January 1, 2017

Chris Dworczyk

WLC

Master notes

OOAD

Contents

[Object-Oriented programming (VIDEO) 3](#_Toc500863422)

[OO Software Development Lifecycle (OO SDLC) 3](#_Toc500863423)

[System Analysis 3](#_Toc500863424)

[Object-Oriented view of the Analysis phase 3](#_Toc500863425)

[System Design 3](#_Toc500863426)

[Object-orietned view of the Design phase 3](#_Toc500863427)

[Aim of taking an object-oriented approach 4](#_Toc500863428)

[Benefits of Using object orientation 4](#_Toc500863429)

[Advantages of OO Methodology 4](#_Toc500863430)

[What is an Object? (VIDEO) 4](#_Toc500863431)

[What is a Class? (VIDEO) 5](#_Toc500863432)

[Summary 6](#_Toc500863433)

[Object-oriented 6](#_Toc500863434)

[Object 6](#_Toc500863435)

[What is Polymorphism (VIDEO) 7](#_Toc500863436)

[Use Case Diagram (VIDEO) 8](#_Toc500863437)

[How do we make objects? 8](#_Toc500863438)

[Createing a class 8](#_Toc500863439)

[4 pillars of oo – a pie 9](#_Toc500863440)

[Abstraction 9](#_Toc500863441)

[Encapsulation 10](#_Toc500863442)

[Inhertitance 10](#_Toc500863443)

[How to identify inheritance 11](#_Toc500863444)

[Class hierarchies 11](#_Toc500863445)

[Inheritance in c# 12](#_Toc500863446)

[Polymorphism 12](#_Toc500863447)

[Summary 12](#_Toc500863448)

[OO Design Process 13](#_Toc500863449)

[Gather requiermetns 13](#_Toc500863450)

[Analysis 13](#_Toc500863451)

[To formally gather requirements (FURPS/FURPS+) 13](#_Toc500863452)

[User focus 13](#_Toc500863453)

[User story 14](#_Toc500863454)

[Use case 14](#_Toc500863455)

[What details are captured 14](#_Toc500863456)

[How do we identify potential use cases? 15](#_Toc500863457)

[Identifying the actros 15](#_Toc500863458)

[Identifying the scenario 15](#_Toc500863459)

[Exmaple of a use case document 16](#_Toc500863460)

[remaining agile 16](#_Toc500863461)

[Diagraming use cases 16](#_Toc500863462)

[Linking use cases 17](#_Toc500863463)

[Summary 17](#_Toc500863464)

[Linking use case (Continued) 17](#_Toc500863465)

[Include 17](#_Toc500863466)

[Include example 18](#_Toc500863467)

[Extends 18](#_Toc500863468)

[Extend example 18](#_Toc500863469)

[Generalization 19](#_Toc500863470)

[Examples of Use Cases 19](#_Toc500863471)

[Hospital reception 19](#_Toc500863472)

[Alarm clock 20](#_Toc500863473)

[restaurant 20](#_Toc500863474)

Week 1

# Object-Oriented programming [(VIDEO)](../1.HND%20-%20OOAD/Slides%20and%20Handouts/Week%201%20Videos/01%2001a%20WhyWeUseObjectOrientation.mp4)

Before the 1980s programs were written in a straight procedural language which meant all the variables and logic were in one place.

This grew hard to maintain as programs got larger so, Object Oriented Programming (OOP) started to be born. How they worked on the other hand is that the program was split into smaller mini programs.

Each object contained their own data and logic and communicated between themselves.

Popular languages like C#, C++, Java, Perl, PHP, Python, Ruby (and many other) are all Object-Oriented programming languages.

## OO Software Development Lifecycle (OO SDLC)

### System Analysis

This is the first phase of any SDLC. In this the developers would interact with users to try and get a better understanding of what is required of them.

Analysis would prepare a model of desired system.

#### Object-Oriented view of the Analysis phase

The primary tasks are:

* Identifying objects (both data and functions)
* Organizing the objects into a model
* Defining the internals of the objects (attributes)
* Defining the behaviour of the objects (actions)
* Describing how he objects interact

***Outcome***: The analysis produces models on how the desired system should function and how it must be developed. The models do not include any implementation.

### System Design

In this stage overall architecture is decided for the system.

