ECS414
Object Oriented Programming

Object Oriented Analysis and Design

Matthew Huntbach
matthew.huntbach@qmul.ac.uk

Coming material

- Large scale software
- Specification, representation and implementation
- Analysis and modelling
- Design Principles
- Clean Code
- Defensive Programming

Large Scale Software

- Commercially developed software is on a hugely greater scale than student exercise software
- Commercially developed software consists of many parts fitted together to make a whole
- Commercially developed software will be written by teams of developers
- Commercially developed software is used by people who are not software experts
- There can be serious consequences if commercially developed software does not perform correctly
- Commercially developed software will often need to be modified to meet new requirements

Managing Large Scale Software

- To manage large scale software we need to be able to break it into smaller parts which can be considered individually
- We need to be able to put parts together in a way that does not cause unexpected interactions
- If a system needs changing, to meet new requirements or because it is performing incorrectly, we need to be able to identify where the changes should be made
- Our software needs to be structured in a way that a small change in requirements does not lead to a big change in the software
- We need to be sure that a small change in one part of the software does not lead to unexpected consequences elsewhere

Software Engineering

- “Software Engineering” is about how to produce large scale software, it will be covered in detail in the second year here
- There are many aspects of producing programs that come under software engineering
- Some aspects of software engineering are to do with actual aspects of code, as we cover here
- Other aspects of software engineering are not directly about code, for example interaction with a customer to find the overall requirements of a program, and the human management of a team of programmers
- We will mention here just briefly some aspects that will be covered in more detail in the software engineering module, in particular design principles as the general model for well structured code

Abstraction - again

- Abstraction means cutting out all but the essential details
- Abstraction is essential when dealing with code on any level beyond the most simple introductory programming
- We need to write code so that it can be seen in terms of individual parts each of which can be viewed and understood in isolation
- We need to be able to use other code, knowing about it only in an abstract way: how it interacts with the code we are writing, not what it does underneath to cause that interaction
- The class and object concept if used properly helps provide that abstract way of viewing code

Client and Contractor

- Another way of describing the relationship between different parts of code is to use the idea of “client” and “contractor”
- In this analogy, the specification for a module (class or method) is considered to be a “contract” agreed between the “client” code which needs some service to be performed, and the “contractor” code which agrees to perform that service according to the conditions given in the contract
- This idea was developed by Bertrand Meyer through the programming language Eiffel (`http://www.eiffel.com`) which pioneered many features later seen in Java
- The term “Design by Contract” is used for program design in this way (note, Meyer has trademarked it)

Object Oriented Programming

- The object-oriented approach to programming has been widely adopted because it has proved a successful way of managing the complexity of large scale software
- It enables us to see program execution in terms of objects interacting rather than a single computer obeying commands to manipulate a single memory
- Sometimes it helps to “anthropomorphise” objects – think of them as agents which “know” things, have “responsibilities”, make “requests”, and so on
- We can hide complexity by establishing a clear distinction between what an object does (its public methods) and how it does it (the code inside its class)
- We can further hide complexity by referring to objects through variables of interface types which show only the capacity in which the object is being used in the code which uses it

Specification

- Before writing code we should have a clear idea of what that code will do in terms of its public interaction with other code
- The specification takes the form of a formal statement about the behaviour of its public methods
- Some experts advocate specification by examples: writing test code for classes before writing the code that implements the classes
- All observable behaviour of an object should fit with its specification
- Specification of objects may be applied to interface types: the observable behaviour of an object accessed through an interface type is that given by its methods which are in the interface type
- A specification should say no more than is necessary for the uses required of an object

Methods

- Specification for a method should say what it returns in terms of the arguments it takes and (unless the method is `static`) the state of the object it is called on
- The state of an object here is as it is viewed in the application code, not its implementation
- Method calls may change the state of the object they are called on, if so the change of state should be specified
- Method calls may change the state of objects passed to them as arguments, if so the change of state should be specified
- Method calls may have an input/output effect, if so that should be specified
- Methods should have a clear single purpose, so be cautious about specifications which involve both state changes and return values

