a pig you could give them Pinky or any other pig.

If you were asked to give someone a mammal you could give them Pinky, any other pig or any other mammal (e.g. any lion, or any mouse, or any cat).

If you were asked to give someone an animal you could give them Pinky, any other pig, any other mammal, or any other animal (bird, fish, insect etc.).

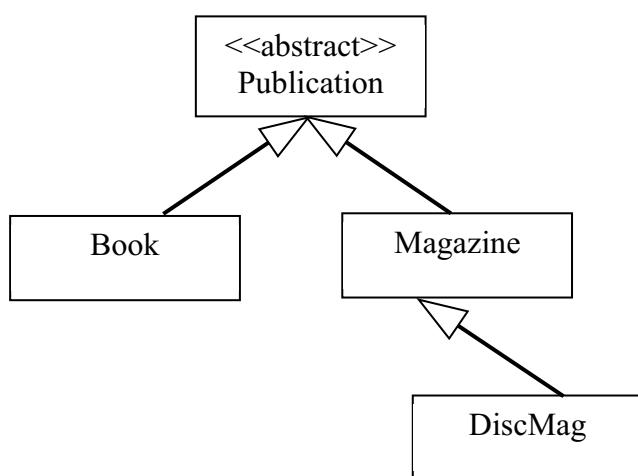
In other words Pinky **is a** pig, a pig **is a** mammal and a mammal **is an** animal.

Thus an object in a classification hierarchy has an '**is a**' relationship with every class from which it is descended and in this hierarchy each classification represents a type of animal.

This is true in Object Oriented programs as well. Every time we define a class we create a new '**type**'. Types determine compatibility between variables, parameters etc.

A subclass type is a subtype of the superclass type and we can substitute a subtype wherever a '**supertype**' is expected. Following this we can substitute objects of a subtype whenever objects of a supertype are required (as in the example above).

The class diagram below shows a hierarchical relationship of types of object – or classes.



- If we want a 'DiscMag', it must be an object of class DiscMag.
- If we want a 'Magazine', it could be an object of class Magazine or an object of class DiscMag (as this is a type of magazine).
- If we want a 'Publication' it could be a Book, Magazine or DiscMag.

In other words we can 'substitute' an object of any subclass where an object of a superclass is required. This is NOT true in reverse!

### Activity 1

In C# the keyword 'new' invokes the constructor of a class (i.e. creates an object of that class). Look at the class diagram above and decide which of the following lines of code would be legal in a C# program where these classes had been implemented:

```
Publication p = new Book(...);  
  
Publication p = new DiscMag(...);  
  
Magazine m = new DiscMag(...);  
  
DiscMag dm = new Magazine(...);  
  
Publication p = new Publication(...);
```

### Feedback 1

Publication p = new Book(...);

Here we are defining a variable p of the general type of 'Publication'. We are then invoking the constructor for the Book class and assigning the result to 'p'. This is OK because Book is a subclass of Publication (i.e. a Book **is a** Publication).

Publication p = new DiscMag(...);

This is OK because DiscMag is a subclass of Magazine which is a subclass of Publication. DiscMag is an indirect subclass of Publication (i.e. a DiscMag **is a** Publication).

Magazine m = new DiscMag(...);

This is OK because DiscMag is a subclass of Magazine

DiscMag dm = new Magazine(...);

This is illegal because Magazine is a **superclass** of DiscMag. Some Magazines are DiscMags but some are not so if a DiscMag is required we cannot hand over any Magazine.

Publication p = new Publication(...);

This is illegal for a different reason. Publication is an abstract class and therefore cannot be instantiated.

## 4.2 SUBSTITUTABILITY

When designing class/type hierarchies, the type mechanism allows us to place a subclass object where a superclass is specified. However this has implications for the design of subclasses – we need to make sure they are genuinely substitutable for the superclass. If a subclass object is substitutable, then clearly it must implement all of the methods of the superclass – this is easy to guarantee as all of the methods defined in the superclass are inherited by the subclass. Thus, while a subclass may have additional methods it must at least have all of the methods defined in the superclass and should therefore be substitutable. However, what happens if a method is overridden in the subclass?

When overriding methods we must ensure that they are still substitutable for the method being replaced. Therefore, when overriding methods, while it is perfectly acceptable to tailor the method to the needs of the subclass, a method should not be overridden with functionality which performs an inherently different operation.

For example, `RecNewIssue()` in `DiscMag` overrides `RecNewIssue()` from `Magazine` but does the same basic job (“fulfils the contract”) as the inherited version with respect to updating the number of copies and the current issue. While it extends that functionality in a way

specifically relevant to DiscMags (by displaying a reminder to check the cover discs), essentially these two methods perform the same operation.

What do we know about a ‘Publication’?

Answer: It’s an object which supports (at least) the operations:

void SellCopy()

String ToString()

and it has properties that allow us to

set the price,

get the number of copies

set the number of copies.

Inheritance guarantees that objects of any subclass of Publications provides at least these.

Note: a subclass can never remove an operation inherited from its superclass(es) – this would break the guarantee. Because subclasses **extend** the capabilities of their superclass, the superclass functionality can be assumed.

The ToString() method returns a string description of an object and it is quite likely that we would choose to override the ToString() method (initially defined within ‘Object’) so that within Publication it would return the title of that publication – this would be a suitable description of a Publication.

We could override it again within Magazine so that the string returned provides a better description of Magazines, perhaps the title and the issue details. However we should not override the ToString() method in order to return the price of a magazine – this would be changing the functionality of the method so that the method performs an inherently different function.

Overriding ToString() so that it performs a different function would break the substitutability principle. Essentially we know that Publications should be able to describe themselves using ToString()...thus a Magazine, which is a publication, should be able to describe itself. If a magazine can’t describe itself then it can’t do everything a publication can do. We must be able to substitute a magazine for a publication therefore when we override methods they must perform the same inherent function. While overridden methods cannot change the inherent functionality they can be tailored to the needs of the subclass i.e. it is perfectly fine if the description provided by the overridden ToString() method is a better description of a Magazine.

### 4.3 POLYMORPHISM

The mechanisms of inheritance and method overriding allows polymorphism to work and this is essential in developing flexible applications.

Because an instance of a subclass is an instance of its superclass we can handle subclass objects as if they were superclass objects. Furthermore because a superclass guarantees certain operations in its subclasses we can invoke those operations without caring which subclass the actual object is an instance of.

This characteristic is termed ‘polymorphism’, originally meaning ‘having multiple shapes’.

Thus, a Publication comes in various shapes...it could be a Book, Magazine or DiscMag. We can invoke the SellCopy() method on any of these Publications irrespective of their specific details.

‘Polymorphism’ is a fancy name for a common idea. Someone who knows how to drive can get into and drive most cars because they have a set of shared key characteristics – steering wheel, gear stick, pedals for clutch, brake and accelerator etc – which the driver knows how to use. There will be lots of differences between any two cars, but you can think of them as subclasses of a superclass which defines these crucial shared ‘operations’.

If ‘p’ ‘is a’ Publication, it might be a Book or a Magazine or a DiscMag.

Whichever it is we know that it has a SellCopy() method.

So we can invoke p.SellCopy() without worrying about what exactly ‘p’ is.

This can make life a lot simpler when we are manipulating objects within an inheritance hierarchy. We can create new types of Publication e.g. a Newspaper and invoke p.SellCopy() on a Newspaper without have to create any functionality within the new class – all the functionality required is already defined in Publication.

Polymorphism makes it very easy to extend the functionality of our programs as we will see now. We will see this again in Chapter 5 and in the case study at the end of this book (Chapter 14).

## 4.4 EXTENSIBILITY

Huge sums of money are spent annually creating new computer programs but over the years even more is spent changing and adapting those programs to meet the changing needs of an organisation. Thus, as professional software engineers we have a duty to facilitate this and help to make those programs easier to maintain and adapt. Of course the application of good programming standards, commenting and layout etc, have a part to play here but polymorphism can also help as it allows the creation of programs that are easily extended.

### CashTill class

Imagine we want to develop a class CashTill which processes a sequence of items being sold. Without polymorphism, we would need separate methods for each type of item:

SellBook (Book pBook)  
SellMagazine (Magazine pMagazine)  
SellDiscMag (DiscMag pDiscMag)

With polymorphism we need only  
SellItem (Publication pPub)

Every subclass is ‘type-compatible’ with its superclass. Therefore, any subclass object can be passed as a Publication parameter.

This also has important implications for extensibility of systems. We can later introduce further subclasses of Publication and these will also be acceptable by the SellItem() method of a CashTill object (even though these subtypes were unknown when the CashTill was implemented).

### **Publications sell themselves!**

Without polymorphism we would need to check for each item ‘p’ so we were calling the right method to sell a copy of that subtype

```
if ‘p’ is a Book call SellCopy() method for Book  
else if ‘p’ is a Magazine call SellCopy() method for Magazine  
else if ‘p’ is a DiscMag call SellCopy() method for DiscMag
```

Instead we trust C# to look at the object ‘p’ at run time, to determine its ‘type’ and its own method for selling itself. Thus we can call:

```
p.SellCopy()
```

and if the object is a Book it will invoke the SellCopy() method for a Book. If ‘p’ is a Magazine, again at runtime C# will determine this and invoke the SellCopy() method for a Magazine.

Polymorphism often allows us to avoid conditional ‘if’ statements – instead the ‘decision’ is made implicitly according to which type of subclass object is actually present.

### **Implementing CashTill**

The code below shows how CashTill can be implemented to make use of Polymorphism.

```
public class CashTill
{
    private double runningTotal;
    public CashTill()
    {
        runningTotal = 0;
    }

    public void SellItem(Publication pPub)
    {
        runningTotal = runningTotal + pPub.Price;
        pPub.SellCopy();
        Console.WriteLine("Sold " + pPub + " @ " +
                           pPub.Price + "\nSubtotal = " +
                           runningTotal);
    }

    public void ShowTotal()
    {
        Console.WriteLine("GRAND TOTAL: " + runningTotal);
    }
}
```

The CashTill has one instance variable – a double to hold the running total of the transaction. The constructor simply initializes this to zero.

The SellItem() method is the key feature of CashTill. It takes a Publication parameter, which may be a Book, Magazine or DiscMag. First, the price of the publication is added to the running total using the Price property defined in the class Publication. Then, the SellCopy() operation is invoked on the publication. Finally, a message is constructed and displayed to the user, e.g.

```
Sold Windowcleaning Weekly (Sept 2017) @ 2.75
Subtotal = 2.75
```

Note: when **pPub** appears in conjunction with the string concatenation operator ‘+’. This implicitly invokes the ToString() method for the subclass of this object, and remember that ToString() is different for books and magazines.

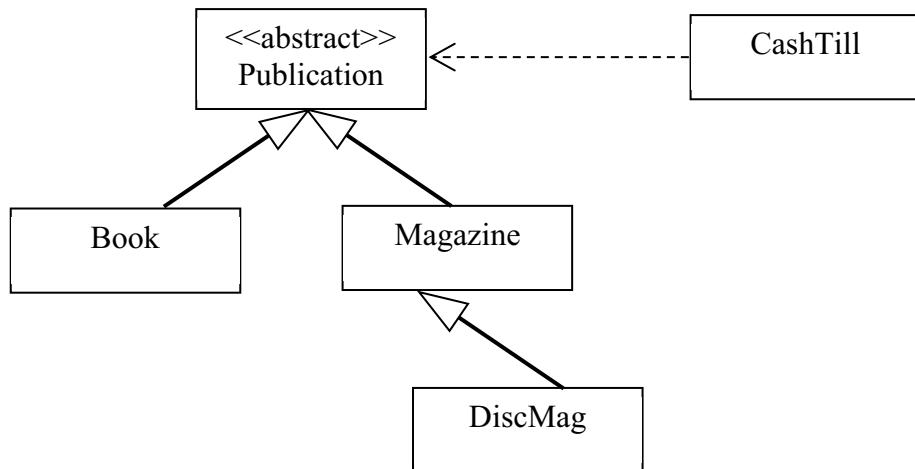
The correct ToString() operation is automatically invoked by C# to return the appropriate string description for the specific object sold!

Thus, if a book is sold the output would contain the title and author e.g.

Sold Hitch Hikers Guide to the Galaxy by D Adams @ 7.50  
Subtotal = 7.50

Thus our cash till can sell any publication of any shape, i.e. any type Book, Magazine or DiscMag, without worrying about any specific features of these classes. This is polymorphism in action!

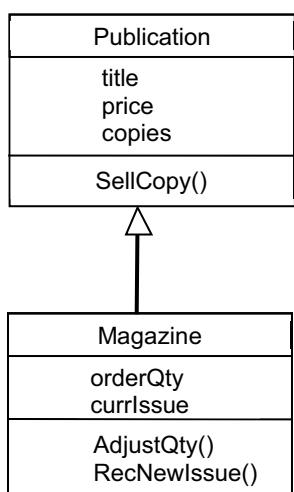
We can show CashTill on a class diagram as below:



Note: CashTill has a dependency on Publication because the SellItem() method is passed a parameter of type Publication. What is actually passed will of course be an object of one of the concrete types descended from Publication.

### Activity 2

Look at the diagram below and, assuming Publication is not an abstract type, decide which of the pairs of operations shown are legal.



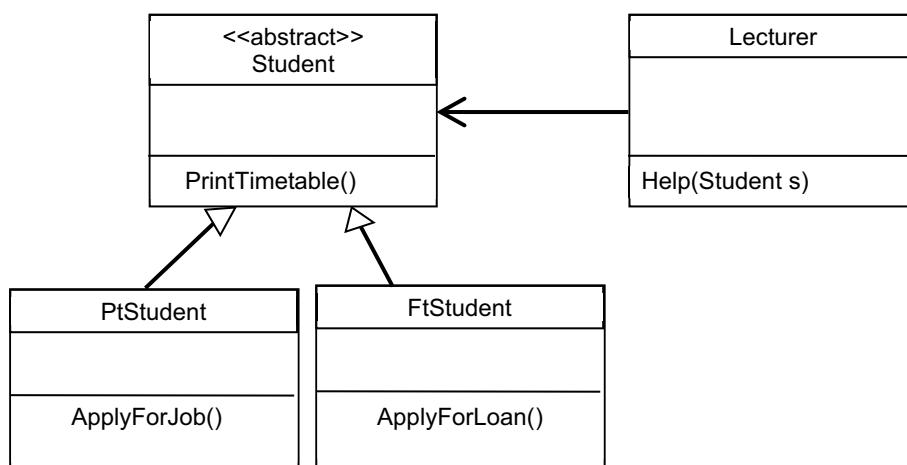
- a) Publication p = new Publication(...);  
p.SellCopy();
- b) Publication p = new Publication(...);  
p.RecNewIssue();
- c) Publication p = new Magazine(...);  
p.SellCopy();
- d) Publication p = new Magazine(...);  
p.RecNewIssue();
- e) Magazine m = new Magazine(...);  
m.RecNewIssue();

### Feedback 2

- a) Legal – you can invoke SellCopy() on a publication
- b) Illegal – the RecNewIssue() method does not exist in publications
- c) Legal – Magazine is a type of Publication and therefore you can assign an object of type Magazine to a variable of type Publication (you can always substitute subtypes where a supertype is requested). Also you can invoke SellCopy() on a publication. The publication happens to be a magazine but this is irrelevant. As far as the compiler knows in this instance 'p' is just a publication.
- d) Illegal – While we can invoke RecNewIssue on a magazine the compiler does not know that 'p' is a magazine...only that it is a publication.
- e) Legal – m is a magazine and we can invoke this method on magazines.

### Activity 3

Look at the diagram below and, noting that Student is an abstract class, decide which of the following code segments are valid...



Note: FtStudent is short for Full Time Student and PtStudent is short for Part Time Student.

- a) Student s = new Student();
 Lecturer l = new Lecturer();
 l.Help(s);
- b) Student s = new FtStudent();
 Lecturer l = new Lecturer();
 l.Help(s);

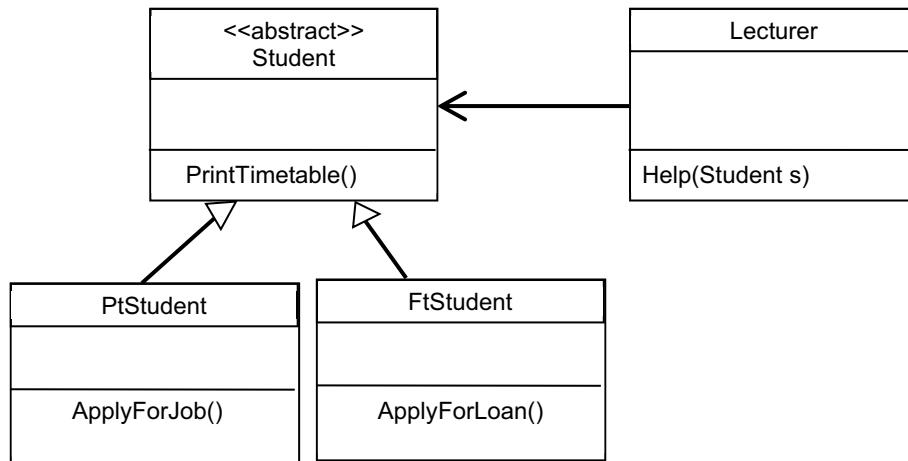
**Feedback 3**

- a) This is not valid as class Student is abstract and cannot be instantiated
- b) This is valid. FtStudent is a type of Student and can be assigned to a variable of type Student. This can then be passed as a parameter to I.Help()

### Activity 4

Taking the same diagram and having invoked the code directly below decide which of the following lines (a or b) would be valid inside the method Help(Student s)...

```
Student s = new FtStudent();
Lecturer l = new Lecturer();
l.help(s);
```



- a) s.PrintTimetable();
- b) s.ApplyForLoan();

### Feedback 4

- a) This is valid – we can invoke this method on a Student object and also on an FtStudent object (as the method is inherited).
- b) Not Valid! While we can invoke this method on a FtStudent object, and we are passing an FtStudent object as a parameter to the Help() method, the Help() method cannot know that the object passed will be a FtStudent (it could be any object of type Student). Therefore there is no guarantee that the object passed will support this method. Hence this line of code would generate a compiler error.

As the class Lecturer only has a relationship with the Student class, not PtStudent or FtStudent, we can only invoke the methods defined in the Student superclass...even if we provided a FtStudent object it can only treat this object as a general Student. It cannot treat it specifically like a FtStudent and cannot use methods only defined in the subclass.

This is an important concept that we will return to as we learn more about the design of polymorphic systems.

## 4.5 INTERFACES

There are two aspects to inheritance:

- The subclass inherits the interface (i.e. access to public members) of its superclass – this makes polymorphism possible.
- The subclass inherits the implementation of its superclass (i.e. instance variables and method implementations) – this saves us copying the superclass details in the subclass definition.

In C#, the use of inheritance (via the ‘:’ symbol) automatically applies both these aspects.

A **subclass** is a **subtype**. Its interface must include all of the interface of its superclass, though the implementation of this can be different (through overriding) and the interface of the subclass may be more extensive with additional features being added.

However, sometimes we may want two classes to share a common interface without putting them in an inheritance hierarchy. This might be because:

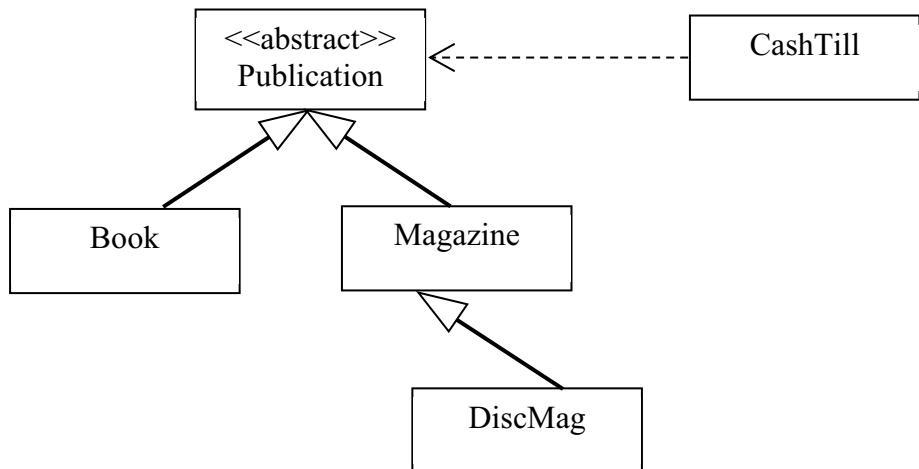
- They aren’t really related by a true ‘is a’ relationship
- We want a class to have interfaces shared with more than one would-be superclass, but C# does not allow such ‘multiple inheritance’
- We want to create a ‘plug and socket’ arrangement between software components, some of which might not even be created at the current time.

This is like making sure that two cars have controls that work in exactly the same way, but leaving it to different engineers to design engines which ‘implement’ the functionality of the car, possibly in quite different ways.

Be careful of the term ‘interface’ – in C# programming it has at least three meanings:

- 1) The public members of a class – the meaning used above
- 2) The “user interface” of a program, often a “Graphical User Interface” – an essentially unrelated meaning
- 3) A specific C# construct which we are about to meet

Recall how the subclasses of Publication provide additional and revised behaviour while retaining the set of operations – i.e. the interface – which it defined.



This is why the CashTill class can deal with a 'Publication' without worrying of which specific subclass it is an instance (remember that Publication is an abstract class – a 'Publication' is in reality **always** a subclass).

## Tickets

Now consider the possibility that in addition to books and magazines, we now want to sell tickets, e.g. for entertainment events, public transport, etc. These are not like Publications because:

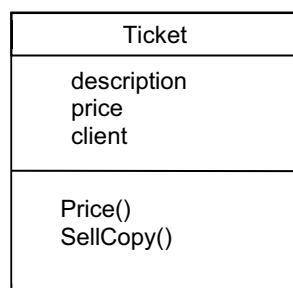
- We don't have a finite 'stock' but print them on demand at the till
- Tickets consist simply of a description, price and client (for whom they are being sold)
- These sales are really a service rather than a product

Tickets seem to have little in common with Publications – they share a small **interface** associated with being sold, but even for this the underlying **implementation** will be different because we will not be decrementing them from a current stock.

For these reasons Ticket and Publication do not seem closely related and thus we do not want to put them in an inheritance hierarchy. However we do want to make them both acceptable to CashTill as things to sell and we need a mechanism for doing this.

Without putting them in an inheritance hierarchy what we want is a more general way of saying “things of this class can be sold” which can be applied to whatever (present and future) classes we wish, thus making the system readily extensible to Tickets and anything else.

While the Ticket class is sufficiently different from a Publication that we don't want to put it in an inheritance hierarchy it does have some similarities – namely it has a SellCopy() method and a property to obtain the price – both of these are needed by a CashTill.



However, the SellCopy() method is very different form the SellCopy() method defined in Publication. To sell a publication the stock had to be reduced by 1 – with a ticket we just need to print it.

```
public void SellCopy()
{
    Console.WriteLine("*****");
    Console.WriteLine("          TICKET VOUCHER      ");
    Console.WriteLine(this.ToString());
    Console.WriteLine("*****");
    Console.WriteLine();
}
```

As the `SellCopy()` method is so different we do not want to inherit its implementation details. Therefore we don't feel that `Ticket` belongs in an inheritance hierarchy with `Publications`. Despite this we do want to be able to check tickets through the till as we can with publications.

Just like publications, tickets provide the operations which `CashTill` needs:

```
SellCopy()  
Price()
```

Consequently the `CashTill` can sell a `Ticket`. In fact, `CashTill` can sell anything that has these methods, not just `Publications`. To enable this to happen we will define this set of operations as an 'Interface' called `ISellableItem` (where 'I' is being used to indicate this refers to an interface not a class).

```
public interface ISellableItem
{
    double Price
    {
        get;
    }
    void SellCopy();
}
```

Note: The interface defines purely the signatures of operations without their implementations.  
Also: while this interface defines the need for a `get` method for 'price' a `set` method is not required and therefore not defined in the interface.

All the methods are implicitly public even if this is not stated, and there can be no instance variables, constructors or code to implement the methods.

In other words, an interface defines the **availability** of specified operations without saying anything about their implementation. That is left to classes which **implement** the interface.

An interface is a sort of contract. The **ISellableItem** interface says “I undertake to provide, at least, methods with these signatures:

```
public void SellCopy ();  
public double Price ();
```

though I might include other things as well”

Where more than one class implements an interface it provides a guaranteed area of commonality which polymorphism can exploit.

Think of a car and a driving game in an arcade. They certainly are not related by any “is a” relationship – they are entirely different kinds of things, one a vehicle, the other an arcade game. But they both implement what we could call a “SteeringWheel interface” which we can use in exactly the same way, even though the implementation (mechanical linkage in the car, video electronics in the game) is very different.

We now need to state that both Publication (and all its subclasses) and Ticket offer the operations defined by this interface:

```
public abstract class Publication : ISellableItem
{
    [...class details...]
}
```

```
public class Ticket : ISellableItem
{
    [...class details...]
}
```

In C#, the same symbol ‘:’ is used when we define a subclass that extends a super class or when we create a class that implements an interface.

### Contrast **implementing an interface** with **extending a superclass**.

- When we extend a superclass, the subclass inherits of both interface and implementation from the superclass.
- When we implement an interface, we give a guarantee that the operations specified by an interface will be provided – this is enough to allow polymorphic handling of all classes which implement a given interface.

### The Polymorphic CashTill

The CashTill class already employs polymorphism: the SellItem() method accepts a parameter of type Publication which allows any of its subclasses to be passed:

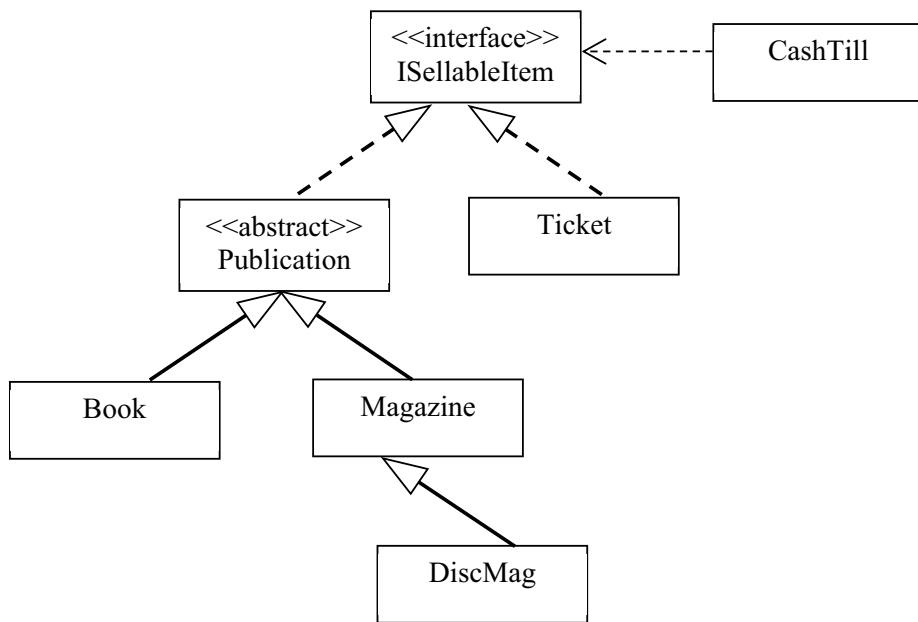
```
public void SellItem (Publication pPub)
```

We now want to broaden this further by accepting anything which implements the ISellableItem interface:

```
public void SellItem(ISellableItem pSI)
```

When the type of variable or parameter is defined as an interface, it works just like a superclass type. Any class which implements the interface is acceptable for assignment to the variable/parameter because the interface is a **type** and all classes implementing it are subtypes of that type.

This is now shown below....



CashTill is no longer directly dependent on class Publication – instead it is dependent on the interface ISellableItem.

Note: We can start the interface name with an 'I' to indicate this is an interface not a class.

The relationships from Publication and Ticket to ISellableItem are like inheritance arrows except that the lines are **dotted** – this shows that each class **implements** the interface.

A class in C# may only inherit from one superclass but can implement as many interfaces as desired. The format for this is:

```
class MyClass : MySuperClass, IMyInterface, IMySecondInterface
```

## 4.6 EXTENSIBILITY AGAIN

Polymorphism allows objects to be handled without regard for their precise class. This can assist in making systems extendable without compromising the encapsulation of the existing design.

For example, we could create new classes for more products or services and so long as they implement the ISellableItem interface the CashTill will be able to process them **without a single change to its code!**

An example could be ‘Sweets’. We could define a class Sweets to represent sweets in a jar. We can define the price of the sweets depending upon the weight and then sell the sweets by subtracting this weight from our total stock. This is not like selling a Publication, where we always subtract 1 from the stock. Nor is this like selling tickets, where we just print them.

However, if we create a class ‘Sweets’ that implements the ISellableItem interface our enhanced polymorphic cash till can sell them because it can sell any sellable item.

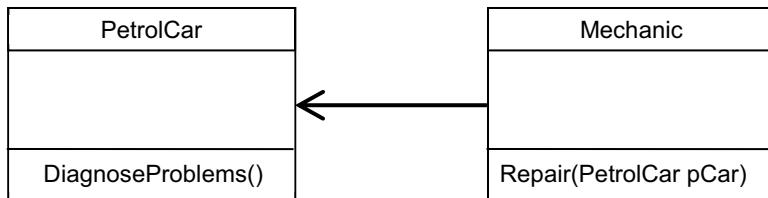
In this case, without polymorphism we would need to add an additional ‘sale’ method to CashTill to handle Tickets, Sweets and further new methods for every new type of product

to be sold. By defining the ISellableItem interface can introduce additional products without affecting CashTill at all. Polymorphism makes it easy to extend our programs and this is very important as it saves effort, time and money.

Interfaces allow software components to plug together more flexibly and extensively, just as many other kinds of plugs and sockets enable audio, video, power and data connections in the everyday world. Think about the number of different electrical appliances which can be plugged into a standard power socket – and imagine how inconvenient it would be if instead, you had to call out an electrician to wire up each new one you bought!

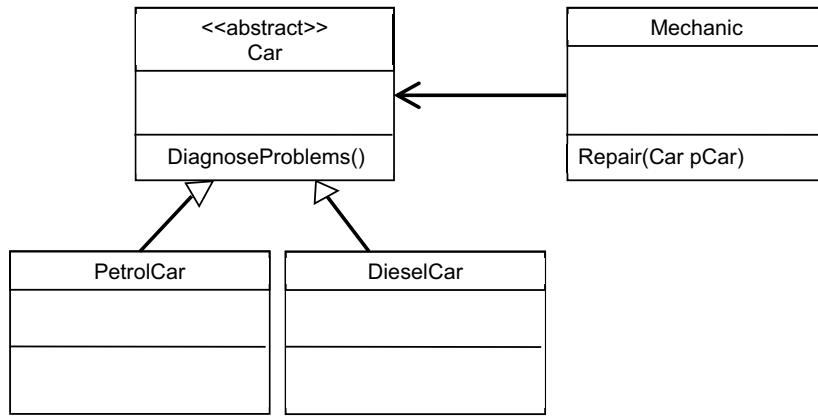
### Activity 5

Adapt the following diagram by adding a class for Diesel cars in such a way that it can be used to illustrate polymorphism.



### Feedback 5

This is one solution to this exercise...



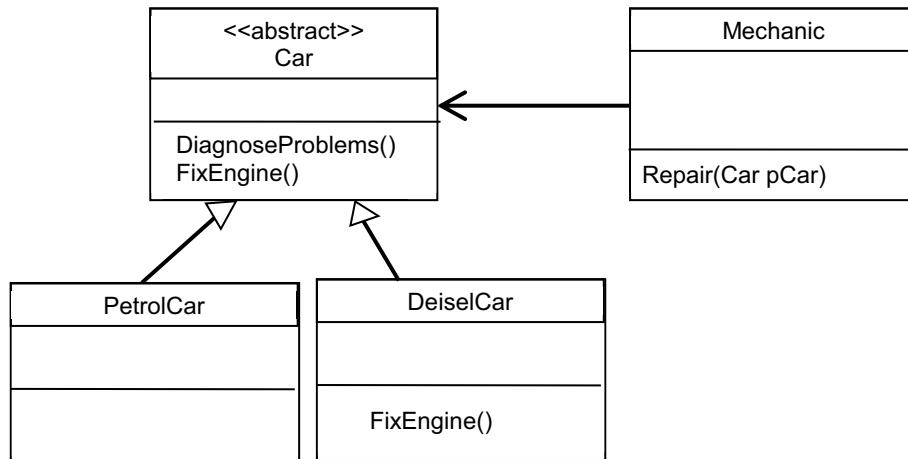
Here, the Mechanic class is directly interacting with the Car class. In doing so it can interact with any subtype of Car (e.g. Petrol, Diesel) or any other type of Car developed in the future (e.g. an electric car). These are all different (different shapes – at least different internally) and yet Mechanic can still interact with them as they are all Cars. This is polymorphic.

If an ElectricCar class was added, Mechanic would still be able to work with it without making any changes to the Mechanic class.

### Activity 6

Assume Car has a FixEngine() method that is overridden in DieselCar but not overridden in PetrolCar (as shown on the diagram below).

Look at this diagram and answer the following questions...



- Would the following line of code be valid inside the **Repair()** method ?  
`pCar.FixEngine();`
- If a **DieselCar** object was passed to the **repair()** method which actual method would be invoked by `pCar.FixEngine();` ?

### Feedback 6

- Yes! We can apply the method **FixEngine()** to any **Car** object as it is defined in the class **Car**.
- This would invoke the overridden method. The method must be defined in the class **Car** else the compiler will complain. However, at run-time the identity the actual object will be checked. As the actual object is a subtype 'DieselCar' the actual method invoked will be the overridden method. Clever stuff given that the **Repair()** method is unaware of which type of car is actually passed!

## 4.7 DISTINGUISHING SUBCLASSES

What if we have an object handled polymorphically but we want to check which subtype it **actually** is? The **is** operator can do this:

*object is class*

This test is **true** if the object is of the specified class (or a subclass), **false** otherwise.

Note: **myDiscMag is Magazine** would be **true** because a DiscMag is a Magazine. **myDiscMag is Publication** would also be **true** because a DiscMag is a Magazine and a Magazine is a Publication.

‘**is**’ can also be used with an interface name on the right, in which case, it tests whether the class implements the interface.

Strictly **is** tests whether the item on the left is of the type (or a subtype of) the type specified on the right.

Using this, we could amend the CashTill class so that it displays a specific message depending upon the object sold.

```
public void SaleType (ISellableItem pSI)
{
    if (pSI is Publication)
    {
        Console.WriteLine("This is a Publication");
    }
    else if (pSI is Ticket)
    {
        Console.WriteLine ("This is a Ticket");
    }
    else
    {
        Console.WriteLine ("This is a an unknown sale type");
    }
}
```

**pSI is Publication** will be true if pSI is any subclass of Publication (i.e. a Book, Magazine or DiscMag). If we wished to we could equally test for a more specific subtype, e.g. **pSI is Book**.

Notice that once we compromise polymorphism by checking for subtypes we also compromise the extendability of the system – new classes (e.g. Sweets) implementing the ISellableItem interface will also require new clauses adding to this if statement, so the change ripples through the system with the consequence that it becomes more costly and error-prone to maintain.

Instead of doing this we should try to package different behaviours into the subclasses themselves, e.g. we could define a **DescribeSelf()** method in the interface ISellableItem. This would then need to be programmed in each class that implements the ISellableItem interface i.e. we could define DescribeSelf() so that a publication returns the string “This is a publication” and a Ticket it returns “This is a ticket”.

Thus, each subtype would return an appropriate message that can be displayed by the cash till. The entire ‘if’ statement in CashTill can then be replaced with one simple line of code as shown below:

```
public void SaleType (ISellableItem pSI)
{
    Console.WriteLine(pSI.DescribeSelf());
}
```

Now when we add new classes, all we would need to do is implement the `DescribeSelf()` method in the new class **and we would not need to change the CashTill class at all**. With a little thought, we can make our system polymorphic and this makes it easier to extend and change.

## 4.8 SUMMARY

Polymorphism allows us to refer to objects according to a superclass, rather than their actual class.

Polymorphism makes it easy to extend our programs by adding additional classes without needing to change other classes.

We can manipulate objects by invoking operations defined for the superclass without worrying about which subclass is involved in any specific case.

C# ensures that the appropriate method for the actual class of the object is invoked at run-time.

Sometimes we want to employ polymorphism without all the classes concerned having to be in an inheritance hierarchy. An interface allows us to do this by specifying common operations. When doing this, there is no inherited implementation – **each class must implement all of the operations defined by the Interface**.

Any number of classes can implement a particular interface.

A class in C# may only inherit from one superclass but can implement multiple interfaces.

# 5 USING POLYMORPHISM EFFECTIVELY

## Introduction

Polymorphism is very powerful but only works if systems are correctly designed. This chapter therefore aims to give you the skills to design an Object Oriented system so that polymorphism works effectively.

## Objectives

By the end of this chapter you will be able to...

- Understand how to apply polymorphism to design effective Object Oriented software.

This chapter consists of nine sections:

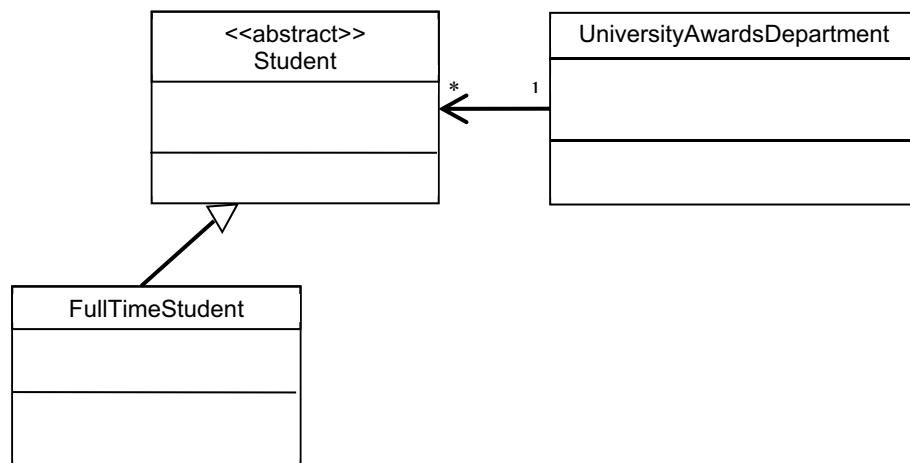
- 1) Responsibility
- 2) What characterises a Polymorphic system?
- 3) Polymorphic Collections
- 4) Non-Polymorphic Components
- 5) A More Realistic (Complex) Polymorphic System (Part 1)
- 6) A More Realistic (Complex) Polymorphic System (Part 2)
- 7) Extending a Polymorphic System Using an Interface
- 8) One Final Note
- 9) Summary

## 5.1 RESPONSIBILITY

Polymorphism is very powerful but only works if systems are designed correctly. An essential requirement for a well-designed Object Oriented system is to place methods in the correct classes. To do this you must understand what each class is responsible for i.e. we must put the methods in the correct classes (which is not always as easy as it sounds).

### Activity 1

Consider a computer system for a University one part of which prints student's degree certificates as described in the following class diagram.



Note: The diagram above indicates a '1 to many' relationship between the University Awards Department class and the Student class. This is because the system must maintain a collection of students (not just one student). We will consider polymorphic collections later in this chapter and learn how to program them in Chapter 9.

For now we will focus on understanding where responsibility lies.

Look at the diagram above and answer the following questions...

- a) In which class would the PrintDegrees() method go?
- b) Who has the responsibility for keeping a student's results...is it the student or the University Awards Department?
- c) If the responsibility for keeping a student's results belongs to the Awards Department – then what happens to this class if we add new types of student (e.g. PartTimeStudent)?

### Feedback 1

A core requirement for developing good Object Oriented systems (systems that can be easily extended) is to put the responsibility for each part of the system where it belongs i.e. put methods in the correct classes.

- a) In which class would the PrintDegrees() method go?

The responsibility for printing degrees belong with the university awards department hence the PrintDegrees() method would go there. This will print all of the degrees that are being awarded at the end of the year.

- b) But who has the responsibility for keeping a student's results...is it the student or the University Awards Department?

In the real world of course a student does not have the responsibility to keep an official record of their results – the University stores these to prevent fraud – but here the Student class is not an actual student. It is a model of a student within a software system that belongs to and is controlled by the University.

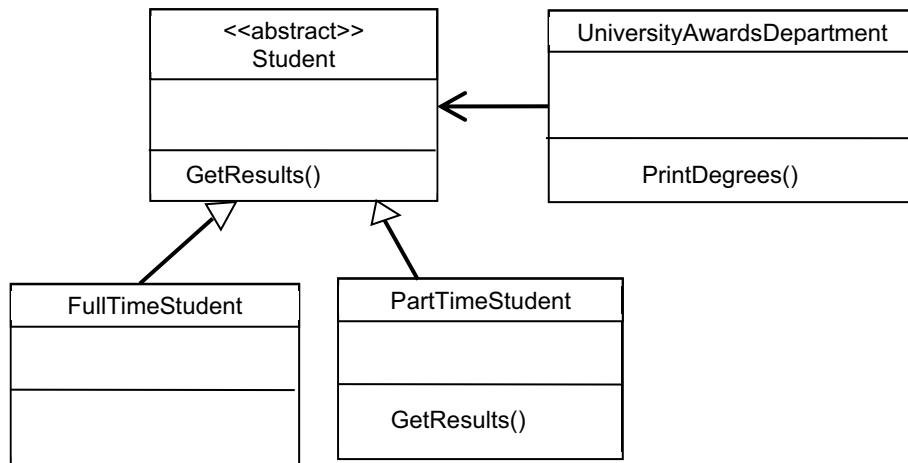
The results belong to a student so the Student class has the responsibility of storing the results for that student.

- c) If the responsibility for keeping a student's results belongs to the Awards Department – then what happens to this class if we add new types of student (e.g. PartTimeStudent)?

If the responsibility for keeping a student's results belongs to the Awards Department then the awards department would need to understand how the results differ for each new type of student. In other words each time we add a new type of student then not only would we need to add a new Student subclass we would also need to reprogram the UniversityAwardsDepartment class. **This would be a bad design!**

This goes against the principle of Polymorphism. Polymorphism should mean our systems are easy to extend as we can add students of different shapes i.e. new types of student.

The system should therefore be as shown below...



Now when a new class of student is added, e.g. PartTimeStudent, it will inherit the GetResults() method defined in the superclass Student. Then the PrintDegrees() method can ask each student for their results and use this information when printing the degree.

Storing student's results in the Student class makes the system very easy to extend.

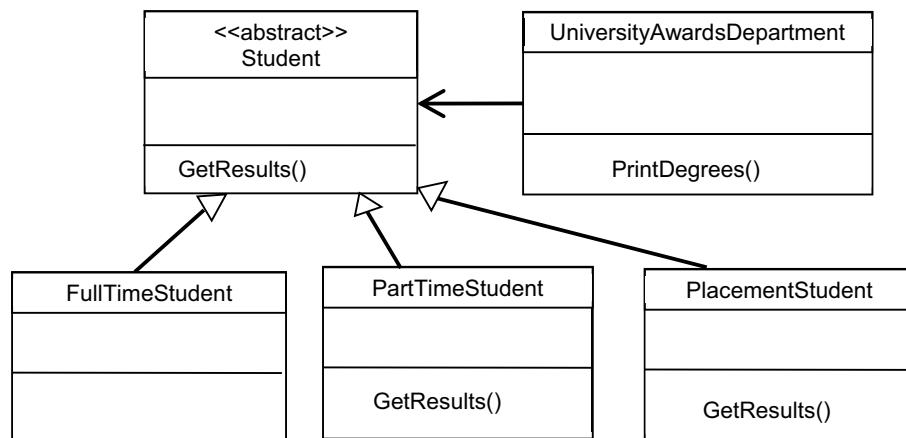
However while the PartTimeStudent class will inherit the GetResults() method it can be overridden (as shown in the diagram above). Imagine a situation where the results for a part-time student are different from a full time student, perhaps because they take fewer subjects.

The University awards department does not need to know about the different types of student. The student themselves know what type of student they are and they can report 'appropriate' results to the awards department. In other words the GetResults() method is overridden in the PartTimeStudent class.

The Print Degrees() method says to each Student 'what are your results?'. A fulltime student invokes the method inherited from the superclass. A part time student invokes the overridden method. Either way appropriate results are reported and the correct results are printed on the degree certificate **and, importantly, the UniversityAwards Department class does not need to be reprogrammed each time a new type of student is added to the system.**

## Activity 2

- a) The University now decides to give students the opportunity to spend a year working for an employer and they want to reflect this in the results when printing the degree certificates. To enable this, the diagram below shows a new class of student has been added to the system – a PlacementStudent.



Does this addition require a change to the PrintDegrees() method?

- b) When printing the degrees the University decides they wants to tailor the entire certificate to reflect the type of student taking that degree. Thus, for placement students the name of the placement provider (i.e. the company) could be placed in a prominent position on the certificate – this requires changing more than the GetResults() method but how can this be done?

Hint: who does the certificate belong to: the student or the University?

## Feedback 2

a) Does this addition require a change to the PrintDegrees() method?

If polymorphism works we should be able to extend a system simply by adding a subclass without the need to change other parts of the system. In this case we can simply add the subclass PlacementStudent and override the GetResults() method.

Now as each certificate is to be printed the correct GetResults() method is invoked for each type of student and the certificates are therefore different, but correct, for each type of student.

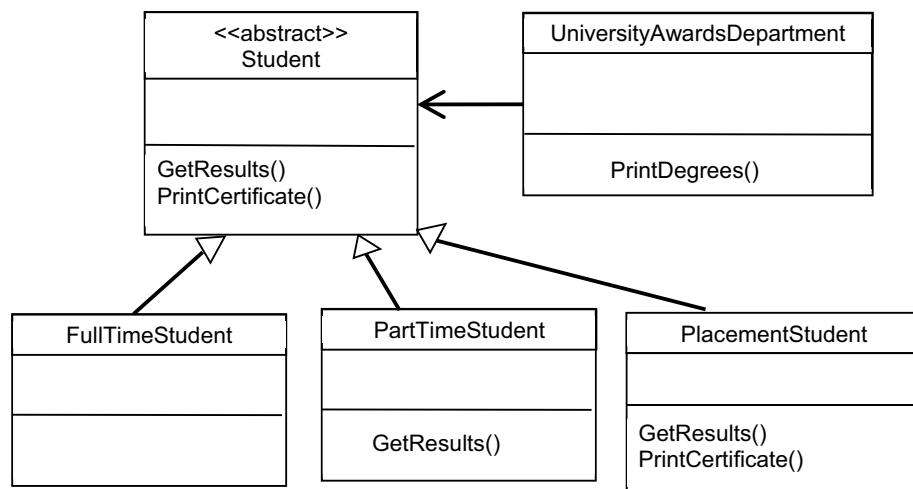
**The UniversityAwardsDepartment class does not need to be reprogrammed each time a new type of student is added.** This class does not 'know' about the different types of student... it just knows they are students and students can report on their own results.

The student themselves know which type of student they are and therefore which GetResults() method to invoke. The computer checks the student type when the system is running and then decides which GetResults() method to use. Thus the GetResults() method in the PlacementStudent class can return a string that reflect the fact that this student did a placement for a specific employer e.g. 'Tom Jones 1<sup>st</sup> class honours degree having done a placement with Google'.

Polymorphism therefore makes the system very easy to extend but only if we put the responsibility where it belongs i.e. we put the methods in the correct classes. A student's results belong to the student and should be stored in the Student class. A GetResults() method can then be added to that class to make the results accessible to other parts of the system. Overriding this method means that new types of student can report results that are appropriate to that type.

b) How can the University tailor the entire certificate, not just the results section, to reflect the type of student taking that degree?

The certificate belongs to the student, not the University, so why not place the responsibility for printing a student's certificate in the Student class (as shown in the diagram below)?



The UniversityAwardsDepartment class will still contain a method to print all the degrees but this will now simply iterate around every student stored in the system asking each student to print their own certificate.

Doing this will mean that all aspects of the certificate can be tailored depending upon the type of student it is intended for.

According to the design above full time and part time students will use the inherited version of the PrintCertificate() method so the certificates will generally look the same but the results section will still vary as the GetResults() method is overridden in the PartTimeStudent class...therefore certificates for full time and part time students will still not be absolutely identical even though the same PrintCertificate() method is invoked.

The certificates for placement students can, however, look very different as the PrintCertificate() method is overridden in the PlacementStudent class.

Polymorphism will work well with this design. New types of students can be added to the system, e.g. foreign students, and overriding the PrintCertificates() method will allow the certificates to be tailored to suit this type of student. **The degrees for foreign students could even be printed in the student's home language...and all of this requires no change to the UniversityAwardsDepartment class or other existing parts of the system.**

Imagine the university now runs short courses and wants to print certificates for students who complete these.

This change would be very simple to implement.... Design a certificate that is appropriate for a student who has completed a short course. Add 'ShortCourseStudent' as a new subclass of Student and override both the GetResults() method and PrintCertificate() method accordingly. The awards department would simply treat short course students like any other student and invoke the PrintCertificate() method when a certificate is required.

Thus, new subclasses can be added and the system extended (and therefore improved) without having to change (re-program) other parts of the system.

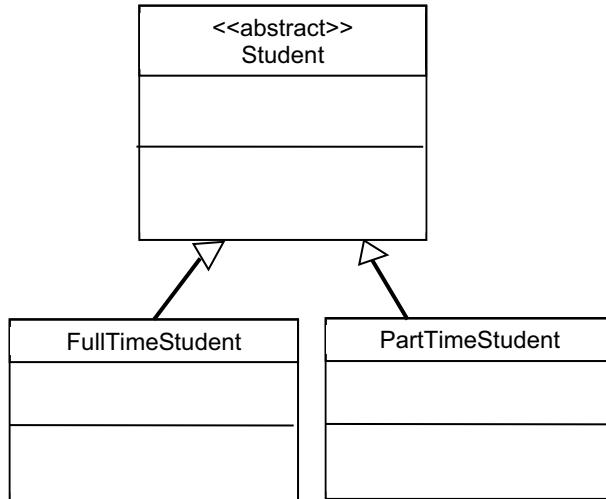
Other parts of the system (not shown) e.g. the library system, the student billing system etc. would also not need to change as new types of student are added. These parts of the system deal with all students polymorphically (without regard to the specific type of student they are). A short course student is fundamentally just like any other student and they deserve a certificate when they finish their course, can borrow items from the library and they need to pay any relevant bills.

**Polymorphism makes systems very easy to extend as a client's/user's needs change but only if we recognise what each class is responsible for and put the methods in the correct place.**

## 5.2 WHAT CHARACTERISES A POLYMORPHIC SYSTEM?

Polymorphism means 'many shapes'. There are two parts of the diagram that together indicate a system is polymorphic.

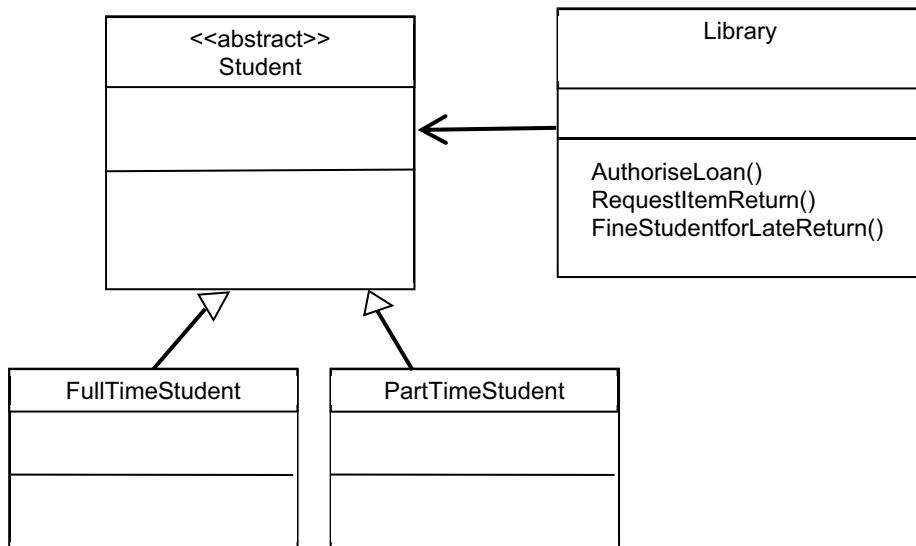
Firstly, there must be an inheritance hierarchy as shown below.



This inheritance hierarchy shows that students come in several types (or shapes). Full time students and part time students are both types of student.

Secondly, there must be a part of the system that deals with the different types of student polymorphically (without regard to their shape) i.e. as if they were just students.

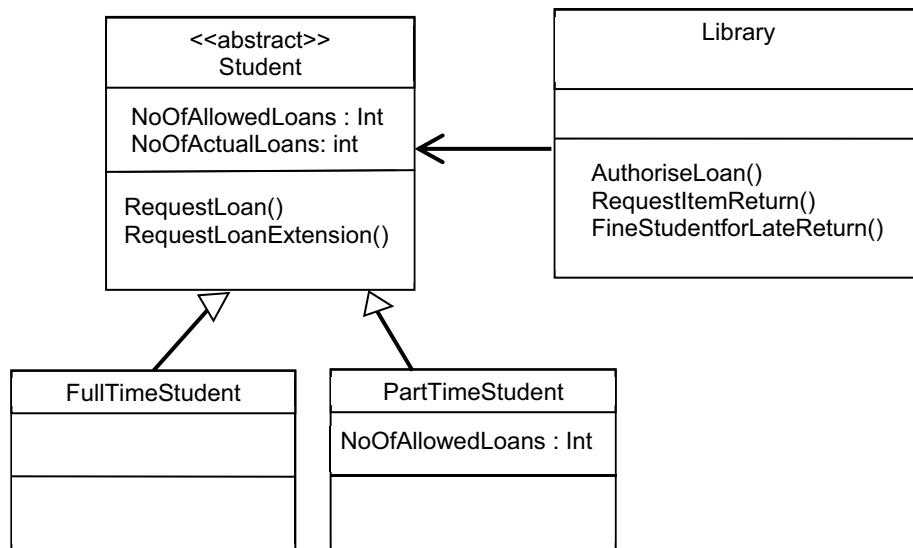
An example of this is shown in the diagram below:



Here, the Library class has an association with the Student class...it has no relationship with specific types of student i.e. it treats all students the same. It authorises loans for them, request items are returned when needed and fines students for late returns.

These methods work in the same way irrespective of the type of student. So when a new type of student is added, e.g. a distance learning student, the library system can automatically treat them as any other student and authorise loans accordingly.

Polymorphism however, does not mean that we must ignore the differences between student types. The diagram below indicates that the number of allowed loans is overridden in the PartTimeStudent class...perhaps because part time students are allowed fewer loans.



Imagine that the number of allowed loans as defined in the parent class as 10 but this number is overridden in the **PartTimeStudent** class as 5. A full time student will use the inherited value and a part time student will use the overridden value – these values can be accessed by the **Library** class when deciding whether or not to authorise a new loan.

Now imagine a new class is added ‘**DistanceLearningStudent**’ with the number of allowed loans overridden at 0 – perhaps because distance learning students study abroad and can’t borrow books from the main library.

As this class is added, thanks to polymorphism, the **Library** class can treat them as they would any other student. The system works without changing the code in the **Library** class but of course no loans would actually get approved to these students.

Polymorphism is therefore indicated by two things working in tandem...

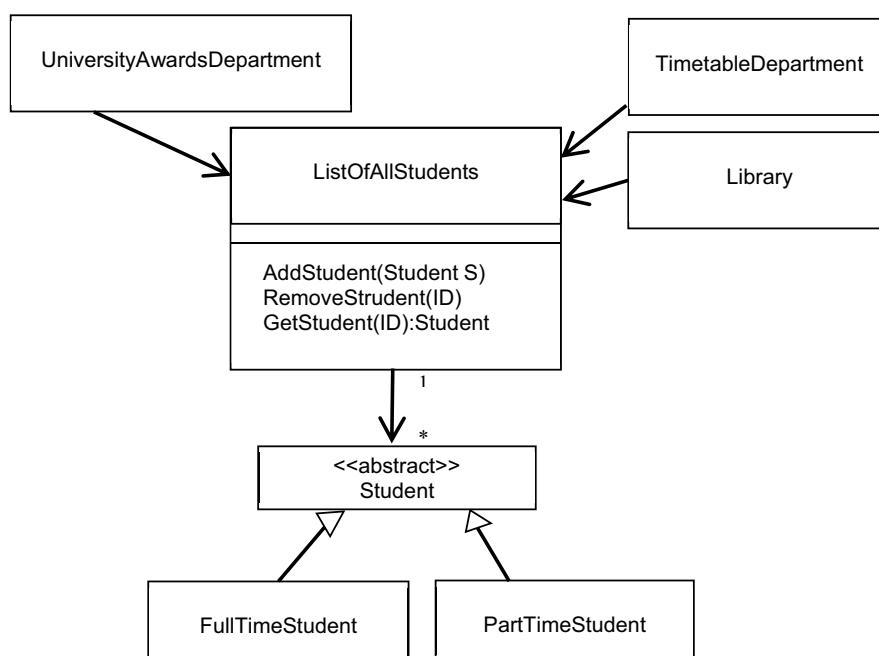
- a) An inheritance hierarchy (indicating many shapes or types) and
- b) An association with the super type class (indicating that parts of the system deal with the types of student without concern for what type they are).

### 5.3 POLYMORPHIC COLLECTIONS

Polymorphism is a powerful mechanism that makes systems very easy to extend.

The diagram below indicates a system designed to keep a polymorphic list of all students... with methods to add a student to the list, remove a student and get a student with a specified ID. A student in this list can be either a full time or part time student – or indeed any new type of student not yet defined.

The list is polymorphic as the mechanism to add and retrieve student's works no matter what type of student we are trying to add or remove.



The other parts of the system, the library, the awards department and the timetabling system can work with this list of students without worrying about which specific type of student they are dealing with.

If a new type of student is defined (e.g. a short course student) then objects of this new type can be added to the list of all students and the other parts of the system will work automatically i.e. lectures and tutorials can be scheduled for them (as they would for any other student), they can borrow items from the library and the awards department can print certificates when they finish their course (and all these classes should work without needing to be changed).

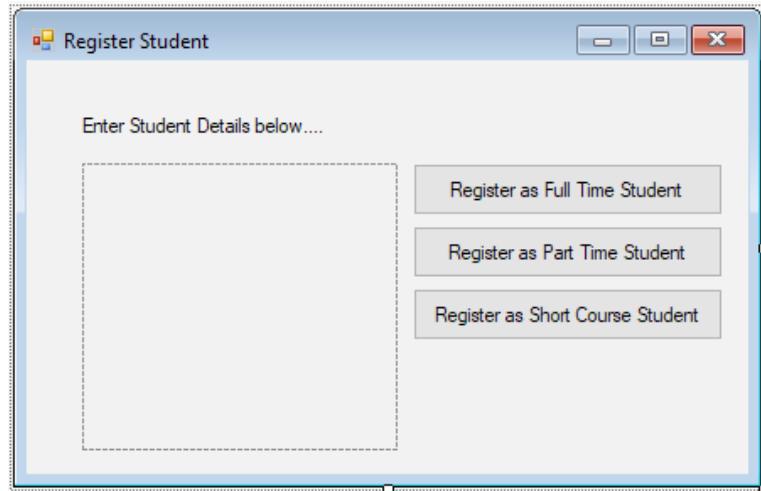
We will learn how to program collections in Chapter 9.