“The system is perceived as a set of interacting smaller sub systems that are in turn composed of a set of interacting objects”

#### Object-orietned view of the Design phase

OO Design includes two stages:

* Object Design
  + Object-oriented design is the discipline of defining the objects and their interactions
* System Design
  + complete architecture of the desired system is designed
  + the emphasis is on the objects comprising the system rather than the processes in the system.

***Outcome***: The design phase produces detailed models on how the desired system should function and how components should interact

### Aim of taking an object-oriented approach

Applying the object-oriented paradigm, we can visually model the solution throughout each development cycle to foster better stakeholder communication and product quality.

You will learn how to visually model your solution.

### Benefits of Using object orientation

* Data hiding(encapsulation)
* Reusability
  + Programmer can spend less time and effort and focus on other aspects of the system due to reusability
* Inheritance

Systems designed using this approach are closer to the real world.

### Advantages of OO Methodology

* Objects are immune to requirement changes therefore easier to make modifications. Normally just affects one object.
* Encourages more re-sue therefore reduce costs and lifecycle times.
* More natural approach and provides structures for thinking and abstracting and leads to a more modular design.

# What is an Object? [(VIDEO)](../1.HND%20-%20OOAD/Slides%20and%20Handouts/Week%201%20Videos/01%2002%20WhatIs_Object.mp4)

OOP was created to make thinking about programming closer to thinking about the real world.

Object is a thing. Is apple an object? Yes. A mug? Yes. We understand that objects are separate from one another. You might have 2 mugs, but they are different, they have their own identity.

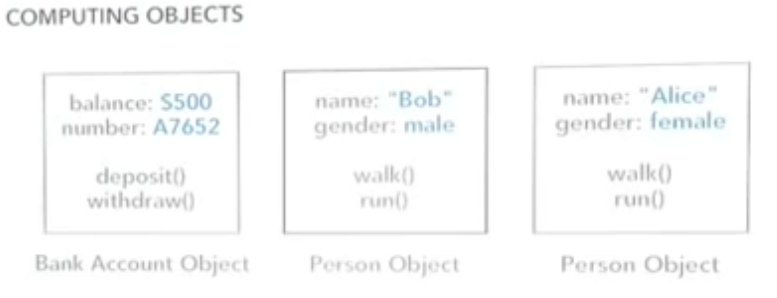
It’s nothing to do with complexity, an aircraft is an object and so I a pencil.

Object might contain other objects, they are still separate objects, we understand that.

Objects have characteristics. A mug might be full or empty, an apple might be green or red. Those are the attributes of the objects.

In real world object have behaviours, a telephone will ring a plane will fly. They’re specific to those objects, an apple couldn’t fly.

Those 3 things identity, attribute and behaviour are the same 3 things that describe an object in a OOP language.

Objects in the language are self-contained. They have identity separate for other object, they have their own attributes that describe their current state and behaviour which is what they can do.

In real world we tend to only use object for things we can see and touch but in computing we can take it further. Sure, in computing program we often have object that represent real world objects like car or house but also a date could be an object, a time, bank account. You can’t touch or feel a bank account in real life but we know want it is. In real life it meets our definition. It has identity, one bank account will be different to any other one. It has attributes (data) like balance, account holder name. It has behaviours like withdraw money, deposit, close account.

***Objects are not always physical item OR visible items.***

How to know if something in your application is an object. It’s easy if it’s something like a car, employee, document. But let’s say you’re building an event management application. What about something like **event**? Would **event** be an object? First clue, is the word a noun?

If you were talking about it in a conversation could you put “the” word in front of it?

the Mug, the Apple, the Television.

Sure! But also,

the bank account, the time, the date, the event.

Those work and could be objects.

You wouldn’t say the saving, the printing, the exploding. Those are verbs. They’re behaviours inside the objects.

# What is a Class? [(VIDEO)](../1.HND%20-%20OOAD/Slides%20and%20Handouts/Week%201%20Videos/01%2003%20WhatIs_Class.mp4)

Can’t talk about objects without talking about classes. We use classes to create objects.