Representation and Method Implementation

- A class will have a representation of the abstract state it is considered to have when used in application code
- Method calls may result in this internal representation being manipulated to represent change of state as viewed in application code
- Java allows methods in a class to manipulate the internal representation not just of the objects they are called on but also of other objects of the same class passed as method arguments
- Method implementations should never cause a change of abstract state unless that is part of their specification
- It should not be possible for code in other classes to access and change the representation except through calling public methods, an object should otherwise be in control of its own internal state

Exposing the Rep

- "Exposing the rep" means any way in which the internal representation could get manipulated directly by outside code
- Not making field variables `private` is an obvious way to expose the rep
- Making field variables `protected` means code in subclasses can access them directly, this means overridden methods could expose the rep, so be cautious about it
- Do not add "getters" and "setters" unless they are part of the specification, as they can expose the rep
- Avoid constructors, getters and setters which would result in internal variables being aliased by external variables, this is a dangerous way of exposing the rep as it is less obvious, you may mistakenly believe that declaring a field as `private` is sufficient protection, but it is not if that field is aliased

Example: A Set of integers

- Suppose we wanted a class which implemented the abstract concept of a set of integers, call it `IntSet`
- The abstract concept of sets is covered in your logic and discrete structured module
- A set either contains a value, or does not contain a value: it does not have the concept of a value occurring multiple numbers of times or of it being stored in a particular position
- To define a set, we might want to define its basic operations:
  - ✧ `add` – adds a new element to a set
  - ✧ `remove` – removes an element from a set
  - ✧ `contains` – says whether a set contains a particular element
  - ✧ `size` – gives the number of elements in a set
- Then decide do we want our set class to be mutable (`add` and `remove` change the object they are called on) or immutable (`add` and `remove` return new objects)

Mutable and Immutable

- If a set is mutable, the `add` and `remove` operations change the set itself, so the method return type should be `void`, for immutable sets the method would return the new set, so its return type would be `IntSet`
- What if you want to remove an element which is not in the set, or add an element which is already there?
- One possibility is to throw an exception
- For a mutable set, another possibility is to leave the set unchanged
- We could have a return type `boolean`, returns `true` if the set state is changed, `false` otherwise
- For an immutable set we could return a copy of the set
- Or we could return an alias to the same set

Representation

- We might decide to represent an `IntSet` by an array of the same size
- The value of the `IntSet` seen as a set is the abstract state, the array is its internal representation
- More sophisticated implementations (for example, using an ordered binary tree) could be covered under the subject "algorithms and data structures"
- So long as our internal representation is `private`, and `IntSet` objects can only be accessed through their public methods, we can write code which uses `IntSet` objects (application code) and the code would still work exactly the same if the implementation code was changed to a more sophisticated implementation – the difference would be just one of efficiency

Design Decisions (1)

- The decision on how to handle adding an element to a set when it is already present or removing it when it is not present is a decision on its external behaviour
- A good specification should say what the external behaviour should be under all possible circumstances, thinking through all the possible special circumstances is an important part of software design
- It is often the case, however, that special circumstances are only identified when the code is written
- In this case, the person implementing the code should document the decision made and query with the person who wrote the specification whether that was the best way to handle it
- In some cases it may be part of a method's specification that it only has defined behaviour under some circumstances, this is called a "partial specification"

Design Decisions (2)

- The decision on how to represent a set internally is not a direct decision on its external behaviour, the logical behaviour of the set operations in terms of how the code that uses set objects should not depend on their internal representation
- The person writing the code which implements the specification is free to choose what the internal representation should be
- Implementations which are efficient in time and space use should always be chosen – this is covered in more detail in algorithms and data structures modules
- In some cases one representation is efficient for some purposes another is efficient for other purposes, so the person specifying the code should be consulted
- There is often a “trade-off” between time efficiency and memory efficiency, so one implementation could be used for applications where speed is important, another for applications where memory use is important

Implementation (1)

```
class IntSet
{
 private int[] array;

 public IntSet()
 {
  array = new int[0];
 }

 public IntSet(int[] a)
 {
  array = a;
 }
…
```

Implementation (2)

```
…
 public boolean contains(int n) {
  for(int i=0; i<array.length; i++)
     if(array[i]==n)
        return true;
  return false;
 }

 public IntSet add(int n) {
  if(this.contains(n)) return this;
  int[] array1 = new int[array.length+1];
  for(int i=0; i<array.length; i++)
     array1[i]=array[i];
  array1[array.length]=n;
  return new IntSet(array1);
 }
…
```

What's wrong?