## 5.4 NON-POLYMORPHIC COMPONENTS

Polymorphism is a powerful and useful mechanism but there will be parts of the system that cannot be polymorphic.

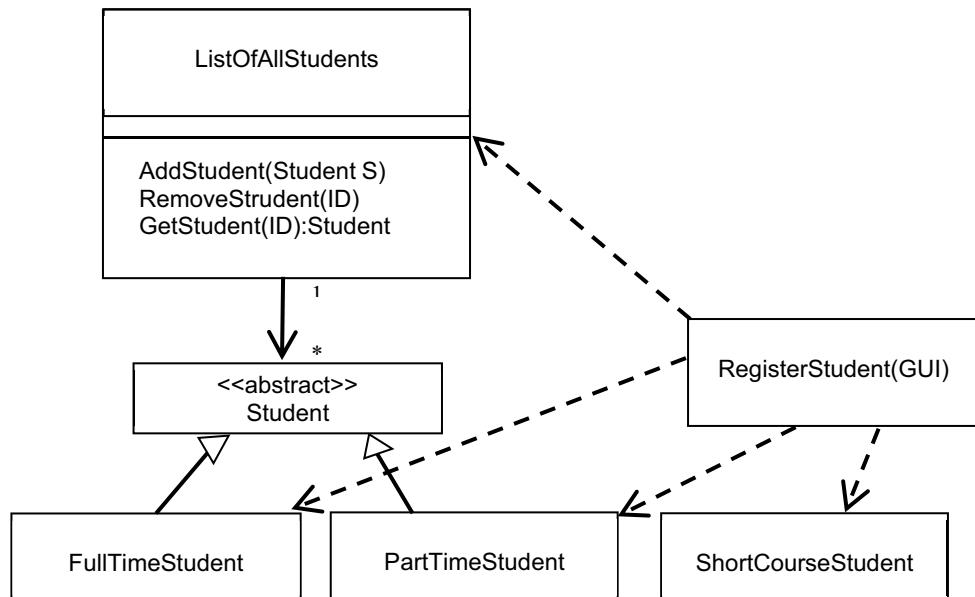
When a new student registers at the university, the constructor for the appropriate class must be invoked. Thus, we must know at the moment of registration what type of student they are...if they are a full time student then the constructor for the FullTimeStudent class must be invoked.

A complete system would need a GUI that allows an administrator to type in the details of a prospective student...name, address, course etc and then registers them adding them to the list of all students.



A GUI to do this, such as the form above, would need separate buttons to register the different types of student. Each button would simply invoke the relevant class constructor using the student details and add the object returned on to the list of all students.

This class would therefore need a relationship with each specific type of student and with the class `ListAllStudents` (as shown in the diagram below).



As the `RegisterStudent` class has a relationship with specific types of student it is not polymorphic. Remember polymorphism means 'many shapes' and according to this diagram full time students do not come in many shapes.

This part of the system cannot be polymorphic.... If a new type of student is created then the GUI will need an extra button added and the code behind this button will need to invoke the constructor for the new subclass.

Still, while the GUI will need to be adapted to cope with new types of student the other parts of the system should not require changes to be made. Polymorphism is still a very powerful mechanism that allows a system to be extended by creating new subclasses while requiring minimalistic changes to large parts of the system.

## **5.5 A MORE REALISTIC (COMPLEX) POLYMORPHIC SYSTEM (PART 1)**

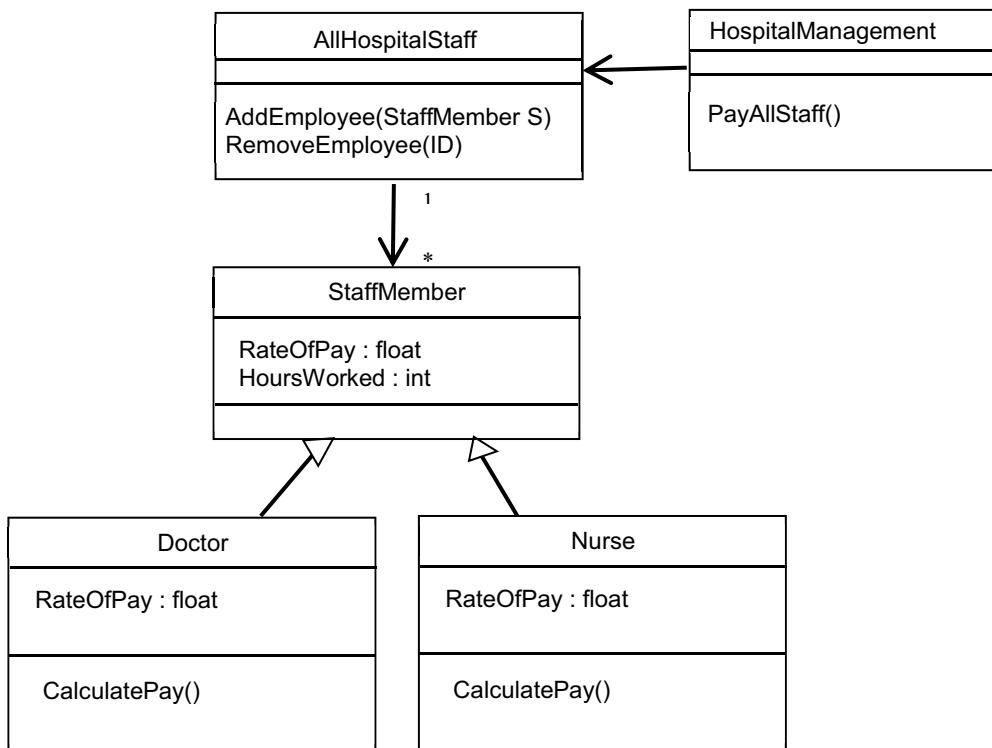
To help ensure you understand how to apply polymorphism let us consider a more complex and realistic system...but we will do this in stages.

Consider a system designed for a hospital. The hospital management want to keep a collection of staff and wants to pay all staff. A class is used to maintain a collection of all staff and a HospitalManagement class iterates around the collection of all staff to pay them.

As the hours worked and rate of pay belong to each staff member they are stored in the StaffMember class...the rate of pay will be overridden in each subclass so each type of staff can be paid an appropriate amount.

Currently the hospital only employs Doctors and Nurses and each of these classes includes a method to calculate the pay for that type of staff member.

All of this is depicted on the diagram below...



### Activity 3

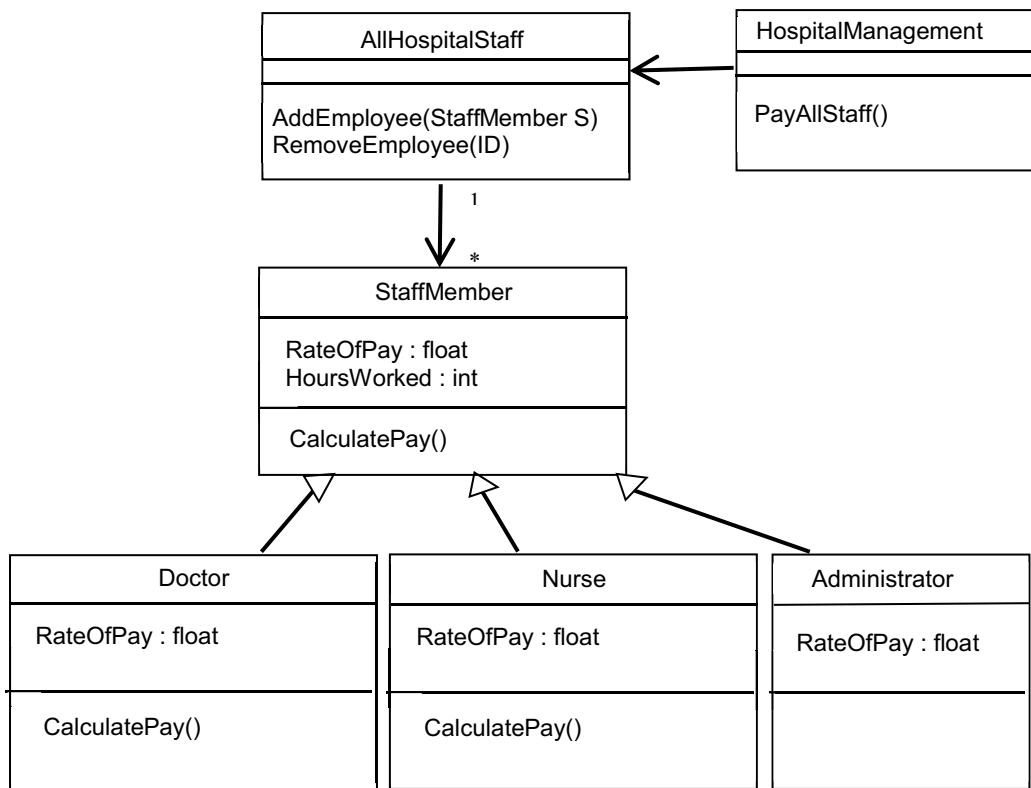
Consider the diagram above and answer the questions below...

- Is this a polymorphic system i.e. does the **HospitalManagement** class treat all staff just as 'staff' and can new staff types be added easily?  
Or does the **HospitalManagement** class have specific relationships with each specific type of staff?
- Can management pay Doctors and Nurses or is something missing from the **StaffMember** class?
- Given the answer to (a) should we be able to add new types of staff, e.g. Administrators, and pay them accordingly without changing the **HospitalManagement** class?

### Feedback 3

- a) The system is polymorphic. The AllHospitalStaff class maintains a collection of all staff irrespective of their specific type. Thus the HospitalManagement class does not know whether a member of staff is a doctor or a nurse...it treats them all just as members of staff.
- b) The system currently cannot pay staff as there is not a CalculatePay() method in the StaffMember class...even though the Doctor and Nurse classes both have a CalculatePay() method the system cannot invoke this method and cannot pay them. If we knew they were doctors etc, we could ask them to calculate their pay but as we only know they are members of staff we therefore do not know that they have a CalculatePay() method – therefore management cannot pay them. We can only treat them as members of staff and invoke methods defined in the StaffMember class.
- c) If we correct this mistake by adding a CalculatePay() method to the StaffMember class, then we would guarantee that all staff can calculate their pay and thus management could pay all staff.

In the diagram shown below this missing method has been added and an Administrator class has been added. Though no methods have as yet been overridden in this new class, an appropriate rate of pay has been added and hospital management can pay administrators using the inherited CalculatePay() method.



**Adding the CalculatePay() method to the StaffMember superclass provides guaranteed functionality for all types of staff.**

## 5.6 A MORE REALISTIC (COMPLEX) POLYMORPHIC SYSTEM (PART 2)

Now we will significantly extend this system and consider an additional complexity... consider the fact that doctors do more than calculate their pay – they diagnose patients and prescribe drugs.

### Activity 4

Consider the following two questions.

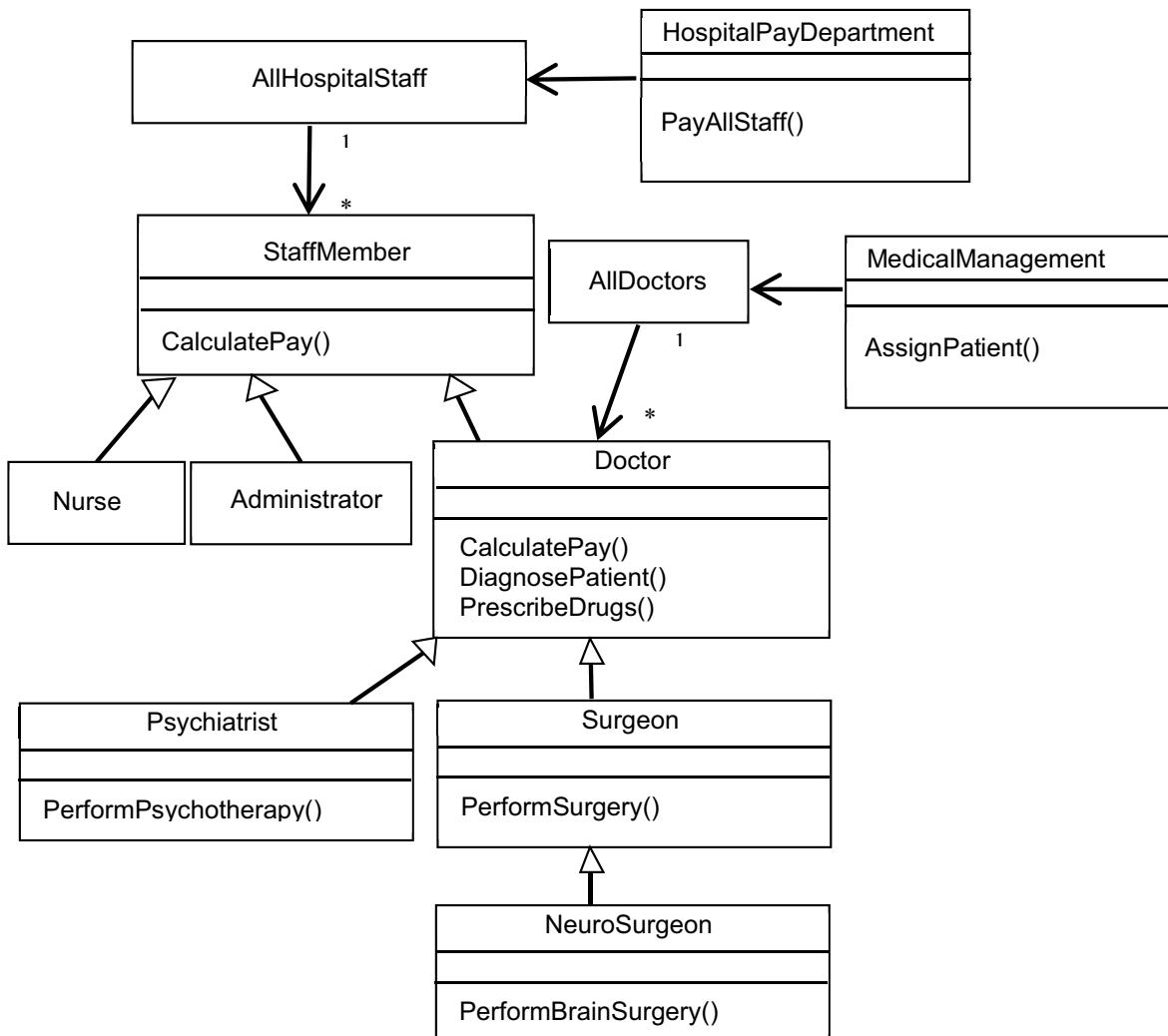
- If we add methods `DiagnosePatient()` and `PrescribeDrugs()` to the `Doctor` class could the `HospitalManagement` class, as shown in the previous diagram, ask doctors to diagnose patients?
- Would administrators in the pay department even want to ask doctors to treat patients?

#### Feedback 4

- a) The HospitalManagement class as currently shown does not have a relationship with or understanding of the Doctor class. They therefore cannot ask these staff to do anything medical – they can only treat them as they would any other member of staff and ask them to do general things e.g. calculate their pay.
- b) Of course administrators in the pay department don't need to understand about the different types of staff, they don't treat doctors differently from any other staff member. They will just ensure that they are paid appropriately.

Other administrators will deal specifically with doctors assigning patients to them etc. Therefore we would almost certainly need to keep two lists (a) a list of all staff so we can pay them etc and (b) a list of doctors so we can deal with them as doctors assigning patients to them etc.

An extended system is shown in the diagram below. To keep it simple some of the classes are shown without any details and the HospitalManagement class has been renamed HospitalPayDepartment.



Looking at the extended hospital system, shown above, you can also see that special types of doctor are now being employed by the hospital so the system has been extended with additional classes `Psychiatrist`, `Surgeon` and `Neurosurgeon` (with `Neurosurgeon` being a specialism, sub type, of `Surgeon`). Additionally, methods have been added so the psychiatrist can perform psychotherapy, the surgeon can perform surgery and the neurosurgeon can perform brain surgery.

### Activity 5

- a) The extended hospital system, shown above, indicates two collections...all hospital staff and all doctors. Are both of these collections polymorphic?
- b) Can the pay department pay all staff, including the new specialist staff, appropriately?
- c) Can the medical management department ask all doctors, including the new specialist staff, to diagnose patients?
- d) Can the medical management department ask a psychiatrist to perform psychotherapy, a surgeon to perform surgery and a neurosurgeon to perform brain surgery?

### Feedback 5

- a) Both of these collections are polymorphic as indicated by the following two relationships...an association (with a superclass)  

The diagram illustrates the inheritance relationship. A downward-pointing arrow from the 'StaffMember' class to the 'Surgeon' and 'Neurosurgeon' classes is labeled 'and inheritance' with an upward-pointing arrow, indicating that the subclasses inherit from the superclass.
- b) The pay department can pay all staff, including the new specialist staff, appropriately. There is a guarantee that all staff have a CalculatePay() method as this is defined in the super class StaffMember...at the very least this will be inherited down and all staff can therefore be paid. Of course this method could be overridden in the subclasses Surgeon and Neurosurgeon with a more appropriate method of calculating pay (perhaps these staff have an element of performance related pay based on the number of operations they perform).
- c) Similarly, the medical management department can ask all doctors (including the new specialist staff) to diagnose patients as they all inherit this method.
- d) **However, the medical management department cannot ask a psychiatrist to perform psychotherapy, cannot ask a surgeon to perform surgery nor a neurosurgeon to perform brain surgery!**

The medical department has a polymorphic relationship with all doctors but not all doctors can perform these specialist procedures. So how do we solve this problem?

One way of solving this problem would be to maintain lots of different lists – a list of all doctors, a list of surgeons etc. as we would then know who the surgeons are we could then ask them to perform surgery. This would however be a bad design!

If a new type of doctor is added e.g. a paediatrician then we would need to reprogram the other parts of the system to maintain a list of paediatricians. This solution is not a polymorphic solution as we are dealing with each specialism, not the general type i.e. the superclass, and we would need to constantly reprogram the system as new specialisms are added.

There is a much better way to solve this problem!

We cannot ask a golfer to play golf or a swimmer to swim a race if all we know is that they are a sports person. If we don't know that a person is a golfer or swimmer then we cannot ask them to perform particular actions related to those particular sports.

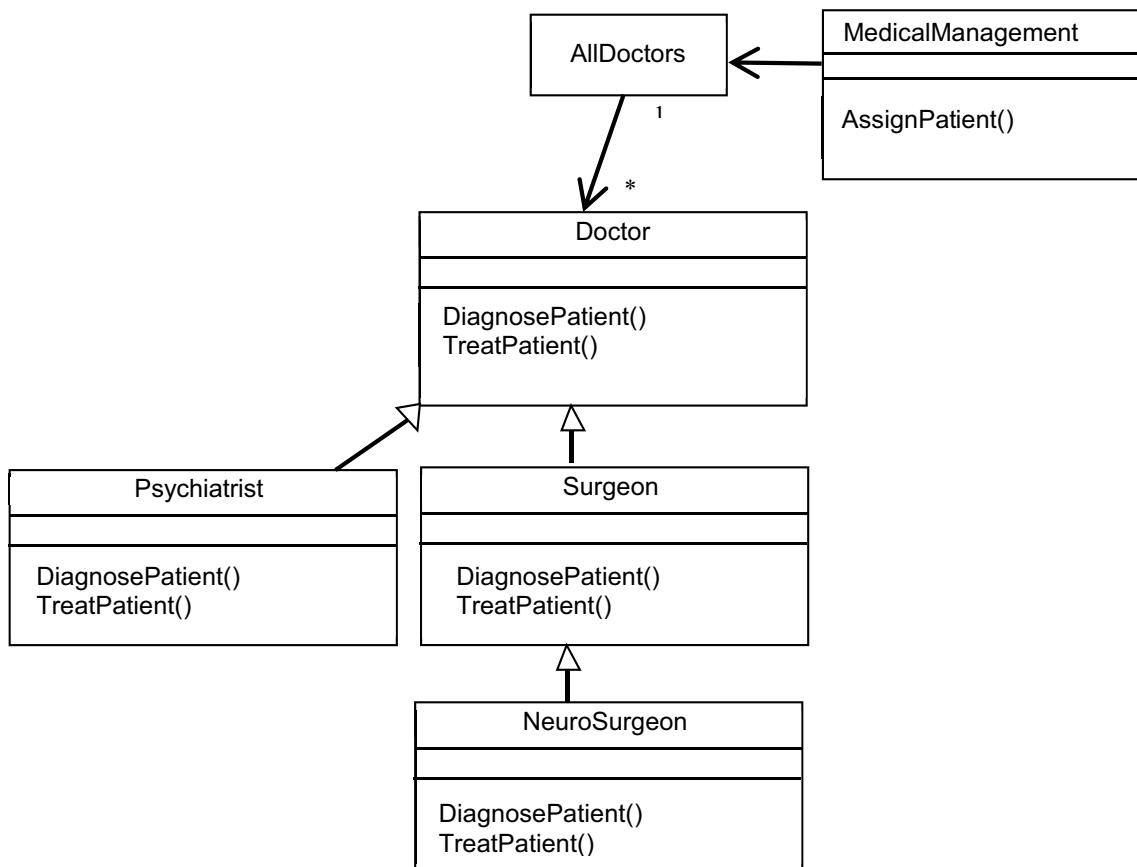
However we can ask them to compete!

A golfer knows what they must do to compete against another golfer, a swimmer, marathon runner etc also know what it means to compete in their specialism.

In the same way a medical administrator cannot ask a surgeon to perform surgery if they do not know they are a surgeon...but they can ask them to treat a patient.

A surgeon knows what they must do when they treat a patient, so does a neurosurgeon and so does a psychiatrist.

Rather than litter our design with specialist methods that we don't need and can't use, the answer is to create general methods in the superclass and override them with appropriate behaviour in the different subclasses (as shown in the diagram below).



In this revised design, shown above, the **PrescribeDrugs()** method in the **Doctor** class has been renamed **TreatPatient()** and both the **DiagnosePatient()** and **TreatPatient()** methods are overridden in all of the subclasses.

Now a medical administrator can ask a psychiatrist, a surgeon and a neurosurgeon to diagnose and treat a patient. The administrator does not need to understand what each specialist member of staff does – the staff member can decide for themselves what they should do to when diagnosing and treating a patient.

The **TreatPatient()** method in the **Surgeon** class will involve surgery in the **Psychiatrist** class the **TreatPatient()** method will do something different but it will still ‘treat a patient’.

One question remains: How does the patient get assigned to a psychiatrist, surgeon or neurosurgeon correctly if the administrator doesn't understand these different specialisms?

### Activity 6

Consider the question...Who is the best person to decide what you are capable of doing? Is it your boss, an administrator in your organisation or yourself?

### Feedback 6

The best person to decide what you are capable of doing is you. It follows then that you are the best person to decide what jobs you are capable of doing.

In the same way, as you are the best person to say what you are capable of, so a psychiatrist, surgeon and neurosurgeon are the best people to decide what they are capable of diagnosing and treating.

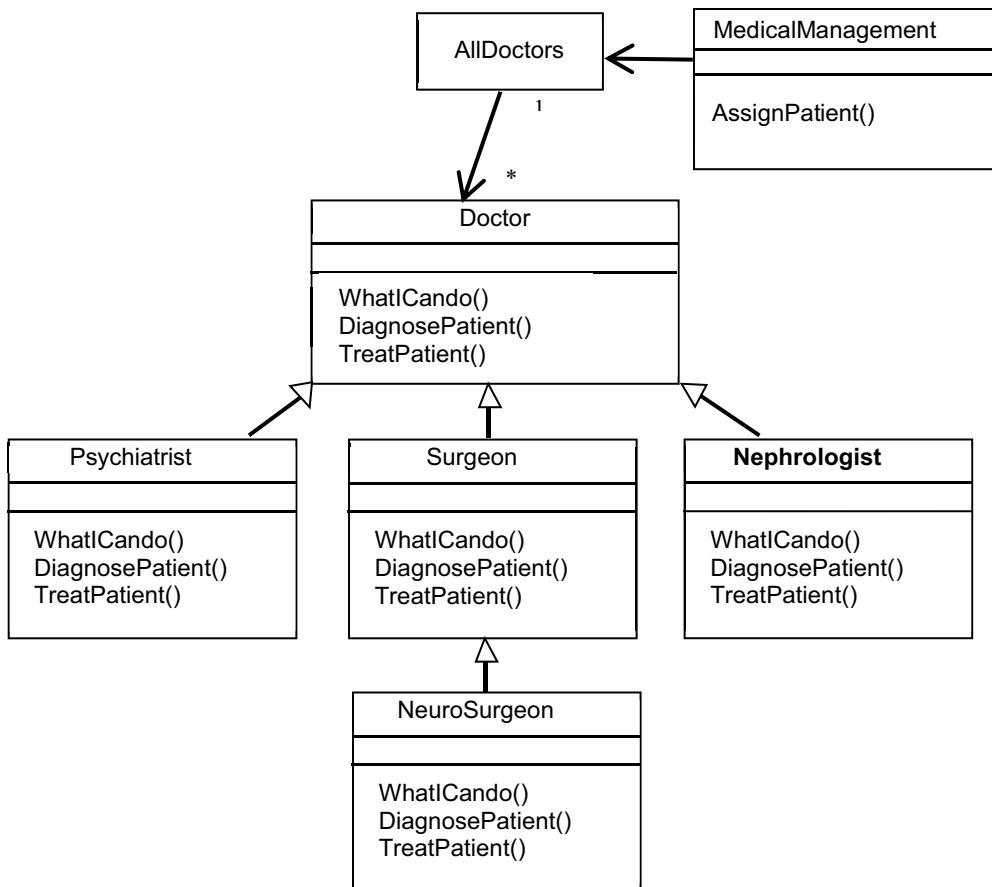
A general doctor could say that they can perform an initial diagnosis, a psychiatrist could say 'I can confirm whether a patient is suffering from a range of mental ailments' and a surgeon could say what they can do.

Thus, we can add a 'What I can do method' to each class and an administrator can then ask each type of doctor what they can do and then match a patient to an appropriate member of staff.

A doctor would say they can perform an initial diagnosis and a new patient would then be referred to that doctor. If the doctor suspected a kidney failure, they would update the patient's medical records accordingly. An administrator could then match this against any available specialist who said they could confirm and treat kidney failure.

Thus, the administrator could invoke general methods that all specialist doctors have without understanding what that specialism involves.

Do you know what a nephrologist does? Neither do I – but they are a specialist and we can enhance our hospital system simply by adding them as a class as shown below.



Now with the addition of a **WhatICando()** method, a medical administrator can ask all doctors, including a nephrologist, what they can do and can then ask them to diagnose and treat appropriate patients.

Thanks to polymorphism new specialist staff can be added to the system very simply. The specialist staff can still do ‘their special thing’ and all staff can be paid appropriately. The pay department does not need updated procedures to pay new types of doctor and the medical administrators don’t need retraining to learn about the new specialisms...they just ask the new specialist staff what they can do.

By designing the system well, the **MedicalManagement** class still has a polymorphic relationship with all doctors...they don’t deal with different specialisms directly and thus the system is very easy to extend yet specialist staff can still do their special thing.

Of course we also need to update the non-polymorphic part of the hospital system, the GUI, so we can register new specialist employees i.e. invoke the relevant constructor and add them to the list of all doctors.

## 5.7 EXTENDING A POLYMORPHIC SYSTEM USING AN INTERFACE

So far we have designed a polymorphic system for a hospital that maintains a list of all staff, allows staff to be paid and allows doctors and other specialist staff to diagnose and treat patients each in their own particular way.

**This may be just what the hospital needs at the moment but a client's needs change and software needs to be adaptable to meet those changing needs.**

As this is polymorphic, new types of staff can easily be added (nurses, cleaners, security staff etc.) and they will be paid appropriately without needing to adapt or rewrite any of the code in the PayDepartment class. Similarly, new specialist doctors can be added, making this system adaptable.

But can we make even more flexible?

### Activity 7

Think for a moment and try to identify someone who is not a member of the hospital staff but who, for some reason, the hospital may need to pay.

### Feedback 7

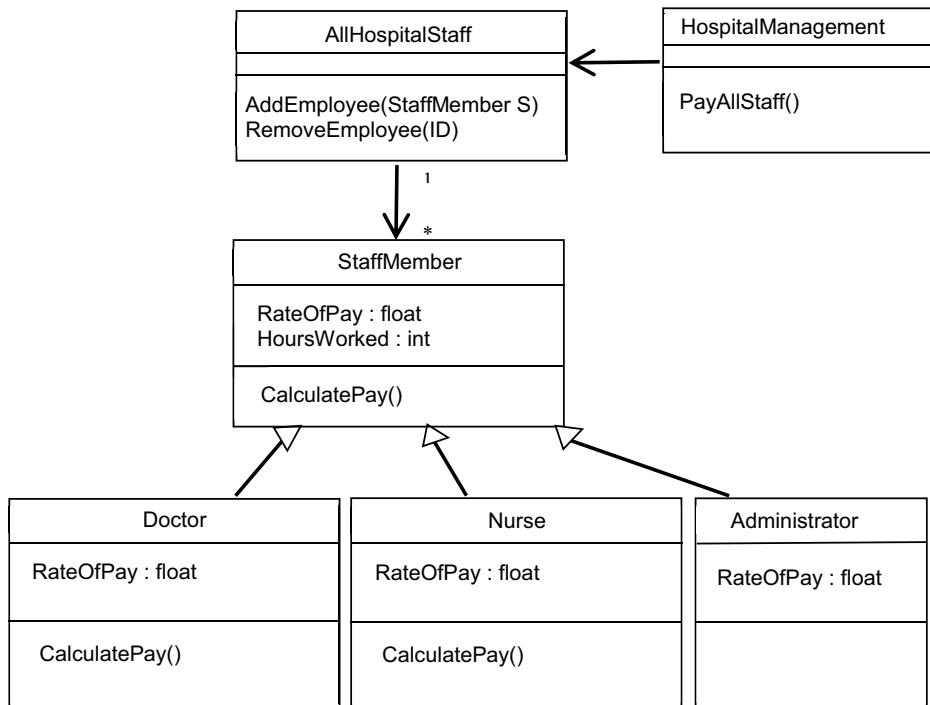
The hospital may need to pay contractors for doing some building work. They may need to reimburse volunteers for expenses.

We can, of course, add new members of staff to the payroll but the hospital could, in the future, identify other people who need to be paid.

Ideally our design would make it easy to add payees who are not staff members. But how do we pay others (contractors etc.) without needing to make ongoing changes to our system?

### Activity 8

Consider the design below which shows different types of staff – all of whom could be paid.



### Activity 8 Continued

Now answer the following questions:

- a) Could we add a Contractor class and a Volunteer class as subclasses of StaffMember?
- b) Would this allow the hospital to pay a contractor's bill and a volunteer's expenses?
- c) Is this a good solution?

### Feedback 8

- a) Yes we could add a Contractor class and a Volunteer class as subclasses of StaffMember.
- b) These classes would inherit a CalculatePay() method and we could override this method with a method that returns the amount of the bill or expenses to be paid. This would therefore allow the hospital to pay a contractors bill and pay a volunteer their expenses.
- c) This however is not a good design for several reasons: (a) A contractor is not a member of hospital staff and therefore they should not inherit the attributes RateOfPay and HoursWorked. However inheritance is not selective – an inherited class will inherit everything from the parent class. (b) Other parts of the hospital system that will interact with staff members (e.g. the holiday scheduling/approval system) and these parts of the system would not necessarily apply to contractors as contractors are not members of staff.

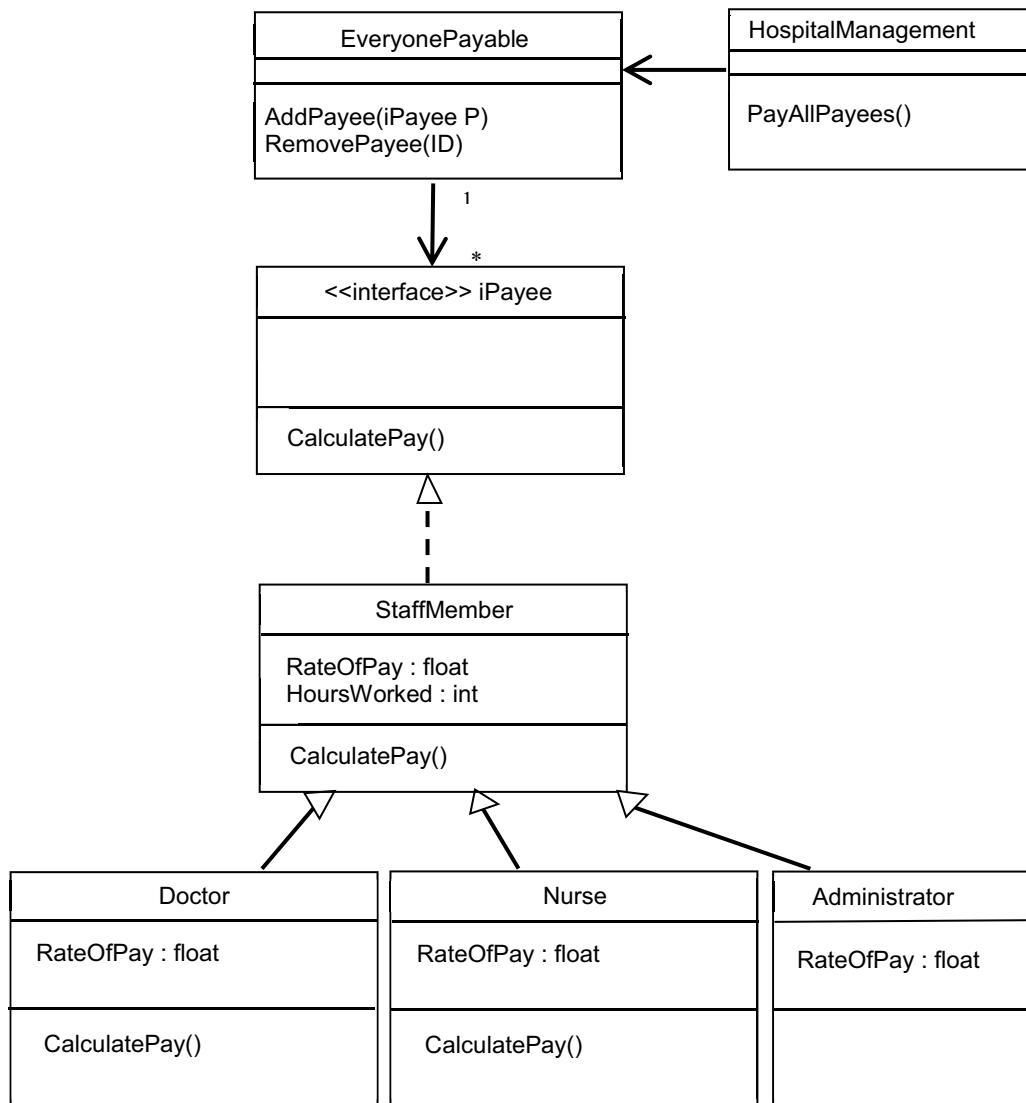
While we could add a Contractor class as a subclass of StaffMember, this would not be a good solution and would lead to complications in other parts of the system.

We need to find a better way of extending the payment possibilities without adding inappropriate classes in the staff member inheritance hierarchy.

The answer is to use an ‘interface’ that allows the hospital management to pay anyone that requires payment...not just staff members.

To enable this we add an interface ‘iPayee’, for anyone who should be paid.

Rather than keep a collection of staff members, we can instead keep a collection of anyone who is payable (see revised design below):



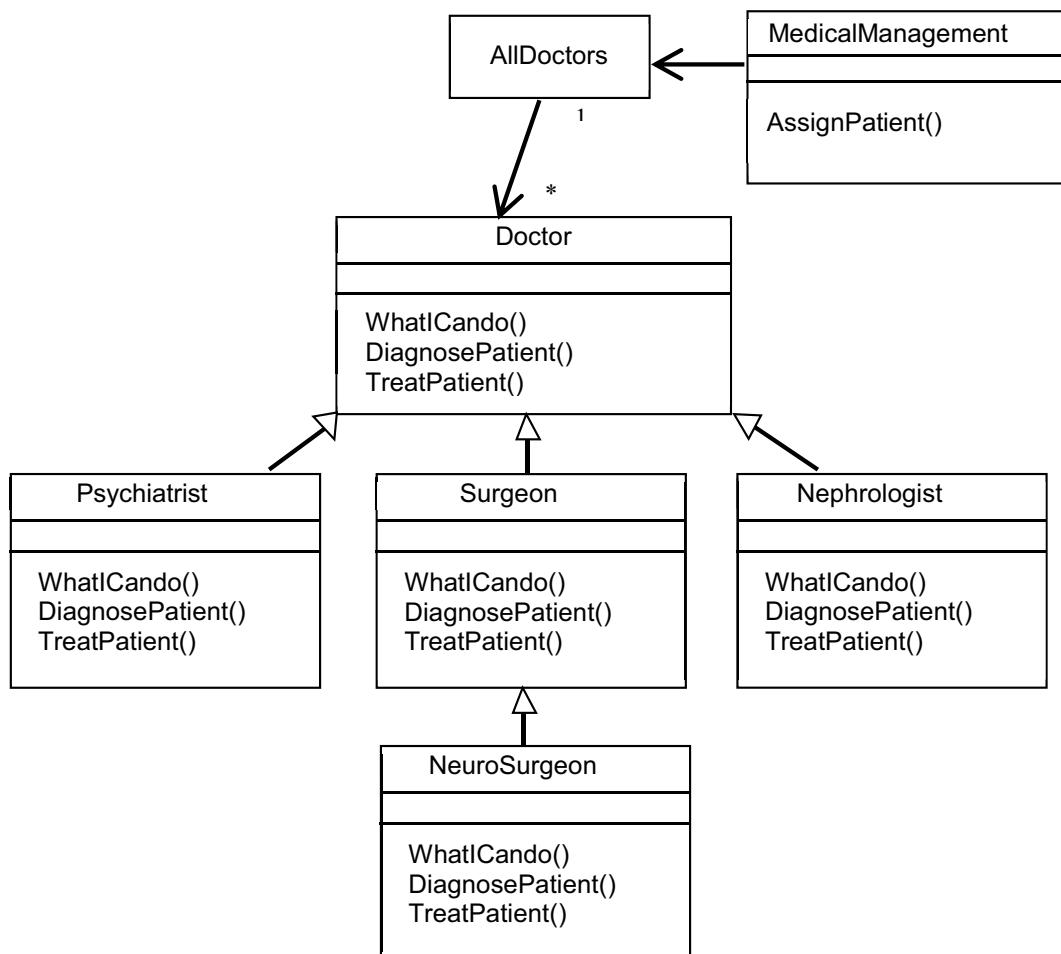
In the design above we have added an interface ‘**iPayee**’ that **StaffMember** implements (note the dashed line indicating implementation rather than inheritance).

The collection has been adapted to keep a record of everyone who needs paying – currently these are all members of staff but this list can now include other people such as contractors – and hospital management now pay everyone who should be paid (not just staff members).

Our system has not functionally changed at all. However it is now more flexible and easy to adapt as the needs of the hospital change. If new types of staff are employed these can still be added as subclasses of **StaffMember** but also, if the hospital identifies new people who need to be paid other than members of staff (e.g. the people who provide services to the hospital (electricity/cleaning...)) then additional classes can be added that implement the **iPayee** interface. Thus, additional payees who are not staff members can now simply be added without causing additional complications.

### Activity 9

Looking at the design of the part of system shown below. Can you identify anywhere else where the addition of an interface can make the system even more flexible and easy to extend?



### Feedback 9

The use of interfaces to aid flexibility is an important topic and we will look again at this in Chapter 8 when we consider the SOLID design principles. For now, we can just consider how using an interface can make any collection more flexible.

The system shows a collection of doctors who diagnose and treat patients. This is a polymorphic collection and we have shown how to add new specialisms where the person can still do their specialist thing.

But what about the possibility of someone who is not a doctor diagnosing and treating patients?

### Activity 10

Can you think of anyone who is not a qualified as a doctor but who still treats a patient?

### Feedback 10

Volunteer first aiders are not fully qualified doctors but they can still diagnose and treat patients. Their diagnosis will be more rudimentary than the diagnosis provided by a doctor but they could still diagnose and treat wounds and they can save lives by diagnosing critical conditions, blocked airways etc., and treating these until the professionals arrive.

It is perhaps unlikely that a hospital would use the services a volunteer first aider but it is possible that the hospital could be involved in training and registering first aiders in which case we may need a simple way of extending our hospital system – and of course we could not add a FirstAider class as a subclass of Doctor as they are not qualified doctors.

Additionally paramedics could be employed by the hospital as qualified first responders. A design decision for these would then need to be made.

Are paramedics qualified doctors (in this case a new sub class can easily be added to accommodate this needed change) or do we need some other say of adding them?

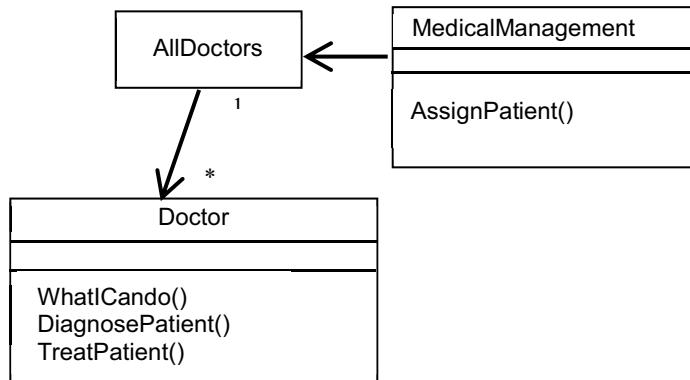
The answer is to add another interface....

By adding an interface ‘iMedic’, we keep a collection of all medics...not just all doctors. This will allow us to simply extend the system by adding new medics, people who are not qualified doctors but who can still diagnose and treat patients.

### Activity 11

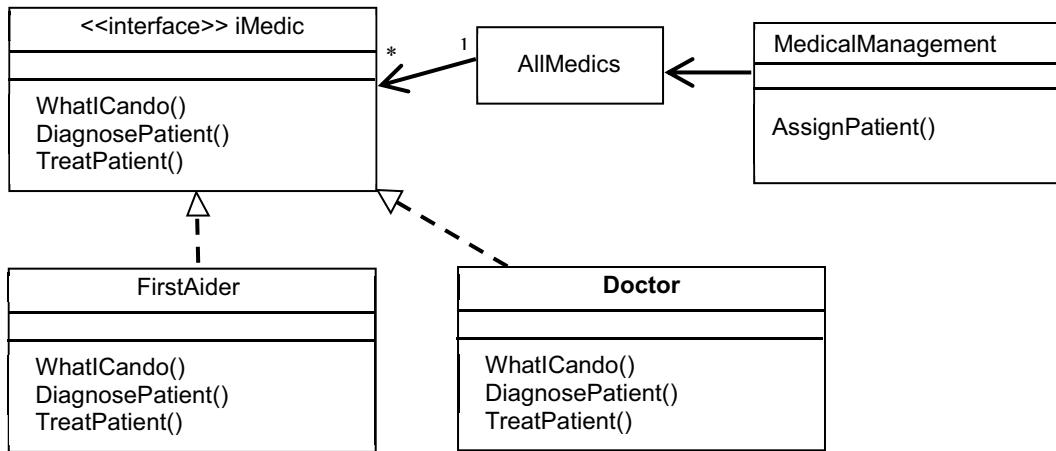
Draw an amended design for the part of the system shown in the diagram below by...

- Adding an interface ‘iMedic’.
- Show that the Doctor class implements this interface.
- Alter the collection so that it maintains a collection of all medics – not just all doctors.
- Add a class ‘FirstAider’ class to the system...a medic who is not a qualified doctor.



### Feedback 11

The design below shows these changes.



Note: Subclasses of `Doctor`, not shown here, do not need to implement the methods they inherit them from the `Doctor` class (though we would expect these methods to be overridden and refined for each of the specialisms). However both the `Doctor` and the new class `FirstAider` **must** implement the methods defined in the interface (as the interface only defines what methods must exist – not how they work).

**By adding the interface `iMedic`, we make our system even more flexible and extendable and thus make it easy to adapt as the needs of the hospital change.**

## 5.8 ONE FINAL NOTE

In this chapter we have shown how to apply polymorphism to make a system for a hospital that can be easily extended and deal with new specialist staff (even when we don't actually know what they do).

In looking at this, we have used classes such as 'Psychiatrist' and 'Surgeon' and we have suggested the inclusion of methods such as `DiagnosePatient()` and `TreatPatient()`.

It is worth remembering, however, that we are designing here a piece of software for a hospital and this class diagram is a model of the software – it is not a model of actual human beings.

The `DiagnosePatient()` method in the `Psychiatrist` class does not define what a psychiatrist does when diagnosing patients. It instead defines what the software does to help the psychiatrist. The software could for example:

- Retrieve the patient's medical records.
- Look through the medical record for drugs that could be causing any side effects reported by the patient.

- Book/schedule any tests or examinations needed.
- Record the results of the psychiatrist's assessment.
- Update the medical records with any planned treatment or discharge notes.

It is because we are modelling a software system that we can place methods such as CalculatePay() in classes that relate to staff...it is still the software, not the member of staff, that is calculating the amount to pay them.

## 5.9 SUMMARY

Polymorphism allows us to refer to objects according to a superclass rather than their actual class which enables systems to be easily extended but only if we:

- Put the responsibility for methods where they belong.
- Design general methods in a superclass that can be overridden appropriately in sub classes.
- Interact polymorphically with the superclass objects without concern for the actual object used (this is determined at runtime).

Classes can manage a polymorphic collection i.e. a collection of supertype objects where the actual objects stored are each of a specific subtype.

Using an interface can make the system more flexible and easy to extend without adding design complications.

One of the claims of Object Oriented design is that it helps produce better models which result in software that is more adaptable. This is critically important as clients' needs constantly change. Hopefully this chapter has helped you appreciate how polymorphism is applied.

Ideas regarding how a client's requirements can be translated into a model will be explored more generally in Chapter 7.

# 6 OVERLOADING METHODS

## Introduction

This chapter will introduce the reader to the concept of method overloading

## Objectives

By the end of this chapter you will be able to...

- Understand the concept of ‘overloading’
- Appreciate the flexibility offered by overloading methods
- Identify overloaded methods in the online API documentation

This chapter consists of the following three sections:

- 1) Overloading Method Names
- 2) Overloading To Aid Flexibility
- 3) Summary

## 6.1 OVERLOADING METHOD NAMES

Historically, in computer programs method names were required to be unique. Thus the compiler could identify which method was being invoked just by looking at its name.

However, several methods were often required to perform very similar functionality. For example, a method could add two integer numbers together and another method may be required to add two floating point numbers. If you have to give these two methods unique names which one would you call ‘Add()’?

In order to give each method a unique name the names would need to be longer and more specific. We could therefore call one method AddInt() and the other AddFloat() but this would lead to a proliferation of names each one describing different methods that are essentially performing the same operation i.e. adding two numbers.

To overcome this problem, in C# you are not required to give each method a unique name – thus both of the methods above could be called Add(). However, if method names are not unique the C# must have some other way of deciding which method to invoke at run time. i.e. when a call is made to Add(number1, number2) the machine must decide which of the two methods to use. It does this by looking at the parameter list.

While the two methods may have the same name, they can still be distinguished by looking at the parameter list:

```
Add(int number1, int number2)  
Add(float number1, float number2)
```

This is resolved at run time by looking at the method call and the actual parameters being passed. If two integers are being passed, then the first method is invoked. However, if two floating point numbers are passed then the second method is used.

Overloading refers to the fact that several methods may share the same name. As method names no longer uniquely identify the method then the name is ‘overloaded’.

## 6.2 OVERLOADING TO AID FLEXIBILITY

Having several methods that essentially perform the same operation, but which take different parameter lists, can lead to enhanced flexibility and robustness in a system.

Imagine a University student management system. A method would probably be required to enrol, or register, a new student. Such a method could have the following signature...

```
EnrollStudent(String name, String address, String coursecode)
```

However, if a student had just arrived in the city and had not yet sorted out where they were living would the University want to refuse to enrol the student? Would it not be better to allow such a student to enrol and set the address to ‘unknown’?

To allow this, the method EnrollStudent() could be overloaded and an alternative method provided as...

```
EnrollStudent(String name, String coursecode)
```

At run time the method invoked will depend upon the parameter list provided. Thus, given a call to

```
EnrollStudent("Fred", "123 Abbey Gardens", "G700")
```

the first method would be used.

A call to EnrollStudent("Fred", "G700") would use the second method but would set the students address as ‘Unknown’. They could, of course, change this data later.

### Activity 1

Imagine a method WithdrawCash() that could be used as part of a banking system. This method could take two parameters: the account identity (a String) and the amount of cash required by the user (int). Thus, the full method signature would be:

```
WithdrawCash(String accountId, int amount).
```

Identify another variation of the WithdrawCash() method that takes a different parameter list that may be a useful variation of the method above.

### Feedback 1

An alternative method also used to withdraw cash could be WithdrawCash(String accountId) where no specified amount is provided but by default £100 is withdrawn.

These methods essentially perform the same operation but by overloading this method we have made the system more flexible – users now have a choice. They can specify the amount of cash to be withdrawn or they can accept the default sum specified.

Overloading methods don't just provide more flexibility for the user they also provide more flexibility for programmers who may have the job of extending the system in the future and thus overloading methods can make the system more future proof and robust to changing requirements.

Constructors can be overloaded as well as ordinary methods.

### Activity 2

Go online to msdn.microsoft.com. This is the website for the Microsoft Developer Network and the contains details of the Application Programmer Interface (API) for the .NET framework (it also contains much more).

Follow the link for the 'Library'

Then look for the link to '.NET Framework Class Library'. This maybe under a section headed 'Development Tools and Languages' or you may need to go though intermediate steps to get to this e.g. via '.NET development and .NET Framework 4'.

When you get into the '.NET Framework Class Library' you should see all of the namespaces (packages) in the .NET framework, on the left, with a brief description of each, on the right.

Follow the link for

System (one of the core namespaces) and then  
String (one of the core classes)

The direct link for this is

<https://docs.microsoft.com/en-gb/dotnet/api/system.string?>

Another way of getting to this is to enter the following search terms into a web browser 'msdn c# String class'

If you have the option then select the C# syntax version...remembering that the .NET framework API is available to programmers in several languages.

Next, look for at the String class documentation to find out how many constructors exist for this class.

## Feedback 2

In .NET version 4 the String class specifies 8 different constructors. They all have the same method name 'String' of course but they can all be differentiated by the different parameters these methods require.

One of these constructors takes and array of characters, a starting position and a length and creates a String object using this subset of characters taken from this array.

`String(Char[], Int32, Int32)`

Initializes a new instance of the String class to the value indicated by an array of Unicode characters, a starting character position within that array, and a length.

Another requires a pointer to an array of characters and creates a new String object that is a copy of the original.

`String(Char*)`

By massively overloading the String constructor the creators of this class have provided flexibility for other programmers who may wish to use these different options when they create strings.

We can make our programs more adaptable by overloading constructors and other methods. Even if we don't initially use all of the different constructors, or methods, by providing them we are making our programs more flexible and adaptable to meet changing requirements.

### Activity 3

Still looking at the String class in the API documentation find other methods that are overloaded.

### Feedback 3

There are a few methods that are not overloaded but most are. These include:

Compare(), CompareTo(), Concat(), Join(), Split() and many others.

Looking at the different Substring() methods, we see that we can find a substring by either specifying the starting point alone or by specifying starting point and length.

Substring(Int32)

Retrieves a substring from this instance. The substring starts at a specified character position and continues to the end of the string.

Substring(Int32, Int32)

Retrieves a substring from this instance. The substring starts at a specified character position and has a specified length.

When we use the Substring() method, the CLR engine will select the correct implementation of this method at run time, depending upon whether or not we have provided one or two parameters.

## 6.3 SUMMARY

Method overloading is the name given to the concept that several methods may exist that essentially perform the same operation and thus have the same name.

The CLR engine distinguishes these by looking at the parameter list. If two or more methods have the same name, then their parameter list must be different.

At run time, each method call (which may be ambiguous) is resolved by the CLR engine by looking at the parameters passed and matching the data types with the method signatures defined in the class.

By overloading constructors, and ordinary methods, we are providing extra flexibility to the programmers who may use our classes in the future. Even if these are not all used initially, providing these can help make the program more flexible to meet changing user requirements.

# 7 OBJECT ORIENTED SOFTWARE ANALYSIS AND DESIGN

## Introduction

This chapter will teach rudimentary analysis and modelling skills through practical examples, leading the reader to an understanding of how to get from a preliminary specification to an Object Oriented architecture.

## Objectives

By the end of this chapter you will be able to...

- Analyse a requirements description
- Identify items outside the scope of the system
- Identify candidate classes, attributes and methods
- Document the resulting Object Oriented architecture

This chapter consists of twelve sections:

- 1) Requirements Analysis
- 2) The Problem
- 3) Listing Nouns and Verbs
- 4) Identifying Things Outside The Scope of The System
- 5) Identifying Synonyms
- 6) Identifying Potential Classes
- 7) Identifying Potential Attributes
- 8) Identifying Potential Methods
- 9) Identifying Common Characteristics
- 10) Refining Our Design using CRC Cards
- 11) Elaborating Classes
- 12) Summary

## 7.1 REQUIREMENTS ANALYSIS

The development of any computer program starts by identifying a need:

- An engineer who specialises in designing bridges may need some software to create three dimensional models of the designs so people can visualise the finished bridge long before it is actually built.
- A manager may need a piece of software to keep track of personnel, what projects they are assigned to, what skills they have and what skills need to be developed etc.

But how do we get from a ‘need’ for some software to an Object Oriented software design that will meet this need?

Some software engineers specialise in Requirement Analysis which is the task of clarifying exactly what is required of the software. Often this is done by iteratively performing the following tasks:

- 1) Interviewing clients and potential users of the system to find out what they say about the system needed.
- 2) Documenting the results of these conversations.

- 3) Identifying the essential features of the required system.
- 4) Producing preliminary designs (and possibly prototypes of the system).
- 5) Evaluating these initial plans with the client and potential users.
- 6) Repeating the steps above until a finished design has evolved.

Performing requirements analysis is a specialised skill that is outside the scope of this text but here we will focus on steps three and four above i.e. given a description of a system how do we convert this into a potential Object Oriented design.

Producing simple and elegant designs is important if we want the software to work well and be easy to develop. However, identifying good designs from weaker designs is not simple and experience is a key factor.

While we can hope to develop preliminary design skills, you will only become good at this with experience.

A novice chess player may know all the rules but it takes experience to learn how to choose good moves from bad moves. Experience is essential to becoming a skilled player. Similarly, experience is essential to becoming skilled at performing user requirements analysis and in producing good designs.

Here we will attempt to develop rudimentary skills. Chapter 8 will focus more on good design principles and how to apply them and in Chapter 13 you will have the opportunity to apply these skills and test your understanding.

Starting with a problem specification we will work through the following steps:

- Listing Nouns and Verbs
- Identifying Things Outside The Scope of The System
- Identifying Synonyms
- Identifying Potential Classes
- Identifying Potential Attributes
- Identifying Potential Methods
- Identifying Common Characteristics
- Refining Our Design using CRC Cards
- Elaborating Classes

By doing this we will be able to take a general description of a problem and generate a feasible, and hopefully elegant, Object Oriented design for a system to meet these needs.

## 7.2 THE PROBLEM

The problem for which we will design a solution is as follows: ‘Develop a small management system for an athletic club organising a marathon.’

For the purpose of this exercise we will assume preliminary requirements analysis has been performed and by interviewing the club managers and the workers, who would use the system, the following textual description has been generated.

*The ‘GetFit’ Athletic Club are organizing their first international marathon in the spring of next year. A field comprising both world-ranking professionals and charity fundraising amateurs (some in fancy dress!) will compete on the 26.2 mile route around an attractive coastal location. As part of the software system which will track runners and announce the results and sponsorship donations, a model is required which represents the key characteristics of the runners (this will be just part of the finished system).*

*Each runner in the marathon has a number. A runner is described as e.g. “Runner 42” where 42 is their number. They finish the race at a specified time recorded in hours, minutes and seconds. Their result status can be checked and will be displayed as either “Not finished” or “Finished in hh:mm:ss”.*

*Every competitor is either a professional runner or an amateur runner.*

*Further to the above, a professional additionally has a world ranking and is described as e.g. “Runner 174 (Ranking 17)”.*

*All amateurs are fundraising for a charity so each additionally has a sponsorship form. When an amateur finishes the race they print a collection list from their sponsorship form.*

*A sponsorship form has the number of sponsors, a list of the sponsors, and a list of amounts sponsored. A sponsor and amount can be added, and a list can be printed showing the sponsors and sponsorship amounts and the total raised.*

*A fancy dress runner is a kind of amateur (with sponsorship etc.) who also has a costume, and is described as e.g. “Runner 316 (Yellow Duck)”.*

### 7.3 LISTING NOUNS AND VERBS

The first step in analysing the description above is to identify the nouns and verbs:

- The nouns indicate entities, or objects, some of these will appear as classes in the final system and some will appear as attributes.
- The verbs indicate actions to be performed some of these will appear in the final system as methods.

Nouns and verbs that are plurals are listed in their singular form (e.g. ‘books’ becomes ‘book’) and noun and verb phrases are used where the nouns\verbs alone are not descriptive enough e.g. the verb ‘print’ is not as clear as ‘print receipt’.

#### Activity 1

Look at the description above and list five nouns and five verbs (use noun and verb phrases where appropriate).

### Feedback 1

The list below is a fairly comprehensive list of the nouns and verbs, not just the first five.

Nouns: GetFit Athletic Club, field, world ranking professional, fund-raising amateur, fancy dress, 26.2 mile route, coastal location, software system, runner, result, sponsorship donation, model, key characteristic, a number, time, result status, competitor, professional runner, amateur runner, world ranking, charity, sponsorship form, collection list, sponsor, sponsorship amount, total raised, costume.

Verbs: Organise, marathon, compete, track runners, announce results, describe (runner), finish race, specify time, check status, display status, describe (professional), print collection list, add (sponsor and amount), print list, describe (fancy dress runner).

## 7.4 IDENTIFYING THINGS OUTSIDE THE SCOPE OF THE SYSTEM

An important part in designing a system is to identify those aspects of the problem that are not relevant or outside the scope of the system. Parts of the description may be purely contextual, i.e. for general information purposes, and thus not something that will directly describe aspects of the system we are designing.

Furthermore, while parts of the description may refer to tasks that are performed by users of the system and thus describe functions that need to be implemented within the system, other parts may describe tasks performed by users while not using the system and thus don't describe functions within the system.

By identifying things in the description that are not relevant we keep the problem as simple as possible.