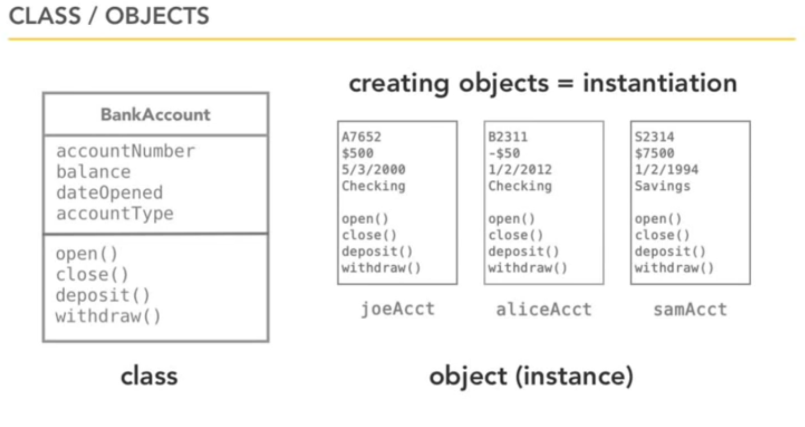
A class describe what the object will be BUT is not the actual object. It’s a blueprint, a detailed description and definition.

If you want to build a house you make a blueprint first. It describes exactly how the house will be and look like but is NOT the house. You then use the blueprint to create the house.

Same in programming we use the class to create the object. And just as we could use to build 1,2 or 100 houses. Same we can create 1 class and create 1,2 or 100 objects.

A class has a name(type): what is it? Employee, bank account, event, player, document.

Attributes(properties/data): what describes it? Width, height, colour, score, file, type, length

Behaviour(operations/method): what can it do? Play, Open, Search, Save, Print, Create, Delete, Close

There are a lot of existing classes ion OO languages to make it easier for you, so you won’t have to rewrite lots of the same code: strings, dates, collections, file I/O, networking and more!

These blocks of code are gathered together and put inside libraries/frameworks:

Java Class Library

.Net Framework BCL

C++ Standard Library

Ruby Standard Library

Python Standard Library

# Summary

### Object-oriented

Once the programmes written were too hard to maintain in a straight procedural language, OO programming was introduced to split it up into smaller sub programs. Each with its own data and logic, communicating with each other.

The benefits are that it lets us design it closer to the real world.

* Data hiding(encapsulation), Reusability, Inheritance.

### Object

3 things to describe an object:

* IDENTITY – have their own independent existence
* ATTRIBUTES – own characteristics that describe them
* BEHAVIOUR - things they can do

**But objects are not always physical or visible therefore difficult to identify objects.**

Ways to identify an object

* NOUN – is the word a noun?
* “the” – could you put the word “the” in front of it?

“An object is any person, place, thing, concept, event, screen, or report applicable to your system. Objects both know things (they have attributes) and they do things (they have methods). A class is a representation of an object and, in many ways, it is simply a template from which objects are created. Classes form the main building blocks of an object-oriented application.  Although thousands of students attend the university, you would only model one class, called *Student*, which would represent the entire collection of students”

Week 2

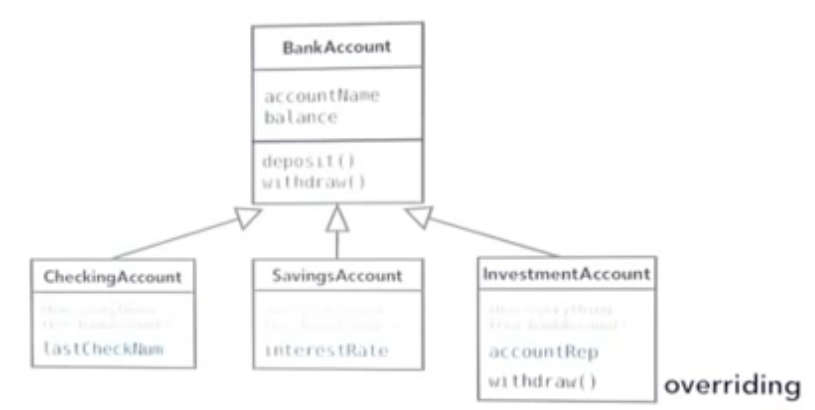
# What is Polymorphism [(VIDEO)](../1.HND%20-%20OOAD/Slides%20and%20Handouts/Week%202%20Videos/01%2007%20WhatIs_Polymorphism.mp4)

“many forms”

It lets us automatically do the correct behaviour even though what we’re working with can take one of many different forms.

The plus sign (+) in a lot of languages this means that 2 variable of integer type will be numerically added, simple! But, if the variables are strings then they will automatically be concatenated. It will automatically do the correct behaviour -but a different behaviour- even though the thing we gave it could have one of many forms.