Exposing the Rep: Example

- We have exposed the rep!
- Consider:

```
int[] collection;
int p;
…
IntSet set = new IntSet(collection);
collection[p]=42;
```

- The assignment changes the array inside the `IntSet` object, meaning it changes the set it is meant to represent
- This is the sort of unexpected and unspecified interaction we want to avoid
- We can avoid it by making the second constructor for `IntSet` copy its `int[]` argument, or making it a `private` constructor for internal use only in the `add` and `remove` methods

Efficiency (1)

- What else was wrong? Consider:

```
public IntSet remove(int n) {
 if(this.contains(n)) {
     int[] array1 = new int[array.length-1];
     int i=0;
     for(; array[i]!=n; i++)
        array1[i]=array[i];
     for(;i<array1.length; i++)
        array1[i]=array[i+1];
     return new IntSet(array1);
    }
 else
   return this;
}
```

- It is inefficient: the call of `contains` goes through the array checking elements against `n`, and then this is done again

Efficiency (2)

- Another version:

```
public IntSet remove(int n) {
 if(array.length==0)
     return this;
 int[] array1 = new int[array.length-1];
 int i=0;
 for(; array[i]!=n&&i<array1.length; i++)
    array1[i]=array[i];
 if(array[i]!=n)
    return this;
 for(i++; i<array.length; i++)
    array1[i-1]=array[i];
 return new IntSet(array1);
}
```

- Another way to improve the efficiency would be to store the integers in numerical order, then the more efficient binary search could be used to check whether an integer is in the array, this would require a version of the method add that keeps the integers stored in order

Representation Invariant (1)

- The requirement that the array for `IntSet` is stored in numerical order is an example of a “representation invariant”
- A representation invariant is a condition on the data structure inside an object which must hold for it to be a valid representation
- Any methods that modify the internal representation must make sure it keeps to the representation invariant requirement
- When new objects are created, their data structure must keep to the representation invariant

Representation Invariant (2)

- Our original version of the code for `IntSet` also had a representation invariant: that no integer occurs more than once in the array
- Although it was not stated, we just assumed it when we wrote the code
- The `add` and `remove` methods keep the representation invariant in the new `IntSet` object they return
- The public constructor which creates an empty `IntSet` object keeps to the representation invariant, there can be no duplicates in an array of length 0
- However, the public constructor which takes an array and puts it as the array inside an `IntSet` object not only exposes the representation, it also leads to the possibility of the representation invariant being broken, because it does not check whether the array contains duplicates
- This would mean the method `remove` would not work correctly, because it is coded on the basis an integer occurs at most once in the array
- The lesson from this is to think carefully about your representation before you start coding

Immutability and Aliasing

- In the version of `IntSet` just given, if `s1` and `s2` are variables of type `IntSet` and `n` is a variable of type `int`, the call:

```
s2 = s1.add(n)
```

  means `s2` may or may not be an alias to `s1`
- That is not a problem, as there is no method that can be called on an `IntSet` that changes the actual object it refers to (i.e. `IntSet` is immutable)
- So, if `s3` is another variable of type `IntSet` and `m` another variable of type `int`, the call:

```
s3 = s1.remove(m)
```

  would not do anything to what `s2` refers to
- However, if `s1.remove(m)` worked in a way that changed the actual object that `s1` refers to rather than creating a new one that represents the change, there would be a problem, as then what `s2` refers to would also get `m` removed if `n` already occurred in what `s1` referred to before the call `s1.add(n)` was made

Mutability

- Here is a version of `add` in `IntSet` that does work by changing the actual object it is called on:

```
public boolean add(int n) {
   if(this.contains(n))
      return true;
   int[] array1 = new int[array.length+1];
   for(int i=0; i<array.length; i++)
      array1[i]=array[i];
   array1[array.length]=n;
   array = array1;
   return true;
}
```

- With this, a call `s1.add(n)` returns a `boolean` saying whether `s1` has been changed
- If `s2=s1` had been done previously, then `s1.add(n)` would change what `s2` refers to, as they would be the same object

Why not just use arrays?