### Activity 2

Look at the description of the problem and the list of nouns and verbs and identify one of each that is outside the scope of the system.

### Feedback 2

Most of the first paragraph is contextual and does not describe functionality we need to implement within the system. We also need to look at other parts of the description to identify parts that are not relevant.

Things outside the scope of the system...

Nouns:

- GetFit Althetic Club – this is the client for whom the system is being developed. It is not an entity we need to model within the system.
- Coastal location – the location of the run is not relevant to the functionality of the system as described. Again we do not need to model this as an object within the system.
- Software system – this is the system we are developing as a whole. It does not describe an entity within the system.

Verbs:

- Organise – this is an activity done by members of the athletic club, these may be users of the system but this is not an activity that they are using the system for.
- Marathon – this is what the runners are doing. It is not something the system needs to do.

Note: 'Finish race' is something that a runner does. However when this happens their finish time must be recorded in the system. Therefore, this is not outside the scope of the system.

## 7.5 IDENTIFYING SYNONYMS

Synonyms are two words that have the same meaning. It is important to identify these in the description of the system. Failure to do so will mean that one entity will be modelled twice which will lead to duplication and confusion.

### Activity 3

Look at the list of nouns and verbs and identify two synonyms, one from the list of nouns and one from the verbs.

### Feedback 3

Synonyms

Nouns:

- world ranking professional = professional runner
- fund-raising amateur = amateur runner
- runner = competitor

Note: Runner is not a synonym of professional runner as some runners are amateurs.

Verbs:

- marathon = compete
- check status = display status
- print collection list = print list
- finish race = record specified time

## 7.6 IDENTIFYING POTENTIAL CLASSES

We have simplified the problem by identifying aspects that are outside the scope of the system and by identifying different parts of the description that are in reality describing the same entities and operations. We can now start to identify potential classes in the system to be implemented.

Some nouns will indicate classes to be implemented and some will indicate attributes of classes.

Good Object Oriented design suggests that data and operations should be packaged together – thus classes represent complex conceptual entities for which we can identify associated data and operations (or methods).

Simple entities, such as an address, have associated data but no operations and thus these can be stored as simple attributes within a related class.

### Activity 4

Look at the list of nouns above and identify five that could become classes.

### Feedback 4

Nouns that could indicate classes are:

- Runner (or Competitor)
- Amateur (or Amateur Runner)
- Professional (or similar)
- FancyDresser (or FancyDressAmateur or similar)
- Sponsorshipform

## 7.7 IDENTIFYING POTENTIAL ATTRIBUTES

Having identified potential classes the other nouns could be used to identify attributes of those classes.

### Activity 5

Look at the list of nouns and identify one that could become an attribute of the class 'Runner' and one for the class 'FancyDresser'.

### Feedback 5

Nouns that could become attributes...

For Runner:

number,  
resultStatus ie. finished (boolean)  
time (hours, minutes, seconds)

For FancyDresser:

costume (String)

Of course we need to identify all of the attributes for all of the classes.

## 7.8 IDENTIFYING POTENTIAL METHODS

Having identified potential classes, we can now use the verbs to identify methods of those classes.

### Activity 6

Look at the list of verbs and identify one that could become a method of the class 'Runner' and one for the class 'FancyDresser'.

### Feedback 6

Verbs that could become methods...

For Runner:

describe (this will actually become an overridden version of ToString())  
finishRace  
displayStatus

For FancyDresser:

describe (ToString() will need to be overridden again to encompass the description of the costume)

Of course, we need to identify all of the methods for all of the classes.

## 7.9 IDENTIFYING COMMON CHARACTERISTICS

Having identified the candidate classes with associated attributes and methods we can start structuring our classes into appropriate inheritance hierarchies by identifying those classes with common characteristics.

### Activity 7

Look at the list of classes below and place four into an appropriate inheritance hierarchy. Identify the one class that would not fit into this hierarchy.

- Runner
- Amateur
- Professional
- FancyDresser
- Sponsorshipform

### Feedback 7

The most general class i.e. the one at the top of the inheritance tree is 'Runner'. Amateur and Professional are subclasses of 'Runner' and FancyDresser is a specific type of Amateur and is, hence, a subclass of Amateur.

We can fit these into an inheritance hierarchy because these classes are all related by an *is\_a* relationship. A FancyDresser *is\_an* Amateur which in turn *is\_a* Runner. A Professional *is\_a* Runner as well.

A SponsorshipForm is not a type of Runner and hence does not fit into this hierarchy. This class will be related to one of the other classes by some form of an association.

Looking at the description, we can see that not all runners have a sponsorship form, only amateurs who are running for charity. There is consequently an association between Amateur and SponsorshipFrom. Of course, FancyDressers inherit the attributes defined in Amateur and hence they automatically have a SponsorshipForm.

## 7.10 REFINING OUR DESIGN USING CRC CARDS

Having identified the main classes in our system and the attributes and methods of these classes, we could now proceed to refine these designs by defining the data types and other small details. We could then document this information on a UML diagram and program the system.

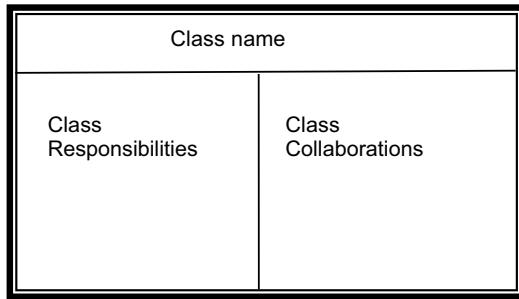
However, in a real world system the problem would be larger and less well-defined than the problem we are working on here and the analysis and refinement of design would therefore be a longer more complex process that we can realistically simulate.

As real world problems are more complex, our initial designs are unlikely to be perfect. Therefore it makes sense to check our designs and to resolve any potential problems before turning these designs into a finished system.

One method of checking our designs is to document our designs using CRC cards and to check if these work by role-playing different scenarios.

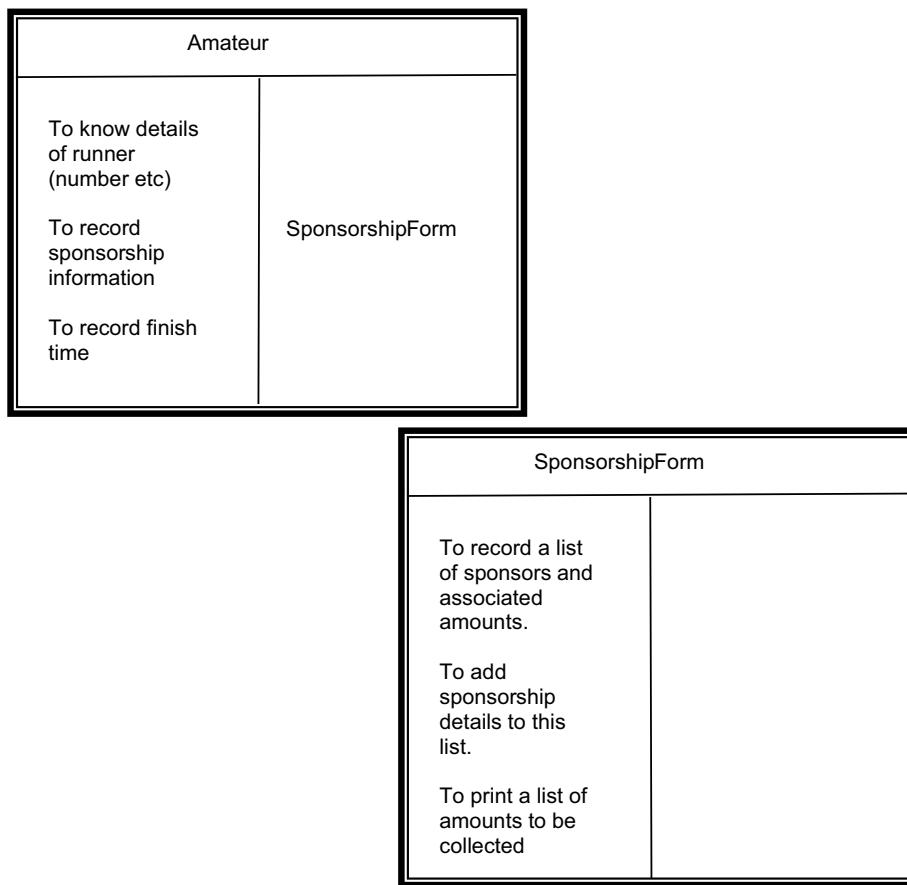
CRC cards are not the only way of doing this and are not part of the UML specification.

CRC stands for Class, Responsibilities and Collaborations. A CRC card is set out below and is made up of three panes with the class responsibilities shown on the left and the collaborations shown on the right.



Responsibilities are the things the class needs to know about (ie the attributes) and the things it should do (ie. the methods) though on a CRC card these are not as precisely defined as on a UML diagram. The collaborations are other classes that this class must work with in order to fulfil its responsibilities.

The diagram below shows CRC cards developed for two classes in the system.



Having developed CRC cards we can role-play a range of scenarios in order to check the system works ‘on paper’. If not we can amend the design before getting into the time consuming process of programming a flawed plan.

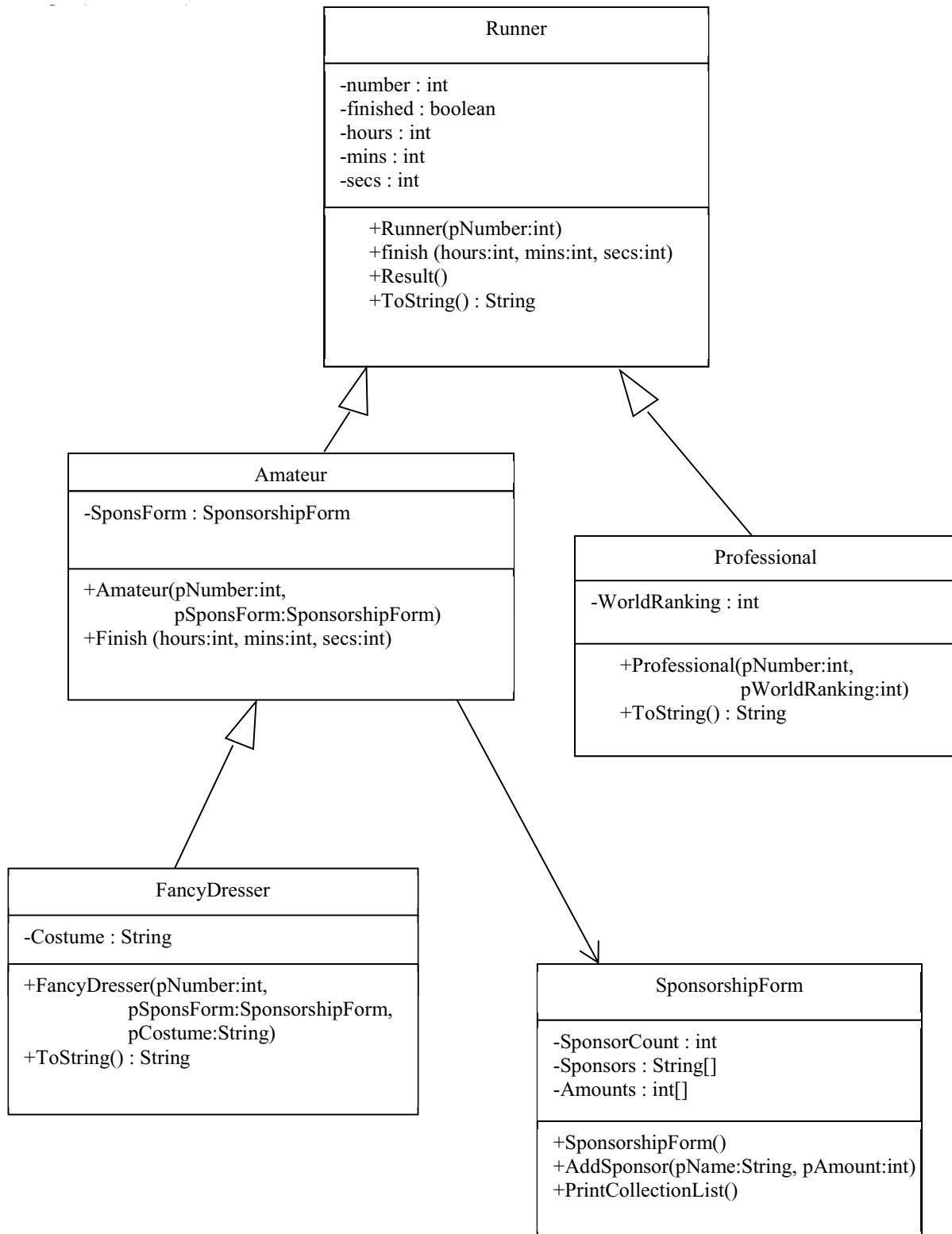
One sample scenario would be when a runner gets an additional sponsor. In this case by looking at the CRC cards above, a Runner is able to record sponsorship information in collaboration with the SponsorshipForm class. The SponsorshipForm class records a list of sponsors and can add additional sponsors to this list. So this part of the design seems to work.

Testing out a range of scenarios may highlight flaws in our system design that we can then fix – long before any time has been wasted by programming weak designs.

## 7.11 ELABORATING CLASSES

Having identified the classes in our system design and documented and tested these using CRC cards, we can now elaborate our CRC cards and document our classes using a UML class diagram. To do this we need to take our general specification documented via CRC cards and resolve any ambiguities e.g. exact data types.

Having elaborated our CRC cards we can now draw a class diagram for proposed design (see below):



## 7.12 SUMMARY

Gathering User Requirements is an essential stage of the software engineering process (and outside the scope of this text).

Turning a complex requirements specification into an elegant simple Object Oriented architectural design is a skilled task that requires experience. However, a good starting point is to follow a simple process set out in this chapter.

Through a sequence of tasks we have seen how to analyse a textual description of a problem. We have:

- Looked for aspects of the description that are outside the scope of the system.
- Identified where the description refers synonymous items using different words.
- Used the nouns and verbs to identify potential classes, attributes and methods.
- Looked at the classes to identify potential inheritance hierarchies and to identify other relationships between classes (e.g. associations).
- Documented the resulting classes using CRC cards and tested the validity of our design by role-playing a range of scenarios and amending our designs as appropriate.
- Finally elaborated these details and documented the results using a class diagram.

We will consider good design principles in the next chapter and the design process, set out in this chapter, will be demonstrated again using the case study described in Chapter 14.

# 8 INTRODUCING THE SOLID DESIGN PRINCIPLES

## Introduction

In Chapter 1 we claimed that the Object Oriented approach to modelling leads to software that has:

- Better maintainability (more comprehensible, less fragile software)
- Better reusability (classes, encapsulated components, can be used in other systems).

But is this always true or only when systems are designed well?

In Chapter 5, we saw how to use polymorphism effectively leading to systems that are adaptable but is this all that is required?

The last chapter attempted to teach rudimentary analysis and modelling skills through practical examples, leading the reader to an understanding of how to get from a preliminary specification to an Object Oriented architecture but how can we be certain this architecture has a good design?

Indeed what is good design?

The aim of this chapter is to introduce the SOLID design principles and to demonstrate how these are applied to ensure the systems we produce are robust and easy to maintain.

This chapter contains practical exercises to help you learn how to apply these principles and Chapter 13 will give you a chance to double check your understanding of the theories explained here.

## Objectives

By the end of this chapter you will be able to understand what the SOLID design principles are and how to apply them.

This chapter consists of seven sections:

- 1) The SOLID design principles
- 2) The Single Responsibility Principle
- 3) The Open/Closed Principle
- 4) The Liskov Substitution Principle
- 5) The Interface Segregation Principle
- 6) The Dependency Inversion Principle
- 7) Summary

## 8.1 THE SOLID DESIGN PRINCIPLES

The SOLID design principles are as follows:

- Single Responsibility Principle**
- Open/Closed Principle**
- Liskov Substitution Principle**
- Interface Segregation Principle**
- Dependency Inversion Principle**

These principles are in essence very simple and based on the combined wisdom and experience of many clever software engineers. However, when clever people write things down they don't always phrase things in a simple to understand way and some of these principles perhaps look more complex than they actually are. Hopefully you will find the explanations and examples here simple to understand.

We will look at each of these principles in turn and try to show how easy they are to apply.

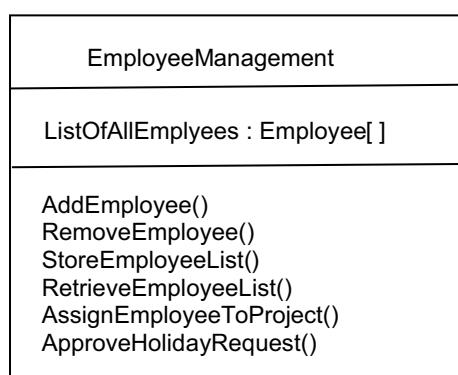
## 8.2 THE SINGLE RESPONSIBILITY PRINCIPLE

The Single Responsibility Principle, the first of the SOLID principles, states that each class should only have one responsibility over a single part of system functionality. A class should therefore have only one reason to change.

If a class has more than one responsibility then the responsibilities become coupled and changes to one responsibility may impair the classes' ability to meet its other responsibilities.

Let's work through an example:

Consider an 'EmployeeManagement' class for a company.... The class is given the responsibility for maintaining a list of all employees, storing and retrieving employee details, assigning employees with appropriate skills to projects and approving an employee's holiday request. This class is documented in the class diagram below:



At first glance this class perhaps seems reasonable as all the responsibilities of this class relate to the management of the employees however complications are caused by the multiple roles this class fulfils.

This class breaks the Single Responsibility Principle by fulfilling multiple roles...

Firstly, it maintains a list of all employees adding and removing employees from this list. It stores and retrieves this list so that employees can be assigned to projects and holidays can be approved. But what happens when the storage and retrieval function is changed?

Assume the company decides to change the way the data is stored...perhaps they want to store the data in the cloud and at the same time enhance the privacy and data protection mechanisms.

Changing the way the data is stored should have no impact on the other functions of this class but the design above could cause real difficulties for the programmer.

If the data is stored in the cloud the retrieval could now fail if access to the cloud is interrupted. To protect against this a programmer could include a Boolean variable within the class to record the success or failure of the data retrieval process (as shown in the revised design below).

EmployeeManagement
ListAllEmployees : Employee[ ] <b>RetrievalSucess : boolean</b>
AddEmployee() RemoveEmployee() StoreEmployeeList() RetrieveEmployeeList() AssignEmployeeToProject() ApproveHolidayRequest()

In a highly maintainable system, small changes to one component should not cause complications to other functions. However, complications can now exist with the system shown above. Now when assigning employees to projects, the system must deal with a potentially incomplete/faulty employee list (if the retrieval process was interrupted). Similarly the holiday approval process could now fail if an unknown member of staff requests a holiday.

The AssignEmployeeToProject() and ApproveHolidayRequest() methods will now need to be amended to check that the data retrieval process was successful. This causes more work for the programmer and increases the chance of errors in the system. Changes to the data storage/retrieval process should not have required changes to these other components of the system.

**These complications have arisen because the Single Responsibility Principle was broken.**

**The more complex and coupled components are the more complex and costly it is to make changes and it becomes more likely you will introduce errors into the system.**

Software Engineers have for many years recognised that highly cohesive systems are more maintainable, easier to fix and easier to understand. The highest level of cohesion being where classes only have one responsibility.

Cohesion is strongly related to the concept of coupling.... A highly cohesive system with only require loosely coupled components which is good. A strongly coupled system is poor as it is more likely that changes to one component will require changes to other parts of the system.

### Activity 1

Follow the links to Wikipedia for an explanation of software cohesion and coupling...

[https://en.wikipedia.org/wiki/Cohesion\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Cohesion_(computer_science))  
[https://en.wikipedia.org/wiki/Coupling\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Coupling_(computer_programming))

### Feedback 1

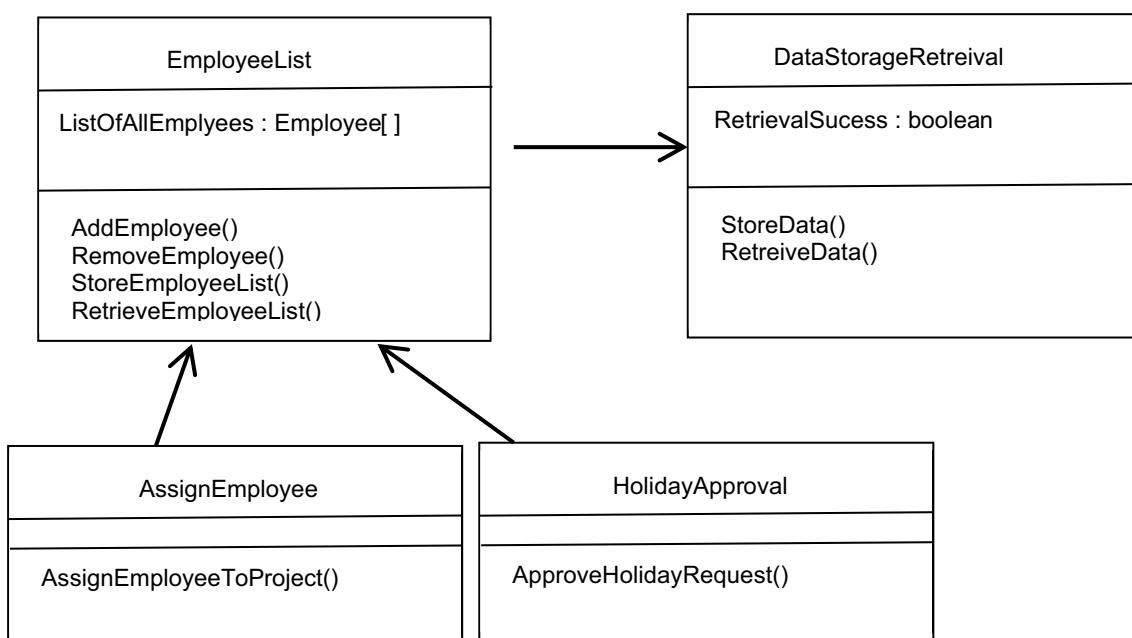
There is no feedback for this activity.

The Single Responsibility Principle takes the idea of a cohesive system to its ultimate conclusion by saying classes should have only one role – one area of responsibility.

The system as designed above, at first glance, looks cohesive as the class deals only with employee functions but it certainly breaks the Single Responsibility Principle and this makes the class more difficult to maintain and introduces potential errors into the system.

To follow the Single Responsibility Principle this class needs to be broken into separate classes – each with one clearly defined area of responsibility, e.g. maintaining the list of employees. Other classes should deal with the other responsibilities of the system.

A revised design is shown below...



In the design above, one class has the responsibility for storing and retrieving data. Another class maintains the list of employees. Another class assigns employees to projects and another approves holiday requests. The four classes do work together and each invokes methods in the other classes when required.

The classes that are responsible for assigning staff to projects and approving holidays maintain a relationship (association or dependency) with the EmployeeList class as they need to use this list when fulfilling their responsibility.

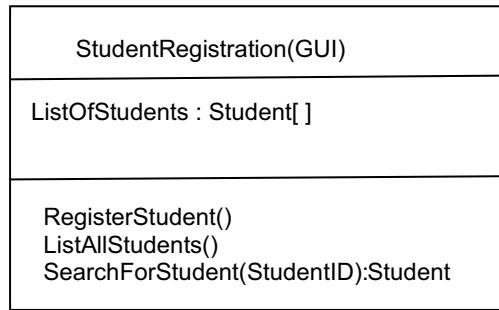
By splitting one large class into four smaller classes each class becomes simpler to understand and simpler to change.

More importantly changes to one class are now far less likely to require changes to other components.

Now the data storage and retrieval methods can be updated without needing to make any changes to the other components in the system.

### Activity 2

Consider the design for a University Student Registration system shown below...



### **Activity 2 Continued**

The system has a main form, a GUI, which is used to enter details of new students.

As each new student enters their details, they are added to the list of all students...which is kept within the main form.

The form also displays the list of all students and allows an individual student to be retrieved from the list and their details to be displayed.

Does this design break the Single Responsibility Principle?

Would this cause complications if the University wanted to provide an alternative interface for a student to register e.g. they could perhaps register via a telephone\spoken interface?

Spend a few minutes and consider what changes you would make to this design.

## Feedback 2

This design does break the Single Responsibility Principle – the form has multiple functions... enter student details, display all students and display the details of an individual student. It also maintains the list of students.

This would cause complications if the University wanted to provide an alternative interface, e.g. a spoken interface, as the class responsible for the alternative interface would need to access data held on the GUI.

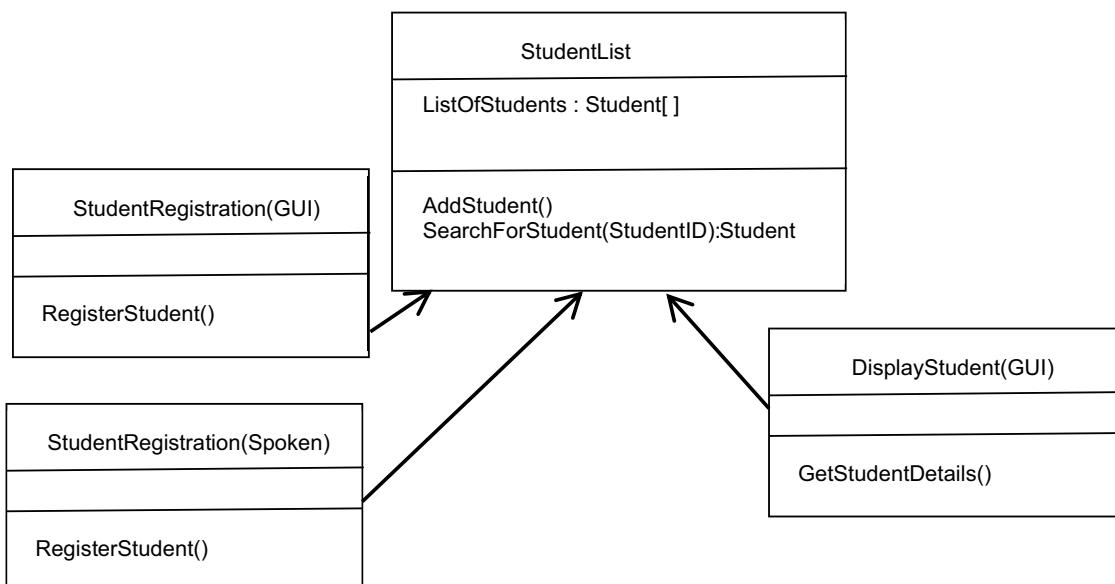
There would also be lots of other complications...if the part of the form used for entering the student's details is expanded this may well require changes to the part of the form displaying the list of all students. These two parts of the form perform different functionality but are tightly coupled.

Putting these different display functions on the same form is very obviously a bad design and I'm sure you would not consider doing this but it illustrates a deeper problem. When a class has multiple responsibilities, changes to one part of the class can very easily require changes to other parts of the class making the system overall more difficult to change.

One very simple improvement in this system is to separate core system functionality from the GUI...the GUI is the interface into the system it should not also maintain the list of students. Another improvement is to split the GUI into parts each with its own area of responsibility.

Thus we should split this system into multiple classes giving each class just one responsibility.

A revised design is shown below...



This design isn't perfect as we have not yet considered the other SOLID principles but it is much improved.

As the student list is maintained in a separate class we can now easily add different interfaces to the system. Now the registration GUI only takes in the students details and invokes methods in the StudentList class. It now become very simple to provide an alternative system interface (spoken, console, mobile etc.) that provides an alternative way of entering data and invokes the functionality implemented in the StudentList class.

Additionally, as the GUI has now been split into separate components, it is also now very unlikely that changes to one part of the GUI will impact needlessly on the design for the other parts.

...much improved!

### 8.3 THE OPEN/CLOSED PRINCIPLE

The second SOLID design principle is the Open/Closed Principle.

The Open and Closed Principle suggests that software entities, such as classes and methods, should be open for extension but closed for modification.

When a simple change results in a ripple effect throughout a program this indicates a rigid design. For example, if we change how functions or libraries work this inevitably means we must make changes to any programs depending on those functions or libraries.

Changes required to programs that use these libraries could be widespread and making these changes could be difficult and very time consuming (therefore very costly).

Libraries and components of systems do need to change to respond a client's changing needs and new software standards etc. The Open and Closed Principle recommends that these changes are achieved by adding new code (extension) rather than modifying existing code i.e. software should be open to extension/closed for changes. Thus, systems that depend on extended software can continue to function using the old parts without change.

The use of old methods that don't conform to current standards may now be discouraged but as the old methods are not actually changed, software that uses them still works.

This Open and Closed Principle applies to inherited classes and is strongly linked to the third SOLID design principle: the Liskov Substitution Principle.

## 8.4 THE LISKOV SUBSTITUTION PRINCIPLE

In Chapter 5 we looked at how to use polymorphism effectively. Polymorphism is a powerful mechanism that allows systems to be extended to meet a client's changing needs without causing a ripple effect i.e. without requiring changes to large parts of the current system.

However for polymorphism to work subtype objects must be substitutable for a super type object. This is the Liskov Substitution Principle – a subtype object must be substitutable for a super type object.

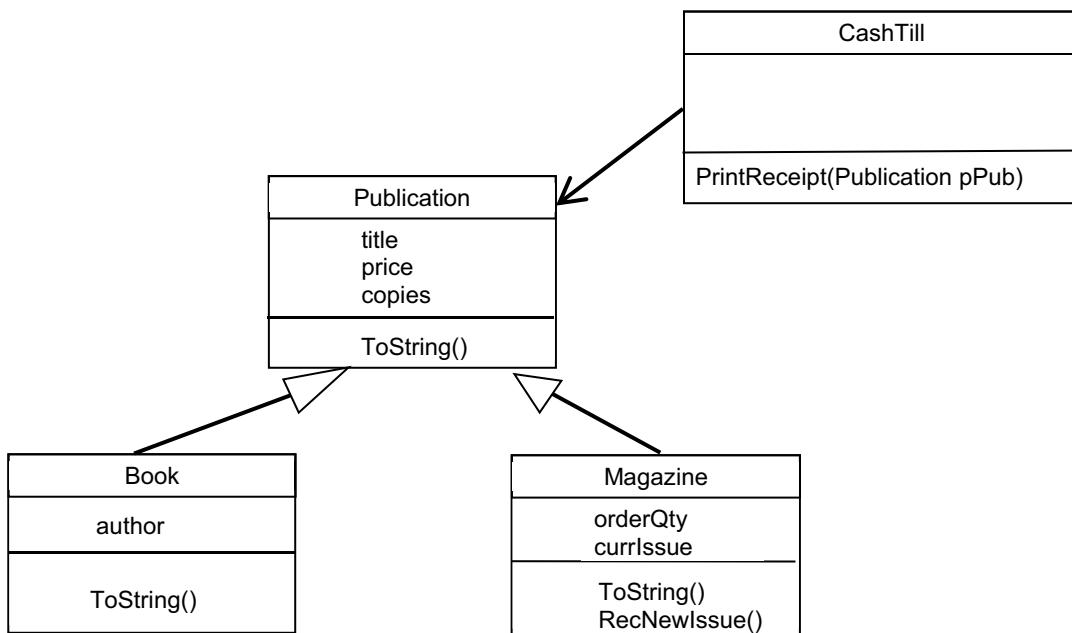
If ChildClass is a subtype of ParentClass, then objects of type ParentClass may be replaced by type ChildClass.

The creation of subtypes which can be substituted for their base type means that subclasses cannot modify functionality defined in the superclass, i.e. they are closed for change, but they can include additional functionality, i.e. be open for extension.

We can still override methods but we cannot change the inherent functionality of an overridden method – we can only make that method more appropriate for the sub type object.

### Activity 3

Consider the system as shown below.



### Activity 3 Continued

The system above includes an inheritance hierarchy where the **ToString()** method has been overridden in the **Publication**, **Book** and **Magazine** classes and an additional method **RecNewIssue()** has been added to the **Magazine** class.

The **ToString()** method has been overridden such that each different type of object will return a different description of itself. This will be used by the **CashTill** when printing receipts.

The code for overriding **ToString()** is shown below.

### In Publication

```

public override String ToString()
{
    return title;
}
  
```

## In Book

```
public override String ToString()
{
    return base.ToString() + " by " + author;
}
```

## In Magazine

```
public override String ToString()
{
    return base.ToString() + " (" + currIssue + ")";
}
```

### Activity 3 Continued

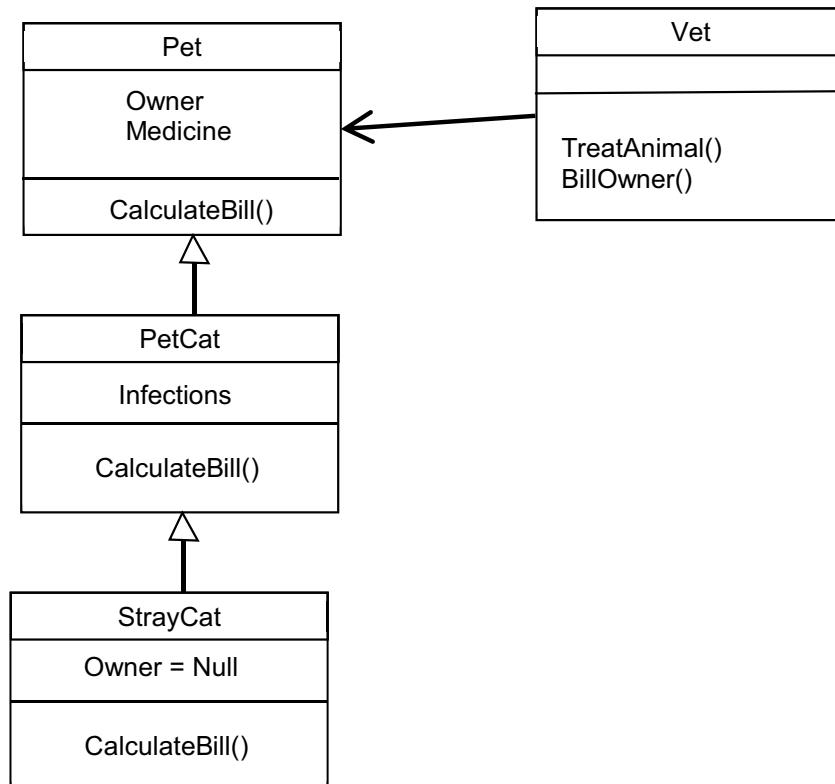
Does the system above follow the Open and Closed Principle?  
Are subclass objects substitutable for super type objects?

### Feedback 3

Feedback on this activity will be provided shortly.

### Activity 4

Now consider a different system (shown below).



In the system above CalculateBill() has been overridden in the PetCat class so that the bill takes account of infections that cats are prone to getting.

The method has been overridden again in StrayCat.

As StrayCats do not have an owner, the Owner attributed has been overridden and set to Null, the CalculateBill() has also been changed as the vet cannot send a bill to the owner... the CalculateBill() method instead calculates the cost of the treatments that will be used as a tax deduction.

No additional methods have been defined in the subclasses.

#### **Activity 4 Continued**

Does this vet system follow the Open and Closed Principle?  
Are subclass objects substitutable for super type objects?

#### **Feedback 3 and 4**

In the publication/cash till system, shown above, an additional method was defined in the magazine class but the overridden methods did not have different functionality – the string returned was just a better description of the subclass object.

This therefore followed the Open/Closed Principle. The magazine class was open for extension – it has extra functionality – but none of the existing functionality was changed.

The book and magazine objects were therefore fully substitutable for a publication. Thus, whenever the cash till wanted a description of the publication sold it did not matter whether a book or a magazine was provided in place of the publication.

A book and a magazine are both fully substitutable for the superclass object and hence polymorphism will work. New sub types can be created and sold without changing the cash till.

In the system for the vet, the CalculateBill() method is overridden in the PetCat class in an appropriate way and a PetCat is fully substitutable for a 'Pet'. However, problems arise with the StrayCat class.

Here the CalculateBill() method has been changed and is doing something quite different – it is no longer calculating a bill for the owner, it is instead calculating a tax deduction. Bad design is also indicated by the attempt to override the Owner attribute and set it to a null value. Inheritance means that the subclass must inherit all attributes and methods of the superclass.... As stray cats do not have an owner this is an attempt to pervert the inheritance process.

The Open/Closed Principle states that code should be open for extension but closed for modification but here we are trying to modify existing functionality in the StrayCat subclass. We are therefore going against the Open and Closed Principle.

**Critically stray cats are not substitutable for pets as they do not have owners so we are breaking the Liskov Substitution Principle.**

The problem becomes clear when the vet is given a stray cat to treat and tries to invoke the BillOwner() method on the stray cat object. This will not work as stray cats do not have an owner and the CalculateBill() method is doing something quite different.

Stray cats are not pet cats, they are not substitutable for pets and do not belong in this inheritance hierarchy.

**For polymorphism to work we must apply the Open and Closed Principle and the Liskov Substitution Principle.**

## 8.5 THE INTERFACE SEGREGATION PRINCIPLE

The Interface Segregation Principle states that we should split fat interfaces into several thin interfaces.

We will first consider what this means and then how to do this. Hopefully the examples below will make this simple to understand.

Remember, an interface defines methods that will be implemented in classes. We want defined methods that can be implemented by different sets of clients and, in doing so, we want to avoid methods that are included for the benefit of just one subclass.

If an interface defines methods, lots of methods, then we call it a ‘fat’ interface. If some of these methods are only needed in one class then every other class that implements this interface will be required to implement these methods. This is often not appropriate.

The solution is instead break the fat interface down into many thin, client-specific, interfaces.

### Activity 5

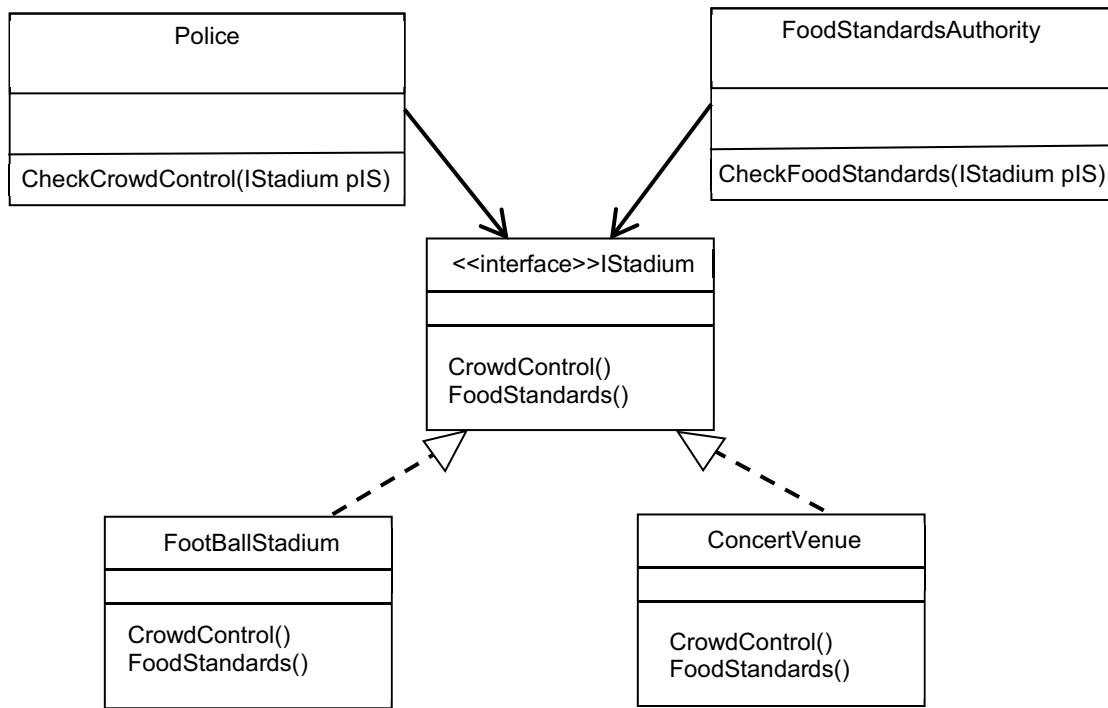
Consider the following scenario...

Venues, such as football stadiums and concert venues, which cater for large crowds need proper crowd control strategies and, as these venues sell food, they also need to implement food standards correctly.

Therefore, an interface is defined that specifies methods that must be implemented in appropriate classes. These methods are implemented in classes representing football stadiums, concert venues and other appropriate classes.

The Police and the governments Food Standards Authority can invoke the methods in any stadium to check that specific venues are implementing the correct standards properly.

This scenario is modelled in the diagram below...



### Activity 5 Continued

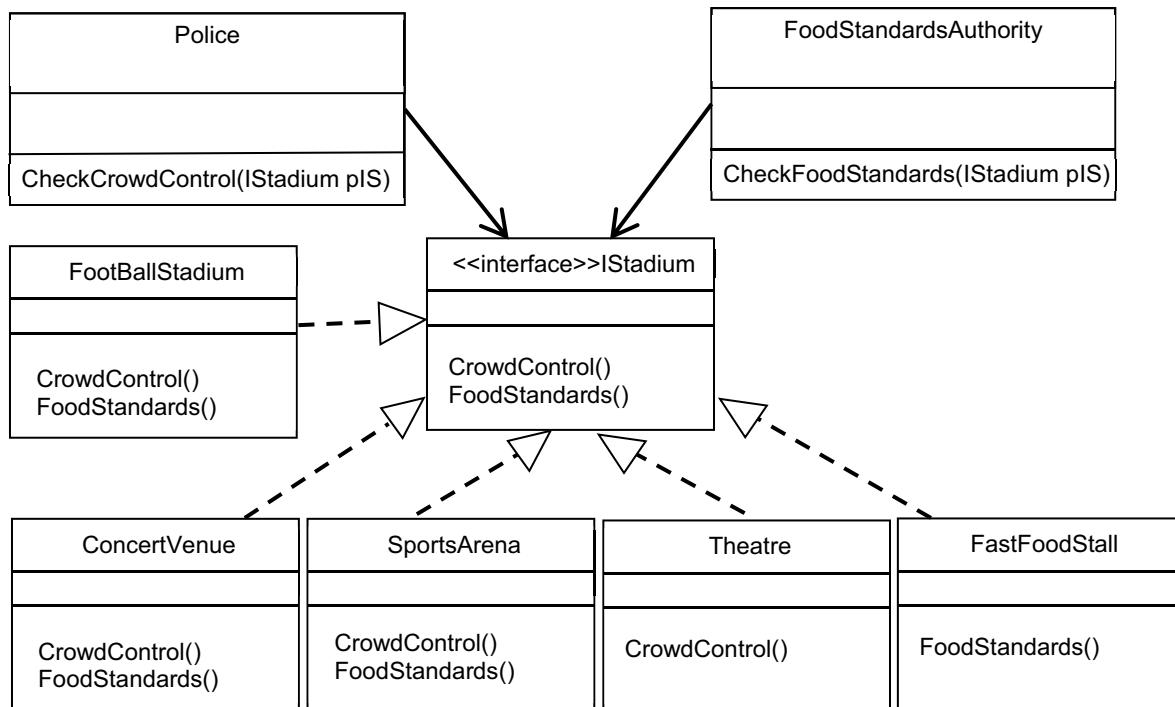
Now consider the following...

A new Sports Arena, which caters for large crowds and wants to sell food, needs to implement both methods and needs to be vetted by both the police and the Food Standards Authority.

A new theatre will accommodate crowds but does not want to sell food and therefore only wants to implement the crowd control method. This venue only needs to be vetted by the police.

A fast food stall only wants to sell food and has no need to implement crowds control strategies. This therefore only needs to be vetted by the Food Standards Authority.

An adapted model to cater for these additions is shown in the diagram below...but is new design OK?



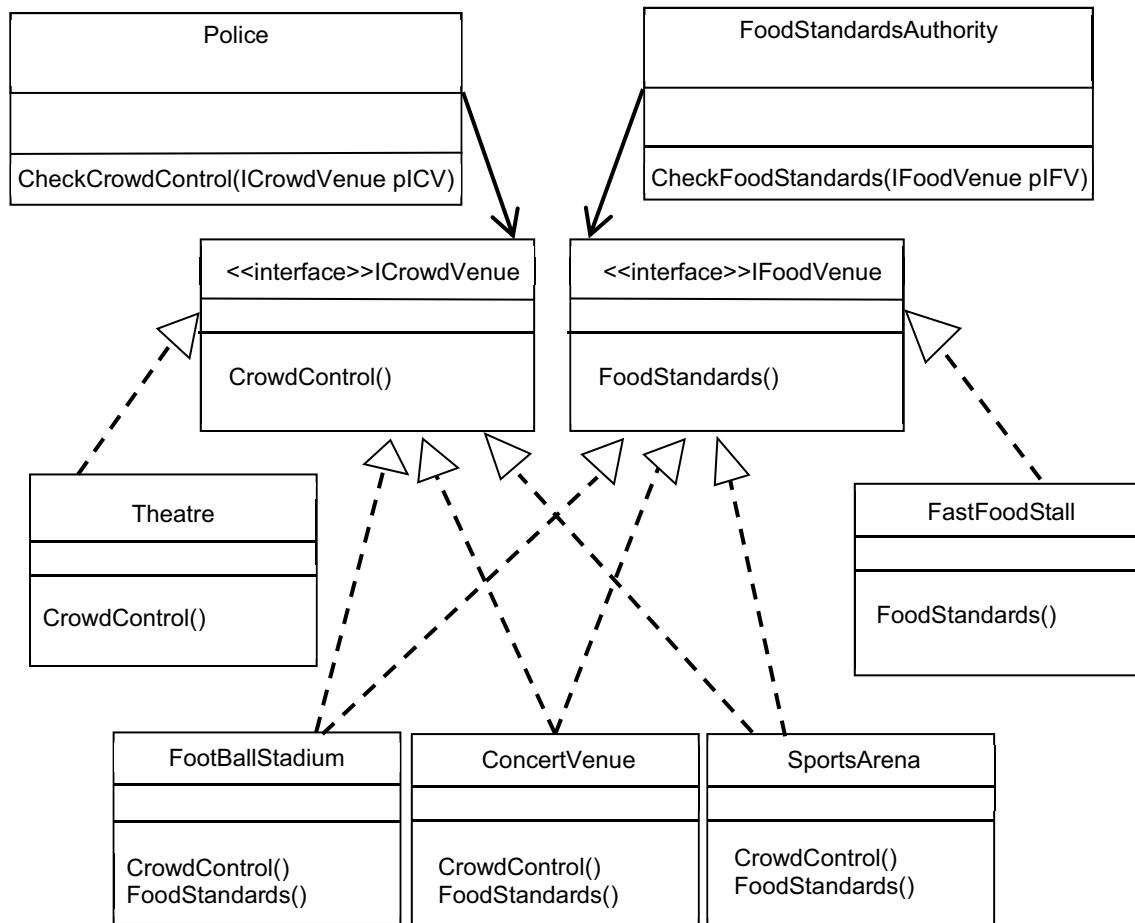
### Feedback 5

This is not a valid design. If a class implements an interface it must implement all of the methods in that interface.

According to this design, a Theatre should implement the FoodStandards() method defined in the interface and can thus be vetted by the Food Standards Authority even though it doesn't sell food and a FastFoodStall would need to implement crowd control strategies.

To fix this design, the fat 'IStadium' interface should be split into two thinner interfaces...one for all food vendors and one for all crowded venues.

This is shown on the diagram below.



### Feedback 5 Continued

With this revised design, the interface has been split into two separate interfaces. The Theatre class only implements the ICrowdVenue interface, these venues will not be vetted by the Food Standards Authority. Similarly a fast food stall only implements the IFoodVenue interface.

The classes FootballStadium, ConcertVenue and SportsArena can implement both interfaces i.e. both sets of functionality and can therefore be vetted by both the police and the Food Standards Authority.

Splitting interfaces, as shown in the example above, promotes code reuse and polymorphism.

### Activity 6

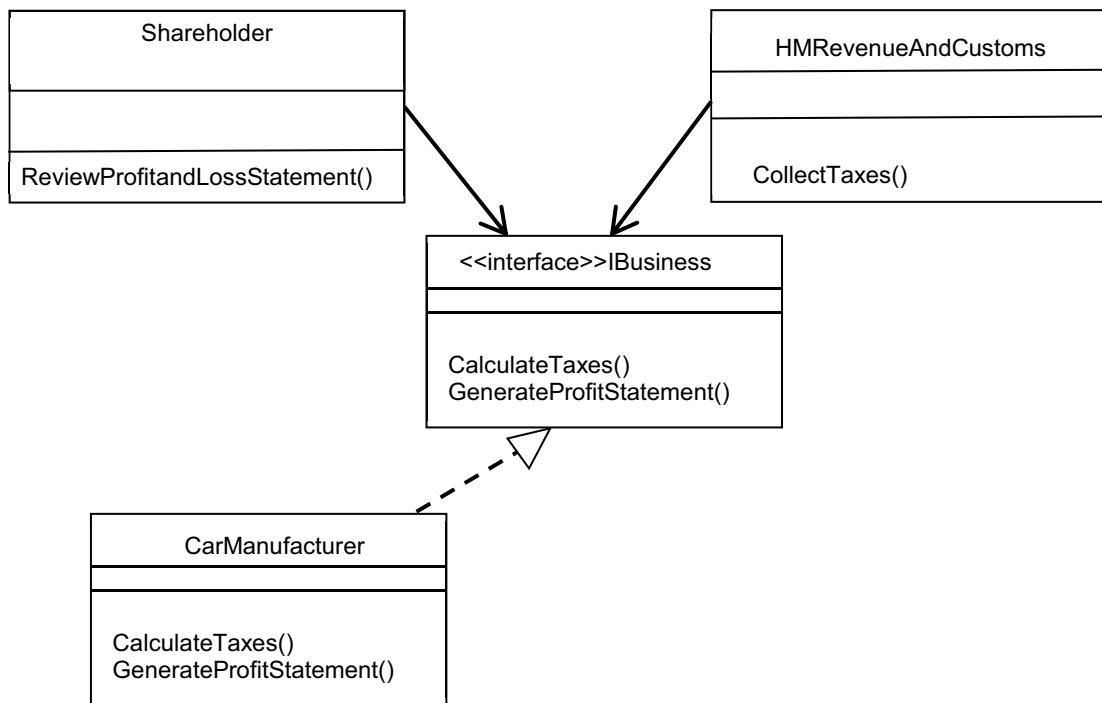
Consider the following scenario and the design below it and consider if that design can be improved.

Businesses often need to provide profit and loss statements for shareholders and all businesses need to calculate tax returns for the governments tax department – in the UK this is the 'HM Revenue & Customs' department.

Thus an interface is defined to specify appropriate methods to be implemented by any business.

This scenario, including an example class for a car manufacturer that will implement these methods, is modelled in the diagram below...but is this the best design?

Consider one way you could improve this design by applying the Interface Segregation Principle.



### Feedback 6

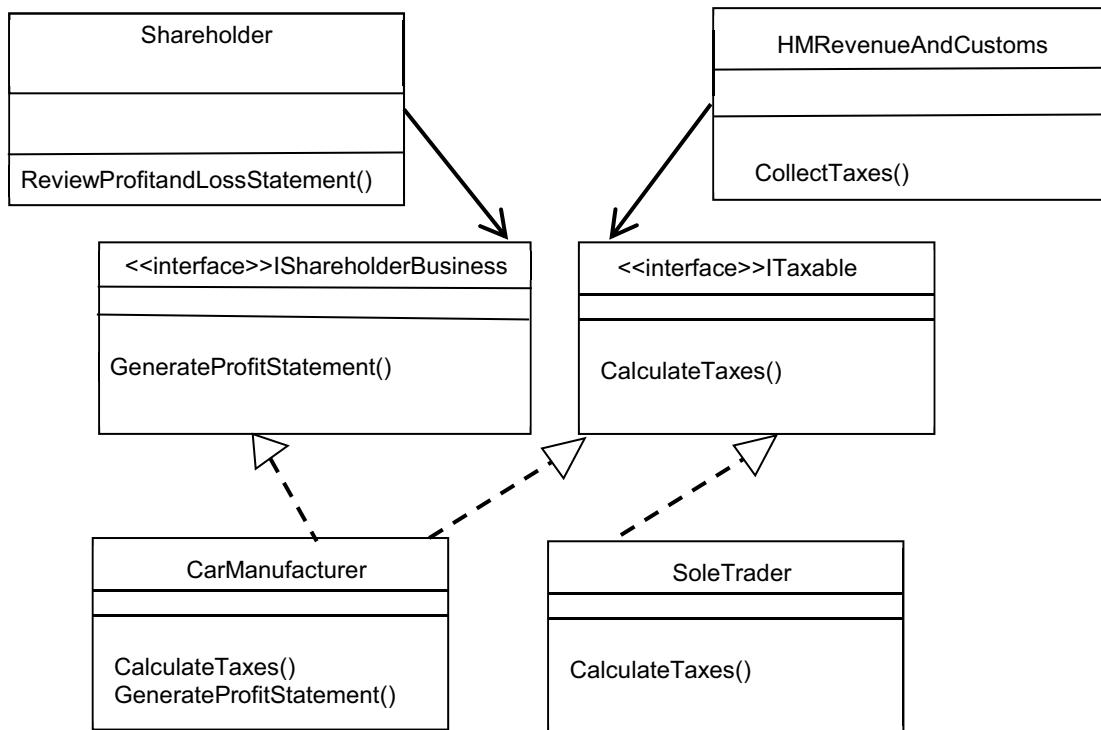
This design above is poor as the interface should be segregated.

Not all interfaces can usefully be segregated and there is no need to split them all into interfaces that specify only one method...but the clue here is that only **some** businesses have shareholders... but the government will want them **all** to pay taxes.

Imagine a self-employed plumber, a sole trader. They don't need to publish a profit and loss statement for shareholders but they do need to pay taxes.

The interface defined above therefore needs to be split into two interfaces so that only appropriate methods need to be implemented by small 1 person traders.

A revised design is shown below...



## 8.6 THE DEPENDENCY INVERSION PRINCIPLE

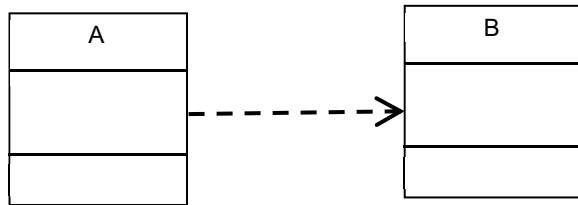
The final SOLID principle is the ‘Dependency Inversion Principle’. This is perhaps the hardest to explain but is no harder to implement and apply than any of the others.

The Dependency Inversion Principle is sometimes explained by an idea called ‘separation of concerns’. This is really the same idea.

In this section we will try to explain this idea firstly by looking at dependency inversion and then by looking at the idea of separating the concerns in the system.

But what is a dependency?

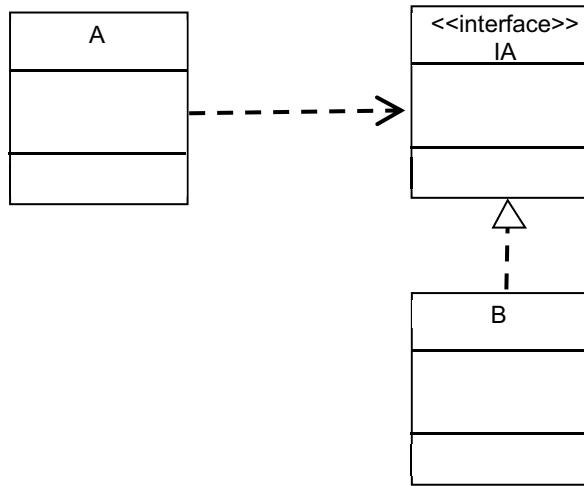
If class A has a method which is passed a parameter object of type B (as shown in the figure below) then A is dependent on the implementation of B. A and B are therefore coupled.



As discussed earlier in this chapter we want very highly cohesive components (the Single Responsibility Principle applies here) with very low levels of coupling as this makes our systems easier to change.

The solution to the coupling shown above, as suggested by the Dependency Inversion Principle, is to break the dependency between class A and class B by introducing an interface IA. Class A instead has a method which is passed a parameter object of type IA. B implements IA and B can be substituted by any class that implements the interface IA.

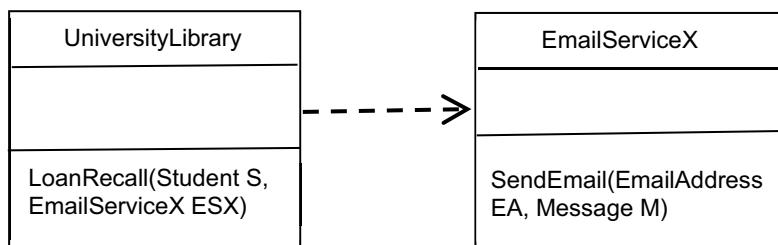
This is shown in the figure below...



Adding this interface inverts the dependency – A is no longer dependent on B, instead B is dependent on the interface IA.

Adding this interface makes code more extensible and A is no longer dependent on B.

Consider an example...



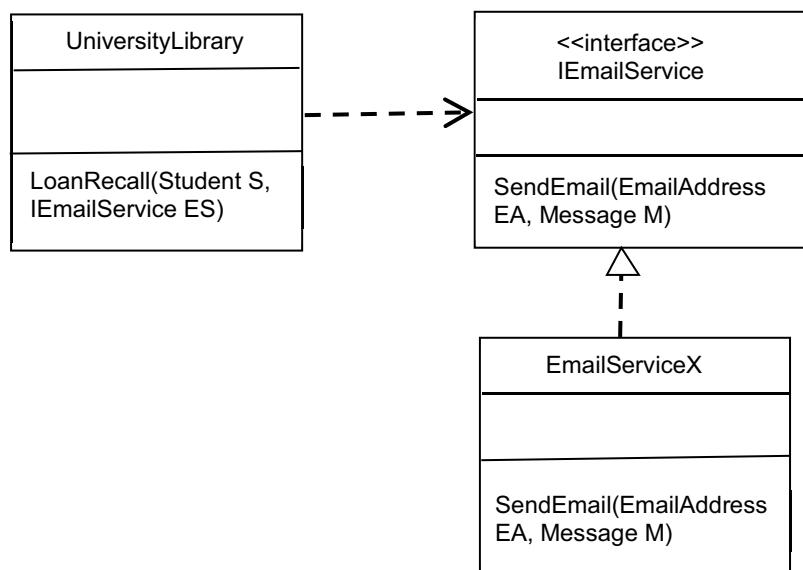
The system above shows part of a University Library system that wants to send a loan recall message to a student when another student makes a request for an item that is already out on loan. This requires the use of an email service and is specifically dependent on EmailServiceX provided by company X.

The LoanRecall() method in the UniversityLibrary class requires an object of this specific email service, ESX, and uses this to send an email to the student concerned by invoking the SendEmail() method 'ESX.SendEmail...'.

If the University were to replace the email service with an alternate email service then the Library class would need to be amended accordingly...and any other classes in the university system that used the email service would also need amending.

Making these classes dependent on the implementation of the email service requires changes to the system when the email service changes. Though the example above is small, in a real system these changes can be complex and costly.

The Dependency Inversion Principle encourages us to invert this dependency by inserting an interface (see below).