Example:

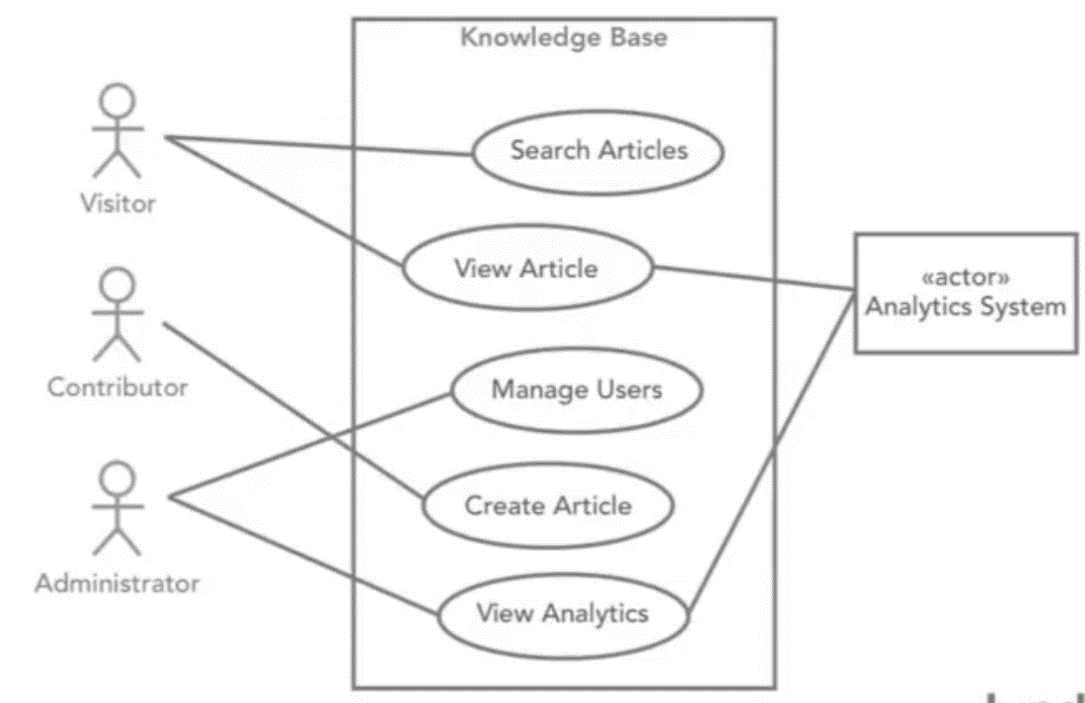
Define a bank account class with attributes like account name, balance. A Withdrawal method and a deposit method. We can then create more specialized sub classes that inherit from the main class like a savings account, checking account or an investment account. They all inherit from the parent so they all share basic definitions like an account name, balance, they can withdraw and deposit, but a savings account may add a savings interest, something the checking account doesn’t have.

Moreover, let’s say that the client said the investment account will take a fee when someone doesn’t give 30 days’ notice to withdraw money. The original withdraw behaviour was defined in the bank account class so its already getting inherited, but it can be specialized by the sub classes called overriding. It just means you write it again.

***You can inherit the behaviour when useful but also override it in other cases!***

Polymorphism lets us work freely from objects created from any of these classes. So, I can have an array of account with 10,000 of these objects loaded into it and I know that if I call the withdraw method on any of them without knowing exactly what class it was initiated from and it would do the correct behaviour form each of them just like the addition sigh would know when to switch between addition and concatenation. Its flexible, polymorphism lets us do the right thing at the right time.

# Use Case Diagram [(VIDEO)](../1.HND%20-%20OOAD/Slides%20and%20Handouts/Week%202%20Videos/03%2004%20DiagrammingUseCases.mp4)

Use case diagram is another diagram that comes from UML.

It sounds like a diagram of a use case. It isn’t,

It’s almost always a diagram of several use cases and multiple actors at the same time and the reason it exists is, so we can get an overview of these and see how they interact all in context.

It’s not the same thing as the use cases in programming.

We have this, so we can see a different perspective.

# How do we make objects?

## Createing a class

* Type
  + Name: what is it?
    - Employee, bank account, event, player, document, album
* Properties data
  + Attributes: what describes it?
    - Width. Height, colour, score, file type, length
  + Behaviour: what can it do?
    - Play, open, search, save, print, create, delete, close

**Example: Bank Account Class**

name: BankAccount

attributes: accountNumber, balance, dateOpened, accountType

behaviour: open, close, deposit, withdraw

Information represented in a CLASS DIAGRAM (UML)

|  |
| --- |
| BankAccount |
| accountNumber  balance  dateOpened accountType |
| open()  close()  deposit()  withdraw() |

|  |
| --- |
| ChrisAccount |
| A7652  £1000  3/5/2010  current |
| open()  close()  deposit()  withdraw() |

## 4 pillars of oo – a pie

We need to understand the 4 principles of OO to help us understand how best to create out classes for our solution.

**Remember:** although we should always think about the 4 principles of OO when designing our solution, we cannot address all of them on the 1st iteration.

**We just need enough to let us move forward successfully!**

 “A PIE”

* Abstraction
* Polymorphism
* Inheritance
* Encapsulation

## Abstraction

An idea/concept that is separate from any specific instance

* You understand the idea of a table without having to say its wooden or glass etc.

We use abstraction to gather the general characteristics and to filter out the details we do not need to solve our problem.

Focus on the essentials

* Ignore the irrelevant
* Ignore the unimportant

**Why abstraction**? - Reduces complexity and increases efficiency

We noted that all cats have general characteristics, which are common to all cats like eyes, a tail, fur, a liking for fish and the ability to make meowing sounds. In addition, each cat has **specific characteristics**, such as **black** fur, a **long** tail, **green** eyes, a love of **salmon**, and a **loud** meow. **These details are known as specifics.**

To draw a basic cat, we **do** need to know that it has a tail, fur and eyes. These characteristics are relevant. We **don't** need to know what sound a cat makes or that it likes fish. These characteristics are irrelevant and can be filtered out.

From the general characteristics we have (tail, fur, eyes) we can build a basic idea of a cat, ie what a cat basically looks like. Once we know what a cat looks like we can describe how to draw a basic cat.

## Encapsulation

**Think capsule…idea of surrounding something to keep contents together and to protect the contents.**

**Keep contents together** – taking attributes and behaviours and bundling them together.

**Protect the contents** – an object should not reveal anything about itself other than that what is necessary for other parts of the application to work. This is known as information/data hiding.

Close off data and inner-workings except for what we choose to make public, normally for input and output.

How much should we hide? – As much as possible!

**Why encapsulation?** – It’s about reducing dependencies between different parts of the application

## Inhertitance

Inheritance allows a software developer to derive a new class from an existing one:

Existing class = parent, super, or base class

Derived class = child or subclass

The child class inherits characteristics of the parent.

* All attributes and method defined for the parent class
* Any changes to the parent filter down to the child class

The child can also have its own unique behaviours and data

**Why inheritance?**

* Code re-use
* Easy modification of model – only change things in one place
* Avoid redundancy leading to smaller, more efficient, easier to understand models.

**Example**

|  |
| --- |
| Person |
| name  email  phone |
| changeEmail () |

**Parent class**

**(super class / base class)**

Don’t want to add customerNumber to Person class as trying to use **abstraction** – focus on the essentials as not all Person objects will be a Customer

|  |
| --- |
| Customer |
| name  email  phone  customerNumber |
| changeEmail() |

**Child class**

**(sub class / derived class)**

**Multiple inheritance used by C++, single inheritance used by C#**

### How to identify inheritance

Inheritance should create an “**IS A…**” relationship

A car IS A vehicle – Yes, inheritance

A bus IS A vehicle – Yes, inheritance

A car IS A bus – No, not inheritance

**Could also say “is a kind of…” or “is a type of…”**

### Class hierarchies

A child class of one parent can be the parent of another child, forming a class hierarchy

Mammal

Mammal

Mammal

Mammal

Mammal

Mammal

Mammal

Mammal

Mammal

### Inheritance in c#

All classes in C# are derived from the object class. If a class is not explicitly defined to be the child of an existing class, it is a direct descendant of the object class

The object class is therefore the ultimate root of all class hierarchies.

|  |
| --- |
| The object class defines methods that will be charged by all objects in C# |
| ToString: converts an object to a string representation  Equals: checks if two objects are the same  GetType: returns the type of a type of object |

A class can override a method defined in Object to have a different behaviour eg:

String class overrides the equals method to compare the content of two strings.

## Polymorphism

Means many forms

A child can override the definition of an inherited method in favour of its own. That is, a child can redefine the method that it inherits from its parents.

The new method must have the same signature as the parent’s method (method name/return type) but can have a different implementation.