By adding an interface, the `LoanRecall()` method is changed. It no longer requires an object of type `EmailServiceX` but instead requires any object that implements the email service interface i.e. an object of type `IEmailService`.

Now the `EmailServiceX` can be replaced by `EmailServiceY` or any other email service as long as it implements the interface. Changing the email service will no longer require changes to the other classes in the system and thus our system is more flexible and easy to change.

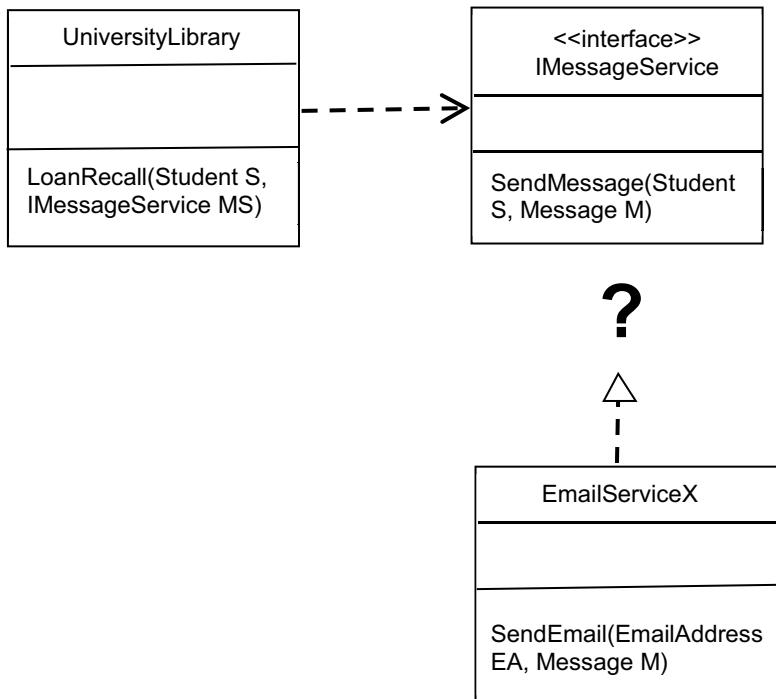
We can, however, take this much further and improve the system by making it even more flexible.

In the example above, the interface `IEmailService` was defined such that it had exactly the same method as implemented by `EmailServiceX` i.e. it had a method called `SendEmail()` which required two parameters: an email address and a message.

What if we wanted to replace EmailServiceX with another email service that had a different method name or required different parameters?

Or what if we wanted to replace the email service with a text messaging service?

Instead of defining the interface based on an existing email service we should instead define the interface based on the needs of the system as shown in the diagram below...



Now IEmailService has been replaced with a more generic interface IMessagService and the LoanRecall() method has been adapted accordingly.

If we examine the interface IMessagService we can see that the system will no longer send an email, it will instead send a message. The system will also no longer use the student email address. It will instead use any of the student's details so it could use the student's email address or it could use their mobile phone number.

Now without changing the UniviersityLibrary class, or other classes, the system is even more flexible and can now use any message service: email, text or something else.

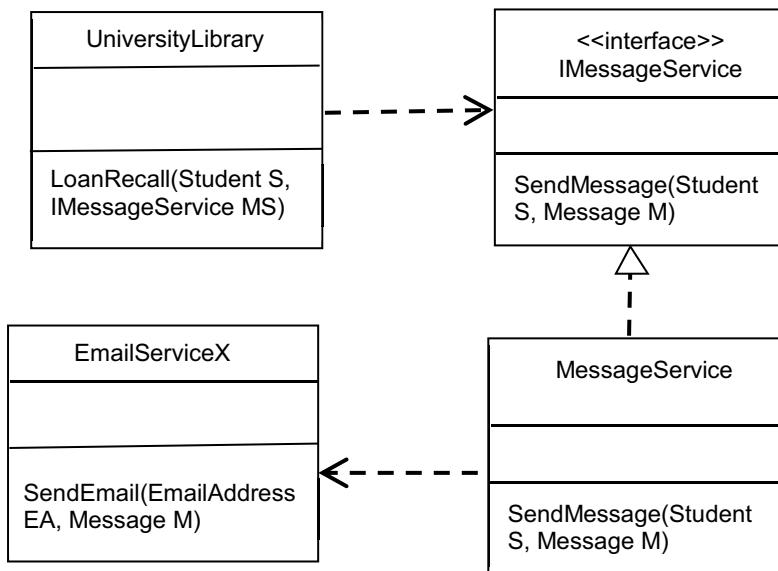
However, one problem still remains to be solved as indicated by the ? in the diagram above.

What if an email or messaging service provides the functionality we need but does not implement the interface exactly as we have specified it?

In the example above, EmailServiceX provides a method SendEmail() it does not have a SendMessage(Student S, Message M) method defined by the interface IMessagService i.e. it does not implement this interface...and of course we cannot expect external services to implement interfaces precisely as we specify them.

The answer to this problem is to insert a small piece of middleware. Middleware is software that sits in between two applications, translating information and allowing them to work together.

An example of this is shown in the diagram below...



The diagram above shows a new class ‘MessageService’. This is a small and very simple piece of middleware that sits between the University system and the actual email service we want to use.

The `MessageService` class implements our interface `IMessageservice` and simply invokes methods provided by the actual email service.

In essence it translates a ‘send message’ request into ‘send email’.

Sample code to illustrate this is shown below...

```
SendMessage(Student S, Message M)
{
    EmailServiceX ESX = new EmailServiceX(...);
    ESX.SendEmail(S.EmailAddress, M);
}
```

The SendMessage() method above obtains an instance of the actual email service we are using and invokes the SendEmail() method passing on the students email address and the message to be sent. Thus, a SendMessage() request results in an email being sent.

We can now very simply replace the email service with a text messaging service, or any other messaging system. While we will need to change the middleware no other changes to our system will be required.

We have broken the dependency our system had on an email service and we can now very simply insert any method of contacting a student.

The Dependency Inversion Principle states that we should not make our systems dependent on the detailed implementation defined in classes. In this case our system should not be dependent on the SendEmail(Address EA, Message M) method.

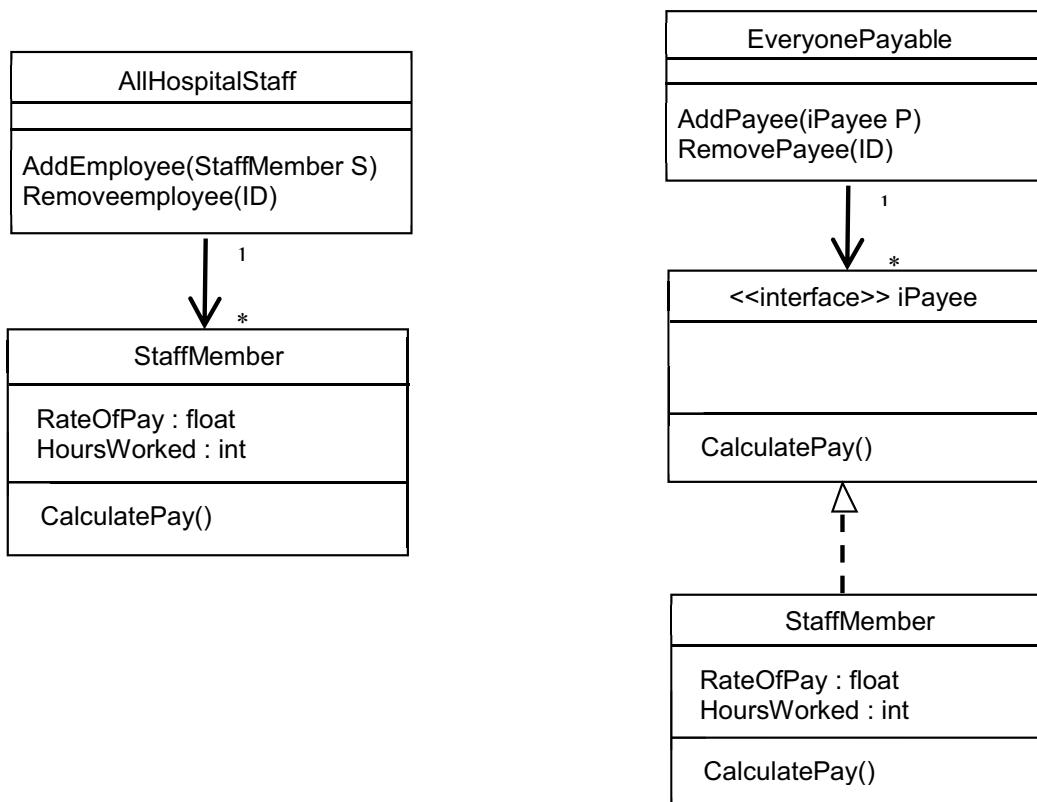
By inserting an interface we have inverted this dependency. Our system is no longer dependent on a specific class. Instead the class that implements the interface is dependent on our interface and our system can use any class that implements this interface.

For external classes we can insert a small piece of middleware to translate our method requests into the actual methods provided.

The idea of ‘Separation of Concerns’ as described by some software engineers is exactly the same idea but expressed slightly differently. This idea says we should use interfaces to separate all of the different components in our system. This reduces coupling and makes our systems more adaptable.

Taken to extremes, this idea says that we should break all associations → in our systems by inserting interfaces.

In Chapter 5 we saw how we could use an interface to make polymorphism even more powerful by replacing a list of all staff with a list of everyone who is payable (see diagram below).



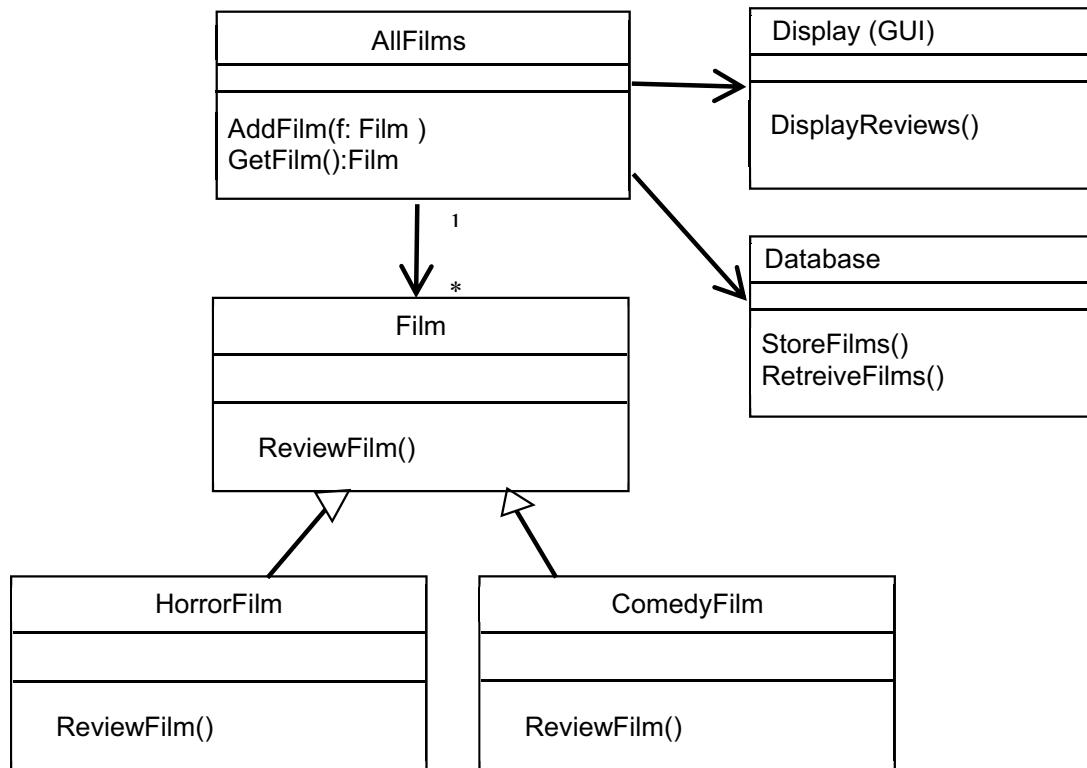
In figure above, on the left, there is an association between the list of all hospital staff and the `StaffMember` class. By inserting an interface, see figure on the right, we are breaking this coupling. The list can now be a list of anyone who is payable...this includes staff, external contractors and anyone else who the hospital may need to pay.

The idea of separating the ‘concerns’ of the system is the same as the Dependency Inversion Principle. Essentially by inserting interfaces we can make our systems more flexible and adaptable to the changing needs of our clients and users.

### Activity 7

A system is required for a reviewer to review films. A list of films is to be maintained by the system (some reviewed and some still to be reviewed). The review process will be different depending on the nature of the film concerned – so horror films will be reviewed on the basis of the tension and the amount of blood and gore, comedies will be reviewed based on the characters, the novelty and the laugh out loud factor. The list of films is to be stored in a database. The system will display the reviews via a GUI.

Consider the following design for this system shown below...



### **Activity 7 Continued**

Looking at the design above answer the following questions:

- 1) How many dependencies can you identify?
- 2) How would you improve this design taking the Dependency Inversion Principle into account or the idea of 'separation of concerns'?

## Feedback 7

The system is already partially flexible as new film types can be added and the ReviewFilm() method overridden accordingly. The list is already polymorphic as it can maintain a list of any type of film, including new types not yet defined.

But we can improve the system by making it much more flexible and adaptable.

1) There are three dependencies in the system and if we can identify them and insert interfaces, that will make the system more flexible.

Firstly the list is dependent on the Film class...if we break this dependency the system can be used to review anything that is reviewable...cars, food, holidays, hotels or anything else.

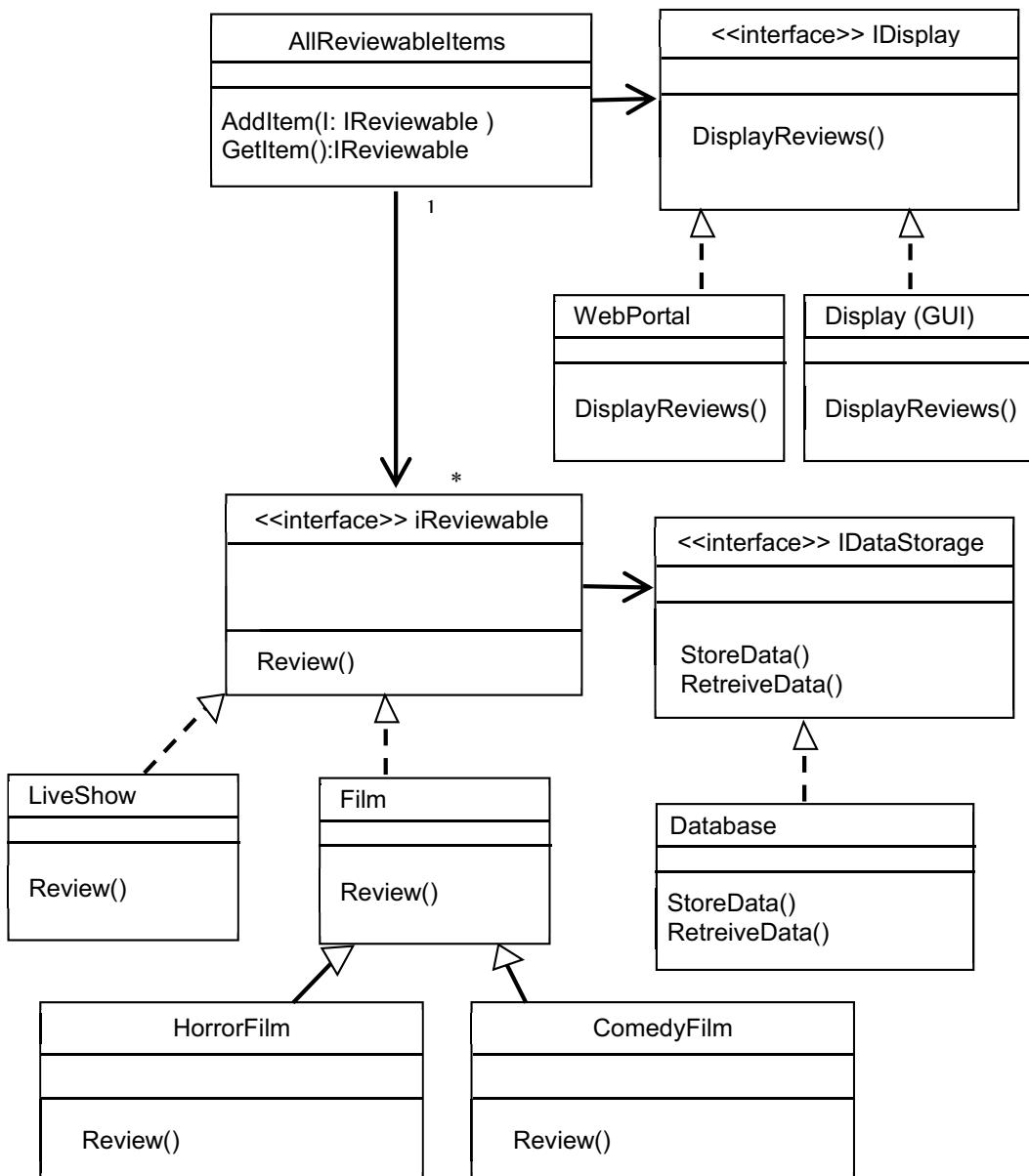
Secondly the system is dependent on the database. If we break this dependency we can change our data storage method without it requiring changes to the other components...perhaps we will store data in a simple text file so reviews can be edited later or perhaps we will want to employ a secure cloud data storage system.

If we want to use an externally provided data storage system, we may also need to insert a small piece of middleware but this will be easy and will not require changes to our other classes.

Finally, the system is dependent on the GUI to display the reviews. If we break this dependency, we can provide alternative ways to view this information...perhaps by displaying reviews using a web or a mobile interface.

2) In the system below three interfaces have been inserted to break all of these dependencies and make our system as flexible as possible.

To demonstrate this flexibility, classes have been added to indicate how easily the system can be extended to a) allow the reviewer to review live shows and b) allow the system to display reviews via a web portal.



The Dependency Inversion Principle says we should invert dependencies by using interfaces so our system is not dependent on low level functionality. The ‘separation of concerns’ says we should separate all of the concerns in our systems.

In reality, these are two ways of stating the same concept...by using interfaces we make our systems easy to adapt to the changing needs of our users.

Ultimately there is not one ‘perfect design’ and there are costs and benefits to be considered: Inserting lots of interfaces adds a little additional complexity at the start but can save lots of time and effort later (and saving time saves money).

Some software engineers would argue you should make your system as adaptable as possible from the very start. Others would argue you should keep your system as simple as possible.

If you chose not to insert an interface you can refactor your system by redesigning it and adding interfaces later but this is not always simple. Sometimes it’s easier to plan ahead and build in the flexibility from the start.

## 8.7 SUMMARY

The SOLID design principles are:

- Single Responsibility Principle**
- Open/Closed Principle**
- Liskov Substitution Principle**
- Interface Segregation Principle**
- Dependency Inversion Principle**

These principles are in essence very simple and when applied make our systems designs better by making them more robust and easier to change – saving time, effort and money.

The Single Responsibility Principles states that each class should only have responsibility over a single part of system functionality. If a class has more than one responsibility then the responsibilities become coupled and changes to one responsibility may cause complications by requiring other parts of the system to be changed.

The more components you need to change the more complex and costly it is and the greater the chance you will introduce errors into the system.

Software Engineers have, for many years, recognised that highly cohesive systems, with low levels of coupling, are more maintainable, easier to fix and easier to understand and the highest level of cohesion comes when a class only has one responsibility.

The Open and Closed Principle suggests that software entities, such as classes and methods, should be open for extension but closed for modification.

When a simple change results in a ripple effect throughout a program this indicates a rigid design.

This principle applies to inherited classes and is linked to the Liskov Substitution Principle. For polymorphism to work subtype objects must be substitutable for a super type object. The creation of subtypes which can be substituted for their base type means that subclasses can extend the base class but they cannot modify existing functionality (though methods can be overridden appropriately).

The Interface Segregation Principle states that we should split fat interfaces into several thin interfaces. This enables reuse and makes our system robust and flexible.

The final SOLID principle is the ‘Dependency Inversion Principle’. This is no harder to implement and apply than any of the others.

The Dependency Inversion Principle or the ‘separation of concerns’ simply means inserting interfaces to make our systems easy to adapt to meet the changing needs of our users.

The application of these principles will be demonstrated again in the case study, see Chapter 14, and you will have another opportunity to develop your understanding of these.

# 9 GENERIC COLLECTIONS AND SERIALIZATION

## Introduction

So far in this book we have:

- Explained the basic concepts of Object Orientation.
- Introduced UML notation and considered the different type of relationship between classes (especially association, dependency, inheritance and implementation).
- Seen how inheritance works and how to use polymorphism effectively.
- Seen how analysis of a problem can lead to an Object Oriented design and how the SOLID design principles can be applied to ensure our designs are good designs that lead to flexible and robust software.

This book will now start to focus less on design and more on practical implementation issues:

- How to implement collections.
- How to serialise/deserialise entire collections.
- C# development tools.
- How to create and use customised exceptions.

This chapter will start by introducing the reader to generic methods. It will then go on to introduce the reader to an essential part of the .NET framework: the classes that implement generic collections. Finally, it will introduce the idea of serialization, a powerful mechanism, and show how a collection can be serialised.

## Objectives

By the end of this chapter you will be able to...

- Understand the concept of Generic Methods
- Understand the concepts of Collections and how to implement different types of Collection
- Understand the concept of Serialiazation and understand how to serialise an entire collection.

This chapter consists of fifteen sections:

- 1) An Introduction to Generic Methods
- 2) An Introduction to Collections
- 3) Different Types of Collections
- 4) Lists
- 5) HashSets
- 6) Dictionaries
- 7) A Simple List Example
- 8) A More Realistic Example Using Lists
- 9) Queues and Stacks
- 10) An Example Using Sets
- 11) Overriding Equals() and GetHashCode()
- 12) An example Using Dictionaries
- 13) Serializing and De-serializing Collections
- 14) The Power of Serialization
- 15) Summary

## 9.1 AN INTRODUCTION TO GENERIC METHODS

We have seen previously how methods are identified at run time by their signature i.e. the name of the method and the list of parameters the method takes.

Thus we can have two methods with the same name. Shown below are two methods that find a highest value...one finds the highest value given two integer numbers, the other finds the highest value of two double numbers.

```
static public int Highest(int o1, int o2)
{
    if (o1 > o2)
        return o1;
    else
        return o2;
}
static public double Highest(double o1, double o2)
{
    if (o1 > o2)
        return o1;
    else
        return o2;
}
```

Given the following code the CLR engine will invoke the first of these methods, then the second. The correct method to invoke is identified by the method and its parameter list.

```
int n1 = 1;  
int n2 = 2;  
Console.WriteLine("The highest is " + Highest(n1, n2));  
  
double n3 = 1.1;  
double n4 = 2.2;  
Console.WriteLine("The highest is " + Highest(n3, n4));
```

Given the following definition of a Student class...

```
class Student
{
    String name;

    public Student(String name)
    {
        this.name = name;
    }

    override public string ToString()
    {
        return name;
    }
}
```

We could even define a version of this method to find the highest student (alphabetically).

```
static public Student Highest(Student o1, Student o2)
{
    if (o1.ToString().CompareTo(o2.ToString()) > 0)
        return o1;
    else
        return o2;
}
```

And invoke this using ..

```
Student s1 = new Student("Alan");
Student s2 = new Student("Clare");
Console.WriteLine("The highest is " + Highest(s1, s2));
```

In doing this we have created three methods that essentially do exactly the same thing only using different parameter types. This leads us to the idea that it would be nice to create just one method that would work with objects of any type.

The method below is a first attempt to do this:

```
static public Object Highest(Object o1, Object o2)
{
    if (o1.ToString().CompareTo(o2.ToString()) > 0)
        return o1;
    else
        return o2;
}
```

This method takes any two ‘Objects’ as parameters. ‘Object’ is the base class of all other classes thus this method can take any two objects of any more specific type and treat these as of the base type ‘Object’. This is polymorphic.

The method above then converts these two ‘Object’s to strings and compares these strings.

Thus this one method can be invoked three times using the following code...

```
int n1 = 1;
int n2 = 2;
Console.WriteLine("The highest is " + Highest(n1, n2));

double n3 = 1.1;
double n4 = 2.2;
Console.WriteLine("The highest is " + Highest(n3, n4));

Student s1 = new Student("Student A");
Student s2 = new Student("Student B");
Console.WriteLine("The highest is " + Highest(s1, s2));
```

In these cases respectively the CLR engine will treat int, double and Student objects as the most general type of thing available ‘Object’. It will then convert this object to a string and compare the strings.

This will work. However in many situations creating methods which take parameters of type ‘Object’ is flawed or at least very limited.

Inside these methods we do not know what type of object was actually passed as a parameter and hence in the example above we do not know what type of object is actually being returned. When two students object are passed the object returned is a student but we cannot invoke Student specific methods on this object unless we first cast the object returned to a Student.

Assume that we want to invoke an ‘AwardMerit()’ method on the ‘Student’ returned via the ‘Highest()’ method. We can do this if we first cast the returned ‘Object’ onto a ‘Student’ object. E.g.

```
(Student) Highest(studentA, studentB).AwardMerit();
```

However the compiler cannot be certain that the returned object is a ‘Student’. Thus the compiler cannot detect the potentially critical error that would occur if we invoked ‘Highest()’ on two integer numbers and then tried to cast the returning integer onto an object of type ‘Student’.

This leads us to the idea that we would like to be able to create a method that will take parameters of **any type** and return values that are again of **any type** but where we will define these types when we invoke these methods. Such methods do exist and they are called Generic methods.

Generic methods are methods where the parameter types are not defined until the method is invoked. Parameter types are specified each time the method is invoked and the compiler can thus still check the code is valid.

In other words a generic method uses a parameterized type – a data type that is determined by a parameter.

In the code below a generic method Highest() has been defined as a method that takes two objects as parameters of unspecified type:

```
static public T Highest<T>(T o1, T o2)
{
    if (o1.ToString().CompareTo(o2.ToString()) > 0)
        return o1;
    else
        return o2;
}
```

We can use this method and each time we invoke it we define the type of object being compared. If two students are compared the compiler will know that the object being returned is also type ‘Student’ and will therefore know it is legal to invoke student specific methods on this object.

When writing a Generic method a list of identifiers for the generic data types must be provided. These are specified after the method’s name and are contained between angle brackets. If multiple generic types are used by a method, their names are separated by commas.

In the example above, the identifier T can stand for any data type. So, when T is used within the brackets in the method’s parameter list to describe data, it might be an int, double, ‘Student’ or any other data type. The only requirement is that the method must work no matter what type of object is passed to it. The Generic method above invokes the ToString() on the parameter and we know this will work for any object as all objects have a ToString() method. We know this because every data type inherits the ToString() method from the ‘Object’ class (or contains its own overriding version). Thus, we know this method above will work for any object of any type.

In the case above only one generic data type is specified. This is used to define the type of both parameters and the return value.

The generic data type identifier for a generic method can be any legal C# identifier, but by convention it is usually an upper case letter. “T” is used to stand for “type”.

When a generic method is defined i.e. one that works with any data type the type is specified when the method is used.

Thus in the code below, while Highest() is a generic method, the compiler can see that a ‘Student’ object is being passed as a parameter and thus the object being returned must also be of type ‘Student’.

```
Student s1 = new Student("Student A");
Student s2 = new Student("Student B");
Student highestStudent = Highest(s1, s2));
```

Given the object being returned is of type ‘Student’, it must be OK to store this in a variable of type Student and it must be OK to invoke and Student methods on the object returned.

Generic methods can therefore work with an object of any type and the compiler can still check for type errors (as the type is specified each time the method is invoked).

Generic classes can also be created and these work in much the same way as generic methods. The creators of the .NET libraries used these mechanisms to create generic collections and these are particularly useful.

While we won’t be writing our own generic methods, or generic classes, we will be making significant use of generic collections, but before using these we need a basic understanding of collections themselves.

## 9.2 AN INTRODUCTION TO COLLECTIONS

Most software systems need to store various groups of entities rather than just individual items. Arrays provide one means of doing this, but in C# collections are much more varied and flexible forms of grouping.

Collections are classes which serve to hold together groups of other objects. The .NET libraries provides several important classes (defined in the System.Collections namespace) that provide several different types of collection and associated functionality.

This work has been developed and significantly improved upon by the creation of generic collections (defined in the System.Collections.Generic namespace).

As generic collections are far more useful than non-generic collections the use of non-generic collections is normally discouraged and this book will focus entirely on generic collections.

### **9.3 DIFFERENT TYPES OF COLLECTIONS**

There are three basic type of collection: List, HashSet and Dictionary.

### Activity 1

Go online to [msdn.microsoft.com](http://msdn.microsoft.com). This is the website for the Microsoft Developer Network and it contains details of the Application Programmer Interface (API) for the .NET framework (it also contains much more).

Follow the links for  
Library (tab near top of page)  
and then the '.NET Framework Class Library' (this maybe listed under a section for 'Development tools and Languages').

You may need to drill down deeper to get into the '.NET Framework Class Library' itself but when there you will see all of the namespaces (packages) in the .NET framework on the left with a description of each on the right.

Follow the link for  
`System.Collections.Generic`

At the time of writing this document the following link will take you directly to this:  
<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic?view=netframework-4.7.1>

Alternatively you can get to the same location by searching for 'msdn .net C# generic collection classes' in any web browser.

Look at the class documentation and identify any other collections that you may find useful.

### Feedback 1

Learning to navigate the NET Framework Class Library and understanding the technical documentation is a skill any professional programmer needs to develop. While you will probably not understand everything initially don't let this stop you from looking.

By looking at this you will see useful classes and for each class you will see the alternative constructors and the methods provided.

One of the classes you may have identified is `SortedDictionary`...this is just like a dictionary that we will cover but one where the elements are automatically sorted for us. We will use a sorted dictionary in the large case study (Chapter 14).

Other useful collections include `LinkedList`, `Queue`, and `Stack`. Discussions of these are beyond the scope of this text but if you have an understanding of Queues, or Linked Lists then it is worth knowing that these have been implemented for you as part of the .NET framework.

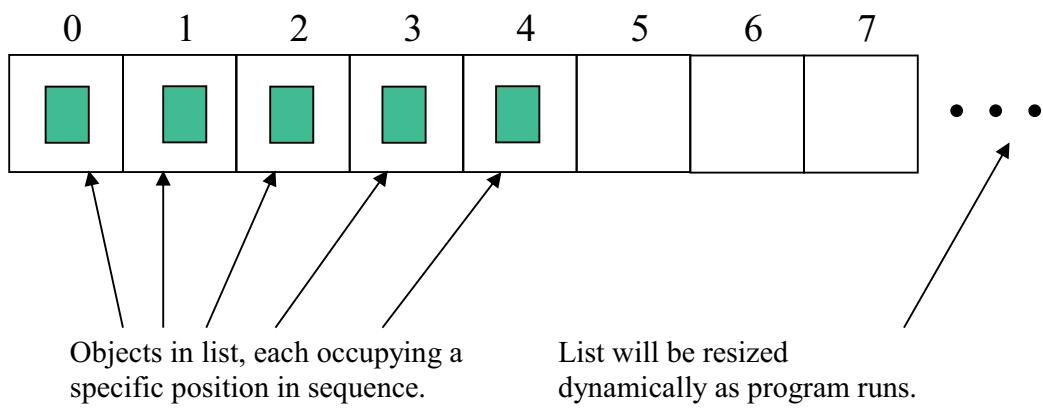
We will now look at what are, arguably, the three most important collections: `List`, `HashSet` and `Dictionary`. We will first discuss what they do and then we will look at how to implement them in C#.

## 9.4 LISTS

Lists are the most commonly used collection. Where you might have used an array, a List will often provide a more convenient method of handling the data.

Lists are very general-purpose data structures, with each item occupying a specific position. They are in many ways like arrays, but are more flexible as they are automatically resized as data is added. They are also much easier to manipulate than arrays as many useful methods have been created that do the bulk of the work for you.

Lists store items in a particular sequence (though not necessarily sorted into any meaningful order) and duplicate items are permitted.

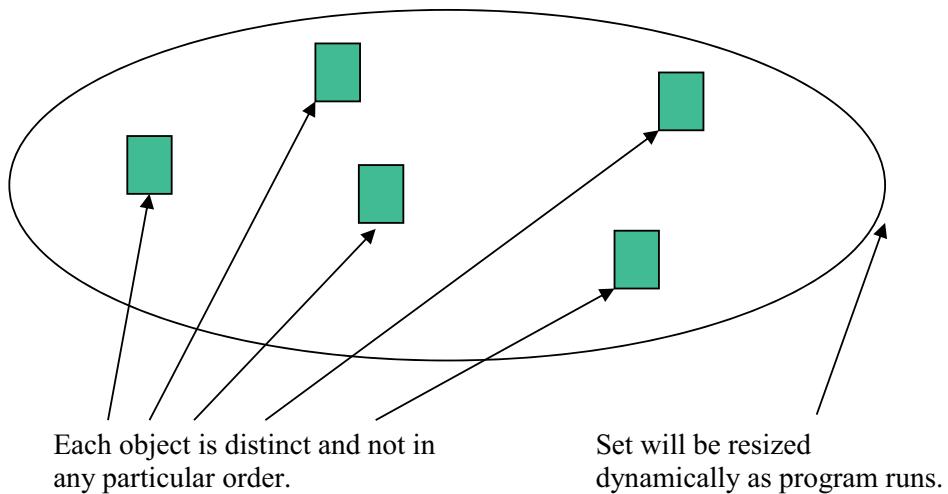


## 9.5 HASHSETS

A HashSet is one implementation of a set. A set is like a ‘bag’ of objects rather than a list. They are based on the mathematical idea of a ‘set’ – a grouping of ‘members’ that can contain zero, one or many distinct items.

Unlike a list, duplicate items are not permitted and a set does not arrange items in order.

Like lists, sets are resized automatically as items are added, so they will never run out of space.



Many of the operations available for a List are also available for a Set. However:

- We **cannot** add an object at a specific position since the elements are not in any order.
- We **cannot** ‘replace’ an item for the same reason (though we can add one and delete another).
- It is possible to retrieve all the items in a set but the order in which they will be retrieved is unknown.
- It is meaningless to find what position an element is in.

## 9.6 DICTIONARIES

Dictionaries are rather different from Lists and Sets because instead of storing individual objects they store pairs of objects. The pair is made up of a ‘key’ and a ‘value’. The key is something which identifies the pair. The value is a piece of information or an object associated with the key.

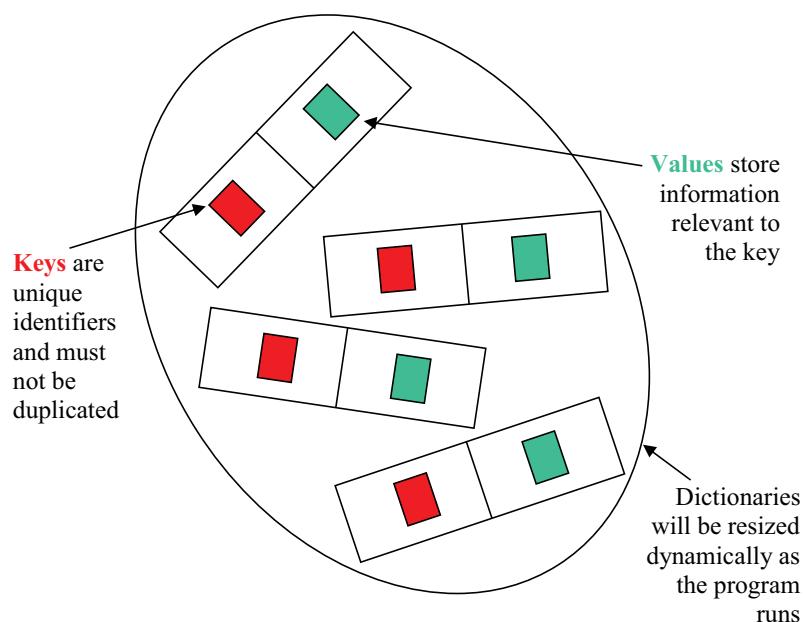
In language dictionaries, that we are all familiar with, the ‘key’ is the word you look up and the ‘value’ the meaning of that word. Of course in language dictionaries the entries are sorted into word (or key) order. If that was not the case it would be very difficult to find a particular word in a dictionary.

Another example of a dictionary would be an address book: in an address book the keys would be people’s names and the values their address (including phone, email etc.). There is only one value for each key, but since values are objects they can contain several pieces of data.

Duplicate keys are not permitted, though duplicate values are. So in the previous example if you looked up two people in the address book you may find them living at the same address – but one person would not have two homes.

Like a Set, a Dictionary does not arrange items in order (but a SortedDictionary does, in order of keys) and like lists and sets, dictionaries are resized automatically as items are added or deleted.

The following figure illustrates a Dictionary:



### Activity 2

For each of the following problems write down which would be most appropriate: a List, a Set or a Dictionary.

- 1) We want to record the members of a club.
- 2) We want to keep a record of the money we spend (and what it was spent on).
- 3) We want to record bank account details – each identified by a bank account number.

### Feedback 2

1. For this we would use a set. Members can be added and removed as required and the members are in no particular order.
2. For this we would use a list – this would record the items bought in the order in which they were purchased. Importantly as lists allows duplicate items we could buy two identical items (perhaps two identical toys for as birthday presents for two different children).
3. We would not have two identical Bank accounts so a set seems appropriate. However, as each is identified by a unique account number, a dictionary would be the most appropriate choice.

## 9.7 A SIMPLE LIST EXAMPLE

In this section of the book we will demonstrate a simple use of the List collection class.

### Activity 3

Go online to msdn.microsoft.com. Find API for the List class defined in the System.Collections.Generic namespace.

List three of the methods you think would be useful.

### Feedback 3

Some of the commonly used methods include...

Add() – which adds an object onto the end of the list,  
Clear() – which removes all the elements from the list,  
Contains() – which returns ‘true’ if this list contains the specified object,  
Insert() – which inserts an element at the specified position,  
Remove() – which removes the first occurrence of an object from a list,  
Sort() – which sorts the element of the list.

Note: Some methods, such as Sort(), are overloaded. Sort() can either sort the entire list or sort just part of the list specified by a range.

The code cof names i.e. Strings.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ListOfNames
{
    class Program
    {

        // create names which will store a list of names
        private static List<String> names;

        static void Main(string[] args)
        {
            names = new List<String>(); // Create empty list

            string name;
            for (int i = 0; i < 3; i++) //Enter three names
            {
                name = Console.ReadLine();
                names.Add(name); // Add each name onto list
            }

            foreach (String n in names) // Itterate through list
            displaying each element.
            {
                Console.WriteLine(n);
            }

            Console.WriteLine("Please enter name to search for");
            String searchname = Console.ReadLine();
            if (names.Contains(searchname)) // Demonstrate Contains
            method works.
            {
                Console.WriteLine("Name found");
            }
            else
            {
                Console.WriteLine("Name not found");
            }
            Console.ReadLine();
        }
    }
}
```

The code above is fairly self explanatory however, perhaps a few parts are worthy of clarification in particular the creation of the list.

Firstly, as we are using we are using a generic List class, part of the System.Collections.Generic namespace, this can be a list of any object but the type must be specified as the list is created.

Thus the code segment below creates ‘names’ a variable that will hold a List of Strings.

Note: The type is specified within the angled brackets < > after the word ‘List’.

```
private static List<String> names;
```

This was created as a Static variable simply for the reason that we are using this directly from within ‘Main’ and we won’t actually be creating an object of the ‘Main’ class.

When we invoke the constructor for the List class, we must again specify the type of list being created i.e. a list of strings.

```
names = new List<String>(); // Create empty list
```

In the remainder of the code we a) Iteratively add three names to the list, b) Display the list and c) Use the Contains() method to search for a specific name.

Each element of the list can be accessed in much the same way as elements of an array are accessed (e.g. Console.WriteLine(names[0])) but C# provides an improved ‘for loop’ construct that will iterate through the elements of the list giving us access to each element as it does so.

Thus, as the code below iterates through the list ‘n’ is set to each element in turn.

```
foreach (String n in names)
{
    Console.WriteLine(n);
}
```

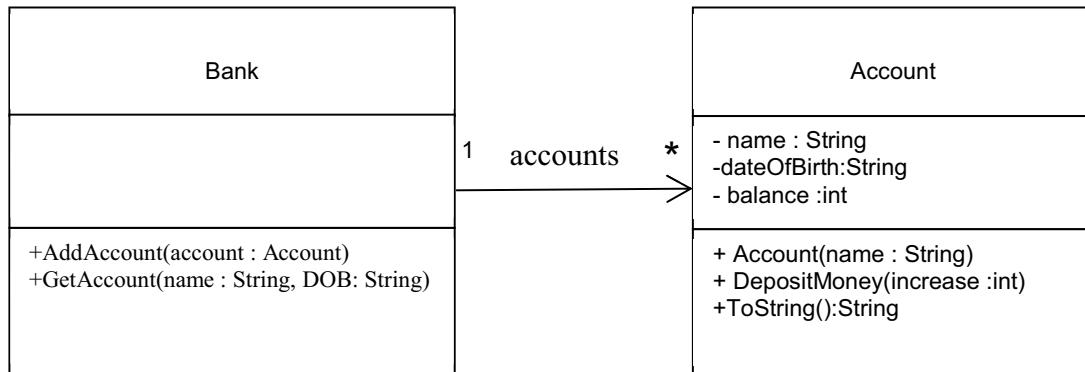
Note: Elements in a list are stored in order. Hence when the list is displayed the order remains the same.

While this program stores and manipulates a list of Strings the program could just as easily store a list of Publications – or any object constructed from a class we create.

## 9.8 A MORE REALISTIC EXAMPLE USING LISTS

A more realistic example of lists would come where we are storing more complex objects of our own devising (not just Strings).

Consider the UML class diagram below:



The diagram above represents the basics of a banking system (admittedly a very simplistic banking system). In this system, a class is used to simulate a Bank and maintain a collection of Accounts (note the 1 to many relationship). For the moment we will use a List to manage this collection.

Each element of this list will represent a single Account object. One Bank object will maintain a list all accounts.

Each time an account is created, the initial balance will be zero but the owner will be able to deposit money when required. So we can focus on the essential purpose of this exercise, to demonstrate the use of a list, we won't bother adding other obviously required functionality (to withdraw money etc.).

When each individual account is created we will add it to the list of accounts and we will also allow an individual account to be retrieved so that the account holder can deposit money into their account.

We will demonstrate this with a Main method that runs through a fixed routine a) Adding a few accounts, b) Displaying these accounts and c) Retrieving an account, d) Adding money to this account and e) Displaying the list of accounts again.

We will need an accessor method in Bank so that the Main method can access the list of accounts to display them. In C# we will implement this using a public property.

The code for the Account class is shown below and should need no explanation.

```
class Account
{
    private String name;
    public String Name
    {
        get { return name; }
    }

    private String dateOfBirth;
    public String DateOfBirth
    {
        get { return dateOfBirth; }
    }

    private int balance;

    public Account(String name, String dob)
    {
        this.name = name;
        this.dateOfBirth = dob;
        balance = 0;
    }

    public void DepositMoney(int increase)
    {
        balance = balance + increase;
    }

    public override String ToString()
    {
        return «Name: « + name + «\nBalance : « + balance;
    }
}
```

The code for the Bank class is given below and this demonstrates how easy it is to use the collection class ‘List’.

```
class Bank
{
    private List<Account> accounts;
    public List<Account> Accounts
    {
        get { return accounts; }
    }

    public Bank()
    {
        accounts = new List<Account>();
    }

    public void AddAccount(Account account)
    {
        accounts.Add(account);
    }

    public Account GetAccount(String name, String dob)
    {
        Account FoundAccount = null;

        foreach (Account a in accounts)
        {
            if ((a.Name == name) && (a.DateOfBirth==dob))
            {
                FoundAccount = a;
            }
        }
        return FoundAccount;
    }
}
```

Firstly the constructor for the Bank class (shown above) creates an empty list of accounts. As with any list, this automatically resizes itself, so unlike an array we do not need to worry about running out of space.

Also, when elements of the list are removed, blank spaces within the list are automatically deleted. While this is not hard to do with arrays it is a time consuming process and this is handled automatically for us when we use Lists.

Adding an account becomes a trivial task of invoking the Add() method (see the AddAccount() method shown above).

Finally retrieving an individual account is fairly simple to do by iterating through the list until the account we are looking for is found (see GetAccount() method).

Note: This list manages a generic collection of complex objects and, as the type is specified at the time the list is created, when an object is retrieved from the list the CLR knows the type of the object retrieved. Methods associated with that object can then be invoked on that object.

This can be demonstrated via the following 'Main' method:

```
static void Main(string[] args)
{
    Bank b = new Bank();
    Account a;

    a = new Account(«Alice», »06/12/1963»);
    b.AddAccount(a);
    a = new Account(«Bert», «14/08/1990»);
    b.AddAccount(a);
    a = new Account(«Claire», »1/1/2000»);
    b.AddAccount(a);

    // Display the accounts
    foreach (Account acc in b.Accounts)
    {
        Console.WriteLine(acc.ToString());
    }

    Account a1 = b.GetAccount(«Bert», «14/08/1990»);
    a1.DepositMoney(100);

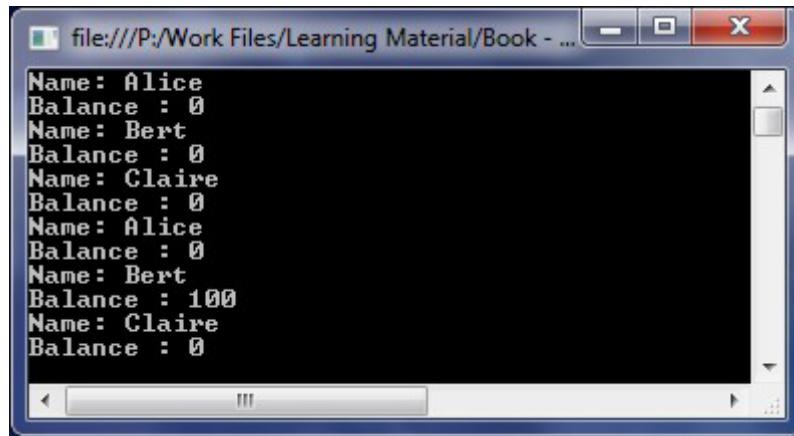
    // Display the accounts again
    foreach (Account acc in b.Accounts)
    {
        Console.WriteLine(acc.ToString());
    }

    Console.ReadLine();
}
```

The code above performs the following actions:

- Firstly it invokes the constructor for the Bank class which in turn creates an empty list of Accounts.
- It then creates three accounts and adds these to the list in the Bank object.
- The list of accounts is then displayed.
- An individual bank account is then retrieved from the list and money is deposited into this account.
- Finally the list of accounts is displayed again.

The output from running this program is shown below...



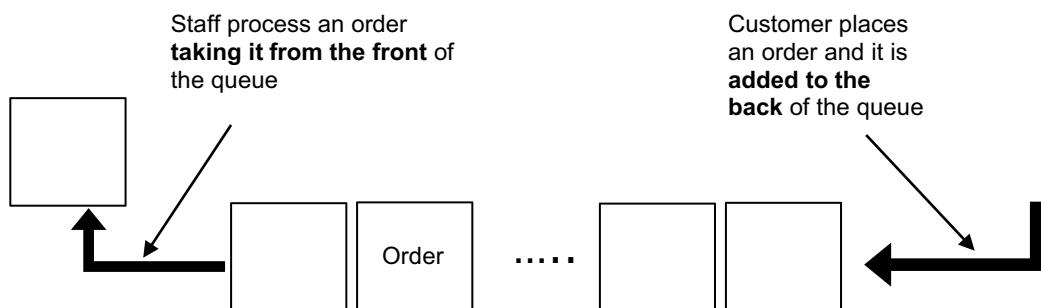
The screenshot shows a Windows application window titled "file:///P:/Work Files/Learning Material/Book - ...". Inside the window, there is a list of objects, each consisting of two lines: "Name: <name>" and "Balance : <balance>". The objects listed are Alice (Balance 0), Bert (Balance 0), Claire (Balance 0), Alice (Balance 0), Bert (Balance 100), and Claire (Balance 0). The window has a standard title bar, minimize, maximize, and close buttons.

## 9.9 QUEUES AND STACKS

Before we move on, and look the use of sets, it is worth briefly considering two very useful variations of a list: namely a queue and a stack.

Queues acts like a list but where items can only be added at the end of list and items are only taken off from the start of the list.

Consider a concurrent application: Customers place order on a queue via the web and staff process the orders (first come first served). A queue, shown below, is also known as First In, First Out (FIFO) structure.



Queues are simple to create:

```
Queue<Order> OrdersToBeProcessed = new Queue<Order>();
```

And are simple to use.

The Enqueue() method adds an object to end of queue e.g.

```
OrdersToBeProcessed.Enqueue(NewOrder);
```

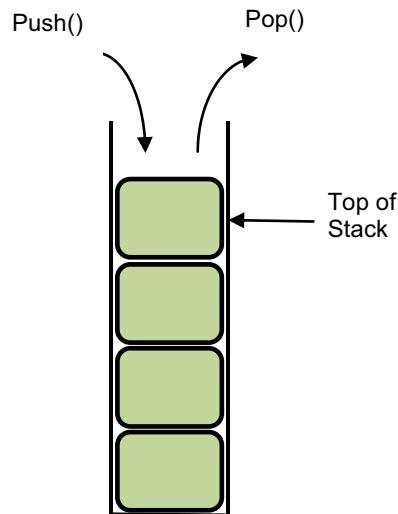
Dequeue() removes and returns the object at the beginning of the queue e.g.

```
Order = OrdersToBeProcessed.Dequeue();
```

Other methods include Clear() and Contains().

Note: The existence of System.Collections.Concurrent namespace for creating concurrent applications (beyond the scope of this book)

Stacks are very similar to queues but are Last In, First Out (LIFO) structures.



Objects are added (pushed) and taken (popped) from the top of the stack. As each item is added the stack gets deeper. The main methods used with stacks are Push() and Pop().

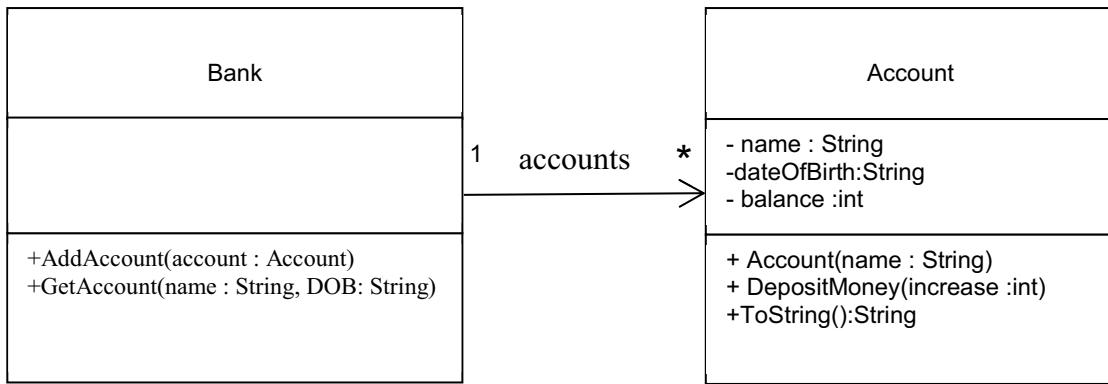
A common use for stacks is an “undo” mechanism, as used for example in word processors. This is accomplished by keeping all text changes in a stack. The last change made is the first to be undone.

## 9.10 AN EXAMPLE USING SETS

In section 9.8 we used a list to store a collection of bank accounts. This is not the most sensible choice as it allows duplicate accounts to be created and the bank would get very confused if two identical accounts were created (when money was deposited into an account which account should it be added to?).

Furthermore, the list stores objects in a particular order, in this case the order the account were created, but this order is irrelevant as it will not help us find a particular account belonging to an individual.

The essential problem remains the same – to store a collection of bank accounts.



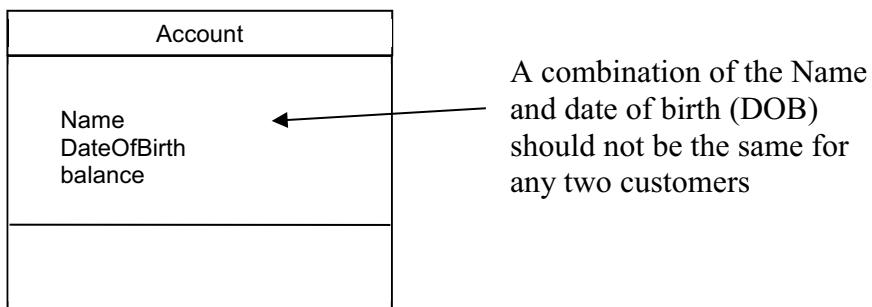
A much better choice for this collection would be to store the collection of accounts as a Set, because sets do not allow duplicates to be created.

## 9.11 OVERRIDING EQUALS() AND GETHASHCODE()

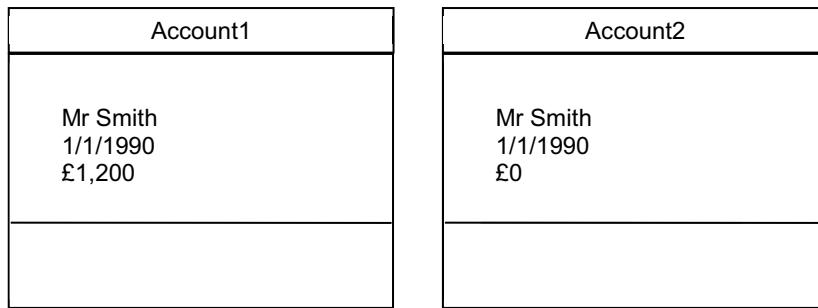
Using a set is no more complex than using a list but there is one problem that we must first resolve whenever we want to create a set of objects.

Consider a set of two bank accounts, one for Mr Smith and one for Mrs Jones.

To be certain that no two accounts are duplicates we would give each a unique account number but for now we will assume that no two customers born on the same day will have the same name.



Therefore it should not be possible to create a second account where the account holder has the same name and DOB as a previous account holder. Thus, in the example below it should not be possible to create Account 2.



However, unless told otherwise, C# will treat the two objects above as different objects as both objects have different names: **Account1** and **Account2**. Thus, while sets do not allow duplicates objects both of these accounts could be created and added to a set because the computer does not recognize them as the same i.e. as duplicates.

To overcome this problem we need to override the `Equals()` method, defined in the `Object` class, to say these objects are the same because the Name and DOB are the same.

We can do this by adding the following method to the `Account` class...

```
public override bool Equals(object obj)
{
    Account a = (Account)obj;
    return ((name==a.Name) && (dateOfBirth==a.DateOfBirth));
}
```

This overrides `Object.Equals()` for the `Account` class.

Now when the system checks to see if two accounts are the same it will use our overridden `Equals()` method. The system will say two object are the same if the account holders name and DOB are the same.

Note: Even though it will always be an `Account` object passed as a parameter, we have to make the parameter an `Object` type (and then cast it to `Account`) in order to override the `Equals()` method inherited from `Object` which has the signature

### **public bool Equals (Object obj)**

(You can check this in the by looking for the ‘`Object`’ class in the ‘`System`’ namespace of the .NET API)

If we gave this method a signature with an `Account` type parameter it would not override `Object.Equals(Object obj)`. It would in fact overload the method by providing an alternative method. As the system is expecting to use a method with a signature of `public bool Equals (Object obj)` it would not use a method with a signature of `public bool Equals (Account obj)`.

Therefore, we need to override **public bool Equals (Object obj)** and cast the parameter to an `Account`. We can then extract the name and DOB of the account holder to compare them with those of the current object.

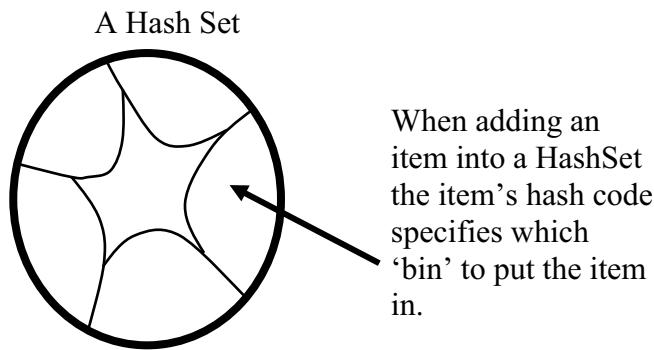
To be safe when overrding `Equals()`, we should also check that the parameter has a valid object or casting it to an `Account` will fail – the code below has an additional if statement added to perform this check.

The ‘if’ statement returns false if a null object is passed in as a parameter or if it cannot determine the type of that object.

```
public override bool Equals(object obj)
{
    if ((obj == null) || (GetType() != obj.GetType()))
    {
        return false;
    }
    Account a = (Account)obj;
    return ( (name==a.Name) && (dateOfBirth==a.DateOfBirth));
}
```

There is one additional complication to be resolved, this concerns how objects are stored in sets. To do this C# uses a hash code.

To make sets work efficiently they use something called a hashtable. Hashtables are beyond the scope of this book but essentially when adding an item into a HashSet the item's hash code specifies which 'bin' to put the item in (see the figure below).



When looking for duplicates, only one bin is searched (not the whole set). If an identical item is not in the bin being searched the computer will not 'see it' and it will then allow the duplicate item to be added to the set.

To prevent this, identical items must be associated with the same bin meaning two accounts with the same name and DOB should generate the same hash code.

Currently this will not be the case as the hash code is generated using the object name (e.g. Account1). Thus two accounts, 'Account 1' and 'Account 2' could still be stored in the set even if the name and DOB is the same as they could be associated with different bins.

To overcome this problem we need to override the GetHashCode() method so that a hash code is generated using the Name and DOB rather than the object name (just as we needed to override the Equals() method).

We can ensure that the hash code generated is based on the name and DOB by overriding the GetHashCode() method, in the Account class, as shown below...

```
public override int GetHashCode()
{
    return (name + dateOfBirth).GetHashCode();
}
```

The simplest way to redefine GetHashCode() for an object is to join together the instance values which define equality as a single string and then take the hashcode of that string. In this case equality is defined by the name of the account holder and DOB which taken together should be unique. Thus the code above generates one string by adding the name and DOB together. It then generates the hashcode of that string.

It looks a little strange, but we can use the GetHashCode() method on this String even though we are overriding the GetHashCode() method for objects of type Account.

GetHashCode() is guaranteed to produce the same hash code for two Strings that are the same. Occasionally the **same** hash code may be produced for **different** key values, but that is not a problem.

By overriding Equals() and GetHashCode() methods C# will prevent objects with duplicate data (in this case with duplicate name and dates of birth) from being added to the set.

The full code for the class Account is given below with the two overridden methods added at the end.