Same object has different implementation and it is the type of the object executing the method that determines run time not during compilation and is known as dynamic binding,

# Summary

We model solutions based on interacting objects

The classes created are merely templates from which the object(s) is instantiated from.

**“A PIE”**

* Abstraction – attributes/methods that are separate from any specific instance
  + Generalisation – focus on the essentials
* Encapsulation – keep contents together and protect them
  + Only reveal what is necessary for other parts of the application to work
* Inheritance - derive a new class from an existing one
  + Code re-use, avoid redundancy, easy to modify
* Polymorphism - A child class can override the definition of an inherited method in favour of its own

Week 3

# OO Design Process

We need to go through a process to identify the classes/objects for our application:

1. Gather requirements
2. Describe the application
3. Identify the main objects
4. Describe the interactions
5. Create a class diagram

## Gather requiermetns

### Analysis

Functional – what is required to do i.e. features/capabilities

Non-functional – what else? i.e. security, performance, documentation

### To formally gather requirements (FURPS/FURPS+)

Functional requirements,

Usability requirements

Reliability requirements

Performance requirements

Supportability requirements

+ design requirements, implementation requirements, interface requirements, physical requirements

N.B. agile approach – first pass, grab minimal set of requirements to feed into next part of the process. Some requirements will be TBD and will be defined later

## User focus

Now we have our requirements i.e. “application must do…”

We need to focus now on the **USER –** how does the user accomplish a goal?

2 formats typically used for this purpose

* User story
* Use case

### User story

Describes a single goal from the user’s perspective

* What do they want to do and why?
* 1 or 2 sentences typically written on index cards
* Hollow a format:

As a (type of user) I want (goal) so that (reason – this part is optional)

e.g.

As a bank customer I want to change my PIN online so that I don’t have to go into a branch.

This would be done at the start of the project as a placeholder for future conversations and as a reminder that we need to discuss.

Always used as input for next stage in process

### Use case

A use case is a record of a conversation that has happened and details steps for a goal

Full format process would use “Use Case”

|  |  |
| --- | --- |
| User Stories - Short | Use Cases - Long |
| 1 index card one goal, no details | A document with multiple goals and details |
| Informal “placeholder” | Casual to formal “record” |

Always used as input for next stage in process

#### What details are captured

Title: **what is the goal**

Short phrase with active verb e.g. register new member, transfer funds

Actor: **who desires it?**

Who is having this interaction? E.g. customer, member, ACME system (non-human)

Scenario: **how is it accomplished?**

As a paragraph, details of accomplishing a goal step-by-step. Could write a numbered list of steps BUT remember this is NOT pseudocode!

*Extensions* – for alternative flows i.e. when situation does not follow “normal” e.g. item out of stock

*Precondition* – add a precondition to the scenario e.g. must have minimum of 1 item in shopping list cart

*Post conditions, triggers, stakeholders*

#### How do we identify potential use cases?

Ask your stakeholders the following questions from the point of view of the actors:

1. What are users in this role trying to accomplish?
2. To fulfil this role what do users need to be able to do?
3. What are the main tasks of users in this role?
4. What information do users in this role need to examine, create or change?
5. What do users in this role need to be informed of by the system?
6. What do users in this role need to inform the system about?

Constantine L.L. & Lockwood L.A.D. (1999). *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. 1st ed. New York: Addison-Wesley

#### Identifying the actros

Actor – anything with behaviour that lives outside of the application but has a goal they wish to accomplish.

Dose your application need to interact with any external system or organisation?

For human actors – do you need to distinguish between roles/security groups? (visitor, member, admin, owner)

Think about job titles, departments e.g. data entry staff security team.

#### Identifying the scenario

A goal an actor can accomplish in a single encounter (may be several steps) e.g. checking out shopping is part of an overall goal and not a use case in its own right

Use cases can have multiple scenarios

Normal “sunny day” scenario

Extensions – alternative paths. Don’t think about bizarre events just typical events that may occur.

Choose readability and ease of creation over formality



System connects to external payment process over HTTPS and uses JSON to submit the payment



system validates payment information

#### Exmaple of a use case document

## remaining agile

It is very easy for use case modelling to become un-agile. To prevent this from happening we need to focus on creating artefacts that are just barely good enough, they don’t need to be perfect.

Only need enough to move forward!

Strongly written use case with a few mistakes is FAR better than an over complicated detailed list that confuses and bores the audience.