```
class Account
{
    private String name;
    public String Name
    {
        get { return name; }
    }

    private String dateOfBirth;
    public String DateOfBirth
    {
        get { return dateOfBirth; }
    }

    private int balance;

    public Account(String name, String dob)
    {
        this.name = name;
        this.dateOfBirth = dob;
        balance = 0;
    }

    public void DepositMoney(int increase)
    {
        balance = balance + increase;
    }

    public override String ToString()
    {
        return "Name: " + name + "\nBalance : " + balance;
    }

    public override bool Equals(object obj)
    {
        if ((obj == null) || (GetType() != obj.GetType()))
        {
            return false;
        }
        Account a = (Account)obj;
        return ( (name==a.Name) && (dateOfBirth==a.DateOfBirth));
    }
}
```

```
public override int GetHashCode()
{
    return (name + dateOfBirth).GetHashCode();
}
```

The code above is identical to the previous version of the Account class (when we used lists) except for the addition the overridden methods Equals() and GetHashCode(). When storing any object in a set we must override both of these methods to prevent duplicates being stored.

The code for the Bank class is identical apart from the first few lines which creates a typed HashSet instead of a typed List.

```
class Bank
{
    private HashSet<Account> accounts;
    public HashSet<Account> Accounts
    {
        get { return accounts; }
    }

    public Bank()
    {
        accounts = new HashSet<Account>();
    }

    public void AddAccount(Account account)
    {
        accounts.Add(account);
    }

    public Account GetAccount(String name, String dob)
    {
        Account FoundAccount = null;

        foreach (Account a in accounts)
        {
            if ((a.Name == name) && (a.DateOfBirth==dob))
            {
                FoundAccount = a;
            }
        }
        return FoundAccount;
    }
}
```

Finally, the Main method used to demonstrate this is shown below:

```
static void Main(string[] args)
{
    Bank b = new Bank();
    Account a;

    a = new Account("Bert", "06/12/1963");
    b.AddAccount(a);

    // try to create and add a duplicate account-this shouldn't work
    a = new Account("Bert", "06/12/1963");
    b.AddAccount(a);

    a = new Account("Alice", "14/08/1990");
    b.AddAccount(a);
    a = new Account("Claire", "1/1/2000");
    b.AddAccount(a);

    // Display the accounts
    foreach (Account acc in b.Accounts)
    {
        Console.WriteLine(acc.ToString());
    }

    Account a1 = b.GetAccount("Bert", "06/12/1963");
    a1.DepositMoney(100);

    // Display the accounts again
    foreach (Account acc in b.Accounts)
    {
        Console.WriteLine(acc.ToString());
    }

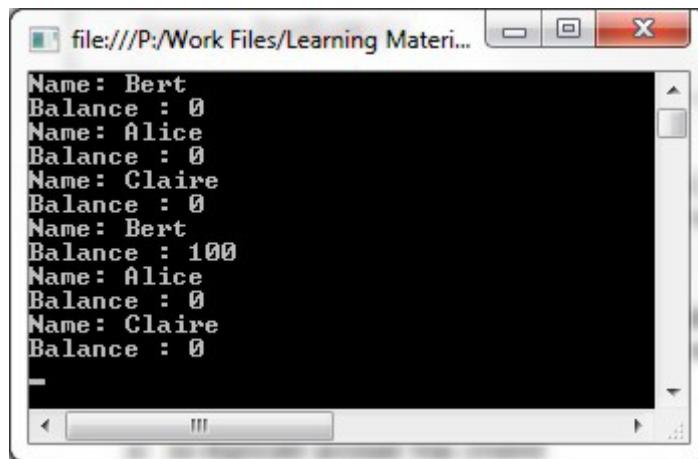
    Console.ReadLine();
}
```

This Main method is very similar to the Main method used to test lists. Several accounts are created, displayed, an account is retrieved and, after a deposit into this account has been made, the list is displayed again.

Two changes should be noted. First an attempt is made to create a duplicate account – this should not work. The list of accounts has been created in a non alphabetical order.

By looking at the program output, shown below, we can see that:

- No duplicate account was created, so overriding Equals() and GetHashCode() worked.
- The accounts are listed in the order created but not in an meaningful order. Entries in sets are not stored in any order and therefore we cannot be sure which order they will be retrieved in.



```
Name: Bert
Balance : 0
Name: Alice
Balance : 0
Name: Claire
Balance : 0
Name: Bert
Balance : 100
Name: Alice
Balance : 0
Name: Claire
Balance : 0
```

### Activity 4

Earlier we found that some of the useful methods for Lists included...

Add() – which adds an object onto the end of the list.  
Clear() – which removes all the elements from the list.  
Contains() – which returns true if this list contains the specified object.  
Insert() – which inserts an element at the specified position.  
Remove() – which removes the first occurrence of an object from a list.  
Sort() – which sorts the element of the list.

Go online to [msdn.microsoft.com](http://msdn.microsoft.com). Find API for the HashSet class defined in the System.Collections.Generic namespace and find out which of the methods in the List class have equivalent methods in the HashSet class.

### Feedback 4

You should find Methods Add(), Clear() Contains() and Remove() methods also exist for a HashSet.

Insert() and Sort() do not exist. As sets are unsorted objects you cannot sort them and as they have no specific order it make no sense to try to insert an object at a specified position.

There is much commonality between the different types of collection. Having learnt how to use one it is relatively easy to learn another.

### Activity 5

The code below will find a String in a set of Strings (called StringSet). Amend this so this will work find a Publication in a set of Publications (called PublicationSet).

```
public void FindString(String s)
{
    boolean found;

    found = StringSet.Contains(s);

    if (found)
    {
        Console.WriteLine("Element " + s + " found in set");
    }
    else
    {
        Console.WriteLine("Element " + s + " NOT found in set");
    }
}
```

### Feedback 5

```
public void FindPublication(Publication p)
{
    boolean found;

    found = PublicationSet.Contains(p);

    if (found)
    {
        Console.WriteLine("Element " + p + " found in set");
    }
    else
    {
        Console.WriteLine("Element " + p + " NOT found in set");
    }
}
```

### Activity 6

Consider the set of Publications that would need to be created for the code in Activity 5 to work and answer the following questions.

- 1) Could such a set be used to store a collection of books?
- 2) Could it store a combination of books and magazines?
- 3) If a book was found in the set, what would the following line of code do?

Console.WriteLine("Element " + p + " found in set");

### Feedback 6

- 1) Yes. Books are a subtype of publication and could be stored in a set of publications.
- 2) Yes. For the same reasons we could store a combination of books and magazines objects (or any other type of publication). This collection is therefore a polymorphic collection.
- 3) 'p' would invoke the `ToString()` method on the publication. The CLR engine would determine at run time that this was in fact a book and assuming the `ToString()` method had been overridden for book, to return the title and author, this would make up part of the message displayed. Thus, polymorphically the message would change depending upon which type of publication was found in the collection.

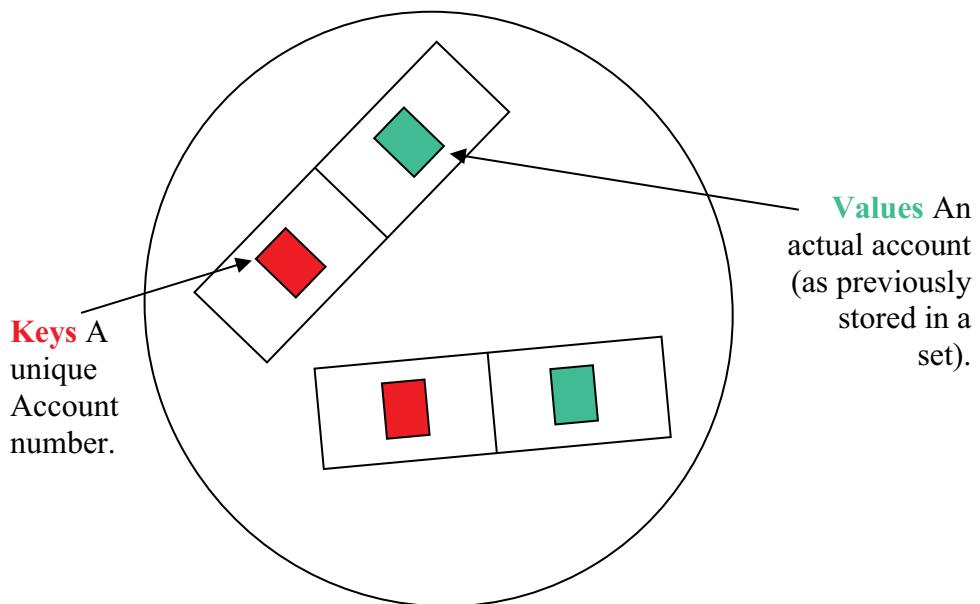
## 9.12 AN EXAMPLE USING DICTIONARIES

In the previous example we assumed no bank accounts would be created for two customers with the same name and DOB. However, it is possible that two customers could have the same name and DOB.

To make accounts truly unique we could amend the definition of Account to contain an account number. We could write some code to automatically generate account numbers that are unique and we could continue to store these using a set. However, as we will often retrieve an account using an account number, a dictionary would be a more efficient collection for this application.

A dictionary is like a set but where each item stored is made up of a key/value pair.

In this case the key would be a unique account number and the value would be the associated bank account object (see figure below).



In the previous example we stored a collection of Accounts in a set, we do not need to make any changes to the Account class if we wish to store these in a dictionary.

We could remove the overridden methods for Equals() and GetHashCode() from the Account class as a dictionary can contain duplicate values. Alternatively, we could leave these methods in should we wish to.

In a dictionary it is the keys that are unique.

There are some significant changes to the Bank class as it now uses a Dictionary. The code for the Bank class is shown below:

```
class Bank
{
    private Dictionary<int,Account> accounts;
    public Dictionary<int,Account> Accounts
    {
        get { return accounts; }
    }

    public Bank()
    {
        accounts = new Dictionary<int, Account>();
    }

    public void AddAccount(int number,Account account)
    {
        try
        {
            accounts.Add(number, account);
        }
        catch (Exception)
        {
        }
    }

    public Account GetAccount(int number)
    {
        Account a;
        accounts.TryGetValue(number, out a);
        return a;
    }
}
```

In the code above when creating the dictionary you can see that two data types are specified: Firstly, the type for the dictionary key is specified – in our case we will use a simple ‘int’ for an account number. Secondly, the data type for the value is specified – in our case the value is a complete account object. Hence the segment of code below:

```
private Dictionary<int,Account> accounts;
```

The dictionary class defines two very useful methods for us, Add() and TryGetValue().

The Add() method requires two parameters, the ‘Key’ and the ‘Value’. Hence we must provide both the account number and the account object to be added to the dictionary:

```
accounts.Add(number, account);
```

Note: This method will generate an exception if the ‘key’ is not unique and this error should be trapped and dealt with...though in this simple case we have decided to take no action should this occur.

The TryGetValue() method requires an input argument, the key it is searching for, and it then sets the value of the second argument to the value associated with that key. In our example it sets the second parameter to the account associated with the account number.

Note: This will be null if the account number does not exist. See code below:

```
accounts.TryGetValue(number, out a);
```

The Main method used to test this dictionary is given below:

```
static void Main(string[] args)
{
    Bank b = new Bank();
    Account a;
    int an; // account number;

    a = new Account("Bert", "06/12/1963");
    b.AddAccount(1, a);

    // try to create and add a duplicate account-this shouldn't work
    a = new Account("Bert", "06/12/1963");
    b.AddAccount(1, a);

    a = new Account("Alice", "14/08/1990");
    b.AddAccount(2, a);
    a = new Account("Claire", "1/1/2000");
    b.AddAccount(3, a);

    // Display the accounts
    foreach (KeyValuePair<int, Account> kvp in b.Accounts)
    {
        an = kvp.Key;
        a = kvp.Value;

        Console.WriteLine("Account number: " + an +
                           "\nAccount details: " + a.ToString());
    }

    a = b.GetAccount(1);
    a.DepositMoney(100);

    foreach (KeyValuePair<int, Account> kvp in b.Accounts)
    {
        an = kvp.Key;
        a = kvp.Value;
        Console.WriteLine("Account number: " + an +
                           "\nAccount details: " + a.ToString());
    }

    Console.ReadLine();
}
```

Functionally the 'Main' method above is virtually identical to the Main method used to test the Set example.

The code above performs the following operations:

- Several accounts are created, each with an account number, including one attempt to create a duplicate.
- The collection of accounts is displayed, in this case including an account number.
- An account is retrieved and amended.
- The collection is displayed again.

To keep this example short we have not protected against the possibility that the account being retrieved cannot be found.

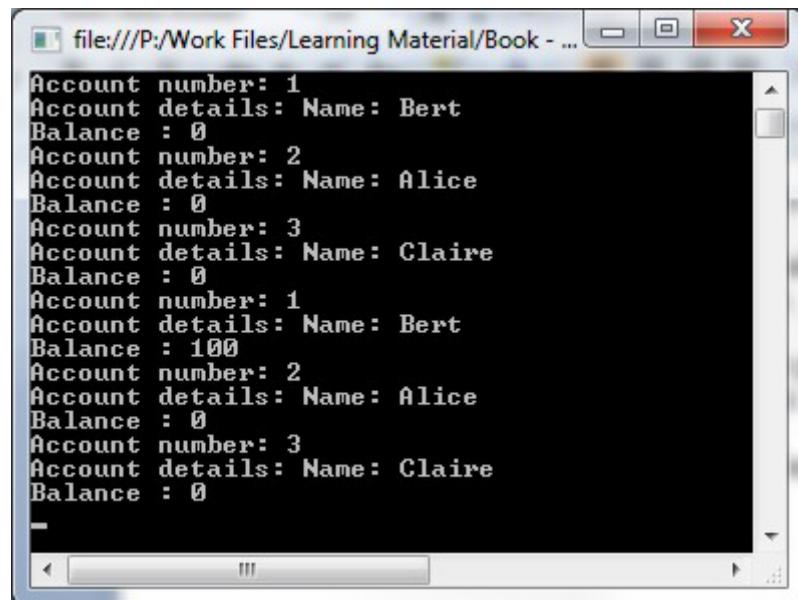
The following lines of code which display the contents of the collection are perhaps worthy of a fuller explanation:

```
foreach (KeyValuePair<int, Account> kvp in b.Accounts)
{
    an = kvp.Key;
    a = kvp.Value;
    Console.WriteLine("Account number: " + an +
                      "\nAccount details: " + a.ToString());
}
```

Each object in the collection is made up of a key/value pair. Thus when we iterate around the collection each iteration returns a key/value pair.

Above we define 'kvp' a variable made up of a key/value pair where the key is an 'int' (i.e. an account number) and the value is an Account object. We split this pair into its component parts and store these in the respective variables (account number 'an' and account 'a'). We then display the account number and invoke the `ToString()` method on the account, using the result to display details of the account.

The output from this program is shown below:



```
Account number: 1
Account details: Name: Bert
Balance : 0
Account number: 2
Account details: Name: Alice
Balance : 0
Account number: 3
Account details: Name: Claire
Balance : 0
Account number: 1
Account details: Name: Bert
Balance : 100
Account number: 2
Account details: Name: Alice
Balance : 0
Account number: 3
Account details: Name: Claire
Balance : 0
-
```

As we would expect, the system did not allow us to create accounts with duplicate account numbers.

If you look at the Dictionary class in the System.Collections.Generic namespace of the .NET framework you will see a host of other useful methods that have been created to manage Dictionaries. These include Clear(), ContainsKey(), ContainsValue() and Remove().

## 9.13 SERIALIZING AND DE-SERIALIZING COLLECTIONS

Collections, Lists, Sets and Dictionaries among others, are very powerful flexible mechanisms for storing collections of objects.

They automatically resize themselves and contain methods that save significant programming effort when adding members, retrieving members, removing members, searching for members, sorting collections etc.

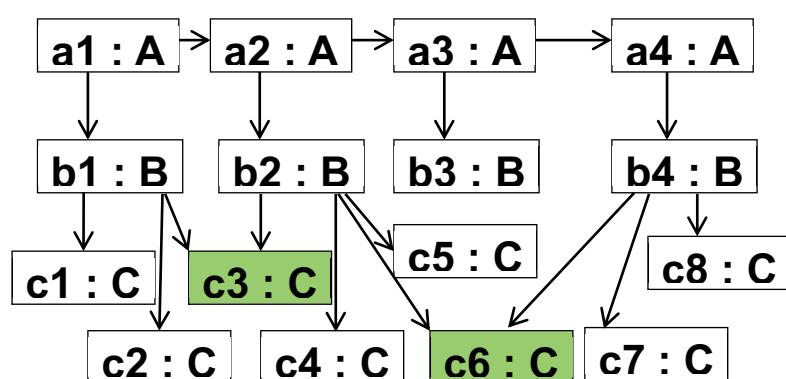
They become even more powerful when combined with the serialization/de-serialization facilities provided by the .NET framework.

Using traditional file handling routines textual data can be stored and retrieved from files.

Serialization allows whole objects to be stored with one simple command (once an appropriate file has been opened).

C# objects frequently contain references to other objects, they create what is known formally as a 'graph' – a network of connections. The serialization mechanism follows these references and also serializes the objects referenced...and objects those objects reference...etc.

An object graph is shown below:



In this graph we can see that object a1 refers to objects a2 and b1. These in turn refer to other objects. Thus if we were to fully save (or serialize) a1 we would need to store all the

details of this object including details of all the other objects this refers to... and all the objects these refer to and so on.

Serialisation will follow these links automatically.

Notice that c3 and c6 are referred to by two objects. The serialization mechanism is smart enough to realise that the same object is occurring when it encounters it for the second time and, instead of writing out a duplicate, it simply writes out a reference to the original object. By the same means a ‘cycle’ in the graph – i.e. a circle or references returning to the same place – will not cause the serialization mechanism to go into an infinite loop!

This mechanism assumes that classes A, B and C are all serializable. Some classes are not. For instance those that rely on reading from a file cannot be serialised as the file may not exist when we attempt to deserialise an object of that class. In practice most of the classes we create will be serializable.

To demonstrate the power of the serialization mechanism we will amend the bank account system developed in section 9.12 where a dictionary was used to store a collection of accounts.

With one instruction we will store an object of type Bank. In doing so, the system will also store the collection Accounts and in doing so it will store **all** Account objects.

When we retrieve the Bank object all Account objects will also be retrieved and the collection reconstructed.

In order to do this no changes need to be made to the Account class or the Bank class though we must tell the compiler these classes can be serialised.

To do this we place the keyword `serializable` in front of each class as shown below...

```
[Serializable]
class Account
{
    // code from body of class omitted

}

[Serializable]
class Bank
{
    // code from body of class omitted
}
```

In the ‘Main’ method we need to add additional using statements as we are using additional parts of the .NET framework (see below). We will then invoke the serialization\de-serialization process and test this works.

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using System.IO;
```

A complete ‘Main’ method that will demonstrate this working is provided below. Much of this is either self-explanatory or contains code we are already familiar with. The new parts that relate to the serialization/de-serialization will be explained afterwards.

```
static void Main(string[] args)
{
    Bank b = new Bank();
    Account a;
    int an; // account number;

    a = new Account("Bert", "06/12/1963");
    b.AddAccount(1, a);
    a = new Account("Alice", "14/08/1990");
    b.AddAccount(2, a);
    a = new Account("Claire", "1/1/2000");
    b.AddAccount(3, a);

    Console.WriteLine("Stored data");
    foreach (KeyValuePair<int, Account> kvp in b.Accounts)
    {
        an = kvp.Key;
        a = kvp.Value;

        Console.WriteLine("Account number: " + an +
                           "\nAccount details: " + a.ToString());
    }

    FileStream outFile = new FileStream("AccountData",
                                         FileMode.Create, FileAccess.Write);
    BinaryFormatter bFormatter = new BinaryFormatter();
    bFormatter.Serialize(outFile, b);
    outFile.Close();
    outFile.Dispose();

    Bank b2 = new Bank();
    FileStream inFile = new FileStream("AccountData", FileMode.Open,
                                       FileAccess.Read);
    b2 = (Bank)bFormatter.Deserialize(inFile);
    inFile.Close();
    inFile.Dispose();
```

```
Console.WriteLine("\nRetrieved data");
foreach (KeyValuePair<int, Account> kvp in b2.Accounts)
{
    an = kvp.Key;
    a = kvp.Value;

    Console.WriteLine("Account number: " + an +
                      "\nAccount details: " + a.ToString());
}

Console.ReadLine();
}
```

The code above performs the following steps:

- Firstly several bank accounts are created, added to the collection and the collection is then displayed.
- Next an output file is created.
- A binary formatter is created as this is used by the serialization process.
- With one line the bank object ‘b’ is serialised and in doing so all account objects, stored as a collection inside b, are serialized (see code below).

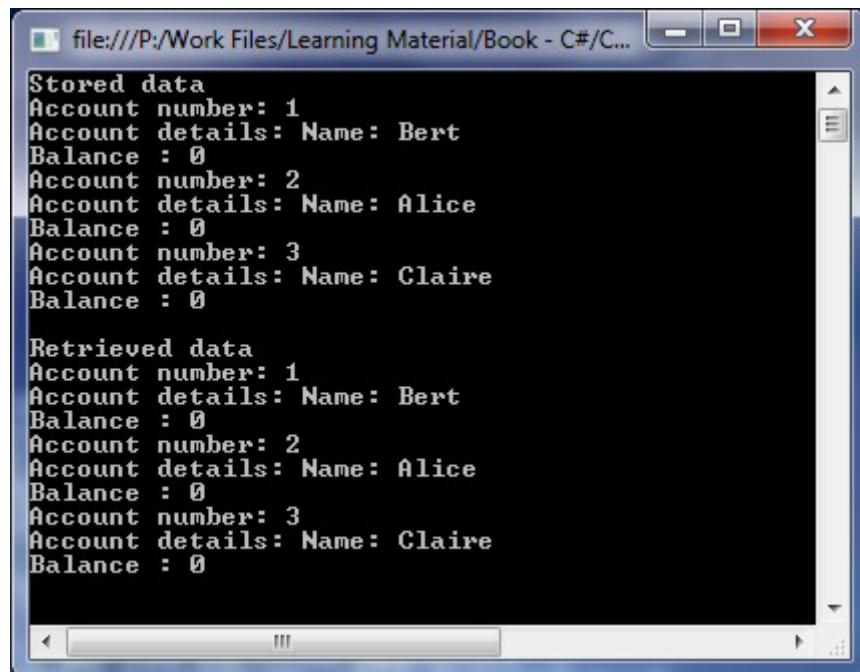
```
bFormatter.Serialize(outFile, b);
```

- In order to prove this has worked a new, empty Bank object is created ‘b2’.
- The de-serialization process is invoked and the output cast into a Bank type. The results are then stored in the object ‘b2’ (see code below).

```
b2 = (Bank)bFormatter.Deserialize(inFile);
```

- Finally the contents of ‘b2’ are displayed to show they are consistent with the data originally created.

The results of running this program are shown below.



A screenshot of a Windows Command Prompt window titled "file:///P:/Work Files/Learning Material/Book - C#/C...". The window displays two sections of text: "Stored data" and "Retrieved data", each containing three account entries. The accounts are defined by their number, name, and balance, all of which are currently zero.

```
Stored data
Account number: 1
Account details: Name: Bert
Balance : 0
Account number: 2
Account details: Name: Alice
Balance : 0
Account number: 3
Account details: Name: Claire
Balance : 0

Retrieved data
Account number: 1
Account details: Name: Bert
Balance : 0
Account number: 2
Account details: Name: Alice
Balance : 0
Account number: 3
Account details: Name: Claire
Balance : 0
```

## 9.14 THE POWER OF SERIALIZATION

Serialisation is a powerful mechanism that saves a considerable amount of time especially when dealing with more complex collections of objects than the simple collection of bank accounts shown in the previous section.

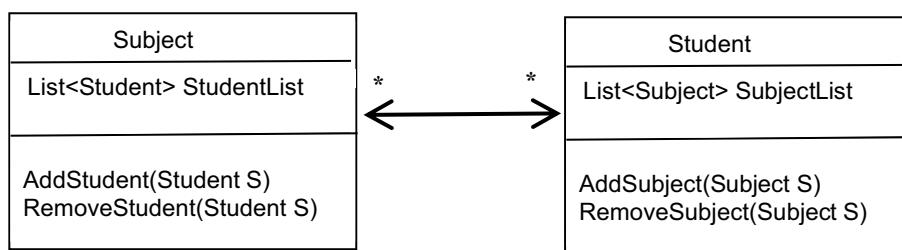
Consider the following scenario:

A University maintains a list of students and also a list of classes/subjects taught.

These lists are not separate unconnected lists: Students register for specific subjects and occasionally de-register if they want to take a different subject instead...so each subject must maintain a list of students who are taking that subject.

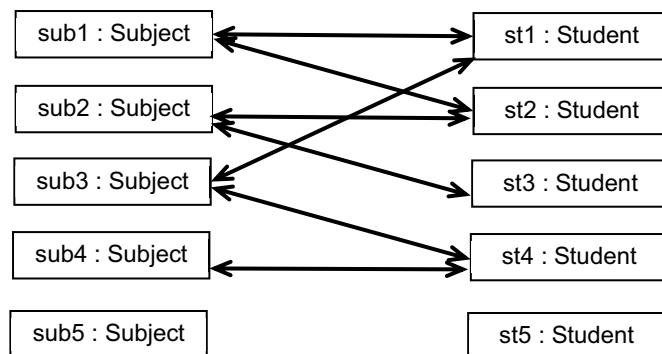
Similarly each student must maintain a list of subjects taken by that student...so the university can check which subjects they are taking.

A class diagram to represent this system is shown below:



Note: the class **Subject** contains a list of **Students** and in the class **Student** contains a list of **subjects**.

The objects, i.e. the subjects and the students, at one particular moment in time can be represented by an object diagram, see example below:



The figure above shows a list of subjects, sub1, sub2, sub3 etc., and a list of students, st1, st2, st3 etc.

The arrows on the diagram show which objects are related to other objects. Thus, subject sub1 is being studied by student st1 and st2. Student st1 is taking subject sub1, sub2 and also sub3.

Currently, no student is registered to take subject sub5 and student st5 is not taking any subjects.

### Activity 7

Consider the object diagram shown above and answer the questions below:

- 1) How would you store and retrieve all of this data using a text file?
- 2) Would a program to do this need to change if we added an extra field to one of the classes, e.g. if we add a mobile telephone number to the student class?
- 3) On retrieval would the objects get reconnected automatically or would we need to write some program code to remake the connections?
- 4) Would this be simpler if we were to use object serialization to store the objects?
- 5) If the object sub1 were serialized which objects in the diagram above would be stored and which would not?
- 6) Could we make it simple and ensure all of the objects are stored and retrieved with just a few lines of code?

### Feedback 7

- 1) How would you store and retrieve all of this data using a text file?

Writing the code to store all of this data to a text file would be a long and fiddly process.

We would need to write some code that would iterate around the collection of subjects and for each subject write out the details of that subject and the list of the students taking that subject to a text file. In essence we are taking the data out of each object and storing this data manually – one step at a time.

We would also need to iterate around the list of students writing the details of each student out to a file.

The file could end up looking like the following...

SUBJECTS

sub1

Title of subject

Details of subject

Name of teacher

List of students taking this subject...st1, st2 etc. etc.

(details repeated for each subject)

STUDENTS

st1

Students name

Students ID number

...and possibly the list of subjects taken by the student (though this can be inferred from the list already stored).

We would also need to read in the data associated with each object in the system and invoke the constructor for that object type to recreate each associated object. As the code to read in the data is dependent on the file structure this is a fiddly part of the system to code.

- 2) Would a program to do this need to change if we added an extra field, e.g. mobile telephone number, to the student class?

If you changed the data associated with a class, e.g. added a mobile telephone number, then you would need to change the code that writes the data to a file and you would need to change the associated code that reads in the data and invokes the constructor.

**This would be even more complex if we made more substantial changes to our system e.g. if we added new types of student...sub classes of the Student class.**

- 3) On retrieval would the objects get reconnected automatically or would we need to write some program code to remake the connections?

As we would need to recreate each object, one at a time, we would also need to reconnect them. After recreating all of the subject objects and all of the student objects we would then need to use the student list to register the correct students on to each subject and add the subjects to the student records...i.e. we would need to manually recreate all of the objects and all of the connections.

Essentially we would be taking all of the data from the file and recreating the entire object graph one step at a time.

- 4) Would this be simpler if we were to use object serialization to store the objects?

This whole process would be massively simplified if instead of using a text file to store the data we used object serialization to store the objects. With just a few lines of code we could serialize all of these objects and with another few lines we could de-serialise the objects.

As the object graph is de-serialised all of the connections are remade automatically.

We would not need to invoke constructors to recreate the objects.

We would not need to remake the connections i.e. we would not need to say which students are taking which classes.

And best of all...if the structure of the student class is changed by adding an extra field, or if we add new sub types of student (new sub classes), then we would not need to change our code at all.

The serialisation process does not need to be told about the structure of the objects or the object graph. The serialisation and de-serialisation process is totally automated and if the object graph changes no changes are required in our code.

All of this saves a massive amount of programming time and effort and reduces the risk of errors being introduced into our code as the classes change.

- 5) If the object sub1 were serialized which objects in the diagram above would be stored and which would not?

Serializing sub1 will store this object and all objects associated with it... thus st1 and st2 are stored. Storing st1 and st2 will in turn store sub2 and sub3 as these are associated with these students.

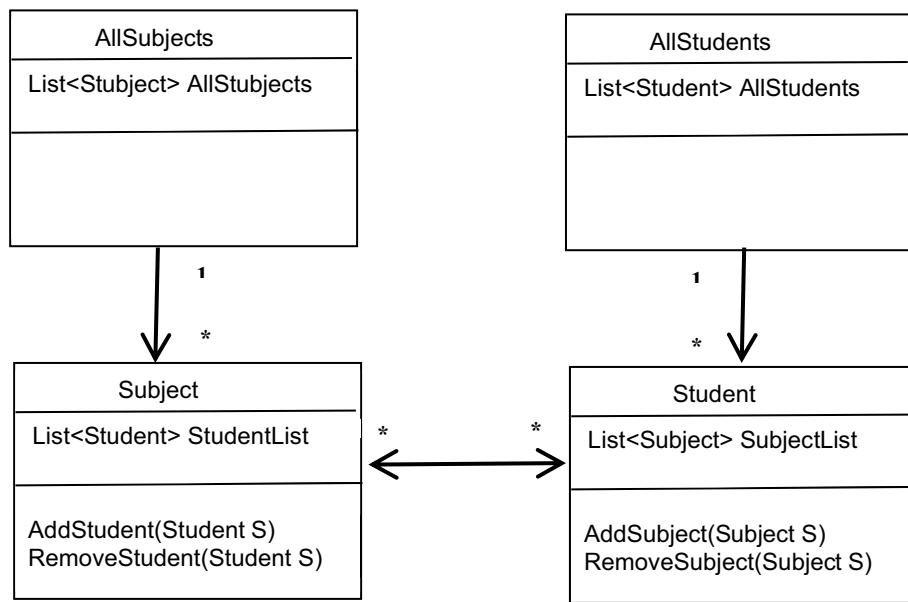
In turn all associated objects will be stored. However, sub5 and st5 are not associated with anything and therefore these objects would not be stored.

We need to find a simple way of storing all of the subjects and all of the students – even if a subject does not have any students enrolled on them and even if a student is not currently enrolled on any classes.

- 6) Could we make it simple and ensure all of the objects are stored and retrieved with just a few lines of code?

It is simple to ensure that all of the objects are stored – including those not currently connected in the object graph shown above.

The answer is to maintain a list of all subjects and a list of all students (as shown in the class diagram below).



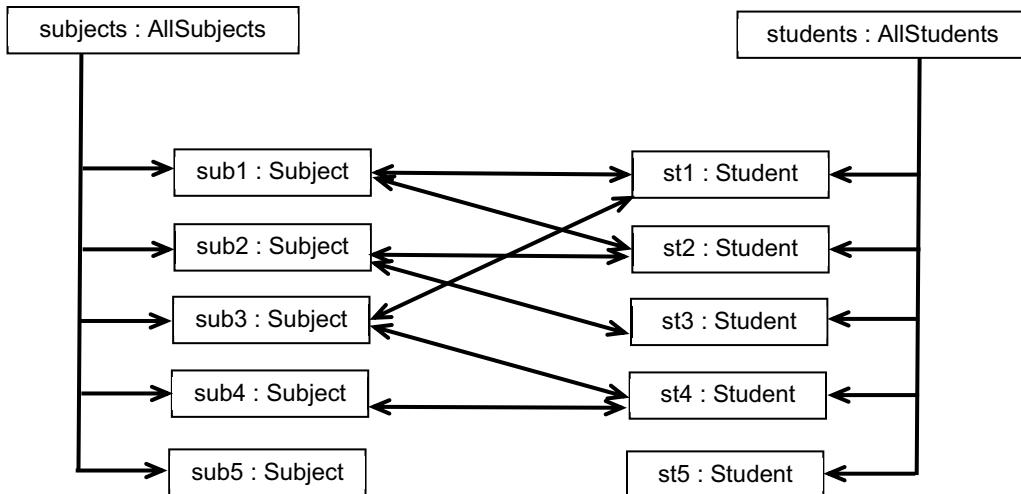
### Feedback 7 Continued

In the system we would then create one object of type 'AllSubjects' and one object of type 'AllStudents'. As new subjects and new students are created they would be added to these lists.

Serialising these two list objects would then store all of the associated objects i.e. all of the subjects and all students – including those that would not previously have been stored.

Of course all subject-student connections would be stored at the same time and de-serializing these two objects would recreate both lists, all of the objects within the lists and all of the connections between these objects.

The revised object diagram shown below indicates how serialisation will now store all of the objects including sub5 and st5 as they are now all associated with the two list objects.

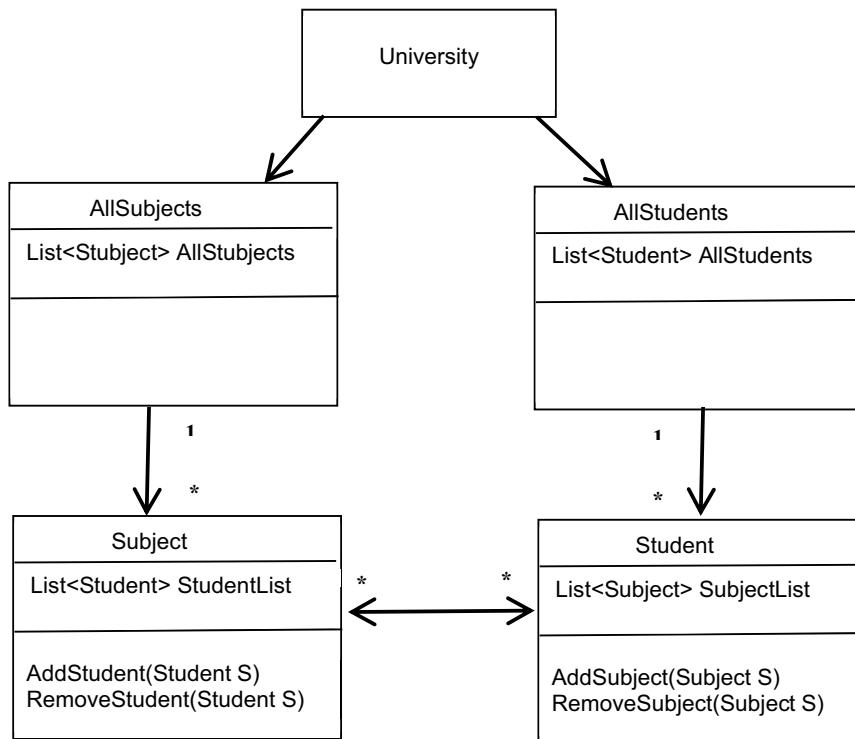


### Feedback 7 Continued

Serialising the lists shown above would essentially only take two lines of code but we could even take this one step further...

If we created a 'University' class, as shown in the class diagram below, we could then create one object of that class and use that to store the two lists...we could then serialise all of the objects in our system by serializing this one object i.e. in one simple step.

Whether creating this extra class is worthwhile, or not, really depends upon how much other data is stored within it and what useful methods it would contain.



**Serialization is a powerful and time saving mechanism as it allows entire object graphs to be stored in a single step.**

## 9.15 SUMMARY

The .NET Collections classes provide ready-made methods for storing collections of objects. These almost completely make the use of arrays redundant!

There are ‘untyped’ collections and ‘typed’ collections. Typed collections, or generic collections, were developed based upon the idea of generic methods and generic classes. Generic collections are much more useful than untyped collections and thus we have not shown the use untyped collections here.

Collection classes include List, HashSet and Dictionary. Each of these define appropriate methods, many of which are common across all of the collection classes.

Special attention is required when defining objects to be stored in Sets (or as keys in Dictionaries) to define the meaning of ‘duplicate’. For these we need to override the Equals() and GetHashCode() methods inherited from Object.

Serialization is a very powerful mechanism that allows the entire contents of a collection and all related objects to be stored with one simple command.

The serialisation process does not need to be reprogrammed as our system changes, i.e. as classes are restructured and new subclasses are created, which saves time. Furthermore, as we don't need to code our own data storage methods, serialisation reduces the chance of errors in our code.

A further example of the use of collections and the serialisation process will be provided in Chapter 14, a larger case study at the end of this book. This will demonstrate the use of lists, sets and dictionaries for a small but more realistic example application. The code for this will be available to download and inspect if required.

# 10 C# DEVELOPMENT TOOLS

## Introduction

This chapter will introduce you to several development tools that support the development of large scale C# systems. We will also consider the importance of documentation and show how tools can be used to generate documentation for systems you create (almost automatically). Automated documentation is essential to support modern agile Software Engineering methods (as discussed in Chapter 12).

## Objectives

By the end of this chapter you will be able to...

- Find details of several professional and free interactive development environments.
- Understand the importance of the software documentation tools and the value of embedding XML comments within your code.
- Write XML comments and generate automatic documentation for your programs.

This chapter consists of eight sections:

- 1) Tools for Writing C# Programs...
- 2) Microsoft Visual Studio
- 3) SharpDevelop
- 4) Automatic Documentation
- 5) Sandcastle Help File Builder
- 6) GhostDoc
- 7) Adding Namespace Comments
- 8) Summary

## 10.1 TOOLS FOR WRITING C# PROGRAMS

Whatever mode of execution is employed (see Chapter 1), programmers can work with a variety of tools to create source code. It is possible to write C# programs using simple discrete tools such as a plain text editor (e.g. Notepad) and a separate compiler invoked manually as required. However virtually all programmers would use a powerful Integrated

Development Environment (IDE) which use compilers and other standard tools behind a seamless interface.

Even more sophisticated tools Computer Aided Software Engineering (CASE) tools exist which integrate the implementation process with other phases of the software development lifecycle. CASE tools could take UML class diagrams, generated as part of the software analysis and design phase, and generate classes and method stubs automatically saving some of the effort required to write the C# code (i.e. the implementation phase).

Some CASE tools also help by automating the software testing phase.

Overtime, as more powerful tools are being created, IDEs have begun to incorporate some of the features previously found only in CASE tools i.e. they do much more than allow programs to be written, edited and compiled. One such tool is Microsoft Visual Studio.

## 10.2 MICROSOFT VISUAL STUDIO

Moving to an ‘industrial strength’ IDE is an important stepping stone in your progress as a software developer, like riding a bicycle without stabilisers for the first time. With some practice, you will soon find it offers lots of helpful and time-saving facilities that you will not want to work without again.

Microsoft Visual Studio is a powerful IDE platform that supports the development of .NET programs written in a range of languages not just C#.

Details of this can be found via <https://www.visualstudio.com/>

Visual Studio comes in different varieties but perhaps the two most common are Visual Studio Community edition and the Enterprise edition. The Enterprise edition was used to create all of the code in this book however the Community edition is free and still contains all of the features needed by most students.

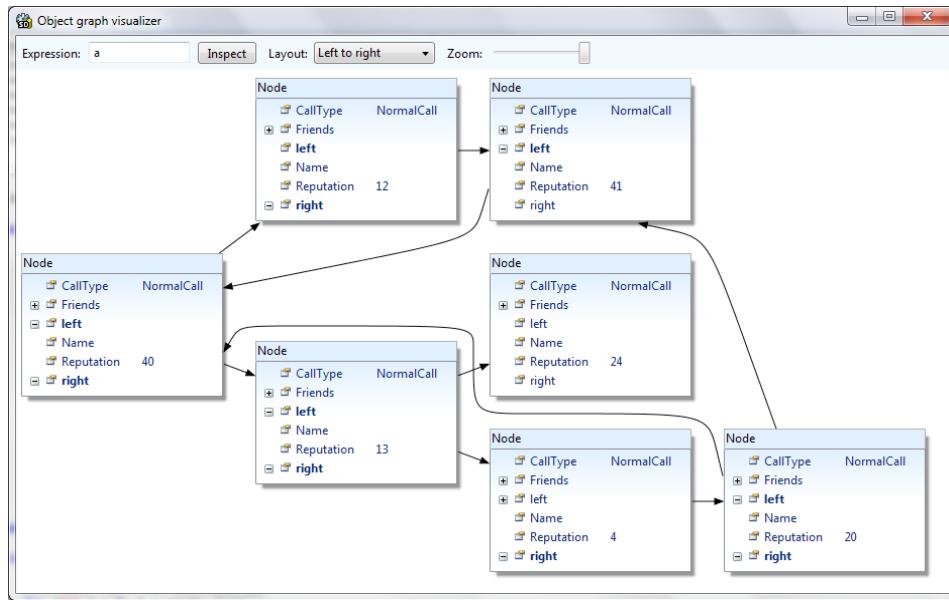
Visual Studio Community edition can be downloaded from the website listed above.

### 10.3 SHARPDEVELOP

Another very powerful and free tool that can be used to develop C# programs is SharpDevelop (or #develop). It is open-source so you can download both the source code and executable versions.

SharpDevelop includes support for several .NET languages including C# and VB.NET. It includes unit testing facilities (via optional plug ins), code completion facilities and powerful debugging facilities.

To help debugging, SharpDevelop includes an Object graph visualiser (see below). This visualiser displays a visual representation of your data structures that is dynamically updated when stepping through your code.



For more information on SharpDevelop or to download it go to <http://www.icsharpcode.net/OpenSource/SD/Default.aspx>

It should be noted that other free tools exist including Xamarin ([www.xamarin.com](http://www.xamarin.com)), an IDE which offers multiplatform support including iOS, Android and Windows.

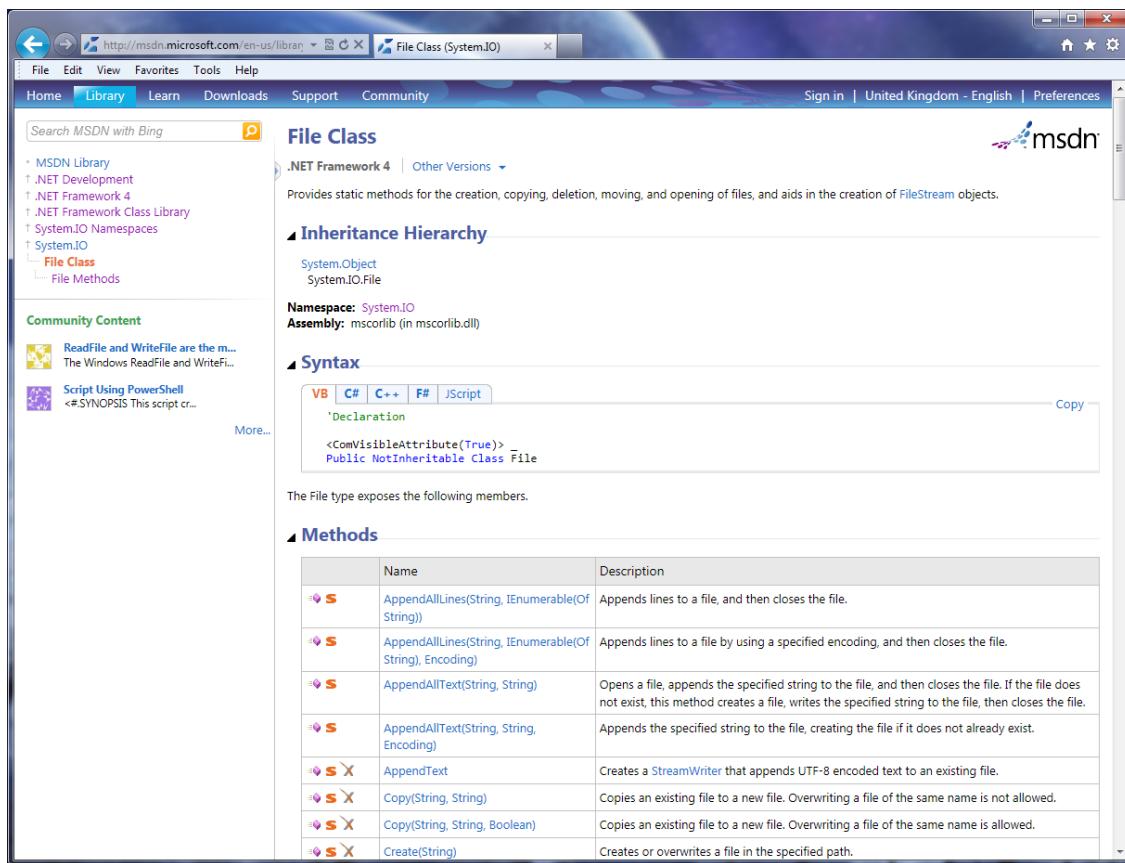
## 10.4 AUTOMATIC DOCUMENTATION

One particularly useful feature found within some IDE's is the ability to generate automatic documentation for the programs we create.

Programmers for many years have been burdened with the tedious and time consuming task of writing documentation. Some of this documentation is intended for users and explain what a program does and how to use it. Other documentation is intended for future programmers who will need to amend and adapt the program so that its functionality will change as the needs of an organisation change. These programmers need to know what the program does and how it is structured. They need to know:

- What packages it contains?
- What classes exist in each of these packages and what these classes do?
- What methods exist in each class and for each of these methods
  - What do the method do?
  - What parameters does it require?
  - What, if any, value is returned?

Tools can't produce a user guide but they can provide a technical description of the program. Tools can analyse C# source files for the programs we write and produce documentation, either as a set of web pages (HTML files) or in some other format. Technical documentation should contain similar information as you would find when looking MSDN to find details of the .NET libraries.



This website provides, as a set of indexed web pages, essential details of the .NET libraries including all packages, classes, methods and method parameters and return values. This extremely useful information was not generated by hand but generated using the automatic tools.

Tools can produce similar detailed documentation for any C# program you create and this would help future programmers amend and update your programs. However this relies on properly formatted (and informative!) XML-style comments in source files, including tags such as @author, @param etc.

Because this documentation is generated automatically, at the push of a button, it saves programmers from a tedious, time consuming and error prone task. Furthermore the documentation can be updated whenever a program is changed.

XML, Extensible Markup Language, is a way of specifying and sharing structured information.

By adding meaningful XML comments to the code we write tools can produce automatic documentation for us. Poor attention to commenting of source code will result in poor documentation. On the other hand, if the commenting is done properly then the reference documentation is produced for virtually no effort at all.

XML comments should therefore be added at the start of every class using the standard tags `<summary>` and `<remarks>` to give an overview of that class in the following format.

```
/// <summary>
/// Provide a short description of the class and its role here...
/// </summary>
/// <remarks>Author Simon Kendal
/// Version 1.2 (19th Dec 2017)</remarks>
```

Similar comments should be provided for every method using the `/// <param name="name of parameter">`, and `/// <returns>` tags to describe each parameter and to describe the value returned. The details of each parameter, starting with the name of the parameter, should be provided on separate lines as shown below.

```
/// <summary>
/// A description of the method
/// </summary>
/// <param name="name of first parameter">A description of that parameter </param>
/// <param name="name of 2nd parameter">A description of that parameter </param>
/// <returns> A description of the value returned by a method. /// </returns>
```

### Activity 1

The method below takes two integer numbers and adds them together. This value is returned by the method. Write an XML comment, using appropriate tags, to describe this method.

```
public int add(int number1, int number2)
{
    return (number1 + number2);
}
```

### Feedback 1

```
/// <summary>
/// This method adds two integer numbers.
/// </summary>
/// <param name="number1">The first number.</param>
/// <param name="number2">The second number.</param>
/// <returns>The sum of the two numbers. </returns>
```

Automatic tools cannot analyse comments to determine if these provide an accurate description of the methods. However tools can do far more than cut and paste a programmers comments onto a web document.

Tools can analyse method signatures and compare this with the tags in the comments to check for logical errors. If a method requires three parameters but the comment only describes two then this error can be detected and reported to the programmer, see example below:

"Parameter 'name' has no matching param tag in the XML comment for 'Name of class here' (but other parameters do)".

By analysing the code the tools can also provide additional information about the class, the methods that have been inherited and the methods that have been overridden.

By using automatic tools reference documentation can be produced for programmers using the classes concerned. This documentation will not include comments within methods intended for programmers editing the class source code in maintenance work.

Automatic documentation tools do not exist as standard within either Microsoft Visual Studio or SharpDevelop. However both of these IDE's can be integrated with Sandcastle Help File Builder...this then enables automatic documentation to be generated. Additionally another tool exists, called GhostDoc, which further helps by generating the necessary XML comments.

## 10.5 SANDCASTLE HELP FILE BUILDER

To be precise Sandcastle Help File Builder (SHFB) does not produce automatic documentation for your code...but it adds a graphical front end to 'Sandcastle'. 'Sandcastle' is a difficult to use tool that lacks a GUI, and has a complex installation routine but will generate automatic documentation.

Sandcastle Help File builder is a GUI that will drive Sandcastle and will also automate the installation of Sandcastle. Sandcastle Help File Builder can be downloaded from GitHub via <https://github.com/EWSoftware/SHFB>

Running this software will install

- Sandcastle.
- Updates patches and additional files.
- Compilers for different help file formats – only the Help/HTML compiler is required.
- Sandcastle Help File Software itself.

When running SHFB you need to specify an XML file with comments generated from your code and the C# code itself i.e. a '.sln' file.

To generate the XML file from within Visual Studio, select the project Properties/Build/ XML comments.

Note: Visual Studio will complain if comments are missing from your code.

## 10.6 GHOSTDOC

SHFB is very useful as it generates automatic documentation for your programs assuming of course appropriate XML comments exist in the body of the code.

GhostDoc is one piece of software that will automatically add XML comments to your C# code. Though the comments should be edited to ensure they are meaningful.

GhostDoc can be downloaded from <http://submain.com/products/ghostdoc.aspx>

## 10.7 ADDING NAMESPACE COMMENTS

One final complication exists regarding namespace comments.

A room exists and it is possible to go outside of that room and put a sign on the door.

A building exists and we can do the same i.e. go outside and put a sign on the building but while the Universe exists we cannot go outside and put a sign on it...because outside does not exist!

A similar problem exists with namespace comments.

We know where a class starts in our code so we can put an XML comment at the start to describe that class. Similarly, we can put comments at the start of methods to describe those methods...but while namespaces exist we cannot define the start of these. So if we segment our larger systems into namespaces, as we should, we cannot put comments at the start of a namespace to describe that namespace.

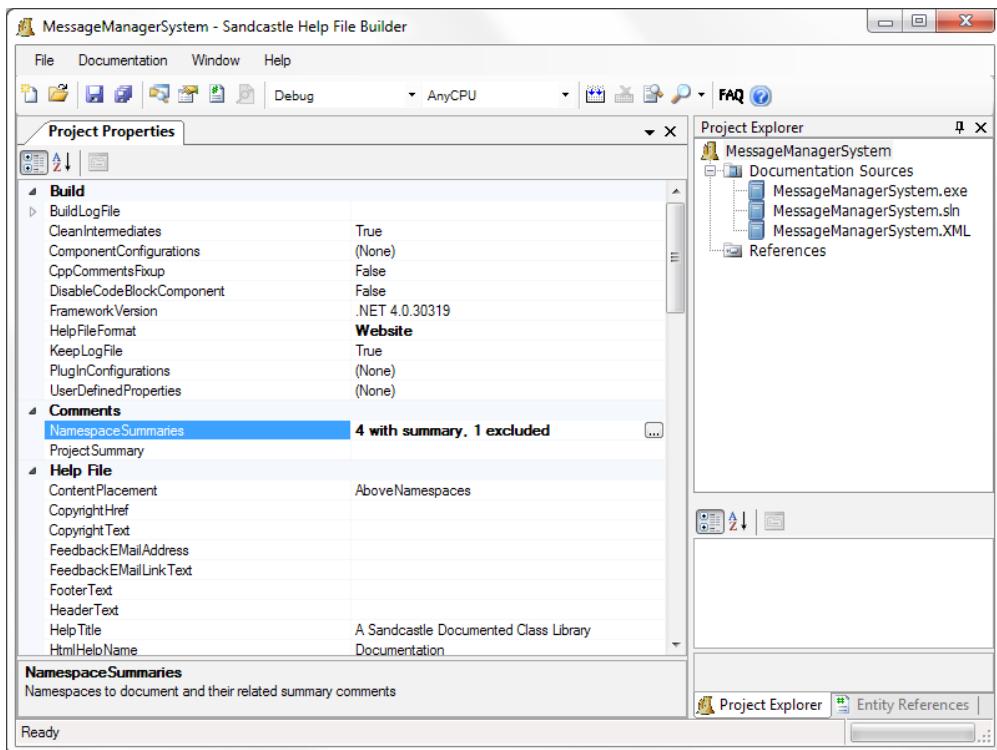
If we look at part of the MSDN documentation (see figure below) we see that it describes name spaces and we should do the same for our programs.

The screenshot shows a Microsoft Internet Explorer browser window displaying the MSDN documentation for the `System.Collections` namespace. The URL in the address bar is `http://msdn.microsoft.com/en-us/library/g`. The page title is **System.Collections Namespaces**. On the left, there's a navigation tree under ".NET Framework 4" with nodes like `System.Collections`, `System.Collections.Concurrent`, `System.Collections.Generic`, `System.Collections.ObjectModel`, and `System.Collections.Specialized`. A sidebar on the left has a "Community Content" section with a link to "Add code samples and tips to enhance this topic". The main content area contains a table titled "**Namespaces**" with five rows, each detailing a specific namespace:

Namespace	Description
<code>System.Collections</code>	The <code>System.Collections</code> namespace contains interfaces and classes that define various collections of objects, such as lists, queues, bit arrays, hash tables and dictionaries.
<code>System.Collections.Concurrent</code>	The <code>System.Collections.Concurrent</code> namespace provides several thread-safe collection classes that should be used in place of the corresponding types in the <code>System.Collections</code> and <code>System.Collections.Generic</code> namespaces whenever multiple threads are accessing the collection concurrently.
<code>System.Collections.Generic</code>	The <code>System.Collections.Generic</code> namespace contains interfaces and classes that define generic collections, which allow users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections.
<code>System.Collections.ObjectModel</code>	The <code>System.Collections.ObjectModel</code> namespace contains classes that can be used as collections in the object model of a reusable library. Use these classes when properties or methods return collections.
<code>System.Collections.Specialized</code>	The <code>System.Collections.Specialized</code> namespace contains specialized and strongly-typed collections; for example, a linked list dictionary, a bit vector, and collections that contain only strings.

But if we cannot define the start of a namespace we must put those comments elsewhere.

SHFB provides a solution to this problem by allowing us to enter the namespace comments for our system directly into SHFB.



While it may involve several pieces of software and take a while to set up the ability to generate automatic documentation can save hours and hours of effort. What is more each time we edit and amend our programs we can then generate automatic documentation within a few button pushes.

Other programmers may need to amend and update our the programs we create. As professional programmers we have a duty to the to provide documentation to make this task as easy as possible. Automatic tools, such as SHFB and GhostDoc, can help us do this.

We will discuss the use of Agile methodologies in Chapter 12. Agile methodologies promote rapid changes in order to enhance the software we develop. If we make rapid changes to software the associated documentation also needs to change rapidly and the tools that generate automatic documentation become even more important.

## 10.8 SUMMARY

We can go ‘back to basics’ creating C# programs using a text editor and stand-alone compilers (freely available) but the use of a professional IDE can offer additional facilities to support programmers.

Specialist tools are available for aspects of development such as GUI design, diagramming and documentation. Some IDEs go beyond the basics and provide some of these facilities directly within the IDE.

Visual Studio Community edition and SharpDevelop are two free powerful IDEs which support the development of C# and other .NET programs.

These tools can initially seem daunting as they provides extensive support for professional development including code formatting and refactoring though online help does exist. Both of these tools provide some level of automatic code generate, e.g. the main method, to make the job of the programmer easier.

Professional programmers have a duty to create up to date documentation. The SHFB software is an extremely useful and timesaving by helping to create this documentation but it does require the programmer to inserting meaningful XML comments into their code (for all classes and all public methods).

GhostDoc helps the programmer by adding XML comments though they will need some manual editing.

Automatic documentation becomes even more important when using an Agile methodology.

# 11 CREATING AND USING CUSTOMISED EXCEPTIONS

## Introduction

If you have written C# programs that make use of the file handling facilities you will have probably written code to catch exceptions i.e. you will have used try\catch blocks. This chapter explains the importance of creating your own exceptions and shows how to do this by extending the Exception Class and using the ‘Throw’ mechanism.

## Objectives

By the end of this chapter you will be able to...

- Appreciate the importance of exceptions.
- Understand how to create your own exceptions.
- Understand how to throw these exceptions.
- Understand how to handle exception hierarchies.

This chapter consists of eight sections:

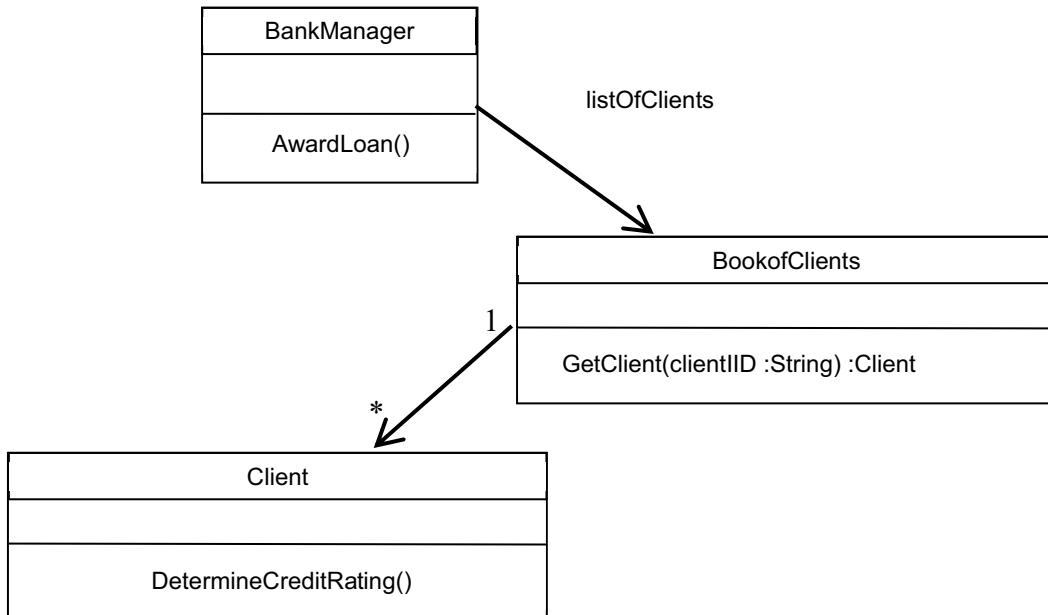
- 1) Understanding the Importance of Exceptions
- 2) Kinds of Exception
- 3) Extending the ApplicationException Class
- 4) Throwing Exceptions
- 5) Catching Exceptions
- 6) Exception Hierarchies
- 7) A ‘Finally’ Block
- 8) Summary

## 11.1 UNDERSTANDING THE IMPORTANCE OF EXCEPTIONS

Exception handling is a critical part of writing C# programs. The authors of the file handling classes within the C# language knew this and created routines that made use of C# exception handling facilities – but are these really important? and do these facilities matter to programmers who write their own applications using C#?