## Diagraming use cases

* Stick figures to represent an actor
* Ellipses around use case titles
* Rectangles around all use cases to show system boundary (can also be useful to use boxes to denote releases)
* Draw lines (association) between actors and use cases they interact with
  1. If actor supplies information, initiates the use case, or receives information because of a use case then we need an association
  2. N.B. this dose not represent flow of information therefore no arrow heads (information flow can be modelled using UML activity diagrams)
  3. Not all actors are human some may be an external system. This should be represented by a box (rather than a stick figure), but include the notation <<actor>> at the top of the box

## Linking use cases

* Association relationship
* Generalization relationships
  + One element (child) “is based on” another element (parent)
* Include relationships
  + One use case (base) includes the functionality of another (inclusion case)
* Extend relationship
  + One use case (extension) extends the behaviour of another (base)

# Summary

* What is the overall aim of using use cases?
  + To eliminate rework due to requirements misunderstandings
  + Use cases in conjunction with techniques like storyboarding, help to build explicit shared understanding between the clients and the development team

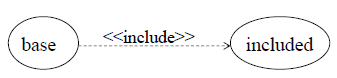
Week 4

# Linking use case (Continued)

## Include

If you have a piece of behaviour that is similar across many use cases break this out as a separate use-case and let the other ones “include” it.

Included use case never stand alone!



<<include>> means always included. The association is a dashed line annotated with <<include>>, pointing towards the use case being included

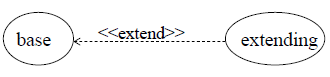
### Include example

In this case we have a use case (book assessment and book lesson) in which both include the check availability use case.

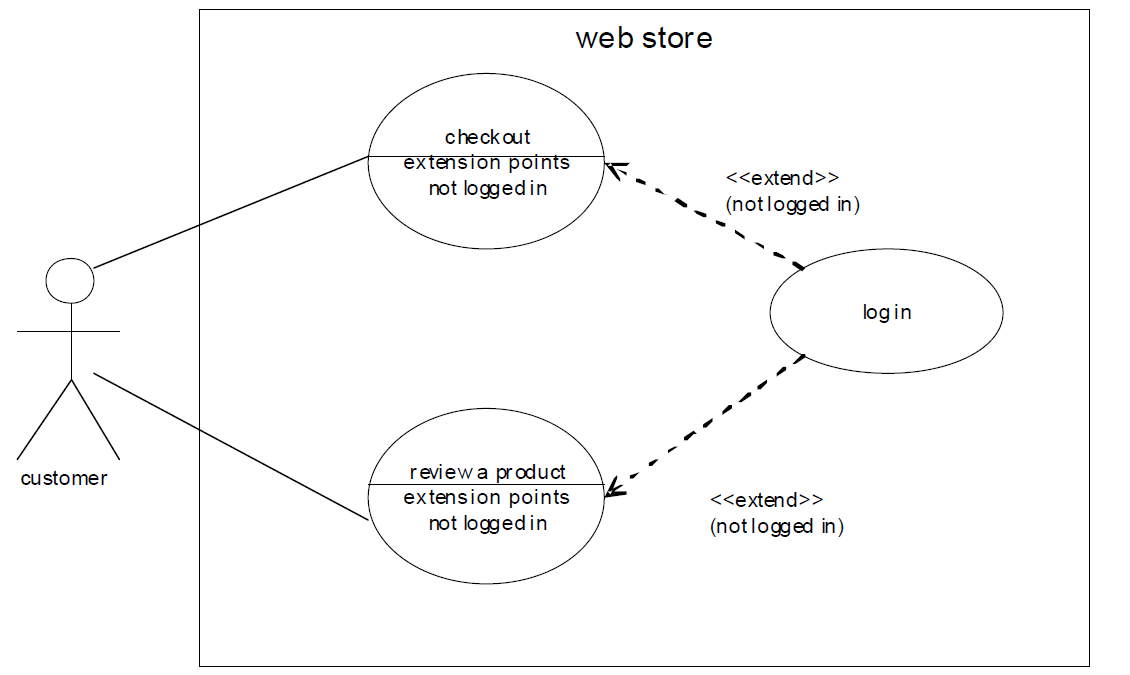
## Extends

This use case is like previous one but dose a little bit more.

The base use case may stand alone, but under certain conditions its behaviour may be extended by the behaviour of another use case.