**Activity 1**

Imagine part of a banking program made up of three classes, and three methods as shown below...



The system shown above is driven by the **BankManager** class. The **AwardLoan()** method is invoked, either via the interface or from another method. This method is intended to accept or reject a loan application.

The **BookofClients** class maintains a set of account holders...people are added to this set if they open an account and of course they can be removed. However, the only method of interest to us is the **GetClient()** method. This method requires a string parameter (a client ID) and either returns a client object (if the client has an account at that bank) – or returns NULL (if the client does not exist).

The **Client** class has only one method of interest, **DetermineCreditRating()**. This method is invoked to determine a client's credit rating – this is used by the **BankManager** class to decide if a loan should be approved or not.

Considering the scenario above, look at the snippet of code below...

```
Client c = listOfClients.GetClient(clientID) ;
c.DetermineCreditRating();
```

This fragment of code would exist in the **AwardLoan()** method. Firstly, it would invoke the **GetClient()** method, passing a client ID as a parameter. This method would return the appropriate client object (assuming of course that a client with this ID exists) which is then stored in a local variable 'c'. Having obtained a client, the **DetermineCreditRating()** method would be invoked on this client.

Look at these two lines of code. Can you identify any potential problems with them?

### Feedback 1

If a client with the specified ID exists, the code above will work. However, if a client does not exist with the specified ID the GetClient() method will return NULL.

The second line of code would then cause a run time error (specifically a NullReferenceException) as it tries to invoke the DetermineCreditRating() method on a non-existent client and the program would crash at this point.

### Activity 2

Consider the following amendment to this code and decide if this would fix the problem.

```
Client c = listOfClients.GetClient(pClientID) ;  
If (c !=NULL) {  
    c.DetermineCreditRating();  
}
```

### Feedback 2

If the code was amended to allow for the possible NULL value returned it would work. However, this protection is insecure as it relies on the programmer to spot this potential critical error.

When writing the GetClient() method the author was fully aware that a client may not be found and in this case decided to return a NULL value. However, this relies on every programmer who ever uses this method to recognise and protect against this eventuality.

If any programmer using this method failed to protect against a NULL return, then their program could crash – potentially in this case losing the bank large sums of money. Of course in other applications, such as an aircraft control system, a program crash could have life threatening results.

A more secure programming method is required to ensure that that a potential crash situation is always dealt with!

Such a mechanism exists – it is a mechanism called ‘exceptions’.

By using this mechanism we can trap any potential errors in our code – preventing and managing crash situations.

In the situation above we could write code that would catch `NullReferenceExceptions` and decide how our program should respond to these. Of course we may not know specifically what generated the `NullReferenceException` so we may not know how our program should respond. We could at least shut down our program in a neat and tidy way, explaining to the user we are doing this because of the error generated. Doing this would be far better than allowing our program to crash without explanation.

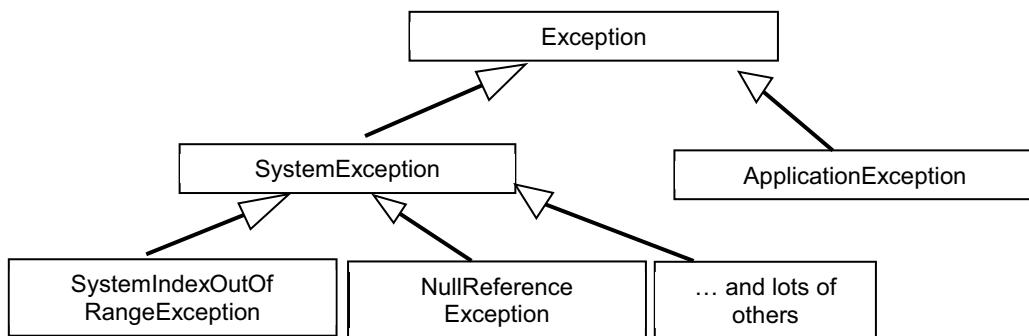
In the example above we could be much cleverer still: the `GetClient()` method could be written in such a way that it generates a new type of exception '`UnknownClient`' exception. We could catch this specific exception and knowing exactly what the problem is, we could define a much better response.... In this case, we could explain that no client exists for the specified ID, we could then ask the user to re-enter their client ID and the program could continue.

## 11.2 KINDS OF EXCEPTION

In C#, all exceptions are derived from the `Exception` class (or one of its subclasses)

Most derive from the following classes:

- `Exception` (the parent of the following two)
- `SystemException` (predefined Common Language Runtime exceptions)
- `ApplicationException` (user-defined application exceptions)



In order to generate meaningful exceptions we need to extend the `Exception` base class built into the .NET framework.

The `Exception` class has already been extended to create a `SystemException` class and an `ApplicationException` class.

The SystemException class is used to generate exceptions within the .NET framework such as NullReferenceException.

When generating exceptions within our application, such as UnknownClientException, we should extend the ApplicationException class

Subclasses of ApplicationException are used to catch and deal with potential problems when running our applications. To do this we must 1) Create appropriate sub classes of ApplicationException class 2) Generate exception objects using a **throw** clause when appropriate and 3) Catch and deal with these exceptions using a **try/catch** block.

### 11.3 EXTENDING THE APPLICATIONEXCEPTION CLASS

When writing our own methods, we should look for potential failure situations (e.g. value that cannot be returned, errors that may occur in calculation etc.). When a potential error occurs, we should generate an ‘ApplicationException’ (i.e. an object of the ApplicationException class). However, it is best to first define a subclass of the ApplicationException (i.e. to create a specialised class) and throw an object of this subtype.

A new exception is just like any new subclass: in this case it is a subclass of the ApplicationException class.

In the case above, an error could occur if no client is found with a specified ID. Therefore, we could create a new exception class called ‘UnknownClientException’.

Exception classes have a ‘Message’ property we can make use of.

There are several overloaded constructors for the ApplicationException class. One of these requires a String and it uses this to initialize a new instance of the ApplicationException class with a specified error message.

Thus we could create a subclass called UnknownClientException and override this constructor. When we create an object of this class we can use this string to give details of the problem that caused the exception to be generated. This string will be stored and later accessed via the Message property.

The code to create this new class is given below:

```
public class UnknownClientException : ApplicationException
{
    public UnknownClientException(String message)
        : base(message)
    {
    }
}
```

In some respects this looks rather odd. Here we are creating a subclass of ApplicationException but our subclass does not contain any new methods – nor does it override any existing methods. Thus its functionality is identical to the superclass. However it is a subtype with a meaningful and descriptive name.

If subclasses of ApplicationException did not exist we would only be able to catch the most general type of exception i.e. an ApplicationException object. Thus, we would only be able to write a catch block that would catch every single type of exception generated by our system.

Having defined a subclass we instead have a choice... a) We could define a catch block to catch objects of the most general type ‘Exception’ i.e. it would catch **all exceptions** or b) We could define a catch block that would catch **all application exceptions** or c) We can be more specific and catch **only UnknownClientExceptions** and ignore other types of exception.

By looking online at the MSDN library we can see that many predefined subclasses of SystemException already exist. There are many of these including:

- ArithmeticException
  - DivideByZeroException
  - OverflowException
- IOException
  - DriveNotFoundException
  - FileLoadException
  - FileNotFoundException
  - PathTooLongException
- InvalidCastException
- InvalidDataException
- InvalidOperationException

Thus, we could:

- Write a catch block that would react to all of these exceptions by catching a SystemException.
- Write a catch block that would catch any type of input\output exceptions, IOExceptions, and ignore all others.
- We could be even more specific and catch only FileNotFoundExceptions.

Catching specific exceptions allows us to take specific remedial action e.g. given a FileNotFoundException we could explain to the user of a program that the file was missing and allow them to specify an alternative file.

Catching general exceptions allows us to ensure no potentially fatal error causes our program to crash though we may not be able to take very useful remedial action e.g. given any Exception we could close our program down but without knowing what caused the exception we could not take specific remedial action.

In exactly the same way we can define sub classes of ApplicationException and catch these exceptions as and when appropriate.

## 11.4 THROWING EXCEPTIONS

Having defined our own exception classes we must then ensure our methods generate, or throw, these exceptions when appropriate. For example, we would instruct a GetClient() method to throw an UnknownClientException when a client cannot be found with the specified ID.

To do this we must create an object of UnknownClientException using the keyword ‘new’. When doing so we can pass an error message as a parameter to the constructor of the exception class as shown in the code below...

```
new UnknownClientException("ClientBook.GetClient(): unknown client ID:  
+ clientID);
```

The code above generates an instance of the UnknownClientException class and provides the constructor with a String message. This message specifies the name of the Class/Method which generated the exception and provides some additional information that can inform the user about the circumstances that caused the error e.g. the clients ID.

Having generated an exception object we use the keyword ‘throw’ to throw this exception at the appropriate point within the body of the method.

```
public Client GetClient(int clientID)
{
    .
    .
    .
    code missing
    .
    .
    .
    throw new UnknownClientException("ClientBook.GetClient(): unknown
client ID:" + clientID);
}
```

In the example above if a client is found the method will return the client object (though this code is not shown). However if a client has not been found the constructor for UnknownClientException is invoked, using ‘new’. This constructor requires a String parameter – and the string we are passing here is an error message that is trying to be informative and helpful.

The message is specifying:

- The class which generated the exception (i.e. ClientBook).
- The method within this class (i.e. GetClient()).
- Some text which explains what caused the exception.
- The value of the parameter for which a client could not be found.

By defining an UnknownClientException we are enabling methods calling this one to catch and deal with potentially serious errors.

## 11.5 CATCHING EXCEPTIONS

Having specified to the compiler that this method may generate an exception we are enabling other programmers to protect against potentially critical errors by placing calls to this method within a try/catch block. The code in the try block will be terminated if an exception is generated and the code in the catch block will be initiated instead.

Thus in the example above the AwardLoan() method can decide what to do if no client with the specified ID is found...

```
try
{
    Client c = listOfClients.GetClient(clientID) ;
    c.determineCreditRating();

    // add code to award or reject a loan application based on this
    // credit rating

}

catch (UnknownClientException uce)
{
    Console.WriteLine("INTERNAL ERROR IN BankManager.AwardLoan()\n"
                      + "Exception details: " + uce.Message);
}
```

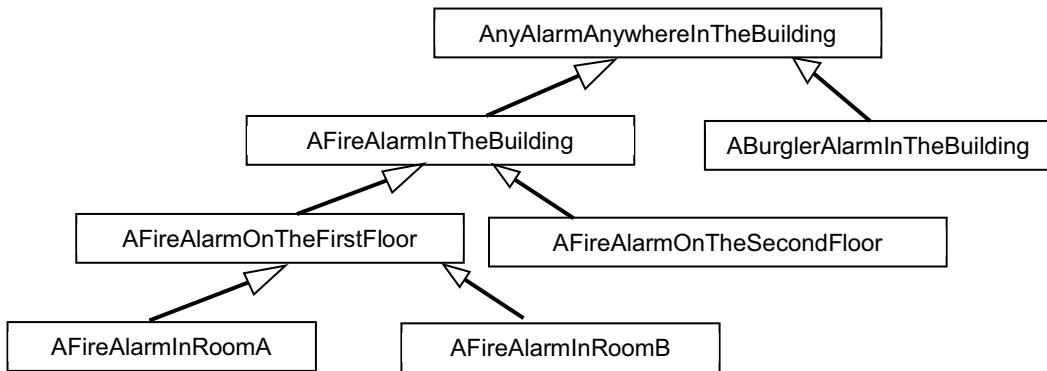
Now, instead of crashing when a client with a specified ID is not found, the UnknownClientException we have deliberately thrown will be handled by the CLR engine which will terminate the code in the **try** block and invoke the code in the **catch** block, which in this case will display a message warning the user about the problem.

## 11.6 EXCEPTION HIERARCHIES

As we have seen there are subclasses of Exception and we can create an entire exception hierarchy.

A block of code could potentially throw multiple exceptions and these can be handled by separate catch blocks. However when an exception is thrown only the first appropriate catch block is executed and therefore care needs to be taken with the order of catch blocks to ensure that general exceptions do not take precedence over the more specialised exceptions.

Consider the following hierarchy of alarms:



The higher we are up the tree the more general the alarm is and the less specific is the action we can take.

- If a fire alarms goes off in a particular room we can evaluate the building and we can call the fire brigade sending them to the exact location.
- If we know that a fire is somewhere on a particular floor, but we don't know the exact location, then we can evaluate the building and we can send the fire brigade to that floor but we can't send them to the exact location.
- If all we know is there a fire somewhere in the building then we can evacuate the building and we can still call the fire brigade but the fire brigade will need to check the entire building.
- At the most general level: if all we know is that some alarm has gone off somewhere, but we don't know what that alarm is and we don't know where, all we can reasonably do is send someone to investigate.

As only the first appropriate catch block is executed when we are dealing with an exception hierarchy we should catch and deal with the most specific exception first.

Effectively we can only instigate one set of responses but this allows us to take the most specific and relevant actions:

- If FireAlarmInRoomA heard...
- If FireAlarmInRoomB heard...
- If FireAlarmOnFirstFloor heard...
- If FireAlarmOnSecondFloor heard...
- If FireAlarmInBuilding heard...

If all else fails than catch the most general exception and close the program neatly – saving error data to a log file is better than crashing.

## 11.7 A 'FINALLY' BLOCK

Finally: a ‘finally’ block can be written to be executed whether an exception has been thrown or not.

This is normally used for clean-up tasks that must occur whether an error was detected and caught or not.

It is necessary because sometimes the normal statements after the catch block don’t execute because:

- An unhandled exception might occur
- The try or catch block might contain a statement which quits the program

See example below:

```
try
{
    // open, read and process a file
}
catch (IOException e)
{
    // output error message
    // exit program
}
finally
{
    // if file is open close it
}
```

## 11.8 SUMMARY

Exceptions provide a mechanism to deal with abnormal situations which occur during program execution.

The exception mechanism will allow other programmers, who use our methods, to recognise and deal with error situations but they can still use ‘if’ statements to handle expected errors.

When writing classes and methods, which may become part of a large application, we should create sub classes of the class ApplicationException and throw exception objects when appropriate.

We can create an entire hierarchy of customised exceptions. When catching these programmers should catch and deal with the specific exceptions first.

When exceptions are generated the code in a catch block will be initiated – this code could take remedial action or terminate the program generating an appropriate error message. In either case at least the program doesn’t just ‘stop’.

A ‘finally’ block can be added to deal with clean-up tasks.

**By making use of the exception mechanism we are protecting against potentially life threatening program failure.**

An example of use of exceptions in a fully working system can be found in the case study (Chapter 14).

# 12 AGILE PROGRAMMING

This chapter will explain the need for agile methodologies and consider different agile practises: SCRUM, Extreme Programming (XP) and Test Driven Development. We will show how the application of Object Oriented design is essential to support an agile approach to software development. Finally, we will show how modern IDE's, such as Visual Studio, offer tools to support agile programming and, in particular, we will examine refactoring and the automatic testing framework within Visual Studio.

## Objectives

By the end of this chapter you will be able to...

- Understand how Agile methods compare with older methods.
- Appreciate the importance of the claims made for Agile development.
- Understand different approaches to Agile development.
- Understand how Object Orientation provides essential support for Agile methods.
- Understand the need for refactoring and how a modern IDE supports this.
- Understand the advantages of automated unit testing and understand how to create automated tests.
- Understand the claims made for Test Driven Development.

This chapter consists of twenty-two sections:

- 1) In the Beginning...
- 2) Old Software Lifecycles
- 3) The Changing World and the Need for Agile Methods
- 4) Agile Approaches
- 5) Scrum
- 6) Extreme Programming (XP)
- 7) Applying Object Orientated Design in Support of Agile Methods
- 8) Refactoring
- 9) Examples of Refactoring
- 10) Support for Refactoring
- 11) Unit Testing
- 12) Automated Unit Testing
- 13) Regression Testing
- 14) Unit Testing in Visual Studio

- 15) Examples of Assertions
- 16) Several Test Examples
- 17) Running Tests
- 18) Test Driven Development (TDD)
- 19) TDD Cycles
- 20) Claims for TDD
- 21) Further Help with Unit Testing
- 22) Summary

## 12.1 IN THE BEGINNING...

In the beginning there was chaos and God said ‘Let there be light’ and from the chaos order started to appear. There wasn’t much to see at this point but at least we could see it.

The world was created, humans evolved and someone decided that computers would be a good idea. These required computer programs but no one knew how to create these and the chaos returned.

Worse still “lots of the people were mean, and most of them were miserable, even the ones with digital watches. Many were increasingly of the opinion that they’d all made a big mistake in coming down from the trees in the first place.” (Hitchhikers Guide to the Galaxy by Douglas Adams).

To bring order to the chaos, ancient and wise beings developed the Waterfall Lifecycle and this made some people happy...but the world continued to evolve.

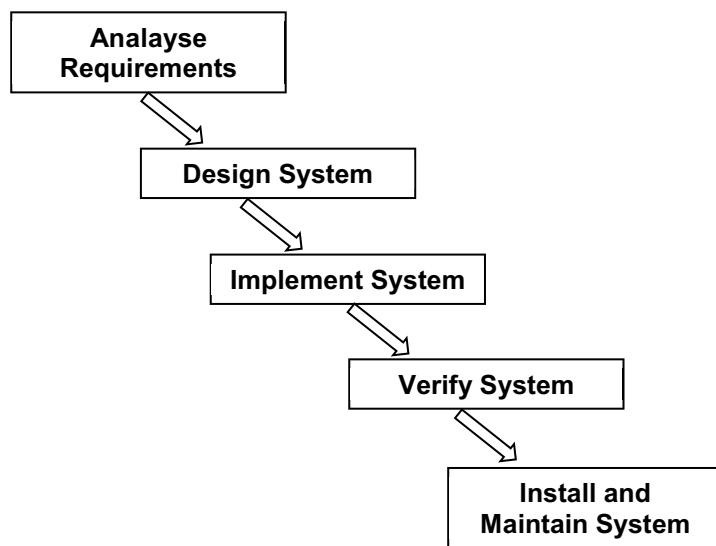
The development of the World Wide Web was a revolution which had massive ramifications for society, the way we communicate, the way we socialise and the way we work. Software Engineers, being people, also changed the way they work and new software lifecycles were needed to reflect these changes.

This chapter explores the role of programmers, the development of new ‘agile’ methodologies and the support Object Oriented can provide to the modern Software Engineer.

## 12.2 OLD SOFTWARE LIFECYCLES

The first well established software lifecycle was the Waterfall model. This was first proposed in the 1950s and while it is sometimes strongly criticised today it should be remembered that it was a really positive attempt to provide structure and control to a chaotic software development process.

The Waterfall Lifecycle, as shown below, suggests a structured approach is needed when developing software: the next task should only start after the last task has finished.



Criticisms of the Waterfall Lifecycle include:

- It only works if a client's requirements are clear and unchanging. If a software engineer misunderstands the requirements, or if a client isn't themselves clear (perhaps because they are not sure what they want from the finished system) then this will lead to a flawed requirements document, leading to a flawed design and a flawed implementation.
- The client does not see samples of software until late in the lifecycle so any problems cannot be rectified early in the process.
- Changes to requirements can be identified very late in the process, i.e. during the verification stage. These changes can require substantial changes to the design and require significant parts of the system to be reprogrammed and implementing these changes can be difficult and expensive.
- Finally, testing is left until the end of the process and if the software needs to be shipped quickly, to meet client deadlines, then the testing could be rushed. If testing and debugging is not done properly, the finished system could be faulty and this could cause serious problems for the users.

The first of these criticisms is perhaps partly unfair as the Waterfall model was adapted to include prototyping as part of requirements gathering. Producing paper based prototypes can be a simple and cheap way of clarifying/confirming requirements and of overcoming some of these limitations.

Still, even if prototyping is used to help gather requirements, the other criticisms are fair. This life cycle

- Does not allow software to be demonstrated to the client until very late in the development process (which limits communication and confidence building).
- Does not allow us to respond to rapidly changing clients' needs.
- Testing is done at the end and can be rushed.

Let's consider the use of this life cycle by looking at an analogy:

### Activity 1

Imagine the scenario below:

A young married couple buy a plot of land and decide to have a new 3 bedroom house built on it. A builder asked them what they want this house to look like and then had a formal set of plans drawn up. The couple look at these plans and they look good on paper. The builder then builds the house starting with the foundations, then the walls, roof and finally the internals (the electrics etc.). They then decorated the house given the colour scheme etc. specified by the couple. Building the house took longer than expected, and a deadline had to be met, so they were not able to check everything properly but they did at least check that the building met all legal safety standards.

The builder hands over the finished house and the first impressions are good but the couple do identify a few small things that needed fixing. However, when the couple start moving in they realise the kitchen is too small for the large family gatherings they hold.

At their request the builder makes the kitchen bigger but to do this they need to knock some walls down, rebuild them, move some of the electrical cables and redecorate...an expensive time consuming process.

The couple are now happy with the house but years later, after having several children, they decide the house is too small and they arrange for an extension to be built onto the house.

Now answer the following questions:

- 1) Did the builder follow a structured lifecycle (requirements analysis, design, development, etc.)?
- 2) Why were the couple not happy with the first version of the house?
- 3) Who was at fault for the problems (the couple or the builder)?
- 4) How could these problems have been avoided?
- 5) As the family grew the house was no longer suitable and an extension needed to be built. Does this indicate a problem with the original design for the house? Should it have been built immediately with more bedrooms?
- 6) Would the house be suitable if the couple's needs rapidly change (perhaps they look after groups of foster children and for some months they need 5 or 6 bedrooms/ perhaps on some occasions they need facilities for looking after children with special needs)?

### Feedback 1

- 1) Did the builder follow a structured lifecycle (requirements analysis, design, development, etc.)?

The builder did follow a structured process that was just like a Waterfall Lifecycle and mostly this worked well. The house plans matched the requirements as they were given and the builders were able to build the house to match the designs though testing was rushed at the end and some faults were not fixed.

- 2) Why were the couple not happy with the first version of the house?

The couple were not happy with the first version of the house because the design was wrong and ultimately this caused an expensive time consuming change.

- 3) Who was at fault for the problems (the couple or the builder)?

The design was wrong because the couple did not communicate the need for a large kitchen but this is not the fault of the couple concerned nor was it the builders fault.

So who was at fault?

No one! Sometimes it's really hard to think of all the project requirements at the start. The couple did consider their basic needs, e.g. 3 bedrooms, but it is not easy to imagine exactly what a house will look like and it is certainly not easy to visualise what living in the house will be like when all you can see is an empty plot of land.

And how could the builder know the couple hosted large family events and needed a larger kitchen?

Given this it is not surprising that the couple later realised the house did not perfectly meet their needs.

4) How could these problems have been avoided?

One way of avoiding these problems would have been to produce an early prototype of the house – before the formal plans were developed. If the couple had a chance to play with a virtual reality house simulator they could have looked inside the kitchen and seen what it would be like to live in. An early prototype would have allowed them to consider all of their needs and to realise that they needed a bigger kitchen – it still would not have guaranteed a perfect set of requirements but it would have helped.

5) As the family grew the house was no longer suitable and an extension needed to be built. Does this indicate a problem with the original design for the house? Should it have been built immediately with more bedrooms?

A client's needs will change over time and this does not indicate a problem with the original house. It would not have been good to build the house with more bedrooms as this would have been very expensive and not everyone needs more bedrooms.

As a client's needs change over time, software will need to be amended accordingly and the Waterfall Lifecycle has a maintenance phase to deal with this.

6) Would the house be suitable if the couple's needs rapidly change (perhaps they look after groups of foster children and for some months they need 5 or 6 bedrooms/ perhaps on some occasions they need facilities for looking after children with special needs)?

The house would not be suitable if the couple's needs rapidly change. Indeed, there is a big problem with this development process if a client's needs rapidly change.

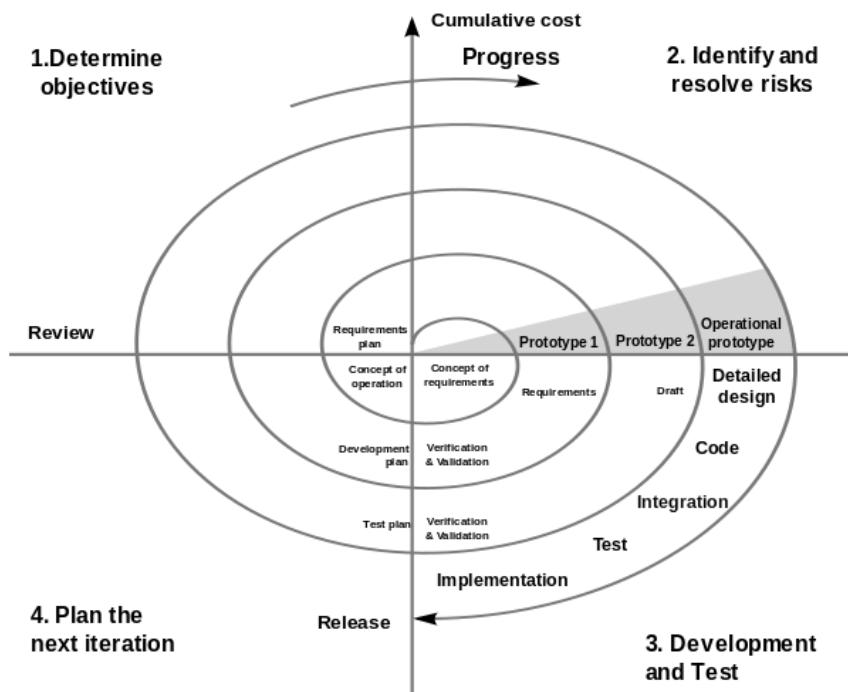
The Waterfall Lifecycle only works if a client's initial requirements are clear and unchanging. While the requirements will change over a long period of time the lifecycle cannot cope with rapid changes.

We will discuss later how dealing with rapid change is becoming more of a necessity and how modern agile methods help with this. We will also consider how Object Oriented design and development tools help.

For now perhaps, we should applaud the positives provided by the structured approach of the Waterfall Lifecycle while noting its use is risky.

- It does depend on gathering a perfect set of requirements at the start of a project and this is far from easy.
- It is a hazardous approach to software development as only at the end, when the product is delivered, will we find out if the client's requirements were correctly obtained.

The development of a spiral lifecycle, shown below, as first proposed by Boehm in the 1980s, was an attempt to overcome the limitations of the waterfall model.



This lifecycle is sometimes simplified and explained as an incremental development approach to software development where small prototypes can be shown to a client and these used to gather further requirements for the design and implementation of the next iteration of the software. The software will evolve until it reaches its goal.

The iterative approach to gathering requirements is, in itself, a big plus. However this simplified explanation ignores one important advantage of the spiral approach.

The spiral approach includes a planning phase where risks can be assessed and the project plans and control mechanisms can be changed as the project develops...thus the spiral approach allows a project to be flexibly managed depending on different phases of the project.

In a way, the spiral approach is a precursor to modern agile methods that build upon these ideas of flexible project management.

## Activity 2

Imagine the scenario below:

Our builders are now asked to build the world's first nuclear power plant. This is a high risk complex project. Additionally, as no one has ever built such a power plant before and there are many unknowns.

Now answer the following questions:

- 1) What advantages would a spiral life cycle provide over a waterfall model for this project?
- 2) Would we be confident of the final timescales and costs involved in this project?
- 3) If we were to build subsequent power plants, would we be able to use the knowledge and experience gained from the first project to develop a clearer set of requirements and designs that could perhaps allow us to build subsequent power plants using a waterfall model?

## Feedback 2

- 1) What advantages would a spiral life cycle provide over a waterfall model for this project?

The spiral approach would provide many advantages.

Firstly, at each stage of the project risks could be assessed and plans put in place to mitigate against these risks.

Secondly, the builders would be able to start by building the core components over several iterations and would refine their understanding as these develop.

Thirdly, as the core components are refined and finished their final shape will become clear it will then become easier the design and build ancillary equipment to work with the core systems.

Finally, the clients will, in iterative stages, be able to see how well the project is developing. This will either give them confidence in the project or, if things are going wrong, it will allow them to intervene and make changes.

- 2) Would we be confident of the final timescales and costs involved in this project?

This is a high risk project with many unknowns and we could not with any confidence predict the project completion date or the final cost. However, the iterative nature of the project would allow the project to be evaluated on a regular basis and in the worst case scenario the project could be cancelled which would at least save some of the money.

3) If we were to build subsequent power plants would we be able to use the knowledge and experience gained from the first project to develop a clearer set of requirements and designs that could perhaps allow us to build subsequent power plants using a waterfall model?

The knowledge gained from the first project would allow a much clearer set of requirements/designs to be produced for future projects. This could allow those project to proceed in a less cyclical manner, possibly using a waterfall approach, and this would allow us to better predict final costs and timescales.

While the spiral approach is designed to cater for high risk projects, where there is considerable level of uncertainty, it is still criticised. Its critics claim:

- It is complex/costly and not suitable for low risk projects.
- This lifecycle makes it difficult to estimate final costs and timescales of a project as it is uncertain how many iterations will be involved.

These criticisms are significant and fair and there is an additional criticism to consider.

The spiral approach, like the waterfall model, is still based on the idea that we can specify an end product.

This is indicated in the scenario above where the knowledge and experience gained from building the first power plant can be used to build subsequent plants. We can only do this because we can predict the need for power years in advance and we can specify what new power plants should generally do i.e. we know that by a specified date, years from now, we need a power plant capable of producing megawatts of power. Additionally, we generally know what this power plant should look like.

**Both the structured and iterative approach to software development provide some benefits but the world has changed and modern agile methods are now needed.**

### 12.3 THE CHANGING WORLD AND THE NEED FOR AGILE METHODS

Both the Waterfall Lifecycle and the Spiral lifecycle were based on the idea of developing a final piece of software that meets a client's requirements.

The old idea of developing a finished software product was reasonable but it does not fit the current world.

- What if there were no ‘end product’?
- What if the software was always needing to be changed and adapted to meet a client’s rapidly changing needs?
- What if software was never ‘finished’ but always in a state of development and continuous refinement?

Before the birth of the WWW, software was developed with the idea of an end point i.e. developing a finished system to meet a client’s needs.

This was a necessity caused by the lack of internet communications. Software was shipped out to clients on disks – floppy discs or CD-ROMS. Sending out updates was a difficult process: discs needed to be sent out to everyone who had bought the product and the updates needed to be manually installed for each client.

Not surprisingly, sincere efforts were made to ship finished software without any bugs though we have seen how testing can be rushed and not always done properly.

Of course new versions of the software were released with additional features but these were only released on a yearly basis, or sometimes longer between releases, and each software release was either a major update or more like a complete new system.

An example of this release cycle were the early versions of Microsoft Word which was first released in 1990 with newer versions in 1991 and 1992.

The same applies to the releases of operating systems: DOS was released in 1981, Windows was released in 1983 (this being a graphical front end to DOS) and Windows 95 was released in 1995.

Each release was not just a software update – it was a complete new product.

### Activity 3

If you are old enough, consider any computer game or other software you bought prior to the internet.

Was its release similar to that of the software described above?

How does that compare with the release of current computer games or other software?

### Feedback 3

Historically, computer games were distributed via disks or cartridges. These would often plug directly into games consoles and were distributed as finished products which were never intended to be updated.

An example of these were the 'Might and Magic' role playing games, versions of which were released in 1988, 1991 and 1992. Each version was really a completely new game.

Now computer games are regularly updated over the internet. Sometimes these updates are simple bug fixes but sometimes these updates add additional features.

Sometimes the updates provide significant additional functionality and these are not given away for free – an example of these includes the 'Sims' games which regularly releases large expansion packs. Sims 4 was released in 2015, this was a completely new game and not an extension to Sims 3, but by the end of 2017 nine expansion packs had been released for Sims 4.

While each expansion pack needs to be purchased separately, these are not new games – each provides significant extra functionality to the base game and can be downloaded over the internet.

There is no end product to the Sims 4 game...it is constantly evolving and being extended.

With the birth of the internet providing software updates became easy and software is no longer conceived and sold as a 'finished product'.

Instead, software evolves and is extended over time to meet the desires of users.

Interestingly, gamers are at times given the opportunity to vote for the most preferred expansion packs – games developers can ensure they are meeting the needs and desires of their users.

The same principles apply to software developed for businesses.

Modern Software Engineers see the need to constantly and quickly adapt software to help the business concerned to perform to its maximum capability.

Thus, instead of selling a 'finished product' and then work on yearly updates, Software Engineers work with a company all of the time – shipping parts of an ever changing product, constantly evaluating the clients current needs and shipping the next product update to meet the client's most important needs.

Software is therefore shipped in small components that are released regularly and the client's systems are therefore constantly being updated to help their business gain maximum benefit.

The development shipping cycle has gone from providing annual, or biennial, updates to monthly or even weekly updates.

This is all made possible by the internet but it does need processes to manage software releases and it does require agile methodologies.

## 12.4 AGILE APPROACHES

Traditional development approaches emphasized detailed advance planning and a linear progression through the software lifecycle *Code late, get it right first time*

'Agile' development approaches emphasize flexible cyclic development with the system constantly evolving to meet a client's ever changing needs with no final solution: *Code early, fix and improve it as you go along.*

Is the Waterfall lifecycle **really** successful in enabling large, complex projects to proceed from start to finish without ever looking back? Advocates of agile approaches contend that these better fit the reality of software development.

Agile development not only helps the client by providing them with the maximum advantage, it helps software engineers maintain a regular working pattern.

The Waterfall approach required the analysis tasks to be followed by the design tasks, then the programming and finally the testing tasks. For large projects this caused peaks and troughs in the work patterns for specialist staff – designers would have a lot of work to do at the start of a project and testers a lot of work to do at the end.

To manage these peaks and troughs when the designers had finished they would typically be moved on to new projects, thus keeping them busy, but they would not then be available if design changes are needed to the current project.

The agile approach allows much shorter development cycles, with monthly rather than annual updates, with designers, programmers and testers working together all of the time. This keeps everyone working at a steady pace, improves communication and improves productivity.

Agile development is becoming normal working practice for Software Engineers but there are still some people using older lifecycles and there are still discussions being held over different agile approaches.

Whatever agile approach is taken agile programming requires the proper application of Object Orientation principles and programming tools that will enable software to change and evolve.

In this chapter we will briefly consider the Scrum approach to agile development and two specific tools provided by modern IDEs that support agile programming: refactoring and testing tools.

## 12.5 SCRUM

Scrum is only one approach to agile development, other popular approaches include Kanban, Extreme Programming (XP) and Test Driven Development (TDD).

The purpose of this section is not to explain Scrum in detail but to give a very brief overview of it and see how this relates to and requires the application of good Object Oriented design principles.

More information on Scrum can be found from the Scrum organisation <https://www.scrum.org/> and the Scrum Alliance <https://www.scrumalliance.org>

Scrum is based on several ideas that are not directly related to the development of software. These ideas include:

- Developing and supporting self-organising teams of people that reflect on their performance and can reorganise and respond to changes.
- Improving the way the teams communicate with daily face-to-face meetings.
- Ensuring regular collaboration with customers in order to understand their changing needs rather than sticking to initial plans and designs.

Specifically regarding software, it encourages:

- Welcoming changing requirements for the customer's competitive advantage.
- Continuous delivery software with shorter timescales.
- Business people and developers working together daily throughout the project.

Scrum projects are managed in sprints, by a sprint team, with each sprint usually lasting for only a few weeks or one month.

The sprint team is an interdisciplinary team made up of programmers, designers, testers and importantly a ‘Product Owner’ – someone who will represent the needs of the client and users. The team is managed by a Scrum Master who will assist the team but who is not a boss and does not control what each team member does.

Each sprint will start by considering a Product Backlog: This is a list of functionality i.e. changes needed by client and prioritised by the client. It is prioritised because the client won’t get everything they want in one sprint...but they will hopefully get one or two of the most important software updates.

After each sprint the product backlog will be updated by the client to reflect their changing needs. The clients’ needs will change because their business will develop – perhaps they will start selling new types of product or perhaps their customers will want additional services. The product backlog will be updated to reflect these changing needs.

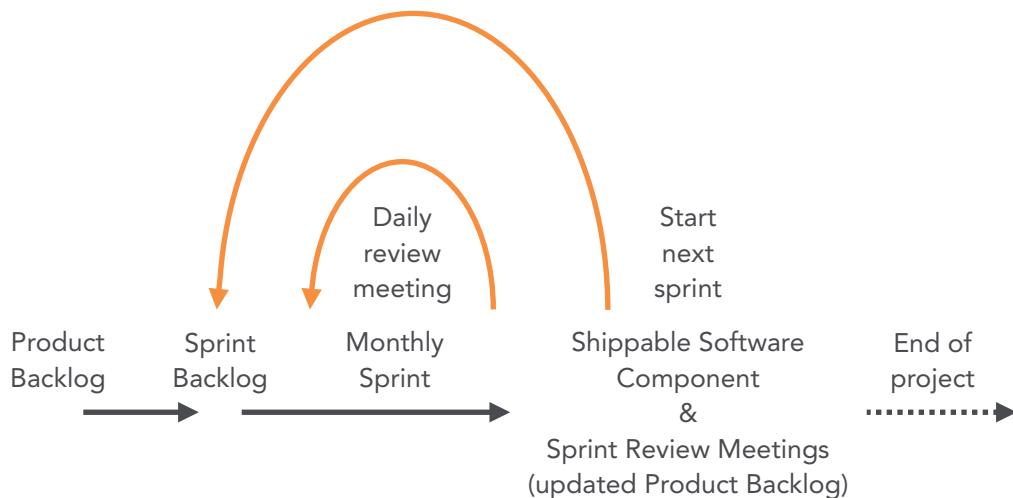
As the sprint progresses the sprint team will create their own prioritised list of work, a Sprint Backlog. They will work on the sprint backlog, designing, coding and testing small components and in doing so create a burn-down chart...a list of the completed work.

Daily meetings will be held with the team, including the product owner, to discuss and evaluate progress.

Each sprint should end in a small product, hopefully a shippable software component, and two reflective meetings:

- A Sprint Review: Involving the client and stakeholders to inspect the product coming out of the sprint, to consider its impact and to consider the changing priorities of the company. The Product Backlog is then refined to reflect the new priorities i.e. what should be done next.
- A Sprint Retrospective: To consider how the team performed, what went well and what didn’t go well, in order to reorganise, improve the team and prepare for the next sprint.

This process can be visualised as follows:



Thus Scrum is an agile process that aims to constantly respond to a client's changing needs and improve software productivity by developing software in regular small components, with designers, programmers and testers working consistently and with self-reflective and improving teams.

## 12.6 EXTREME PROGRAMMING (XP)

Scrum is not the only agile approach to software development and one alternative is Extreme Programming (XP).

Proponents of Extreme Programming suggest taking the best software engineering practices to extreme levels – hence the name Extreme Programming. This could be considered more of a philosophy than a coherent methodology though many would call it a methodology. Either way, many good ideas have come from the proponents of XP.

One idea that comes from the proponents of XP is Paired Programming. Getting another programmer to inspect the code you write, to check it for errors, is undoubtedly a good idea. Taking this idea to its extreme level suggests having another programmer, sitting at your side while you write the code. Having someone constantly checking and immediately inspecting your code must lead to better code with fewer errors in it. Thus, programmers should work in pairs – Paired Programming! To avoid boredom, the pair of programmers swap roles – the checker becoming the programmer and vice versa.

Another good idea taken to extreme levels is testing. Testing software is obviously a good idea and certainly should not be an activity that is left to the end of the software development lifecycle when it can be rushed and therefore not done properly. Many XP proponents therefore propose writing software tests before writing the software – yet another idea taken to extreme levels. This leads to the idea of Test Driven Development (TDD).

Test Driven Development and the related topics of unit testing and automated unit testing are of such importance that they are covered separately in this chapter. For now it is enough to recognise that the idea of TDD came from the proponents of XP.

An idea related to this is that software bugs should not be considered as software bugs but as unwritten tests.

If a bug in the software is found rather than immediately rushing to fix it, first encapsulate this flaw in a software test – then fix it. Capturing this bug in a test means the flaw is not forgotten and, by using the test, we can check future versions of the software to ensure that the fault is never repeated.

Generally, Extreme Programmers share much of their philosophy with the proponents of Scrum:

- Listen to customers and respond very rapidly to their requests for software changes.
- Develop software in an agile way with very short lifecycles.
- Keep software as simple and as well designed as possible, avoiding lots of dependencies in the software, so that it becomes easy to change.

## 12.7 APPLYING OBJECT ORIENTATED DESIGN IN SUPPORT OF AGILE METHODS

Agile methods are good as they allow software to be rapidly developed and changed in order to provide maximum benefit for a client. For this reason, Agile Methods are becoming standard practice among programmers.

**All of this is good but agile methods do require the application of Object Orientated design principles:**

- Firstly code must be designed in a way that it is amenable to change. Object Orientation, Polymorphism and the application of the SOLID design principles are all therefore essential.

- Secondly we must understand the concepts refactoring/the importance of automated unit testing and the associated agile concept of Test Driven Development (both discussed later in this chapter).
- As software is constant evolving automated documentation (see Chapter 10) is an important time saving device but for this to work programmers must add XML comments to their code (and they should also follow other good programming conventions).
- Finally we must employ version control systems to manage our software and help us release regular software updates.

The application of version control systems is beyond the scope of this book. For readers who want to explore this topic, GitHub is one popular version control system that works with Visual Studio and allows people to collaborate on projects, control different versions of the same software and publish software updates (a brief 7 minute GitHub tutorial is available on YouTube <https://www.youtube.com/watch?v=MixBih-LOR4>).

Alternatives to GitHub include:

- Bitbucket (<https://bitbucket.org/>)
- SourceForge (<https://sourceforge.net/>)
- CodePlex (Microsoft's free open source code hosting service (<https://www.codeplex.com/>))
- Cloud Source (provided by Google (<https://cloud.google.com/source-repositories/>))

## 12.8 REFACTORING

A key element of ‘agile’ approaches is ‘refactoring’. This technique accepts that some early design and implementation decisions will turn out to be poor, or at least less than ideal.

Refactoring means changing a system to improve its design and implementation quality without altering its functionality (in traditional development such work was termed ‘preventive maintenance’).

Although the idea of structurally improving existing software is not new, the difference is as follows. In traditional development, it was seen as a remedial action taken when the software design quality had degraded, usually as a result of phases of functional modification and extension. In agile methodologies, refactoring is regarded as a natural healthy part of the development process.

## 12.9 EXAMPLES OF REFACTORING

During the development process, a programmer may realise that a variable within a program has been badly named. However, changing this is not a trivial task.

Changing a local variable will only require changes in one particular method – if a variable with the same name exists in a different method this will not require changing.

Alternatively, changing a public class variable could require changes throughout the system (one reason why the use of public variables are discouraged).

Thus, implementing a seemingly trivial change requires an understanding of the consequences of that change.

Other more complex changes may also be required. These include...

- Moving a method from one class to another
- Splitting out code from one method into a separate method
- Changing the parameter list of a method
- Rearranging the position of class members in an inheritance hierarchy

## 12.10 SUPPORT FOR REFACTORING

Even the simplest refactoring operation, e.g. renaming a class, method, or variable, requires careful analysis to make sure all the necessary changes are made consistently.

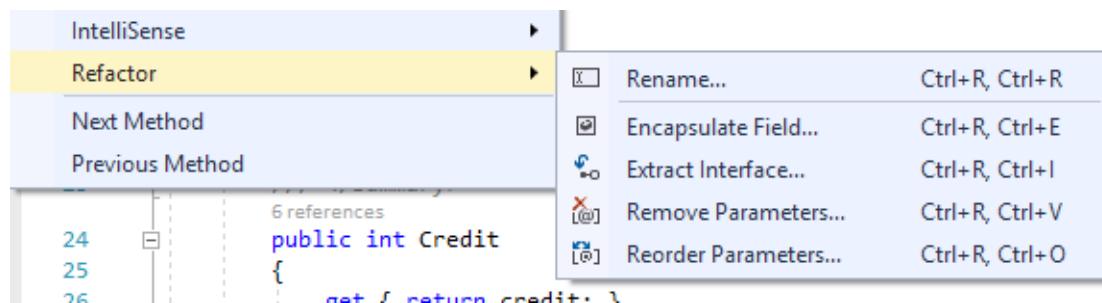
Visual Studio and many other IDEs provide sophisticated automatic support for this activity.

Don't confuse this with a simple text editor find/replace – Visual Studio understands the C# syntax and works intelligently...e.g. if you have local variables with the same name in two different methods and rename one of them, Visual Studio knows that the other is a different variable and does not change it. However, if you rename a public instance variable this may require changes in other classes and even in other namespaces as methods from these classes may access and use this variable.

Changing methods to reorder or remove parameters are also refactoring activities. Doing this will require changes to be made wherever these methods are called throughout the system. Visual Studio will automatically change the method calls as appropriate.

Visual Studio also provides automated support for the creation of an Interface. To do this it extracts an Interface from a class i.e. given a class it can define an interface based on specified features of that class it will then automatically change that class to specify the fact that it implements that new interface. This is a significant restructuring activity that allows us then to create new classes which implement the same interface!

The screen shot below shows some of the refactoring options provided by the Visual Studio IDE.



Another essential facility provided by modern IDEs are automated testing tools.

## 12.11 UNIT TESTING

Unit testing is the idea of testing individual methods of a class in isolation from their eventual context. This idea and the related idea of regression testing, predate agile methods but the automated testing tools that support these are fundamental in allowing TDD to work.

We will therefore consider unit testing, regression testing and automated unit testing before we consider TDD.

This testing is generally ‘wrapped into’ the implementation process:

- Write class
- Write tests
- Run tests, debugging as necessary

A unit test should be independent i.e. it should not require other methods or classes within the system to work.

## 12.12 AUTOMATED UNIT TESTING

To save time, we want to automate unit tests but this will not work if the tests require a human to type in test data or if we need a human to check the programs output. Hence to enable automatic testing we need to set up the test data and we need an automated method for checking the program outputs.

Test cases follow a similar anatomy, no matter which testing framework is used, and this is referred to as the arrange-act-assert cycle. Firstly we ‘arrange’ the test by setting up appropriate test data. Then we ‘act’ i.e. we run the test. Finally we ‘assert’ what the expected output should be. The computer can then compare the actual output against the expected output. If there is a difference then the test indicates that the code is faulty.

Tools and frameworks are available to automate the unit testing process. Using such a tool generally requires a little more effort than running tests once manually and significant benefits arise from the ability to re-run the tests as often as desired just by pushing a button.

This is a great aid to the ‘regression testing’: testing modified code to ensure the modifications don’t break current functionality.

Regression testing was an idea that existed long before agile methods were proposed. However, with agile methods code is modified more often and older code needs to be tested even more often than before – thus automating this process is even more important now.

Additionally, automated unit testing also supports Test Driven Development processes and these play a major role in agile software development methods.

We will explore the Test Driven Development processes within this chapter but before doing so, we will first explore conventional regression testing and the support offered for this via the unit testing framework within Visual Studio.

## 12.13 REGRESSION TESTING

All large software systems need to be adapted to meet changing business needs. Many large systems may have been in use for a decade or more. Over the years the software will need to be updated and improved many times to meet the ever changing needs of the client. For this reason regression testing is required. By running regression tests we want to ensure that changes to the code do not ‘break’ existing functionality. To do this as we write classes we must also write test cases that demonstrate these classes work. Thus we follow a process of...

- Write class
- Write tests
- Run tests, debugging as necessary
- Write more classes
- ...and repeat the process

As we decide its necessary to change some of the earlier classes (or classes which they depend on) due to bugs, changing user requirements, or refactoring, we need to re-run all previous tests to check the new code still passes the older tests. Without regression testing, any modification of existing code is extremely hazardous!

As we regularly need to re-run sets of test cases it is helpful, and hugely timesaving, to have automated testing facilities such as those that exist within Visual Studio.

## 12.14        UNIT TESTING IN VISUAL STUDIO

Visual Studio includes a widely used unit testing framework. This framework requires us to set up the test cases and this takes some time. However, once these have been set up, a thousand test cases can be run at the push of a button and the same set of test cases can be re-run every time the program is amended. This is hugely time saving!

Additionally, Visual Studio 2017 introduced a feature called 'Live Unit Testing'. This executes the unit tests automatically in real time, as you make code changes, to ensure that these changes do not cause parts of the system to break.

The system will also check your code and graphically depict code coverage so that you can see what parts of your system are not covered by any unit tests.

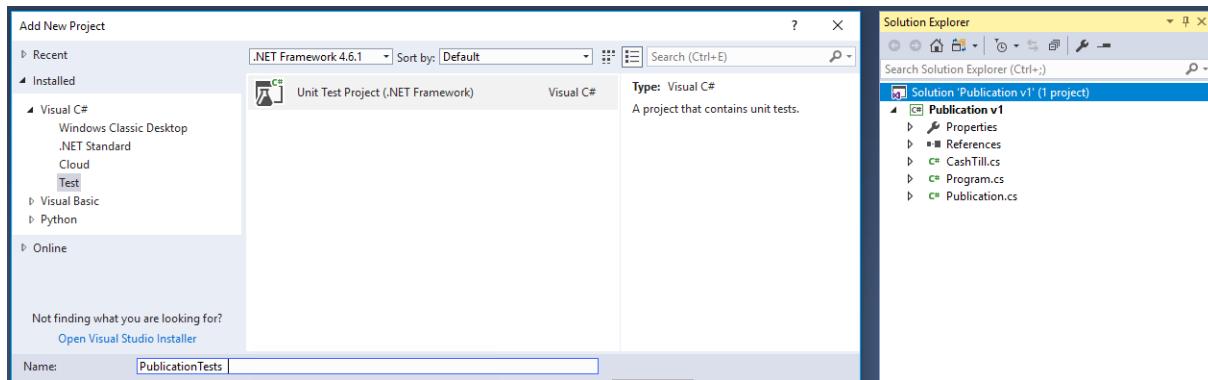
Note: The Community edition of Visual Studio supports unit testing but does not support all of the other tools available e.g. Live Unit Testing.

The system will run the tests and highlight which tests pass and which fail.

Note: Tests that which have 'failed' actually indicate a failure in the program being tested... You could argue that in showing this failure, the test has in fact been successful.

A summary of the process if explained here:

- 1) Firstly, set up a project to store all of the tests for a system. One way of doing this in Visual Studio is to right click on the solution name in the Solution Explorer and choose Add | New Project from the pop-up menu. In the Add New Project dialogue window, choose Visual C#, select and then select TestProject from the sublist giving the test project a suitable name (see figure below).



- 2) Within this create a test fixture for each of the classes we are testing.
- 3) Within each test fixture create multiple tests. Several are probably required for each method being tested.
- 4) Setup the tests i.e. define any initialisation that must be done before the tests are run
- 5) Teardown the tests i.e. define anything that should be done after the tests have been performed.
- 6) Run the tests.

There is a naming convention that makes it easy to relate the tests to the code being tested:

- 1) The project used to store the tests is named after the system being tested (e.g. for example a project storing tests for a bank system could be called BankSystemTests)
- 2) The test fixture is named using the name of the class being tested (e.g. assuming within the bank system there is a class called 'Client' a test fixture would be called TestFixture\_ClientTests).
- 3) The tests are named using the method name being tested \_ the test being applied \_ the expected result (e.g. a method called AddMoney may have a test as follows... AddMoney\_TestAdd5Pounds\_BalanceEquals10).

Having created a test fixture, we need to set up the tests and write the tests themselves. As part of this, we need to specify the correct behaviour of the code being tested. We do this using Assert...() methods which must be true for the test to pass.

We also make use of attributes to provide essential information to Visual Studio so that it can run the tests...

- The [TestClass] attribute indicates that the test class will indeed act as a test fixture. Thus this attribute must be placed at the start of each test fixture.

```
[TestClass]
public class TestFixture_ClientTests
{
    .
    .
    .
}
```

- The [TestInitialize] attribute declares a method that is run before every test case is executed.
- Test cases are methods that must be preceded by the [TestMethod] attribute.
- The [TestCleanup] attribute declares a method that is run after every test case is executed.
- The [ExpectedException(typeof(...))] attribute allows us to specify that we are expecting the code being tested to generate an exception.

We will see examples of each of these shortly.

## 12.15 EXAMPLES OF ASSERTIONS

When setting up test cases we make assertions. An assertion is a statement which should be true if the code has functioned correctly. Example of assertion include...

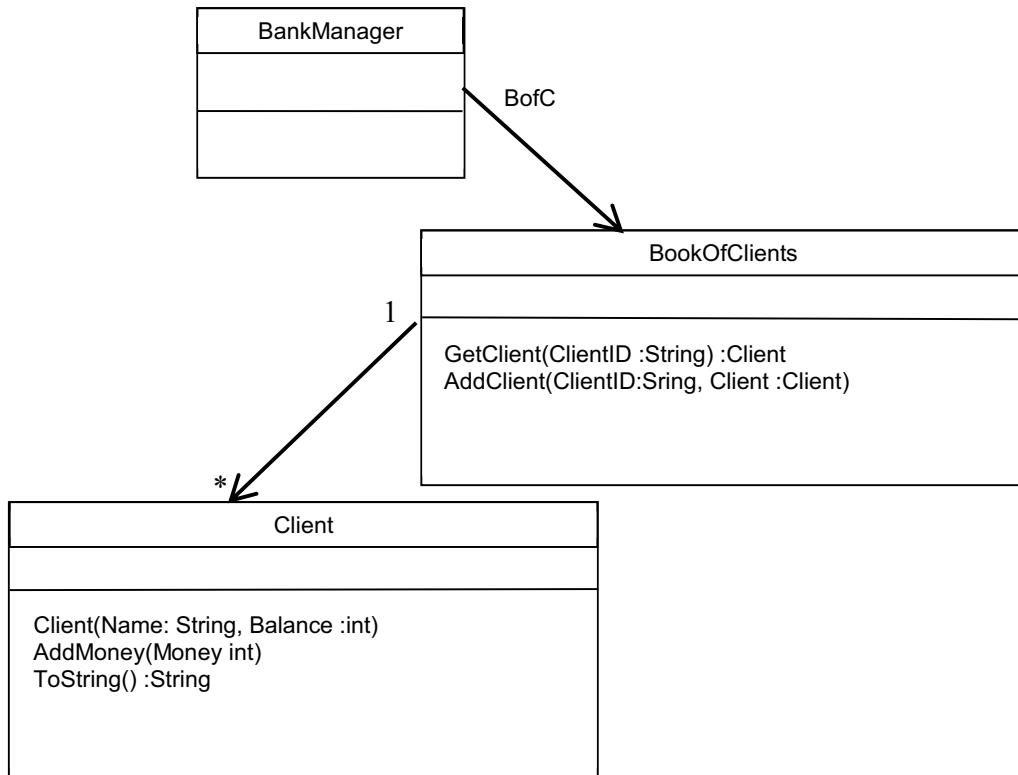
- Assert.AreEqual(...)
- Assert.AreNotEqual(...)
- Assert.Fail()
- Assert.IsTrue()
- Assert.IsFalse(...)

Assert.AreEqual() and Assert.Fail() will be adequate for many testing purposes, as we will see here.

Assert.Fail() indicates that having reached this line means the test has failed! While we say the test has failed this is not strictly true. It is not the test that has failed but our code that has failed. The test successfully found an error in our code.

## 12.16 SEVERAL TEST EXAMPLES

To illustrate the testing framework we will create several test cases to test the functionality of a BankManagement system as represented below:



In this system, a **BankManager** class maintains a book of clients (**BofC**). **Client** objects can be added by the **AddClient()** method which requires an ID for that client and the client object to be added. Clients can be retrieved via the **GetClient()** method which requires a **ClientID** as a parameter and returns a client object (if one exists) or generates an exception (if a client with the specified ID does not exist).

The **Client** class has a constructor that requires the name of the client and the clients balance.

For simplicity sake, we are assuming each client of the bank only has one account (obviously this is not a realistic system) and we do not need an account number as each client will have a unique ID.

We will specifically test the ability to...

- Add a client to the **BookOfClients**
- Trying to lookup a non-existent client
- Increase a Clients balance by invoking the **AddMoney()** method.

This is of course only a small fraction of the test cases which would be needed to thoroughly demonstrate the correct operation of all classes and all methods within the system.

## Testing adding a client

To test a client can be added we need to

1. Set up the test by creating a new empty BookOfClients
2. Create a new client and add this to the BookOfClients
3. Check that the client has been added successfully by trying to retrieve the client just added (this of course should work)
4. We need to test that the client retrieved has the same attributes as the client we just added – to ensure it was not corrupted in the process.

Firstly, we initialise the test by creating a new empty BookOfClients object (BofC) and we clean up the test by setting this to null at the end. The code for this is given below:

```
[TestInitialize]
public void TestInitialize()
{
    BofC = new BankManager.BookOfClients();
}

[TestCleanup]
public void TestCleanup()
{
    BofC = null;
}
```

Next we create our first test method. This method will work by trying to add a client to the empty object BoFC. If the system generates an exception because this client already exists then we know there is a fault in our code...hence under these circumstances we assert that the test has failed. See the test for this below...

```
[TestMethod]
public void AddClient_TestNewClient_ExceptionShouldNotBeGenerated()
{
    BankManager.Client c = new BankManager.Client("Simon", "Room 1234",
"x200", 10);
    try
    {
        BofC.AddClient(1, c);
    }
    catch(BankManager.ClientAlreadyExistsException)
    {
        Assert.Fail("ClientAlreadyExists exception should not be
thrown for new clients");
    }
}
```

If the test above passes, then we know our code has not generated an exception however this does not prove that the client has been added successfully. We need to create other tests to show that we can retrieve the client just added and we need a test to show that in adding/retrieving the client object the attributes have not been corrupted. Multiple tests are required before we can have confidence that the method being tested will actually work!

Note: The name of the test above indicates the name of the method being tested followed by a short description of the test being performed and a description of the expected output. Test names like this, while long, are important. By reading a long list of test names we can work out which parts of our system have been adequately tested and which parts have been missed.

In addition to the tests described above we should also try to add a client with the same ID twice. Our system should of course not allow this to happen. Hence the test succeeds if the code prevents a client being added twice.

Conversely if the system does not generate an exception after adding the same client for the second time then the test should indicate that the code has failed. A test for this is given below...

```
TestMethod]
public void AddClient_TestAddExistingClient()
{
    BankManager.Client c = new BankManager.Client("Simon", "Room
1234", "x200", 10);
    try
    {
        BofC.AddClient(1, c);
        BofC.AddClient(1, c);
        Assert.Fail("ClientAlreadyExists exception should be thrown
if client added twice");
    }
    catch (BankManager.ClientAlreadyExistsException)
    {
    }
}
```

An alternative way of writing the test above is to indicate that the test expects an exception to be generated by using the [ExpectedException(...)] attribute. If this exception is generated and interrupts the test then the test has passed. See an alternative version of the test code below...

```
[TestMethod]
[ExpectedException(typeof(BankManager.ClientAlreadyExistsException))]
public void AddClient_TestAddExistingClient()
{
    BankManager.Client c = new BankManager.Client("Simon", "Room
1234", "x200", 10);
    BofC.AddClient(1, c);
    BofC.AddClient(1, c);
    Assert.Fail("ClientAlreadyExists exception should be thrown if
client added twice");
}
```

## Testing the GetClient method

We must of course test all of the methods in our system including the GetClient() method.

One test we need to perform is to test that an exception is thrown if we try to retrieve a client that does not exist. To test this, we create an new empty BookofClients and try to retrieve a client from this – any client!

In this instance, we would hope our system generates an unknown client exception. Since we have initialised our test by creating an empty client book we should not be able to retrieve any clients.

#### Activity 4

Look at the test code below and decide at which point in this code we could assert the test has shown our system has failed (point A, B, C or D).

```
[TestMethod]
public void GetClient_TestGettingUnknownClient_ShouldGenerateException()
{
    //point A
    try
    {
        // point B
        BofC.GetClient(1);
        // point C
    }
    catch (BankManager.UnknownClientException)
    {
        // point D
    }
}
```

#### Feedback 4

We cannot determine if the GetClient() method has failed before we have invoked it...hence we cannot place an Assert.Fail() at point A or B.

When we do invoke this method, an exception should be generated as our client book is empty. The try block should be terminated at this point and the catch block initiated hence if the code reaches point D the test has succeeded not failed. However, if the code reaches point C it indicates that the expected exception was not generated and hence we can assert that the GetClient() method has failed at this point.

The complete code for this test is given below...

```
[TestMethod]
public void GetClient_TestGettingUnknownClient_ShouldGenerateException
()
{
    try
    {
        BofC.GetClient(1);
        Assert.Fail("UnknownClient exception should be thrown if
client does not exist");
    }
    catch (BankManager.UnknownClientException)
    {
    }
}
```

As well as testing our BookOfClients class we should of course test the other classes in our system.

## Testing the Client Class

As well as testing the BookOfClients class we should test our Client class. The tests for this class should be in a different test fixture and will need to initialise these tests as well...

```
[TestClass]
public class TestFixture_ClientTests
{

    private MessageManagerSystem.Clients.Client c;

    [TestInitialize]
    public void TestInitialize()
    {
        c = new BankManager.Client("Simon", 10);
    }

    [TestCleanup]
    public void TestCleanup()
    {
        c = null;
    }
}
```

In the initialisation we have created a new client with an opening balance of 10.

Strictly speaking, we should have written the Client tests before the BookOfClient tests as we need client objects to test the BookOfClients. In VisualStudio it is possible to ensure these tests are run first.

Below is a simple test to test the AddMoney() method...

```
[TestMethod]
public void AddMoney_TestAdd10ToTheBalance_FinalBalShouldBe20()
{
    c.AddMoney(10);
    Assert.AreEqual(20, c.Money, "Balance after adding 10 is not as
expected. Expected: 20 Actual: "+c.Money);
}
```

The test above uses the `Assert.AreEqual()` method. If the first two parameters are equal, then our code has passed the test. If they are not equal, then our code has failed and the third parameter is the error message to be displayed. To be as helpful as possible the error message specified the balance we expected and the actual balance after the `AddMoney()` method was invoked.

In all of the tests we have written if the test method ends without failing any assertions then the test is passed.

### Testing the `ToString()` method

One way of implicitly testing that **all** the attributes a client have been stored correctly is to test the `ToString()` method returns the value expected. This is a little tricky because the format of the string must match **exactly** including every space, punctuation symbol and newline.

The alternative however is to test the value of every property.

### Activity 5

Assuming the `ToString()` method of the `Client` class is defined as below, create a test method to test the value returned by the `ToString()` method is as expected.

```
public String ToString() {  
    return ("Client name: " + Name + "\nBalance: " + Balance);  
}
```

Hint: We have already initialised client tests by creating a client with a name "Simon" and a balance of 10.

### Feedback 5

One solution to this exercise is given below.

```
[TestMethod]  
public void ToString_TestClientValues()  
{  
  
    Assert.AreEqual ("Client name: Simon\nBalance: 10", c.ToString(),  
    "String returned is not as expected. Expected: Client name:  
    Simon\nBalance: 10 Actual: " + c.ToString());  
  
}
```

Here we assert that the string returned from `c.ToString()` is equal to the string we are expecting. This test is implicitly testing that **all** the attributes have been stored correctly which would be particularly useful if we used it to check a client has been retrieved from the book of clients correctly.

## 12.17 RUNNING TESTS

Having designed a batch of test cases these can be run as often as required at the push of a button.

To do this in Visual Studio we run all of the tests via the Test menu.

## 12.18 TEST DRIVEN DEVELOPMENT (TDD)

While very useful for regression testing, automated unit testing also supports Test Driven Development (TDD) which is a technique mainly associated with ‘agile’ development processes. This has become a hot topic in software engineering

The Test Driven Development approach is to:

1. Write the tests (before writing the methods being tested).
2. Set up automated unit testing, which fails because the classes haven’t yet been written!
3. Write the classes and methods so the tests pass

Note: Writing the tests can actually be done as part of the process of gathering client requirements. Thus user stories can lead to user acceptance tests and these can in turn lead to automated unit tests.

This reversal seems strange at first, but many eminent contributors to software engineering debates believe it is a powerful ‘paradigm shift’.

The task of teaching can be used as an analogy. Which of the following is more focused and leads to better teaching?

- Teach someone everything they should know about a subject and then decide how to test their knowledge.
- Decide specifically what a student should be capable of doing after you have taught them, then decide how to test the student to ensure they are capable of performing this task and finally decide just what you must teach so that they can perform this task and hence, pass the test.

It can be argued that the second approach leads to more focussed teaching. In the same way it is argued by some that Test Driven Development leads to simpler code that focuses just on achieving the functionality required.

## 12.19 TDD CYCLES

When undertaking Test Driven Development the test will initially cause a **compilation** error as the method being tested doesn’t exist!

Creating a stub method enables the test to compile but the test will ‘fail’ because the actual functionality being tested has not been implemented in the method.

We then implement the correct functionality of the method so that the test succeeds.

For a complex method we might have several cycles of: write test, fail, implement functionality, pass, extend test, fail, extend functionality, pass...to build up the solution.

## 12.20 CLAIMS FOR TDD

Among the advantages claimed for TDD are:

- Testing becomes an intrinsic part of development rather than an often hurried afterthought.
- It encourages programmers to write simple code directly addressing the requirements and this helps agile development and rapid code changes.
- A comprehensive suite of unit tests is compiled in parallel with the code development.
- A rapid cycle of “write test, write code, run test”, each for a small developmental increment, increases programmer confidence and productivity.

In conventional software lifecycles, if a software project is running late financial pressures often result in the software being rushed to market not having been fully tested and debugged. With Test Driven Development this is not possible as the tests are written before the system has been implemented.

## 12.21 FURTHER HELP WITH UNIT TESTING

A description of Unit Testing and Live Unit Testing is available via:

<https://docs.microsoft.com/en-gb/visualstudio/test/improve-code-quality>

A walkthrough taking you through the process of creating and running Unit Tests is available via:

<https://docs.microsoft.com/en-gb/visualstudio/test/walkthrough-creating-and-running-unit-tests-for-managed-code>

## 12.22 SUMMARY

'Agile' development approaches emphasize flexible cyclic development with software that constantly evolves to meet a client's changing needs and provide maximum business benefit.

Scrum is one popular approach to agile development that focuses on self-organising and reflective teams working in regular sprints to improve productivity.

The shorter software lifecycles that this encourages has been enabled by the internet and modern communication tools.

Extreme Programmers also propose rapid software development, agile development, paired programming and Test Driven Development.

Agile development does require adaptable software, i.e. code must be designed in a way that it is amenable to change by applying Object Orientation and the SOLID design principles.

Test Driven Development reverses the normal sequence of code and test creation, and plays a major part in 'agile' approaches. Proponents of TDD say this also helps by promoting code that directly meets customer's requirements and is simple and thus, easier to change.

Automated documentation is also important and programmers should therefore add XML comments and follow other good coding conventions.

Refactoring tools help agile development methods.

Unit testing is an important part of software engineering practice whatever style of development process is adopted.

An automated unit testing framework allows unit tests to be regularly repeated as system development progresses.

Finally, version control systems help us to manage our software updates and these are particularly important when following an agile approach to software development.

# 13 CHECKING YOUR UNDERSTANDING

## Introduction

In this book we have:

- Explained the basic concepts of Object Orientation.
- Introduced UML notation and in particular class diagrams.
- Considered the different type of relationship between classes (especially association, dependency, inheritance and implementation).
- Learnt how inheritance works and how methods can be overridden.
- Seen how this leads to polymorphism and how to use polymorphism effectively.
- Shown how analysis of a problem can lead to an Object Oriented design.
- And we have seen how the SOLID design principles can be applied to ensure our designs are good designs that lead to flexible and robust software.

We have also explored practical implementation issues:

- How to implement collections.
- How to serialise/de-serialise entire collections.
- C# development tools.
- How to create and use customised exceptions.

And we have seen how all this applies to modern agile software development approaches.

Throughout the previous chapters, there have been many small activities to help you develop your understanding of the theories explained.

This book is now nearing its end however, before it does, this chapter gives you a chance to check your understanding of these theories and their application.

The use of these theories will also be demonstrated in a large case study in the next chapter.

## Objectives

By the end of this chapter through a series of questions, for which answers are provided, you will be able to test your understanding of UML notation, core Object Oriented theory, the application of the SOLID design principles and practical implementation issues.

This chapter consists of four sections:

- 1) Questions Regarding UML Notation
- 2) Questions Regarding Object Oriented Design
- 3) Questions Regarding Implementation
- 4) Summary

### 13.1 QUESTIONS REGARDING UML NOTATION

UML is a precise notation. Each part of that notation conveys precise instructions to a programmer. Therefore using the correct notation is essential and this section will therefore allow you to check your understanding of the notation.

#### Activity 1

Decide which of the following statements are TRUE.

- 1) In classes we usually make instance variables private but provide public access via properties. We can create private helper methods.
- 2) If we don't place any restriction when accessing via a property we can impose restrictions later without causing other parts of the program to break
- 3) A class can bypass restrictions set in properties by accessing its private members directly.
- 4) - indicates public access, + protected and # private

#### Feedback 1

- 1) True
- 2) True
- 3) True
- 4) False.... + indicates public access, # protected and – private

### Activity 2

Which of the following are NOT valid arrows on a UML class diagram?

- 1) valid
- 2) valid
- 3) valid
- 4) valid
- 5) valid
- 6) All of the above arrows are valid.

### Feedback 2

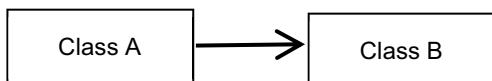
- 1) valid
- 2) valid
- 3) Not valid – do not use this!
- 4) valid
- 5) valid
- 6) Not true – arrow 3 is not valid.

### Activity 3

The diagram below indicates an association relationship between Class A and Class B.

Is the following statement true or false?

Association, as shown, is the weakest relationship indicating that class A in some way uses class B. Typical use: Class A has a method which is passed a parameter object of Class B e.g. BankAccountA.PrintBalance(Printer p);



### Feedback 3

False.

This description applies to a dependency as indicated by

An association is a more permanent relationship that is managed by having a variable of class B inside class A.

### Activity 4

Is the following true or false?

 indicates inheritance

 indicates the implementation of an interface

### Feedback 4

This is true.

All arrows on an UML diagram have a precise meaning and we need to use them correctly.

### Activity 5

In an interface how many methods are actually implemented?

- 1) None
- 2) Very few as we want to keep our interfaces thin.
- 3) As many as required

### Feedback 5

None! In an interface methods are defined but no methods are actually implemented – the methods must be implemented in every class that ‘implements’ the interface.

## 15.2 QUESTIONS REGARDING OBJECT ORIENTED DESIGN

Designing an Object Oriented system well is not easy for novices and some of the design problems below are challenging. They are deliberately thought-provoking to test your understanding of polymorphism and the application of the SOLID design principles.

If you can answer these questions then *Well done!*

However, if all of this theory is new to you and you struggle then **don’t worry**. Instead, consider re-reading Chapter 5 and Chapter 8 as these will refresh your memory and will hopefully help you.

### Activity 6

Which of the following statements are true?

- 1) We should keep our classes simple by focussing on one aspect of functionality only.
- 2) When overriding methods we should not change what they do (only how they do it).
- 3) We should use interfaces to break dependencies in the system.
- 4) We should keep system functionality outside of the GUI.

### Feedback 6

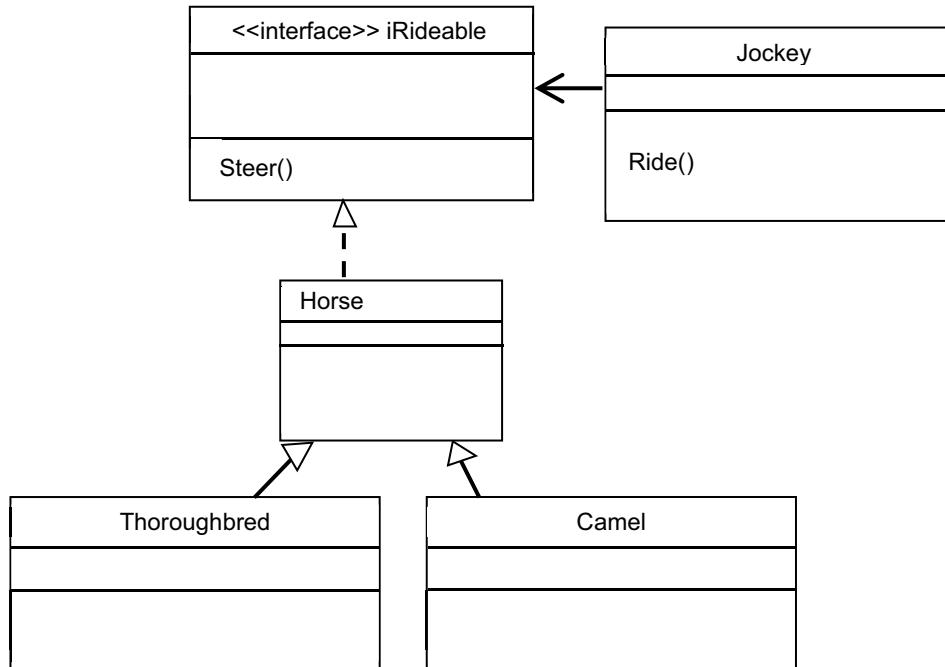
- 1) True – we should keep our classes simple by focussing on one aspect of functionality only. By applying the Single Responsibility Principle to our design we improve the cohesiveness of classes and reduce coupling. Improving cohesiveness makes classes simpler to understand and simpler to change. Reducing coupling decreases the complications caused by making changes to one component of the system – this is especially important if following an agile approach to software development.
- 2) True – when overriding methods we should not change what they do (only how they do it). This is the Open and Closed Principle and the Liskov Substitution Principle in action. This ensures that subtype objects can be substituted where super type objects are expected. Without this polymorphism would not work.
- 3) True – we should use interfaces to break dependencies in the system. This is the application of the Dependency Inversion Principle, or in other words separating concerns of the system. Doing this makes our programs easier to change to meet clients/users changing needs.
- 4) True – we should keep system functionality outside of the GUI. This is the Single Responsibility Principle at work again.

### Activity 7

The design below is for a jockey who can ride anything that is rideable – currently only horses, though other rideable animals can be added later.

There are, however, two very significant errors in the design. Identify these two significant errors and then draw an amended design accordingly.

Note: Details in the diagram that are not relevant have been suppressed.



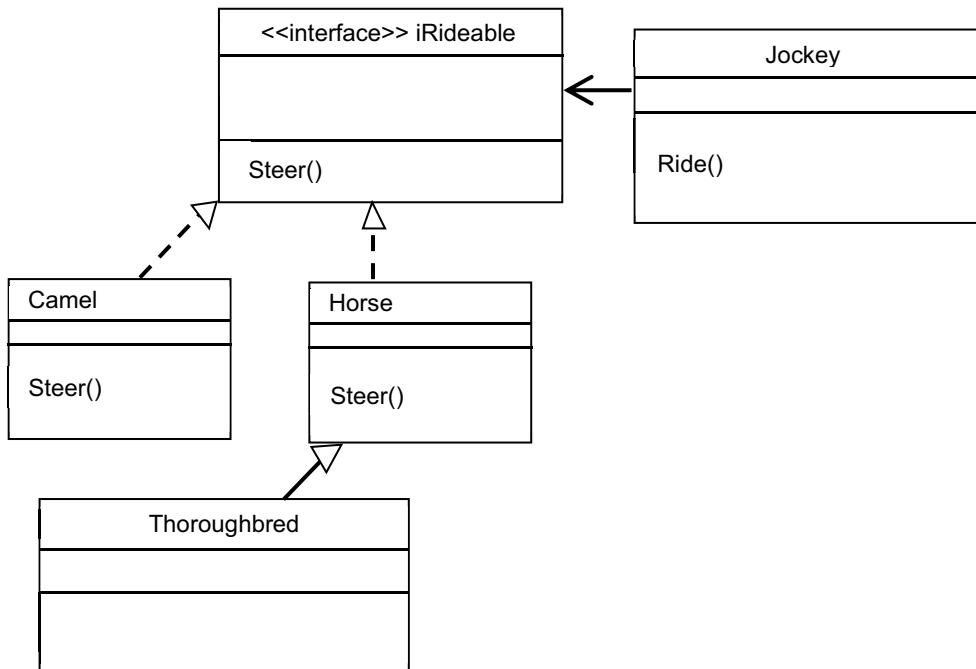
### Feedback 7

The first error is an error in the inheritance hierarchy. A camel is not a horse therefore it cannot inherit the attributes of horse. If the system is to be extended so that the jockey can ride horses and camels then the Camel class should not be in this inheritance hierarchy but should instead implement the methods defined by the interface.

Secondly, the method `Steer()` which is defined in the interface `IRideable` is not implemented in the `Horse` class and it must be. If this is not implemented in the horse class then a Horse is not rideable.

Note: `Steer()` is also not implemented in the `Thoroughbred` class but this is OK because `Thoroughbred` inherits the methods from the `Horse` class so `Thoroughbred` does have a `Steer()` method.

A revised design fixing these two issues is shown below.



### Activity 8

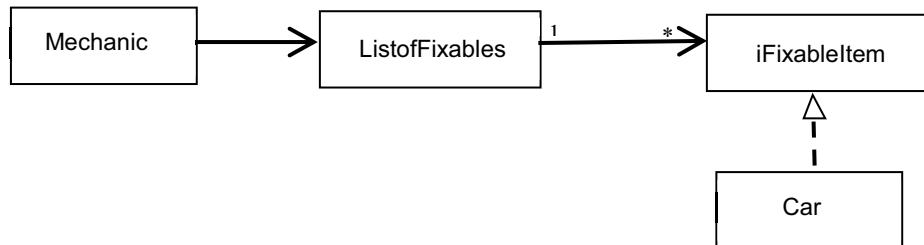
Decide which of the following two options are best...

Assuming we are creating a system for a mechanic who fixes cars should we maintain a list of cars (as shown below)



### Activity 8 Continued

or should we add an interface (see option 2 below) and maintain a list of any fixable object?



### Feedback 8

The Dependency Inversion Principles states you should insert interfaces to separate dependencies in the system thus making the system easy to extend: Design 2 is therefore better.

### Activity 9

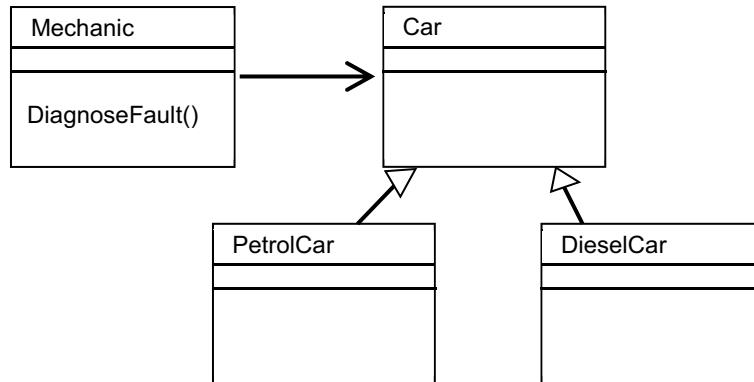
The system below has a significant design flaw – this is not a syntax error, nor is it an error in the inheritance hierarchy.

Can you find it? Can you fix it?

Two hints to help you...

Who does the 'fault' belong to – the Mechanic or the Car?

What will we need to do if we add new classes of Car e.g. ElectricCar or HybridCar?

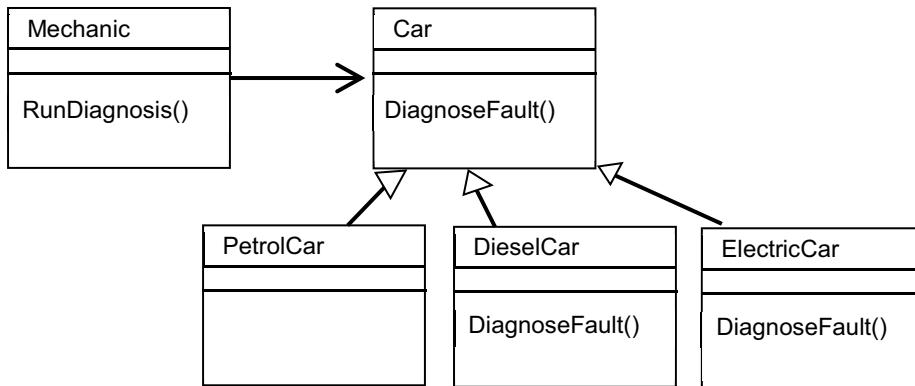


### Feedback 9

The 'fault' belongs to the Car so DiagnoseFault() is not a responsibility of the Mechanic class.

If we add new classes of Car we would, according to the design above, need to reprogram the Mechanic class so it can diagnose the new types of Car...this makes our system very difficult to extend.

The answer is to put DiagnoseFault() where it belongs...see the revised design below.



### **Feedback 9 Continued**

It's the Car's responsibility to diagnose itself.... The Mechanic just tasks it to run its diagnostic method.

The `DiagnoseFault()` method must be defined in the `Car` superclass to guarantee that all `Cars` have it.

Now when a new class of car is added no changes are needed to the `Mechanic` class or other parts of the system. We just override the method so that the new class knows how to diagnose its own faults.

And if `DiagnoseFault()` is not overridden, for example in the `PetrolCar` class above, then it inherits the method from the `Car` superclass and the `Mechanic` class can still invoke the inherited `DiagnoseFault()` method.

The system is now polymorphic and very easy to extend.

...better still why not add an interface and keep a polymorphic collection of anything fixable i.e. anything that implements the `DiagnoseFault()` method. Then we could add other fixable objects e.g. trains, busses etc.

**Object Orientation makes systems very easy to extend  
but only if we design them correctly!**

### **13.3 QUESTIONS REGARDING IMPLEMENTATION**

Having considered syntax issues and design issues, this section will allow you to check your understanding of how to implement an Object Oriented system using C#.

### Activity 10

Decide which of the following statements are TRUE.

- 1) You cannot create an object if a class is Abstract.
- 2) You can assign a subclass object to a super type variable.
- 3) You can assign a super type object to a sub type variable.
- 4) Constructors initialise objects and are usually public.
- 5) All of the statements above are true.

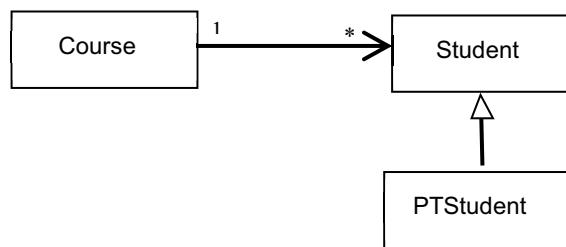
### Feedback 10

- 1) True
- 2) True. All electric cars are cars so you can assign an electric car object to a variable of type car. You can then invoke 'car' methods e.g. Accelerate() on the electric car without worrying what type of car it is. The car itself will decide how to accelerate knowing how itself works. This is why polymorphism works.
- 3) False. Not all cars are electric cars so we cannot assign a car object to an electric car variable.
- 4) True
- 5) False...statement 3 is not true.

### Activity 11

Consider a 'Course' which maintains a collection of Students who take that course...some of whom are part time students (as indicated in the diagram below).

Should this collection of Students be implemented using an array, a list, or a set?



### Feedback 11

We should implement this using a set as we don't want duplicates i.e. no student should be listed on the course twice.

### Activity 12

Which of the following often requires us to override Object.Equals() and Object.GetHashCode().

- 1) List
- 2) Set
- 3) Dictionary
- 4) Both Set and Dictionary
- 5) None of the above

### Feedback 12

We need to override Object.Equals() and Object.GetHashCode() when we are using a set or a dictionary – so duplicates are identified and prevented.

### Activity 13

Decide which of the following statements are TRUE.

- 1) We can, and should, serialise an entire collection in one step.
- 2) When we de-serialise a collection an entire object hierarchy can be returned and all the connections will be remade automatically.
- 3) When serialising objects, to be safe, we should serialise them one at a time.

### Feedback 13

- 1) True.
- 2) True. We can serialise an entire object hierarchy in one step and when we de-serialise it the entire hierarchy is recreated.
- 3) False.

### Activity 14

Decide which of the following statements are TRUE.

- 1) We should develop our application to make use of customised exceptions.
- 2) We should still use traditional error-handling methods, 'if' statements, for errors which can be predicted.
- 3) 'if' statements are not very efficient for handling infrequent error conditions – the exceptions to the rule!
- 4) When catching multiple exceptions we should catch and deal with the most specific exceptions first.
- 5) All of the above are true.

### Feedback 14

All of these statements are true.

### Activity 15

Decide which of the following statements are TRUE.

- 1) Polymorphism allows a system to be easily extended as we can add subclass objects and treat them as superclass objects.
- 2) Polymorphism allows us to ensure our methods are free from procedural errors.
- 3) Polymorphism only works if subclasses are truly substitutable for super classes (i.e. 'is-a' applies).
- 4) We can use interfaces to extend polymorphism beyond a true 'is-a' inheritance hierarchy.
- 5) All of the above are true.

### Feedback 15

Polymorphism is powerful and (1), (3) and (4) are all true. However, it would be nonsense to suggest that Polymorphism allows us to ensure our methods are free from procedural errors. Therefore, testing programs properly is essential and automated unit testing tools are very important.

### Activity 16

Identify the one false statement below:

- 1) Agile' development approaches emphasize rapid flexible cyclic development with software that constantly evolves to meet a client's ever changing needs with no final solution envisaged.
- 2) One of the ideas behind Scrum is to develop and support self-organising teams of people that reflect on their performance and can reorganise and respond to changes.
- 3) Scrum sprints hopefully end with a shippable software component and two meetings: a sprint review and a sprint retrospective.
- 4) XP proponents propose taking good software engineering practices to extreme levels and this includes Paired Programming and Test Driven Development.
- 5) Agile methods require the application of good Object Oriented design, i.e. the application of Polymorphism and the SOLID design principles, to promote code that is robust and easy to change.
- 6) Automated unit testing and automatic documentation are important but not when using agile approaches to software development.

### Feedback 16

- 1) True
- 2) True
- 3) True
- 4) True
- 5) True
- 6) False – Automated unit testing and automatic documentation are important whatever software development lifecycle is used. They are even more important when using agile approaches to software development.

## 13.4 SUMMARY

The purpose of this chapter was to allow you to test your understanding of the Object orientation theory...syntax, design and implementation. This theory will be demonstrated in the next chapter by looking at a case study taking a set of requirements, designing a solution and implementing that solution.

# 14 CASE STUDY

This chapter will bring together all of the previous chapters showing how these essential concepts work in practise through one example case study.

## Objectives

By the end of this chapter you will see...

- how a problem description is turned into a UML model (as described in Chapter 7)
- several example of UML diagrams (as described in Chapter 2).
- an example of the use of inheritance and method overriding (see Chapter 3).
- an example of polymorphism and see how this enables programs to be extended simply (see Chapter 4).
- an example of how generic collections can be effectively used, in particular you will see an example of the use of a set and a dictionary (see Chapter 9).
- an example of these collections can be stored in files very simply using the process of serialization (see Chapter 9).
- examples of exceptions and exception handling (see Chapter 11).
- examples of automated unit testing (see Chapter 12).
- finally you will see the use of the automatic documentation tool (see Chapter 10).

The complete working application, developed as described throughout this chapter, is available to download for free as a compressed file. This file can be unzipped and the project loaded into Visual Studio 2017.

The community edition of Visual Studio is available for free download from <https://www.visualstudio.com/vs/community/>.

The automatic documentation created to describe the system is available as a set of web pages and can therefore be viewed using any web browser.

This chapter consists of sixteen sections:

- 1) The Problem
- 2) Preliminary Analysis
- 3) Further Analysis
- 4) Documenting the Design using UML

- 5) Prototyping the Interface
- 6) Revising the Design to Accommodate Changing Requirements
- 7) Packaging the Classes
- 8) Programming the Message Classes
- 9) Programming the Client Classes
- 10) Creating and Handling UnknownClientException
- 11) Programming the Interface
- 12) Using Test Driven Development and Extending the System
- 13) Generating the Documentation
- 14) The Finished System
- 15) Running the System
- 16) Conclusions

## 14.1 THE PROBLEM

User requirements analysis is a topic of significant importance to the software engineering community and totally outside the scope of this text. The purpose of this chapter is not to show how requirements are obtained but to show how a problem statement is modelled using Object Oriented principles and turned into a complete working system once requirements are gathered.

The problem for which we will design a solution is ‘To develop a message management system for a scrolling display board belonging to a small seaside retailer.’

For the purpose of this exercise we will assume preliminary requirements analysis has been performed by interviewing the shop owner, and the workers who would use the system, and from this the following textual description has been generated:

Rory's Readables is a small shop on the seafront selling a range of convenience goods, especially books and magazines aimed at both the local and tourist trades. It has recently also become a ticket agency for various local entertainment and transport providers.

Rory plans to acquire an LCD message display board mounted above the shopfront which can show scrolling text messages. Rory intends to use this to advertise his own products and offers and also to provide a message display service for fee-paying clients (e.g. private sales, lost and found, staff required etc.)

Each client is given a unique ID number and has a name, an address, a phone number and an amount of credit in 'credit units'. A book of clients is maintained to which clients can be added and in which we can look up a client by their ID.

Each message is for a specific client and comprises the text to be displayed and the number of days for which it should be displayed. The cost of each message is 1 unit per day. No duplicate messages (i.e. the same text for the same client) are permitted.

A set of current messages is to be maintained: new messages can be added, the message set can be displayed on the display board, and at the end of each day a purge is performed – each message has its days remaining decremented and its client's credit reduced by the cost of the message, and any messages which have expired or whose client has no more credit are deleted from the message set.

The software is to be written before the display board is installed – therefore the connection to the board should be via a well-defined interface and a dummy display board implemented in software for testing purposes.

This chapter describes how this problem may be analysed, modelled and a solution programmed – thus demonstrating the techniques discussed throughout this book.

## 14.2 PRELIMINARY ANALYSIS

To perform a preliminary analysis of these requirement as (described in Chapter 7) we must...

- List the nouns and verbs
- Identify things outside the scope of the system
- Identify the synonyms
- Identify the potential classes, attributes and methods
- Identify any common characteristics

From reading this description we can see that the first paragraph is purely contextual and does not describe anything specifically related to the software required. This has therefore been ignored.

## List of Nouns

From the remaining paragraphs we can list the following nouns or noun phrases:

- LCD message display board
- shopfront
- scrolling text message
- client
- ID number
- name
- address
- phone number
- credit
- credit unit
- message
- day
- book of clients
- ID
- text
- number of days
- cost of message (units)
- set of current messages
- message set
- display board
- days remaining
- client's credit
- cost of message
- software
- connection
- interface
- dummy display board

## List of Verbs

and the following verbs:

- acquire
- mount
- show
- advertise
- give (a unique ID)
- display
- add (a client)
- look up
- permit (duplicates – NOT!)
- purge
- decrement
- reduce credit
- expire
- delete
- write (the software)
- install
- implement
- test

## Outside Scope of System

By identifying things outside the scope of the system we simplify the problem...

- Nouns:
  - shopfront
  - software

- Verbs:
  - acquire, mount (the display board)
  - advertise
  - give (a unique ID)
  - write, install, implement, test (the software)

The shopfront is not part of the system and it is not a part of the system to acquire and mount the displayboard, the ID is assigned by the shop owner – not the system, and writing/installing the software is the job of the programmer – it is not part of the system itself.

## Synonyms

The following are synonyms:

- Nouns:
  - LCD message display board = display board
  - scrolling text message = message
  - ID number = ID
  - credit units = client's credit = credit
  - set of current messages = message set
  - days = number of days = days remaining
- Verbs:
  - show = display

By identifying synonyms we avoid needless duplication and confusion in the system.

## Potential Classes, Attributes and Methods

Nouns that describe significant entities for which we can identify properties i.e. data and behaviour i.e. methods could become classes within the system. These include:

- Client
- Message
- ClientBook
- MessageSet
- DisplayBoard
- DummyDisplayBoard

Nouns that would be better as attributes of a class rather than becoming a class themselves:

- For a ‘client’:
  - ID
  - name
  - address
  - phone number
  - credit
- For a ‘message’:
  - text
  - days remaining
  - cost of message

Each of these *could* be modelled as a class (which Client or Message would have as an object attribute), but we decide that each of them is a sufficiently simple piece of information that there is no reason to do so – each one can be a simple attribute (instance variable) of a primitive type (e.g. int) or library class (e.g. String).

This is a **design judgement** – introducing classes for significant entities (Client, Message etc.) which have a set of information and behaviour belonging to them, but not overloading the design with trivially simple classes (e.g. credit which would just contain an ‘int’ instance variable together with a property or accessor method!).

Verbs describe candidate methods and we should be able to identify the classes these could belong to. For instance:

- For a ‘client’:
  - decrease credit
- For a ‘message’:
  - decrement days

The other verbs describing potential methods should also be listed:

- display
- add (client to book)
- add (message to set)
- lookUp (client in book)

- purge
- decrement
- expire
- delete

For each of these the associated class should be identified.

### Common Characteristics

The final step in our preliminary analysis is to identify the common characteristics of classes and to identify if these classes should these be linked in an inheritance hierarchy or linked by an interface.

Looking at the list of candidate classes provided we can see that two classes that share common characteristics:

- DisplayBoard
- DummyDisplayBoard

This either implies these classes should be linked within the system within an inheritance hierarchy or via ‘an interface’ (see section 4.5 Interfaces). In this case the clue is within the description “These will have a ‘connection’ to the rest of the system via a ‘well-defined interface’”.

Ultimately our system should display messages in a real display board however it should first be tested on a dummy display board. For this to work the dummy board must implement the same methods as a real display board.

Thus we should define a C# ‘interface’. No common code would exist between the two classes – hence why we are not putting these within an inheritance hierarchy. However the dummy board and the real display board should both implement the methods defined via a common interface. When our system is working we could replace the dummy board with the real board which implements the same methods. As the connection with the dummy board is via the interface changing the dummy board with the real display board should have no impact on our system.

From our preliminary analysis of the description we have identified candidate classes, interfaces, methods and attributes. The methods and attributes can be associated with classes.

The classes are:

- Client
- Message
- ClientBook
- MessageSet
- DisplayBoard
- DummyDisplayBoard

The Interface is:

- DisplayBoardControl (a name we have made up)

And the methods include:

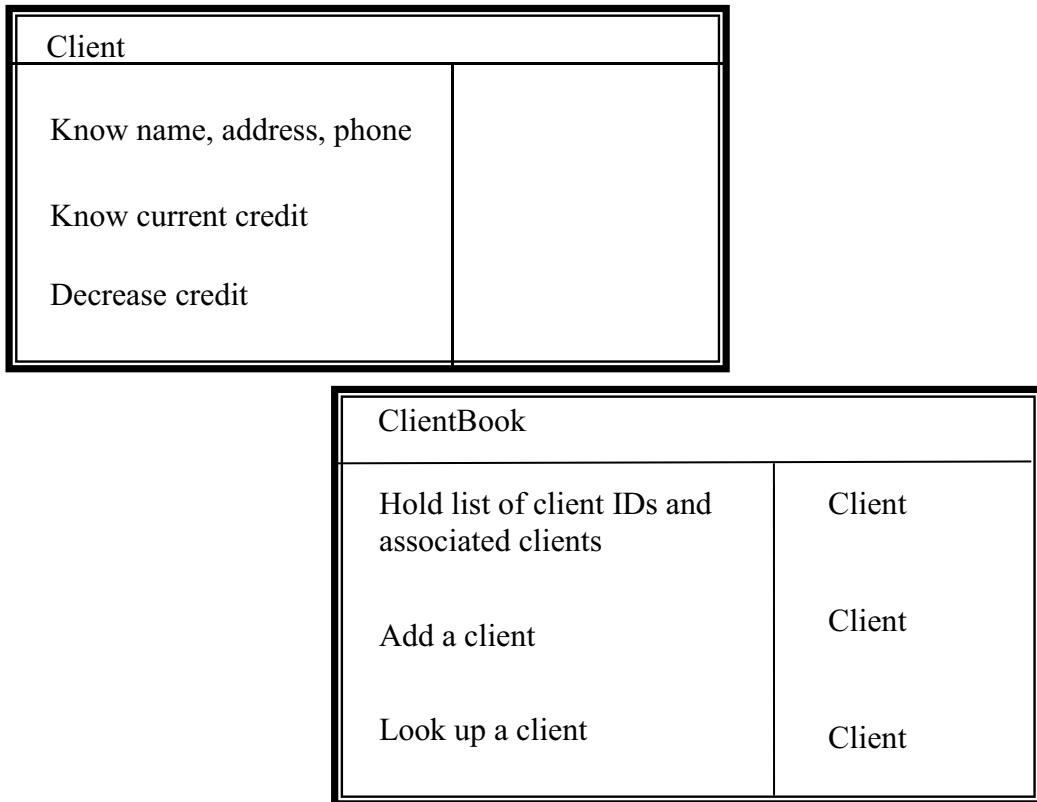
- display
- add (client to book)
- add (message to set)
- lookUp (client in book)
- purge
- decrement
- expire
- delete

### 14.3 FURTHER ANALYSIS

We could now document this proposed design using UML diagrams and program a system accordingly. However before doing so it would be better to find any potential faults in our designs as fixing these faults now would be quicker than fixing the faults after time has been spent programming the system. Thus we should now refine our design using CRC cards and elaborate our classes.

CRC cards (see Chapter 7 section 7.10 and 7.11) allow us to role play various scenarios to check if our designs look feasible before refining these designs and documenting them using UML class diagrams.

The two CRC cards below have been developed to describe the Client and ClientBook classes. The panel on the left shows the class responsibilities and the panel on the right shows the classes they are expected to collaborate with.



We can now use these to roleplay, or test out, a scenario. In this case what happens when we get a new client in the shop? Can this client be created and added to the client book?

To do this the system must perform the following actions:

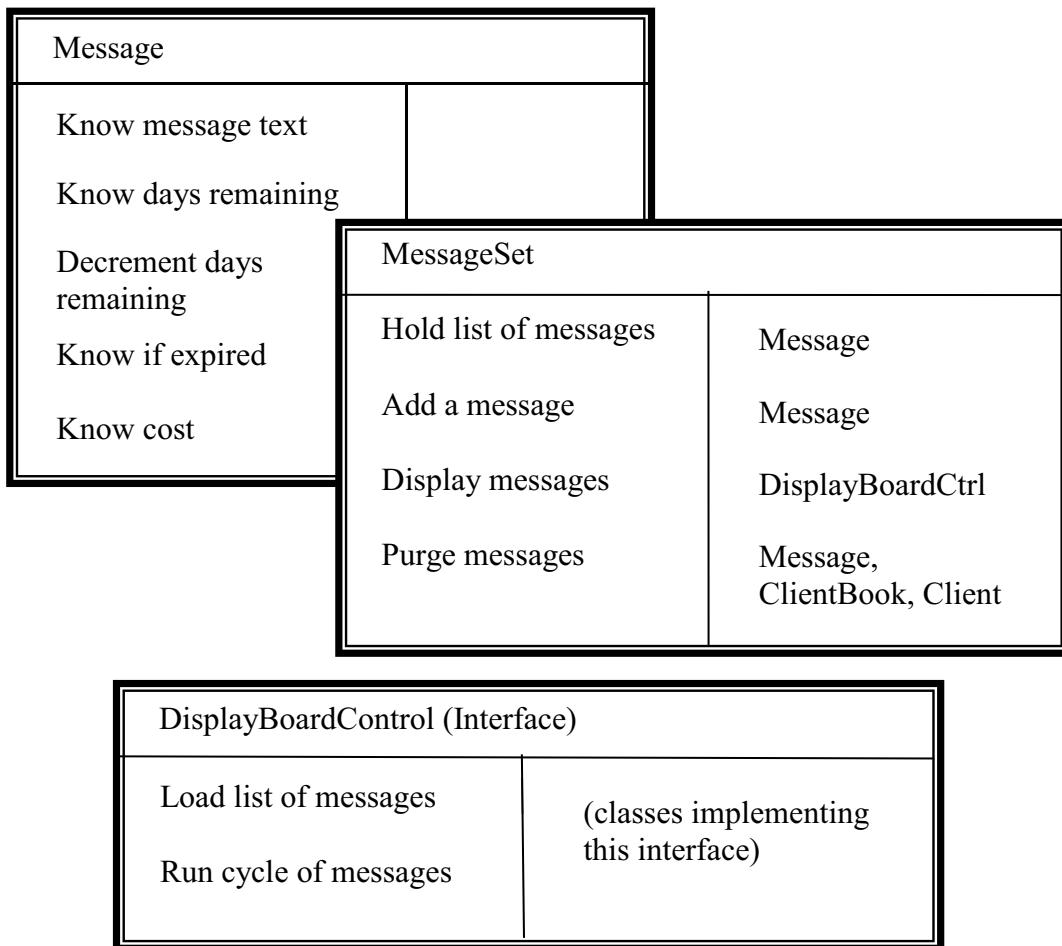
- create a new Client object
- pass it (along with the unique ID to associate with it) to the ClientBook object
- add the client to the client book.

By looking at the CRC cards we can see that:

- the constructor for Client will be able to create a new client object
- The ClientBook has the capability to add a client and
- the ClientBook can hold the IDs associated with each client.

It would therefore appear that this part of the system will work at least in this respect – of course we need to create CRC cards to describe every class and to test the system with a range of scenarios.

Below are three CRC cards to describe the Message and MessageSet classes and the DisplayBoardControl interface.



What we want to ‘test’ here is that messages can be created, added to the MessageSet and displayed on the display.

A point of requirements definition occurs here. There are two possibilities regarding the interface to the display board:

- a) we load one message at a time, display it, then load the next message, and so on.
- b) we load a collection of messages in one go, then tell the board to display them in sequence which it does autonomously.

The correct choice depends on finding out how the real display board actually works.

Note: (a) would mean a simple display board and more complexity for the “Display messages” responsibility of MessageSet, while (b) implies the converse.

For this exercise we will assume the answer to this is (b), hence the responsibilities of scrolling through a set of messages will be assigned to the DisplayBoardControl interface.

Looking at these CRC cards it would appear that we can

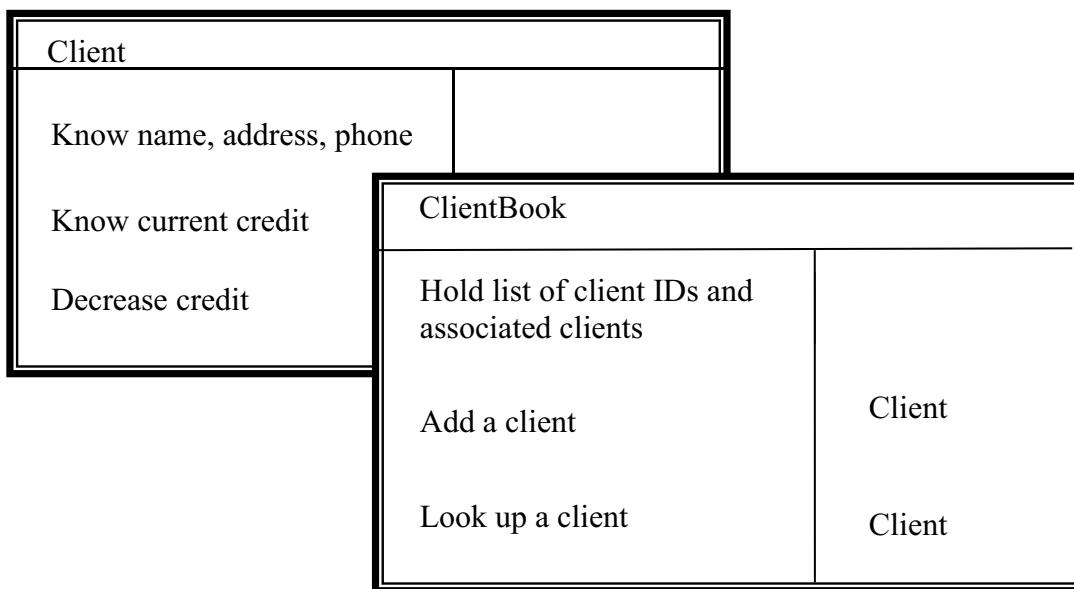
- Create a new message,
- Add this to the message set and
- Display these messages by invoking load ‘list of messages’ and ‘run cycle of messages’

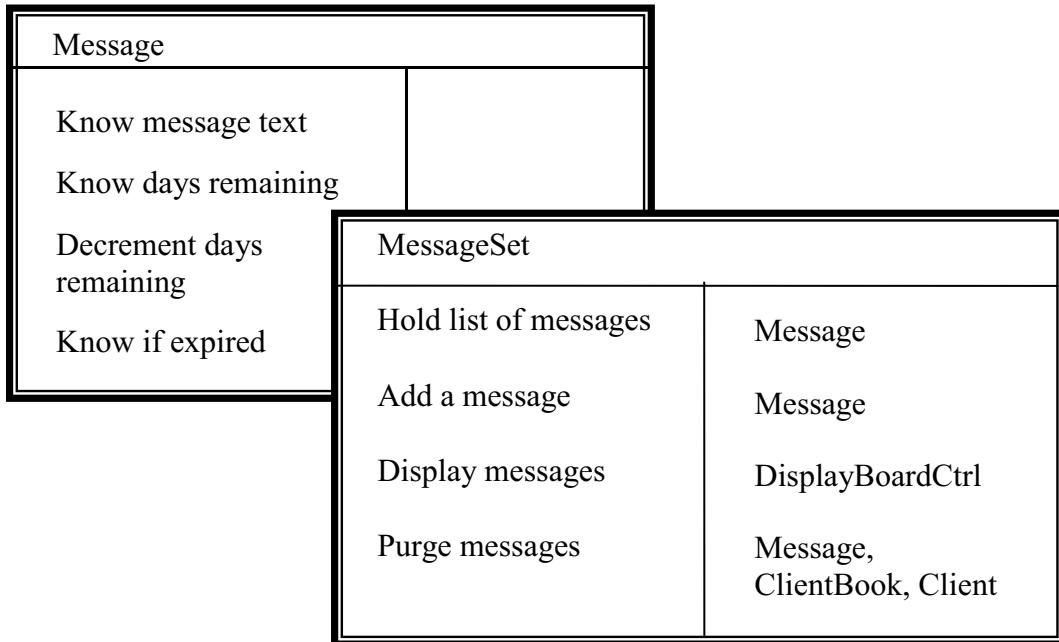
So this part of our design also seems to work.

### The Message Purge Scenario

The final scenario that we want to run though here is the message purge scenario. At the end of the day when messages have been displayed the remaining days of the message need to be decremented and the message will need to be deleted if a) the message has run out of days or b) the client has run out of credit.

CRC cards for the classes involved in this have been drawn below...





### Activity 1

To purge the messages, the MessageBook cycles through its list of messages reducing the credit for the client who 'owns' this message, decrementing the days remaining for that message and deleting messages when appropriate.

Looking at the CRC cards above work through the following steps and identify any potential problems with these classes:

For each message

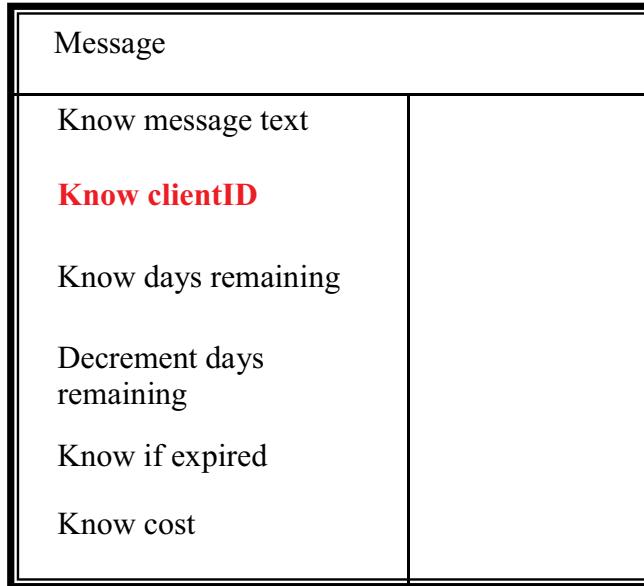
- tell the Message to decrement its days remaining and
- tell the relevant Client to decrease its credit
  - ask the Message for its client ID
  - ask the Message for its cost
  - ask the ClientBook for the client with this ID
  - tell the Client to decrease its credit by the cost of the message
- if either the Client's credit is  $\leq 0$  or the Message is now expired
  - delete the message from the list

### Feedback 1

A problem becomes evident when we try to find the client associated with a message as Message does not know the client ID.

We therefore need to add this responsibility to the Message class.

A revised design for the Message class is given below...



By drawing out CRC cards for each class and interface and by role playing a range of scenarios we have checked and revised our plans for the system – we can now refine these and document these using UML diagrams.

## 14.4 DOCUMENTING THE DESIGN USING UML

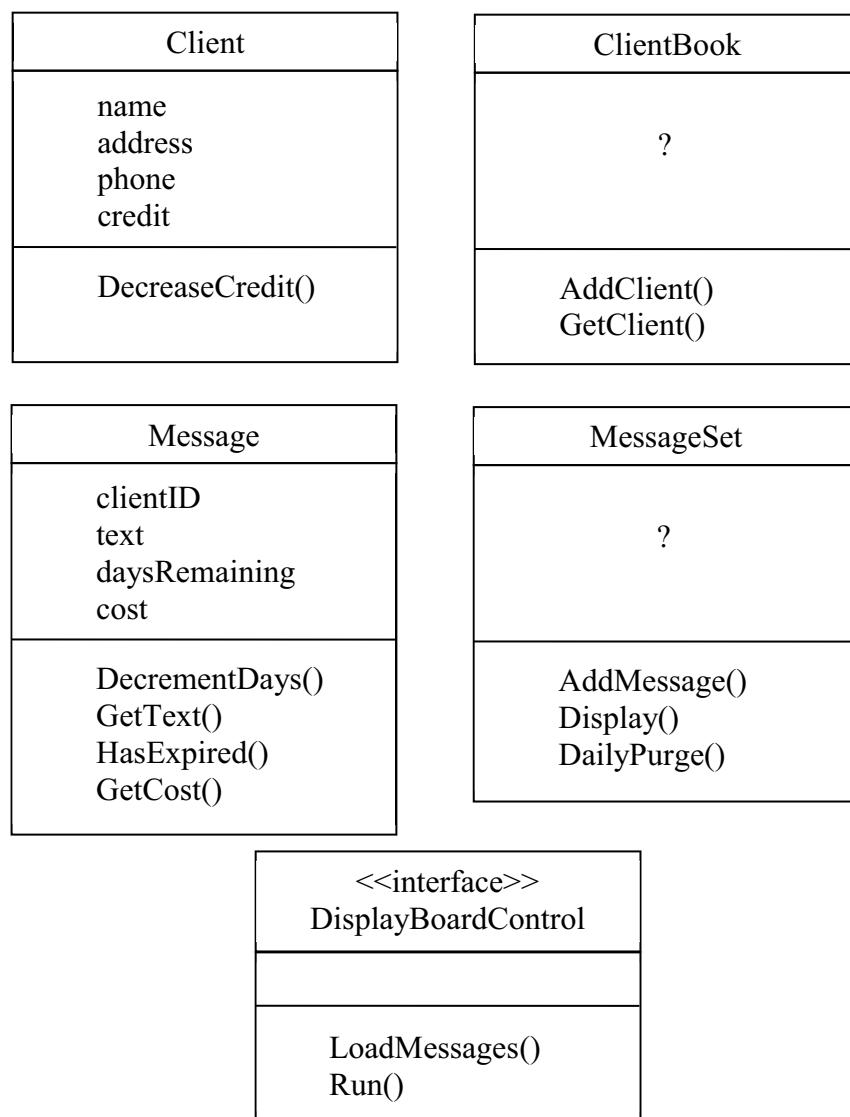
To fully document our designs we need to:

- Determine in detail what attributes go in each class
- Determine how the classes are related and
- Put classes into appropriate packages.

## Elaborating the Classes

Having worked through CRC scenarios we can make an initial assignment of instance variables and methods among our classes, including some accessors and mutators whose necessity has become evident (see diagram below).

Of course as we are going to program this system in C# we will replace our accessor and mutator methods with properties but as this design could be programmed in any Object Oriented language we will leave our design showing appropriate accessor and mutator methods.



We don't know of any simple attributes which ClientBook and MessageSet will require, but they will need to be associated with other classes so we still have some work to do there – hence the ?s (which are not an official part of UML).

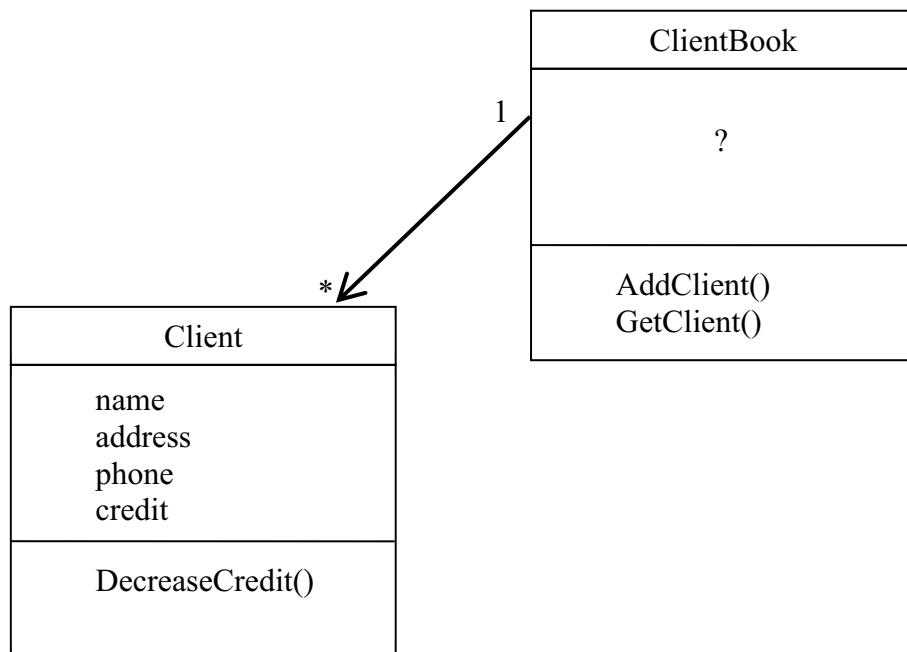
## Relationships Between Classes

We can now start to work out how these classes are related.

Starting with ClientBook and Client: a ClientBook will record details of zero or more clients.

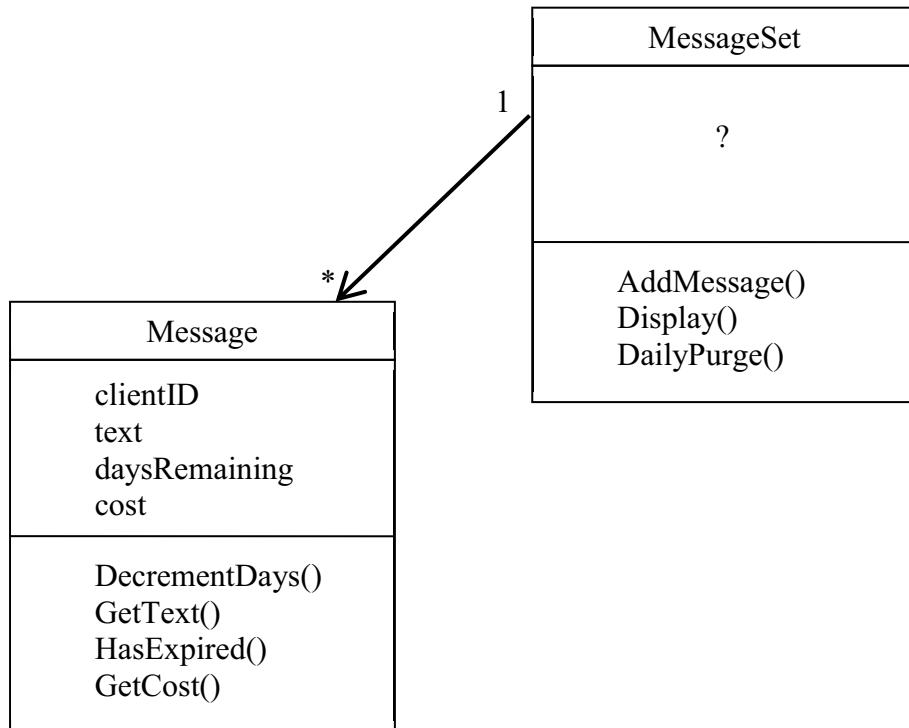
The navigability will be from ClientBook to client because the book “knows about” its Clients (in implementation terms it will have references to them) but the individual Clients will not have references back to the book.

The one-to-many relationship suggests that ClientBook will have a **Collection** (of some kind) of Clients. The specification states that each Client will have a unique ID thus the collection will in fact be a dictionary where each entry is made up of a pair of values – in this case a clientID (an int) and a Client object. As we are likely to be searching and retrieving clients by their ID far more often than we will be adding and deleting clients the system will be more efficient if we make this dictionary a sorted dictionary, where the clients will be stored in order of their ID. With a sorted dictionary adding and deleting elements will be slower as the process is more complex but retrieving elements will be quicker.