<<extend>> means conditionally included. The association is a dashed line annotated with <<extend>> , pointing towards the use case being extended (and not the extending use case)

### Extend example

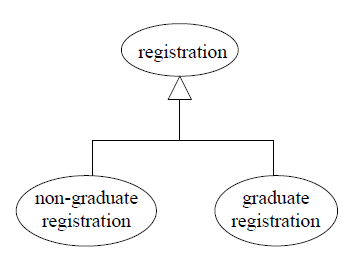
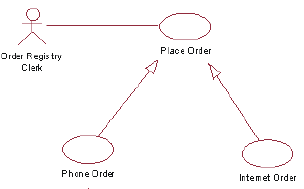


## Generalization

The child use case inherits the behaviour and meaning of the parent use case.

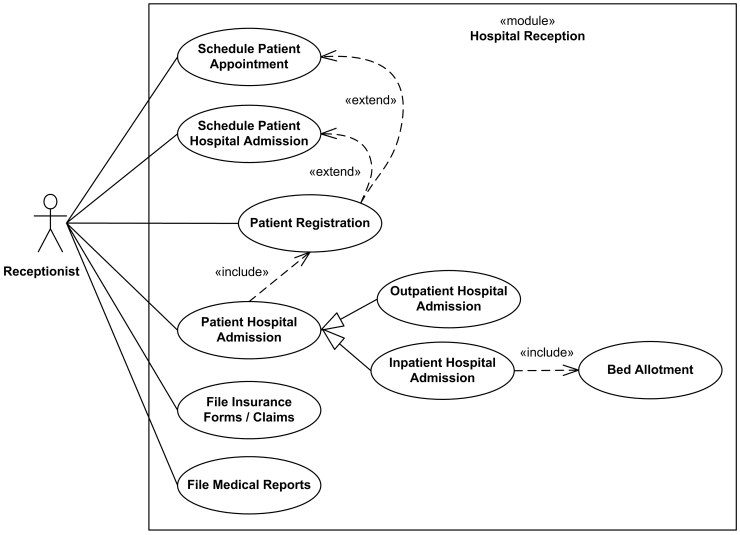
The child may add to or override the behaviour of its parent.

Generalisation shows the child inheriting the behaviour of the parent. The association is a solid line with an empty arrow head pointing towards the parent.

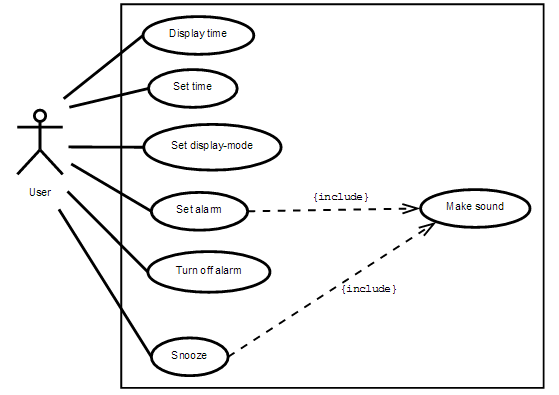


## Examples of Use Cases

### Hospital reception



### Alarm clock



### restaurant

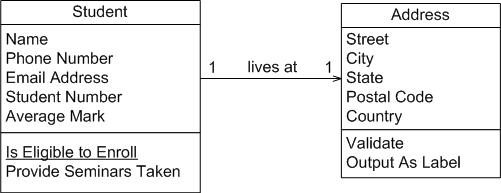
Week 5

# Class Diagrams

|  |
| --- |
| NAME |
| ATTRIBUTES |
| METHODS |

Classes are typically modelled as rectangles with threw sections: the top section for the name of the class, the middle section for the attributes of the class and the bottom section for methods of the class.

## Appropriate level of detail



Addresses are complicated things. They have complex data, containing street and city information for example, and they potentially have behaviour.

Notice how the address class has been modelled to include an attribute for each piece of data it comprises, and two methods have been added: one to verify it is a valid address and one to output it as a label (perhaps for an envelope). By introducing the address class, the student class has become more cohesive. It no longer contains logic (such as validation) that is pertinent to addresses.

The address class could now be reused in other places.

## Associations

To depict the direction in which the label should be read we use a filled triangle called a direction indicator. An example is shown on the offering of association between the seminar and course classes of diagram bellow. This symbol indicates the association should be read “a seminar is an offering of a course” instead of “a course is an offering of a seminar”. Direction indicators should be used whenever it isn’t clear which way a label should be read. My advice however is if your label is not clear then you should consider rewording it.