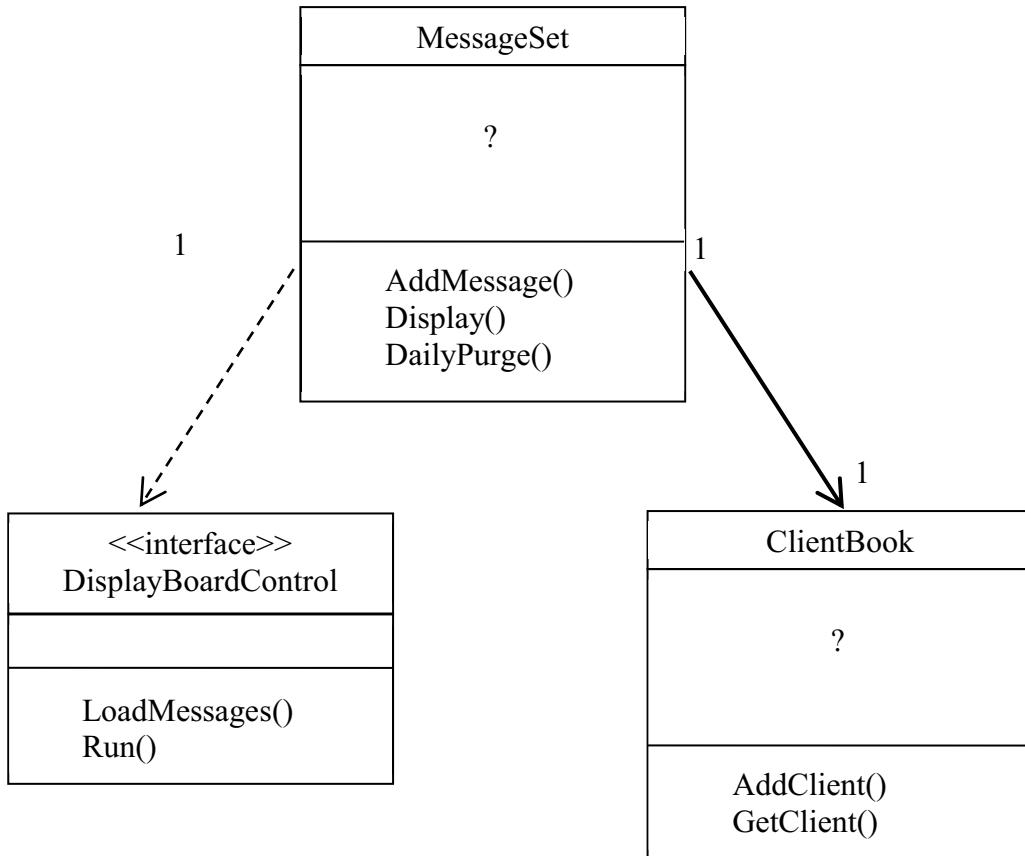
The relationship between MessageSet and Message is very similar to the relationship between ClientBook and Clients.

Although MessageSet appears to have no attributes, its one-to-many association with Message again implies an attribute which is a Collection type. The specification states that messages must be unique but does not imply a key value is required thus a simple set will suffice.



**Relating the Classes: MessageSet, ClientBook, and DisplayBoardControl** Because **MessageSet** is responsible for initiating the display of the messages on the display board it has a dependency on a class implementing the **DisplayBoardControl** interface.

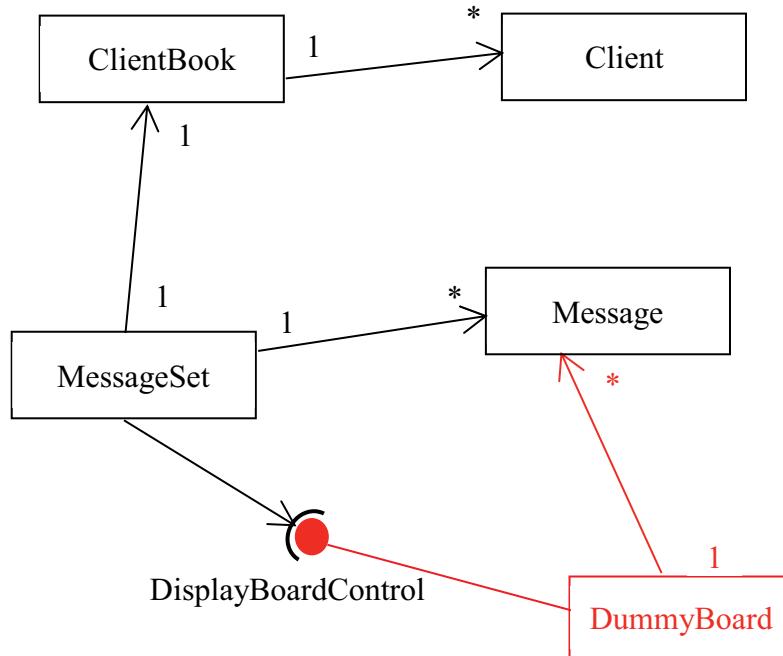
**MessageSet** also has a relationship with **ClientBook** because it needs to access and update Client information when the daily purge is carried out. This is shown below.



### Relating the Classes Overall

The diagram below shows how all of these classes are related. An additional class, **DummyBoard**, has been included which will implement the **DisplayBoardControl** interface for testing purposes.

Since **DummyBoard** will have a collection of messages loaded it also has a one-to-many relationship with **Message**.



Note: the use of the concise “ball and socket” notation for the DisplayBoardControl interface.

While the classes above will form the heart of the system an additional class will be required to drive and manage the system as a whole.

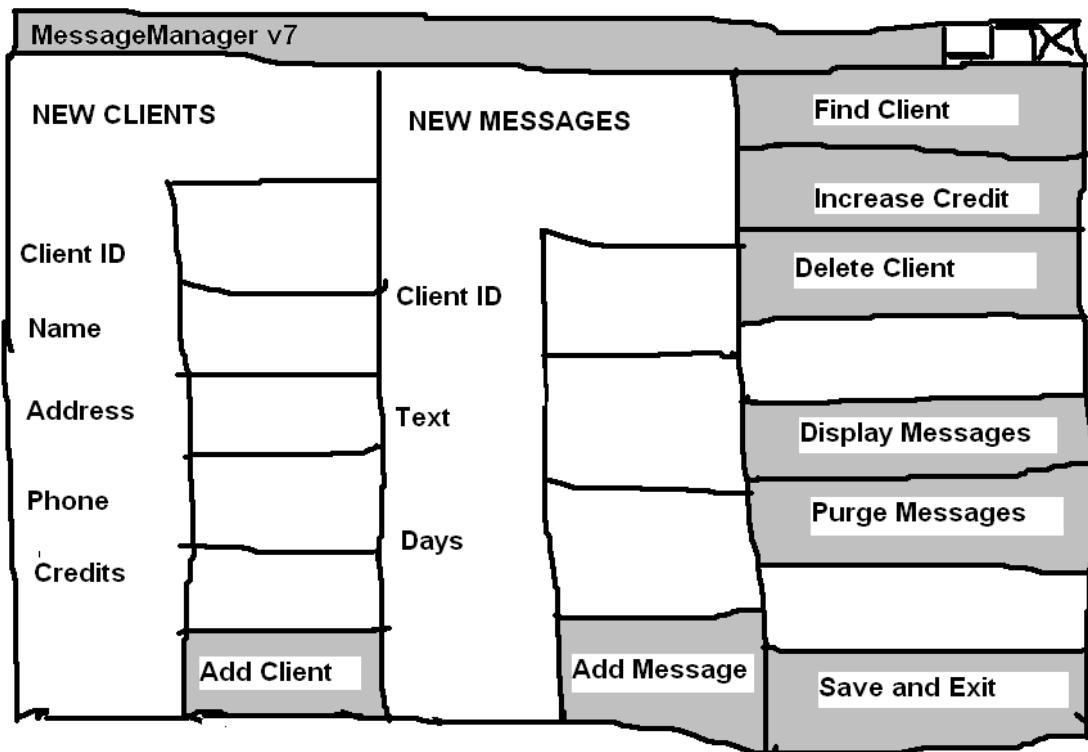
A class ‘ApplicationWindow’ will be created. This will be the GUI that will run the system and this will allow the user will interact with the system adding clients, messages etc.

Other additional functionality, not specified by the shop owner, is implicitly required. At the end of the day the details of the ClientBook and MessageSet will need to be saved to file. This data will need to be restored next time the system is run as the shop owner will clearly not want to enter details of all the clients every time they run the program. This again will be driven from the interface but the body of the code will be devolved to the relevant classes.

## 14.5 PROTOTYPING THE INTERFACE

While methods for gathering user requirements is beyond the scope of this text – it is always a good idea to prototype an interface and get feedback on this before proceeding with the development.

The figure below shows the proposed interface for this system:



This is made up of three areas. From left to right these are a) an area for adding new clients, b) an area for adding new messages and c) an area for buttons dedicated to other essential operations.

## 14.6 REVISING THE DESIGN TO ACCOMMODATE CHANGING REQUIREMENTS

Changing software requirements are a fact of life and Object Oriented programming is intended to help software engineers make program adaptations easier, quicker, cheaper and with less risk of generating errors. The principles of inheritance, method overriding and polymorphism are essential Object Oriented features that help in this manner.

In this project when gaining feedback from the shop owner on the prototype interface they comment that they generally like the interface but that they have an additional system requirement:

Some messages are 'urgent messages'. These should be highlighted on the display by placing three stars before and after the message and the cost of these messages will be twice the cost of ordinary messages. Other than that, urgent messages are just like ordinary messages.

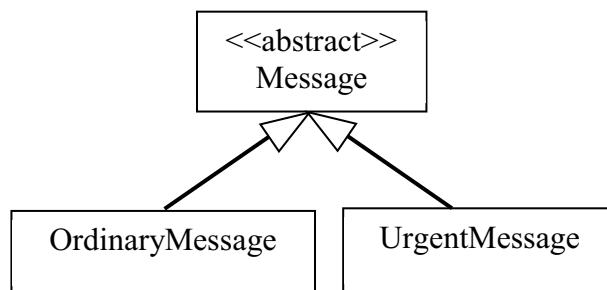
Modifying the interface design to accommodate this change is easy – we can either:

- create a new panel to accommodate the creation of ‘Urgent Messages’ or
- since the data required for an urgent message is identical to normal messages we can just add an extra button to the middle panel ‘Add Urgent Message’.

But how will these extra requirements impact on the underlying classes within the system?

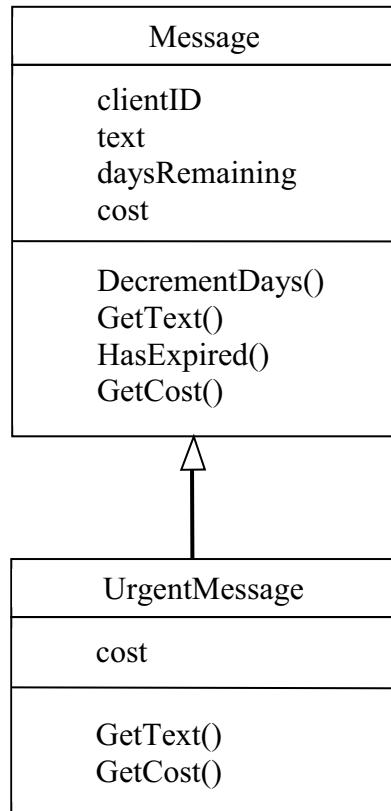
If Object Oriented principles work implementing this additional requirement should be relatively simple. Firstly there is clearly a strong relationship between a ‘Message’ and an ‘Urgent Message’.

If both classes had some unique features but there was a significant overlap in functionality we could introduce an inheritance hierarchy to deal with this:

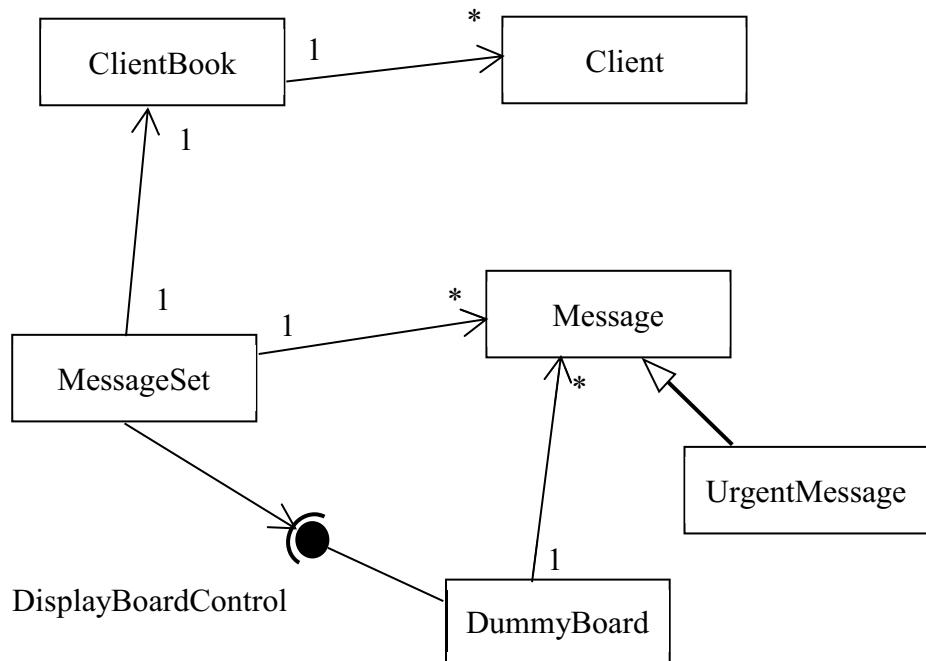


However in this case there are no unique features of an ordinary message – messages have an associated cost, the cost and text can be obtained and new messages can be created. All this is true for urgent messages. An urgent message is just the same as an ordinary message where the text and the cost have been changed slightly. Thus `UrgentMessage` is a type of `Message` and can inherit ALL of the features of `Message` with the cost and text methods being overridden.

Thus the `Message` and `UrgentMessage` classes are related as shown below, with `UrgentMessage` inheriting all of the values and methods associated with `Message` but overriding `GetCost()` and `GetText()` methods to reflect the different cost and text associated with urgent messages.



A revised class diagram is below. But how will this change impact upon other parts of the system?



Thanks to the operation of polymorphism **this change will have no impact at all on any other part of the system!**

Looking at the class diagram above we can see that MessageSet keeps and manages a set of Messages (DummyBoard also keeps a set of messages – once they have been uploaded for display). But what about UrgentMessages?

Urgent messages are just a specific type of message. When the AddMessage() method is invoked within MessageSet it requires an object of type Message i.e. a message to be added – but an object of the subtype UrgentMessage **is still a ‘Message’** so the AddMessage() method would accept an UrgentMessage object.

Therefore, without making any changes at all to MessageSet, MessageSet can maintain a set of all messages to be displayed (both urgent and ordinary)!

Furthermore when the DailyPurge() method is invoked it invokes the GetCost() method on a Message object so that the client can be charged for that message. At run time the Common Language Runtime (CLR) engine will determine whether the object is of type Message or of type UrgentMessage and it will invoke the correct version of the GetCost() method – remember this was overridden in UrgentMessage. This is polymorphism in action!

MessageSet requires messages but, thanks to the application of polymorphism and method overriding, MessageSet will happily deal with any Message subtype as though it were a Message object. If later we decided to create new message types, such as a Christmas or Valentine message, MessageSet would be able to deal with these as well without changing a single line of code!

Thus in this application we are able to extend the system to add the facility for urgent messages by adding only one class and making one small change to the interface.

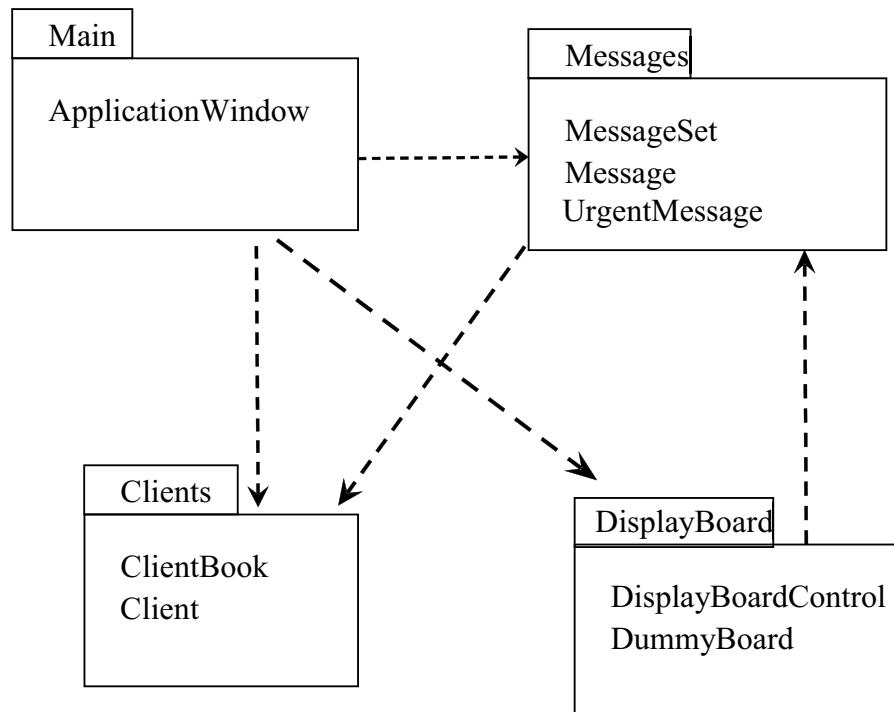
Without the application of polymorphism we would need to have made additional changes to other parts of the system – namely MessageSet and DummyBoard.

Object Orientation has enabled to the system to be extended with minimal effort!

## 14.7 PACKAGING THE CLASSES

Large programs should be segmented into packages as this provides an appropriate level of encapsulation and access control (as described in Chapter 2).

The system being used here to demonstrate the theory in this textbook hardly qualifies as large – nonetheless it has been decided to package related classes together as shown below.



This diagram shows the four packages used and the classes within each package. Also shown are associations between the packages. Not surprisingly the main package, which houses the system interface, is associated with all of the other packages – this is because the interface invokes functionality throughout the system.

Having completed the design, and accommodated changing requirements, we can start implementing the system. This will be done in two phases:

In the first phase a basic system will be implemented which will allow messages and clients to be created, the details written to file and messages to be displayed.

In the second phase the system functionality will be extended to allow clients to be deleted and to allow their credit to be increased. This will be done in a way to allow the demonstration of Test Driven Development (as described in Chapter 12).

## 14.8 PROGRAMMING THE MESSAGE CLASSES

Message, UrgentMessage and MessageSet are relatively straight forward to program.

Message has various instance variables (String: messageText and int: COST, clientID and daysRemaining). It has appropriate properties to access these private attributes (note only daysRemaining needs a setter) and a constructor to initialize the instance variables. It also has the following methods:

```
void DecrementDays()      // to reduce the number of days that the message should  
                          be displayed for  
  
boolean HasExpired()     // to specify if this message should be removed from the  
                          set of messages  
  
String ToString()        // to return the text to be displayed on the displayboard.
```

At some point we will need to store the ClientBook and MessageSet objects to a file. To do this all Client objects and Message objects will also need to be stored hence these classes (including the Message class) will need to be marked as Serializable.

Finally the requirements state that “No duplicate messages (i.e. the same text for the same client) are permitted.”

Therefore Message must override the Equals() and GetHashCode() methods to ensure that duplicates will not be permitted when the messages are stored in a Set.

The complete code for this class is given below – though comments have been excluded for the sake of brevity.

The source code for the full system, fully commented, can be viewed by following the instructions near the end of this chapter.

```
namespace MessageManagerSystem.Messages
{
    [Serializable]
    public class Message
    {
        const int COST = 1;
        public virtual int Cost
        {
            get { return COST; }
        }

        private int clientID;
        public int ClientID
        {
            get { return clientID; }
        }

        private String messageText;
        public String MessageText
        {
            get { return messageText; }
        }

        private int daysRemaining;
        public int DaysRemaining
        {
            get { return daysRemaining; }
            set { daysRemaining = value; }
        }

        public Message(int clientID, String text, int daysRemaining)
        {
            this.clientID = clientID;
            this.messageText = text;
            this.daysRemaining = daysRemaining;
        }
    }
}
```

```
public void DecrementDays()
{
    daysRemaining--;
}

public bool HasExpired()
{
    return (daysRemaining <= 0);
}

public override String ToString()
{
    return (messageText);
}

public override bool Equals(object obj)
{
    if ((obj == null) || (GetType() != obj.GetType()))
    {
        return false;
    }
    Message m = (Message)obj;
    return (clientID.Equals(m.ClientID) &&
            messageText.Equals(m.MessageText));
}

public override int GetHashCode()
{
    return (messageText + clientID).GetHashCode();
}

}
```

The UrgentMessage class is extremely short and sweet as it inherits almost all of its functionality from Message:

```
namespace MessageManagerSystem.Messages
{
    [Serializable]
    public class UrgentMessage : Message
    {
        const int COST = 2;
        public override int Cost
        {
            get { return COST; }
        }

        public UrgentMessage(int clientID, String text, int daysRemaining) : base (clientID, text, daysRemaining)
        {
        }

        public override String ToString()
        {
            return ("*** "+ MessageText + " ***");
        }
    }
}
```

Only the ‘Cost’ property and ToString() methods are overridden as UrgentMessages cost more and the text to be displayed changes. Note: to override the Cost property we must mark it as virtual in the Message class.

The MessageSet class has a one-to-many relationship with Message. This implies a collection type and the fact that duplicate messages are not allowed (at least for the same client) implies the collection should be a Set.

The MessageSet class requires an instance variable to hold the set of messages and it will need access to a ClientBook object as it needs access to the clients when performing a daily purge. The client book object could be stored using an instance variable or passed as a parameter to the DailyPurge() method. As it is only required by this one method the decision was made to pass this as a parameter each time the DailyPurge() is invoked.

A constructor is required to assign a new HashSet() to Messages (the set of messages stored). The following methods are also required:

void AddMessage(Message msgToAdd)	to add a message to the message set
void display(DisplayBoardControl db)	to display the messages each day on a display board...initially a simulated display board
void DailyPurge(ClientBook clients)	to a) decrement the days remaining at the end of each day for each message, b) charge the client for displaying that message and c) remove all messages that have expired.
private bool ToBeDeleted()	a private method used by the DailyPurge() to denote which messages are to be removed from the message set i.e. those that have expired.

Some of the code from this class is shown below:

```
namespace MessageManagerSystem.Messages
{
    [Serializable]
    public class MessageSet
    {
        private HashSet<Message> messages;
        public HashSet<Message> Messages
        {
            get { return messages; }
        }

        public MessageSet()
        {
            messages = new HashSet<Message>();
        }

        public void AddMessage(Message msgToAdd)
        {
            messages.Add(msgToAdd);
        }

        public void Display(IDisplayBoardControl db)
        {
            db.LoadMessages(messages);
            db.Run();
        }

        public void DailyPurge(ClientBook clients)
        {
            // code omitted here
        }

        private bool ToBeDeleted(Message m)
        {
            return m.HasExpired();
        }
    }
}
```

The code above shows the creation of a typed collection of 'Message' called Messages and methods to add and display messages.

The method to display messages requires and object of type DisplayBoardControl to be passed as a parameter. Initially a DummyBoard object will be provided however when a

real display board is purchased then this object will replace the DummyBoard object. This will have no impact on the code within the Display() method as both objects are of the more general type DisplayBoardControl. This is another example of the application of polymorphism.

The DailyPurge() method was excluded from the code above so we could concentrate on this method now.

The DailyPurge() method performs the following actions:

For each message

- Decrement the days remaining for that message
- Find the client who paid for that message
- Find the cost of the message and deduct this from that clients credit

For each message

- If the message has expired or if the client has run out of credit then
  - Set the DaysRemaining for that message to zero.

Remove all expired messages.

See code below for this...

```
public void DailyPurge(ClientBook clients)
{
    Client client;

    // loop through all current messages and decrement credit and days
    foreach( Message m in messages)
    {

        m.DecrementDays(); // deduct 1 from days remaining for message
        try
        {
            // decrease client credit for this message
            client = clients.GetClient(m.ClientID);
            client.DecreaseCredit(m.Cost);
        }
    }
}
```

```
        catch (UnknownClientException uce)
        {
            MessageBox.Show("INTERNAL ERROR IN MessageSet.Purge()
\r\nException Details: " + uce.Source + " \r\nMessage
details " + uce.Message + ": \r\n");
        }

        // loop through all current messages
        // and expire those whose client credit <=0
        foreach (Message m in messages)
        {
            try
            {
                client = clients.GetClient(m.ClientID);
                if (client.Credit <= 0)
                {
                    m.DaysRemaining = 0;
                }
            }
            catch (UnknownClientException)
            {
                // Do nothing as unknown clients have been reported to error
                // log already
            }
        }

        // Remove all expired messages
        messages.RemoveWhere(ToDelete);
    }
}
```

Note: it is possible that a client could not be found – hence the try catch block in the code above. This will be discussed in the next section.

## 14.9 PROGRAMMING THE CLIENT CLASSES

The system needs a Client class, with methods to decrease the client's credit (when a message has been displayed for them). Ultimately it will also need a method to allow a client to pay for and increase their credit but we will not add this functionality in the first version of this system. This class is simple and is very similar to programming the Message class and is therefore not shown here.

Programming the ClientBook class is also similar to programming MessageSet, class however there are a few significant differences:

- All clients have a ClientID so ClientBook uses a sorted dictionary instead of a Set.
- The method GetClient() could fail if no client exists with the specified ClientID. We need to build in protection in case a client cannot be found.
- The method AddClient() could also fail if a client already exists with the specified ID.

The complete ClientBook class (without comments) is shown below. By downloading the finished program this code can be viewed with embedded comments.

```
namespace MessageManagerSystem.Clients
{
    [Serializable]
    public class ClientBook
    {
        private SortedDictionary<int, Client> clients;
        public SortedDictionary<int, Client> Clients
        {
            get { return clients; }
        }

        public ClientBook()
        {
            clients = new SortedDictionary<int, Client>();
        }

        public ClientBook(SortedDictionary<int, Client> clients)
        {
            this.clients = clients;
        }

        public void AddClient(int clientID, Client newClient)
        {
            try
            {
                clients.Add(clientID, newClient);
            }
            catch (ArgumentException)
            {
                throw new ClientAlreadyExistsException
                    ("ClientBook.AddClient(): a client with this ID
                     already exists in system ID:" + clientID);
            }
        }
    }
}
```

```
public Client GetClient(int clientID)
{
    try
    {
        return clients[clientID];
    }
    catch (KeyNotFoundException)
    {
        throw new UnknownClientException
            ("ClientBook.GetClient(): unknown client ID:" +
            clientID);
    }
}
```

The code above shows the constructors to create a ClientBook object which is a sorted dictionary of ClientID, Client objects and the other methods required by the ClientBook class. Further discussion of this is provided below.

## 14.10 CREATING AND HANDLING UNKNOWNCLIENTEXCEPTION

The GetClient() method will generate a KeyNotFoundException if no client exists with the specified ID. If we do not catch and deal with this exception our program will crash! Furthermore our program will crash as we try to invoke the DecreaseCredit() method without having a client object to invoke this method on.

To protect against this we need to:

- Create a new kind of exception (as described in Chapter 11) called UnknownClientException
- tell the ClientBook class to throw this exception if a client is not found
- catch and deal with this exception in the DailyPurge() method.

The first step is simple...

```
public class UnknownClientException : ApplicationException
{
    public UnknownClientException(String message) : base(message)
    {
    }
}
```

We next tell the GetClient() method to generate an UnknownClientException if it catches a KeyNotFoundException (as shown below):

```
public Client GetClient(int clientID)
{
    try
    {
        return clients[clientID];
    }
    catch (KeyNotFoundException)
    {
        throw new UnknownClientException("ClientBook.GetClient() :
            unknown client ID:" + clientID);
    }
}
```

Under the appropriate condition, we invoke the constructor of the exception using the keyword ‘new’ and pass a string message required by the constructor. The object returned by the constructor is then ‘thrown’.

To be helpful the string specifies the method where this exception was generated from and the clientID for which a client was not found. The DailyPurge() method should catch this exception and hopefully deal with it to prevent a crash situation.

The final step is to catch and deal with UnknownClientException within the DailyPurge() method – as shown in section 14.8 (Programming the Message Classes).

If a client does not exist we could remove the message. However in this case we have chosen to be more cautious since we simply don’t know how we have come to have an ‘unowned’ message.

We have therefore decided that if the message has no recognized client we will not to take any action other than to report the error. The message will continue to be displayed (even without having a client to charge!).

If an unowned message has expired we of course still need to remove it from the display set.

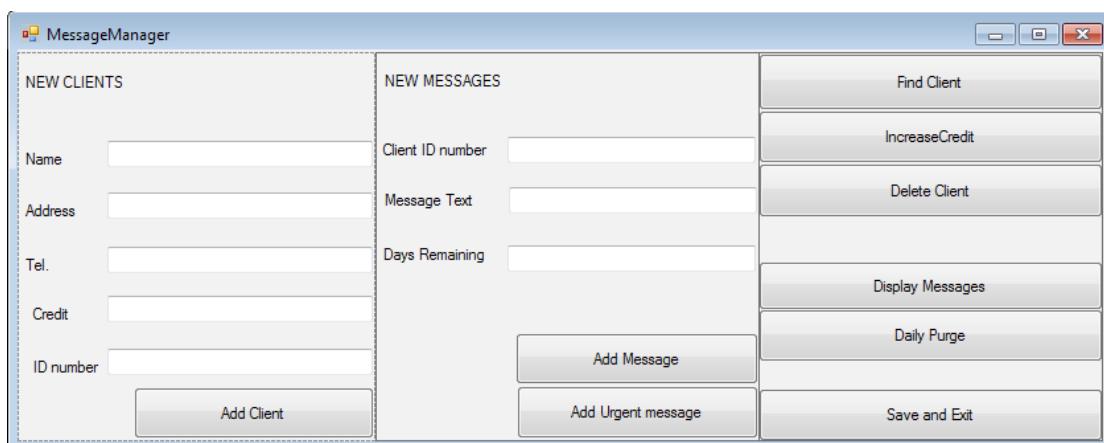
## 14.11 PROGRAMMING THE INTERFACE

We have still to examine the main application window that will be used to ‘drive’ the system.

It performs the following functions:

- it has a permanent reference to the client book and message set.
- it sets up the data file and used for storing the ClientBook and Message Set
- it defines what happens when the system starts and
- it defines what happens when the system shuts down
- it invokes the constructors for the Message, UrgentMessage and Client classes whenever the user wants to add a new message or client to the system.
- it displays the messages on a dummy display board and
- it invokes the DailyPurge() method when requested by the user at the end of each day.

The application window will look as shown below...



Note: this includes buttons for increasing a client’s credit and deleting a client – functionality we realised some time ago that we needed but functionality that was not added to the first version of this system. We will extend our system to add this additional functionality once a basic system is working.

The ApplicationWindow\_Load() method is shown below:

```
private void ApplicationWindow_Load(object sender, EventArgs e)
{
    clientBook = new ClientBook();
    messageSet = new MessageSet();
    try
    {
        FileStream inFile = new FileStream("ClientAndMessageData",
                                            FileMode.Open, FileAccess.Read);
        BinaryFormatter bFormatter = new BinaryFormatter();
        clientBook = (ClientBook)bFormatter.Deserialize(inFile);
        messageSet = (MessageSet)bFormatter.Deserialize(inFile);
        inFile.Close();
        inFile.Dispose();
    }
    catch (FileNotFoundException)
    {
    }
}
```

The ApplicationWindow\_Load() method reconstructs any previously stored ClientBook and MessageSet objects.

The action listener for the SaveAndExit button is shown below...

```
private void btnSaveAndExit_Click(object sender, EventArgs e)
{
    FileStream outFile = new FileStream("ClientAndMessageData",
                                         FileMode.Create, FileAccess.Write);
    BinaryFormatter bFormatter = new BinaryFormatter();

    bFormatter.Serialize(outFile, clientBook);
    bFormatter.Serialize(outFile, messageSet);

    outFile.Close();
    outFile.Dispose();
    this.Close();
}
```

Note: how with just four lines of code above we can create an appropriate output stream and save all client book and message set data – this includes details of all clients and all messages. While we had to mark the relevant classes, ClientBook, Client, MessageSet and Message, as serializable we did not have to write any code to save this data to file.

Shown below is the action listener associated with the FindClient button:

```
private void btnFindClient_Click(object sender, EventArgs e)
{
    Client client;
    try
    {
        InputBox inputBox = new InputBox("Find Client", "Please enter
clients ID.");
        DialogResult dialogResult = inputBox.ShowDialog();

        if (dialogResult == DialogResult.OK)
        {
            int clientID;
            if (!Int32.TryParse(inputBox.Answer, out clientID))
            {
                MessageBox.Show("Invalid client ID, please enter
integer number.");
                return;
            }

            client = clientBook.GetClient(clientID);
            MessageBox.Show(client.ToString());
        }
    }
    catch (UnknownClientException ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

This action listener performs the following tasks:

- It opens a dialog box to ask the user for a clients ID. As C# does not contain predefined methods to create input boxes this uses a form called InputBox that was created specifically for this purpose. If cancel is pressed this form returns an empty string.
- It then checks that OK has been pressed and the ID returned is a valid integer.
- On the client book object it invokes the GetClient() method passing the ID as a parameter.

- Assuming a client object is returned, the `ToString()` method is then invoked to get a string representation of the client and this is passed as a parameter to the `MessageBox.Show()` method (which displays the details of the client with that ID).
- If `GetClient()` fails to find a client with the specified ID it will throw an `UnknownClientException` – this will be caught here and an appropriate message will be displayed. This makes use of the `Message` property of the `Exception` class.

## 14.12 USING TEST DRIVEN DEVELOPMENT AND EXTENDING THE SYSTEM

We now have a working system – though two important methods have yet to be created. We need a method to increase a client's credit – this should be placed within the `Client` class. We also need a method to delete a client, as this means removing them from the client book. This should be placed in the `ClientBook` class.

It has been decided to use Test Driven Development (TDD) to extend the system by providing this functionality (as discussed in Chapter 12 Agile Programming).

In TDD we must:

- 1) Write tests
- 2) Set up automated unit testing, which fails because the classes haven't yet been written!
- 3) Write the classes so the tests pass

After creating these methods we must then adapt the interface so that this will invoke these methods.

To do this we will create two test fixtures...one to test the `ClientBook` class and one to test the `Client` class.

In the `Client` text fixture we will initialise a test by setting up a specified client. We will then write a test which will test if credit can be added and finally we will set the `TestCleanup()` method to remove the client specified.

The code for this is shown below...

```
namespace MessageManagerTests
{
    [TestClass]
    public class TestFixture_ClientTests
    {

        private MessageManagerSystem.Clients.Client c;

        [TestInitialize]
        public void TestInitialize()
        {
            c = new MessageManagerSystem.Clients.Client("Simon", "Room
                1234", "x200", 10);
        }

        [TestCleanup]
        public void TestCleanup()
        {
            c = null;
        }

        [TestMethod]
        public void IncreaseCredit_TestAdd5UnitsOfCredit
            _CreditShouldBe15()
        {
            c.IncreaseCredit(5);
            Assert.AreEqual(15, c.Credit, "Credit after adding 5
                units is not as expected. Expected: 15 Actual:
                "+c.Credit);
        }

    }
}
```

This test creates a client with 10 units of credit, adds an additional 5 units of credit and then checks that this client has 15 units of credit.

One test alone does not sufficiently prove that the IncreaseCredit() method will always work so we may need to define additional tests.

We also need to create test cases to test the DeleteClient() method in the ClientBook class. As this is a separate class we need to create a new test fixture appropriately named and we need to set up a test to test this method.

The code for this is shown blow...

```
namespace MessageManagerTests
{
    [TestClass]
    public class TestFixture_ClientBookTests
    {
        public TestFixture_ClientBookTests()
        {
        }

        private MessageManagerSystem.Clients.ClientBook cb;

        [TestInitialize]
        public void TestInitialize()
        {
            cb = new MessageManagerSystem.Clients.ClientBook();
        }

        [TestCleanup]
        public void TestCleanup()
        {
            cb = null;
        }

        [TestMethod]
        public void GetClient_TestDeleteClient
            _ShouldNotGenerateException()
        {
            MessageManagerSystem.Clients.Client c = new
                MessageManagerSystem.Clients.Client
                    ("Simon", "Room 1234", "x200", 10);
            try
            {
                cb.AddClient(1, c);
                cb.DeleteClient(1);

            }
            catch
                (MessageManagerSystem.Clients.UnknownClientException)
            {
                Assert.Fail("UnknownClient exception should not be
                    thrown if client exists");
            }
        }
}
```

One test we should perform on the DeleteClient() method is to test that it can delete a client...or at least not generate an exception. The test above proves an exception is not thrown inappropriately but it does not demonstrate that the client has been successfully deleted nor does it test what happens if we try to delete a client that does not exist...clearly we need to define some more tests.

Having created appropriate test cases our code will generate compiler errors as the methods IncreaseCredit() and DeleteClient() do not exist.

We must now add these methods to our program and revise them until these tests pass.

The IncreaseCredit() method is given below...

```
public void IncreaseCredit(int extraCredit)
{
    credit = credit + extraCredit;
}
```

And the DeleteClient() method is given below...

```
public void DeleteClient(int clientID)
{
    if(clients.Remove(clientID)==false)
    {
        throw new
        UnknownClientException("ClientBook.DeleteClient() :
        unknown client ID:" + clientID);
    }
}
```

Finally we must amend the system GUI to invoke these methods as required.

Theory suggests that TDD leads to simple code that becomes easy to amend and this is important when following an agile methodology.

In this case by focusing our minds on what the IncreaseCredit() and DeleteClient() methods needs to achieve we reduce the risk of over complicating the code. Of course we may need a range of test cases to make sure the method has all of the essential functionality it needs.

Even if not developing our system using TDD we should define a wide ranging set of test cases for all of the classes within the system. This will ensure that we can undertake regression testing every time we enhance or adapt the system to meet the future and ever changing needs of the client.

Some more tests for the ClientBook class are shown below...

```
[TestMethod]
public void AddClient_TestAddingClient_ShouldNotGenerateException()
{
    MessageManagerSystem.Clients.Client c = new
    MessageManagerSystem.Clients.Client("Simon",
    "Room 1234", "x200", 10);
    try
    {
        cb.AddClient(1, c);
    }
    catch
    (MessageManagerSystem.Clients.ClientAlreadyExistsException)
    {
        Assert.Fail("ClientAlreadyExists exception should not be
        thrown for new clients");
    }

}

[TestMethod]
public void AddClient_TestAddClientTwice_ShouldGenerateException()
{
    MessageManagerSystem.Clients.Client c = new
    MessageManagerSystem.Clients.Client("Simon",
    "Room 1234", "x200", 10);
    try
    {
        cb.AddClient(1, c);
        cb.AddClient(1, c);
        Assert.Fail("ClientAlreadyExists exception should be thrown
        if client added twice");
    }
    catch
    (MessageManagerSystem.Clients.ClientAlreadyExistsException)
    {
    }
}
```

```
[TestMethod]
[ExpectedException(typeof(MessageManagerSystem.Clients.
ClientAlreadyExistsException))]
public void AddClient_TestClientTwice_AlternativeVersion()
{
    MessageManagerSystem.Clients.Client c = new
    MessageManagerSystem.Clients.Client("Simon",
    "Room 1234", "x200", 10);
    cb.AddClient(1, c);
    cb.AddClient(1, c);
    Assert.Fail("ClientAlreadyExists exception should be thrown if
client added twice");
}

[TestMethod]
public void
GetClient_TestGettingUnknownClient_ShouldGenerateException()
{
    try
    {
        cb.GetClient(1);
        Assert.Fail("UnknownClient exception should be thrown if
client does not exist");
    }
    catch (MessageManagerSystem.Clients.UnknownClientException)
    {
    }
}

[TestMethod]
public void GetClient_TestGettingClient_ShouldNotGenerateException()
{
    MessageManagerSystem.Clients.Client c = new
    MessageManagerSystem.Clients.Client("Simon",
    "Room 1234", "x200", 10);
    MessageManagerSystem.Clients.Client c2 = null;
    try
    {
        cb.AddClient(1, c);
        c2=cb.GetClient(1);
    }
}

[TestMethod]
public void
GetClient_TestDeleteUnknownClient_ShouldGenerateException()
{
```

```
try
{
    cb.DeleteClient(1);
    Assert.Fail("UnknownClient exception should be thrown if
client does not exist");
}
Catch (MessageManagerSystem.Clients.UnknownClientException)
{
}
}

catch (MessageManagerSystem.Clients.UnknownClientException)
{
    Assert.Fail("UnknownClient exception should not be thrown if
client exists");
}
}

[TestMethod]
public void GetClient_TestGettingClient_AttributesShouldNotChange()
{
    MessageManagerSystem.Clients.Client c = new
    MessageManagerSystem.Clients.Client("Simon",
    "Room 1234", "x200", 10);
    MessageManagerSystem.Clients.Client c2 = null;
    try
    {
        cb.AddClient(1, c);
        c2 = cb.GetClient(1);
        Assert.AreEqual(c2.Credit, 10, "Value of returned credit not as
expected");
    }
    catch (MessageManagerSystem.Clients.UnknownClientException)
    {
    }
}
```

The tests above show numerous tests with an empty client book. They demonstrate that clients can be added, but not twice. They also demonstrate that clients can be deleted and that exceptions are generated appropriately.

The figure below shows the results from running the tests...

Test run completed Results: 10/10 passed; Item(s) checked: 0	
Result	Test Name
<input type="checkbox"/>	DecreaseCredit_TestRemove5UnitsOfCredit_CreditShouldBe5
<input type="checkbox"/>	AddClient_TestAddClientTwice_ShouldGenerateException
<input type="checkbox"/>	GetClient_TestGettingClient_AttributesShouldNotChange
<input type="checkbox"/>	GetClient_TestGettingClient_ShouldNotGenerateException
<input type="checkbox"/>	AddClient_TestAddClientTwice_AlternativeVersion
<input type="checkbox"/>	IncreaseCredit_TestAdd5UnitsOfCredit_CreditShouldBe15
<input type="checkbox"/>	GetClient_TestDeleteClient_ShouldNotGenerateException
<input type="checkbox"/>	GetClient_TestGettingUnknownClient_ShouldGenerateException
<input type="checkbox"/>	GetClient_TestDeleteUnknownClient_ShouldGenerateException
<input type="checkbox"/>	AddClient_TestAddingClient_ShouldNotGenerateException

By creating automated test fixtures to test all classes and all methods we can run these tests every time the system is adapted to meet the clients changing needs.

### 14.13 GENERATING THE DOCUMENTATION

Documentation is essential and can be generated automatically (as described in Chapter 10 – C# Development Tools) assuming appropriate comments have been placed in the code.

XML comments have been placed in the code to describe all classes, all constructors and all methods. All parameters, return values and exception thrown have also been described.

Three of the comments taken from the Client class are shown below:

```
/// <summary>
/// Manages a collection (sorted dictionary) of clients where each
/// client has an ID number (int).
/// </summary>
/// <remarks>Author Simon Kendal
/// Version 1.0 (5th May 2011)</remarks>
public class ClientBook
{
    private SortedDictionary<int, Client> clients;

    /// <summary>
    /// Gets the clients.
    /// </summary>
    public SortedDictionary<int, Client> Clients
    {
        // ... lines missing ...
    }
}
```

```
/// <summary>

/// Initializes a new empty instance of the <see cref="ClientBook"/> class.
/// </summary>
public ClientBook()
{
    // ... lines missing ...
}

/// <summary>
/// Initializes a new instance of the <see cref="ClientBook"/>
/// class and instantiates this to the disctionary passed.
/// </summary>
/// <param name="clients">A disctionary of Client ID, Client
/// objects.</param>
public ClientBook(SortedDictionary<int, Client> clients)

{
    // ... code omitted ...
}

/// <summary>
/// Adds a client to the client book
/// </summary>
/// <param name="clientID">The client ID.</param>
/// <param name="newClient">The new client.</param>
/// <exception cref="ClientAlreadyExistsException"> Throws
/// exception if a client with ClientID already exists</exception>
public void AddClient(int clientID, Client newClient)
{
    // ... code omitted ...
}
```

Once XML comments have been placed throughout the code and exported, and comments have been added to the Sandcastle Help File Builder tool to describe the namespaces then this tool can be used to generate a set of web pages to describe the system...virtually at the push of a button!

The following picture shows the main help page generated ‘Index.html’ documentation describing the Message Manager System at its highest most general level i.e. the packages or namespaces within the system.

Namespace	Description
<a href="#">MessageManagerSystem.Clients</a>	This namespace contains the classes associated with individual clients, collections of clients and associated exceptions.
<a href="#">MessageManagerSystem.DisplayBoard</a>	This namespace contains the interface that defines the functions of an electronic display board and a class to simulate an actual display board.
<a href="#">MessageManagerSystem.Main</a>	This namespace contains the main application window and any classes associated with the overall control of the application.
<a href="#">MessageManagerSystem.Messages</a>	This namespace contains the classes associated with messages to be displayed, collections of these messages and associated exceptions.

The following picture shows part of the help documentation describing the UrgentMessage class:

**Declaration Syntax**

C#	Visual Basic	Visual C++
----	--------------	------------

```
[SerializableAttribute]
public class UrgentMessage : Message
```

**Members**

All Members	Constructors	Methods	Properties
<input checked="" type="checkbox"/> Public	<input checked="" type="checkbox"/> Instance	<input checked="" type="checkbox"/> Declared	
<input checked="" type="checkbox"/> Protected	<input checked="" type="checkbox"/> Static	<input type="checkbox"/> Inherited	

Icon	Member	Description
	<a href="#">UrgentMessage(Int32, String, Int32)</a>	Initializes a new instance of the <b>UrgentMessage</b> class by calling base constructor.
	<a href="#">Cost</a>	Gets the cost.
	<a href="#">ToString()</a>	Returns a <a href="#">String</a> that represents this instance. Urgent messages have ***'s appended at each end of message text. (Overrides <a href="#">Message.ToString()</a> .)

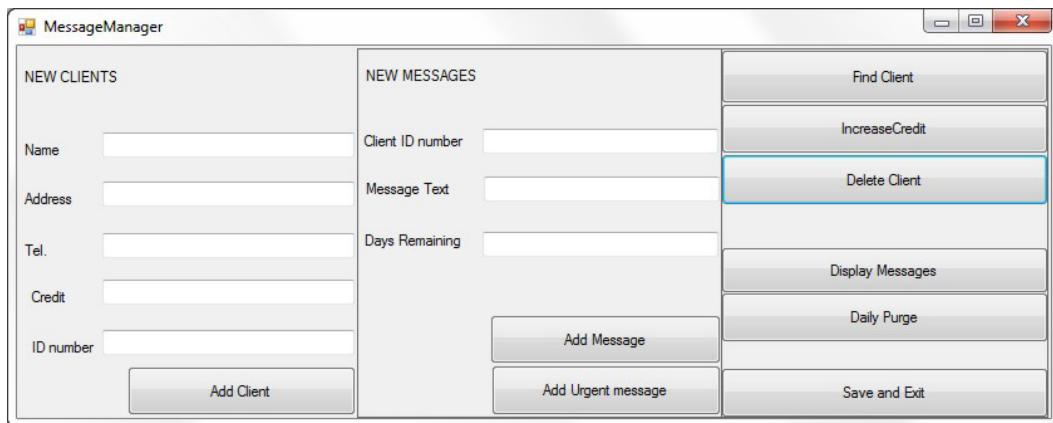
**Remarks**  
Author Simon Kendal Version 1.0 (5th May 2011)

**Inheritance Hierarchy**  
Object  
└ Message  
└ UrgentMessage

## 14.14 THE FINISHED SYSTEM

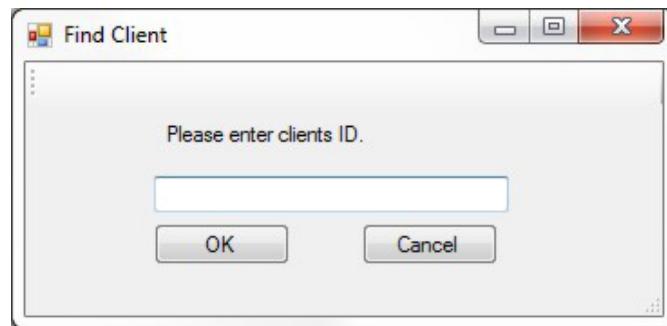
The following screen shots show the finished system.

Firstly the main interface window – this is very similar to the design. The only change was one extra button that was added to allow a message to be designated as an urgent message.

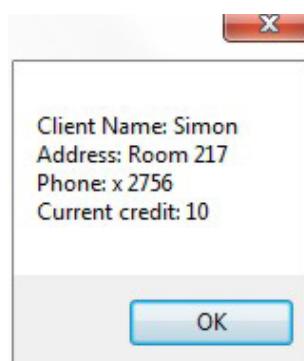


The next two images show the pop up dialogues that appear when the 'Find Client' button is pressed.

Firstly asking for a client ID...



Secondly displaying the client details – assuming a client with this ID has been added.



The ‘Display Messages’ button shows each of the messages on the screen using the DummyBoard class. This is only crudely simulating a real display board and makes no effort to scroll the messages or display them in any graphically interesting way.

Urgent messages look just like ordinary messages except \*\*\*’s are displayed before and after the message.

‘Purge Messages’ invokes the PurgeMessages() method. Mostly this does nothing visible but decrements the days remaining for each message, decreases the client’s credits and deletes the messages if appropriate. Urgent messages are charged at double the rate of ordinary messages. This can be tested by running Find Client before and after doing a daily purge – this should show the clients credit decreasing. If messages exist with an unrecognised client ID and exception will be generated. This exception will be caught by the PurgeMessages() method and an error message will be displayed on the screen.

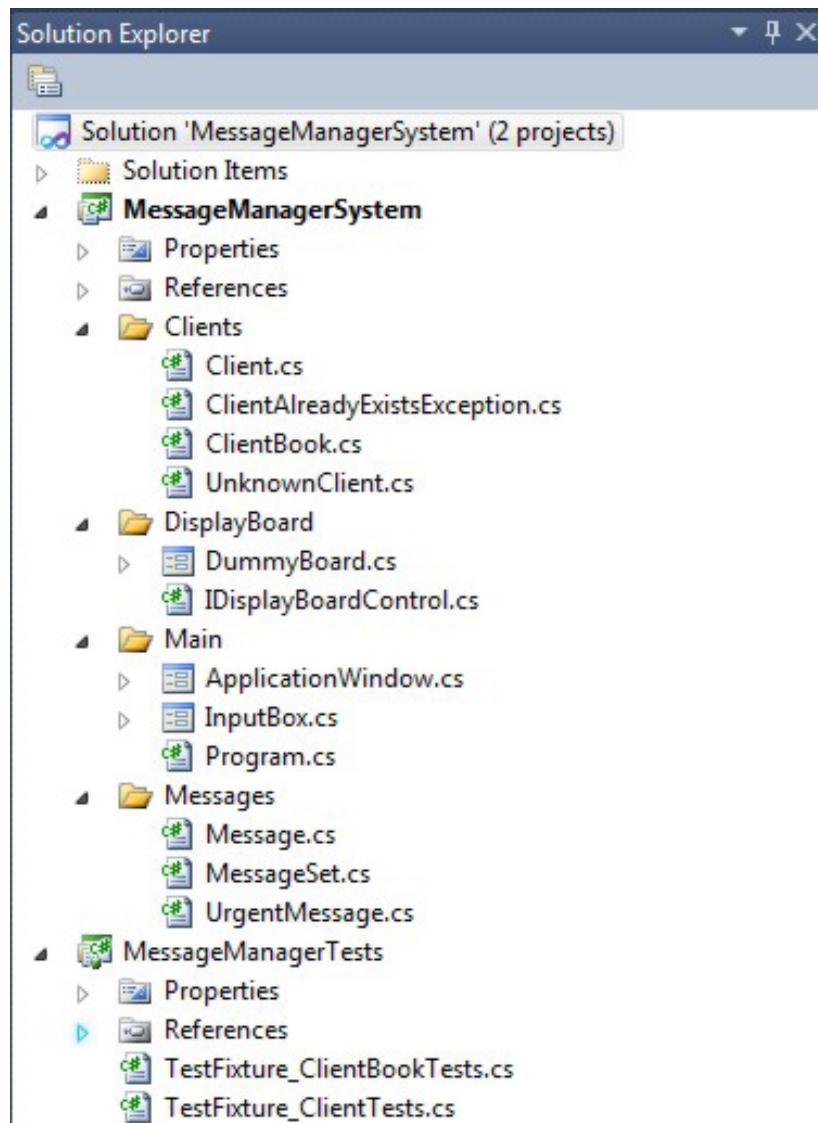
Ultimately of course the idea would be to get the MessageManagerSystem to display the messages on a real display board. This would involve 1) loading the Dynamic Link Library (DLL) for the real display board 2) creating an object of the real display board in place of the dummy display board 3) passing this object when calling the Display() method. i.e. only two lines of the entire MessageManagerSystem would need to be changed!

## 14.15 RUNNING THE SYSTEM

The complete, **fully commented**, source code for the Message Manager system, as described in this chapter, is available with this textbook as a zipped file. To view or run the Message Manager system:

- Install Microsoft Visual Studio. The community edition is free and will be perfectly adequate (<https://www.visualstudio.com/vs/community/>).
- Download and unzip the software available with this book.
- Load the MessageManagerSystem.sln file into Visual Studio, view the code and run within Visual Studio.

In the zip file downloaded are all classes, methods and test cases discussed in this chapter. When viewing the Solution Explorer in Visual Studio you will see all the packages, all of the classes and you should be able to view all of the code with the associated comments (see below).



Also inside this zip file is the automated documentation generated by the Sandcastle tool. To view the documentation go to the 'Documentation' folder and double click on the index.html page. This should load the documentation into your web browser software.

If you install Sandcastle Help File Builder (available for free from <https://github.com/EWSoftware/SHFB>) you will be able to load the file MessageManagerSystem.shfbpro available as part of the Message Manager system download. You will then be able to see that the comments for the namespaces have been added to the project properties and if you adjust the output path to an appropriate path for yourself you will be able to rerun this software and see the documentation generated for yourself.

## 14.16 CONCLUSIONS

The fundamental principles of the Object Oriented development paradigm are

- abstraction
- encapsulation
- generalization/specialization (inheritance)
- polymorphism

These principles are ubiquitous throughout the C# language and the .NET APIs as well as providing a framework for our own software development projects.

A well-established range of tools and reference support is available for Object Oriented development in C#, some of it allied to modern ‘agile’ development approaches.

Throughout this chapter you will hopefully have seen how Object Orientation supports the programmer by:

- using abstraction and encapsulation to enable us to focus on and program different parts of a complex system without worrying about ‘the whole’.
- using inheritance to ‘factor out’ common code
- using polymorphism to make programs easier to change
- using automatic tools to help document and test large software projects.

These principles have been exemplified here using C# but the same principles and benefits apply to all Object Oriented programming languages and the facilities demonstrated here are available in many modern IDE’s.

# 15 FINALLY...

In this book we have covered a lot. We started in Chapter 1 by introducing the concepts on which Object Orientation is based and we compared different software implementation methods.

In Chapter 2, we gave an overview of UML (class, package, object and sequence diagrams) and discussed how precise use of the notation was important.

In Chapter 3, we learnt how to use inheritance and override methods.

In Chapter 4, we considered how important polymorphism was and in Chapter 5 saw how to use polymorphism effectively.

In Chapter 6, we learnt how overloading methods can help create flexible software.

In Chapter 7 and 8, we considered how to design systems well. In particular: How to identify classes, how to refine our designs and how to apply the SOLID design principles.

In Chapter 9 onwards, we started to focus in implementation issues: How to use generic collections, how to override Equals() and GetHashCode(), how to serialise collections and how to use customised exceptions.

In Chapter 12, we discussed old software development lifecycles and compared these with modern agile approaches. We discussed: Scrum, XP, Paired Programming and Test Driven Development. We saw how important automated unit testing is and how to write tests: the arrange-act-assert cycle and how to name tests.

In Chapter 13, you had the chance to check your understanding of this theory and its application.

In Chapter 14, we looked at a case study used to demonstrate this theory in practise.

Throughout this book were numerous activates and feedback to help you. If you have worked through each of these chapters then you have hopefully developed both your understanding of the theory and your practical implmentation skills.

It is a testament to your tenacity and willingness to learn that you have made it this far.

*I hope this book has been useful for you!*

If you want a further explanation of the C# language the latest edition of the 'Pro C#' book by Andrew Troelsen is highly recommended.

Kind regards and best wishes for the future.

*Simon Kendal*

# PEER REVIEW COMMENTS

This book introduces the Object Oriented Programming (OOP) paradigm, combining a theoretical underpinning of the core concepts with practical software development using the C# programming language. Topics covered include UML modelling, design, development and testing methodologies alongside practical programming constructs such as inheritance, polymorphism and data persistence via generic collections and serialization.

Each chapter supports the material covered with a series of practical activities and feedback designed to give the reader a deeper understanding of the topic and practical experience of using the constructs introduced. A range of real-world scenarios help with understanding of the topics and code samples are provided, which can be adapted by the reader for their own software development. The book concludes with a fully-worked-through case study to further cement the knowledge gained.

This book would make an ideal core text for students who have a basic knowledge of procedural programming in C# and wish to expand their experience into OOP. By working through the chapters, completing the practical activities, they will reach the end with not only a theoretical understanding but also having gained practical development experience.

The second edition has been updated with additional content on unified modelling language (UML 2) diagrams, SOLID design principles and industry-adopted agile programming frameworks, making the book current and relevant to a modern software developer.

Miss Elizabeth A. Gandy MSc MBCS  
Programme Leader: BSc Computer Science/BSc  
Games Software Development, Faculty of Computer Science  
University of Sunderland

This 15-chapter book introduces key constructs of object oriented programming (OOP) and modelling, including design methods, development techniques, and test strategies. Beginning with an overview of how programming paradigms have advanced over time, key object oriented concepts (including abstraction, encapsulation, generalization, specialization, inheritance and polymorphism) are comprehensively introduced, theoretically grounded, and practically exemplified using the C# programming language. Each chapter incorporates a series of small exercises, with feedback and model solutions offered, thus enabling readers to practically evaluate their understanding of individual concepts introduced. Additionally, more realistic and comprehensive examples are presented which demonstrate the ability of OOP to enable modular, maintainable, and extensible system design and development. The text culminates in a comprehensive case study which brings together the OOP topics introduced, allowing the reader to integrate and reinforce their learning from previous chapters.

There is plenty of new material in this second edition. Relevant unified modelling language (UML 2) diagrams are introduced, SOLID design principles are presented and practically demonstrated, and industry-adopted agile programming frameworks discussed. The information presented within this book is current, coherently structured, and reinforced with sample code and illustrative content.

Assuming prior knowledge of basic programming concepts and their implementation in C#, this text presents object oriented design and programming in an accessible and progressive manner, therefore constituting a valuable resource for basic users wishing to advance their skills and explore the full potential of OOP.

Dr Kathy Clawson, FHEA  
Senior Lecturer in Software Development School of Computer Science  
Faculty of Computer Science, David Goldman Informatics Centre  
University of Sunderland