If instead of defining class `IntSet` we just used an array of integers (`int[]`) every time we needed a set-like collection

- We would have to keep writing similar code to give the changes to the arrays that represent set-like behaviour
- We would not be able to see easily which array is meant to represent a set and which is meant to represent something else
- Our code would be a huge mass of for-loops
- It would be a lot of work if we decided to change from arrays to another representation
- We would not be able to guarantee that an array that is meant to represent a set will be manipulated as a set in other methods it is passed to
- We could not add new set operations (maybe union and intersection) and have them immediately available to other code

Analysis and Modelling

Analysis

- The first stage in constructing a large scale software system is to determine the requirements
- The software has a purpose, it is being constructed to meet the needs of a particular customer, or to be marketed to potential customers – the “user”
- Customers do not see it in terms of code, they see it as an application they can run
- The design process starts from an analysis of the operations the user wants from the point of view of the user
- The user interacts with the system through a Human-Computer-Interface, which is not the same as a Java interface
- Design of the HCI aspects of a system is a huge and separate topic, which we are not covering in this module

Model-View-Controller

- The Model-View-Controller pattern is generally used for modern software systems, it means keeping a clear separation between those parts of a system which interact with a human user, and those parts which store and manipulate the information underlying the system's services
- Model – does the computational work of the system, often involves accessing and manipulating a database
- View – the part of the system which interacts with the human user, provides some sort of display which the user can manipulate to send instructions to the system
- Controller – links the View with the Model, translates instructions given by the user in terms of clicks, text typing, and so on into calls of methods on code in the Model, translates values returned by these calls into display on the system's human interface

Graphical User Interface Programming

- The object-oriented approach works well for those aspects of coding to do with constructing the human interface
- The components the user sees on the screen translate well to objects in object-oriented terms
- The objects which directly represent the visible components “hide” the complexity of the calculations needed to turn the idea of the components as the human user sees them into representation on the screen
- By “hide” is meant the objects which represent the screen components have methods which represent the human interaction with these components, they will call other code which deals with the underlying technical issues of screen presentation
- So programmers dealing directly with HCI issues do not need to be concerned with the technical details and exact calculations needed for the display, they can rely on the library code they are using calling the code which handles that

Model

- This is about design of the “Model”, as suggested previously the first step is to establish a distinction of this from user interface issues
- The term “Model” suggests that this is a representation of something else, like a “model car” or a “model village”
- The originator of the term used it in this way, he wrote “There should be a one-to-one correspondence between the model and its parts on the one hand, and the represented world as perceived by the owner of the model on the other hand”
  `http://heim.ifi.uio.no/~trygver/`
- This fits in with the origin of object oriented programming as “simulation”
- The idea is that the computer system represents some aspects of the “real world”, and its users use it to draw conclusions from those aspects
- More correctly, the computer model represents the mental model of the user, which is a model of the real world, or in some applications an imagined world

Programming as Modelling

- In this way of thinking, programming is about building a model of a world
- We need to identify the entities in this world
- We need to decide what aspects of those entities are necessary for the purpose of our model, and what can be left out
- Entities are seen in terms of how they interact with other entities
- Entities have “responsibilities”, which may include holding knowledge
- Entities have “collaborators”, other entities they “know” and work with
- There are often multiple entities described by a general template or “class”
- Entities might have the responsibility of answering our questions or reporting things to us as “users”

Use Cases

- A use case is a description of a complete interaction of a human user with a software system
- It will consist of a sequence of user actions done through the human computer interface and the response of the system to those actions
- It may include variations: the user or system may react differently depending on circumstances
- Use cases should cover typical uses of the system
- Use cases should also cover cases where the user makes mistakes
- Older styles of software development (“Waterfall”) recommend comprehensive collection of use cases, described formally, before design and development: the aim is to have everything that is needed to produce a complete system
- Modern styles (“Agile”) recommend fewer and less formal use cases initially, but more interaction with the customer later to develop more use cases: the aim is to develop the system through frequent releases of new versions, adding new features in response to feedback

Entity relationships

- Classes are descriptions of types of entities, to consider the classes that will make up a program, consider the nouns (words that name things) in descriptions of its proposed behaviour – the use cases
- Methods are descriptions of entity actions, to consider the methods that classes have, consider the verbs (words that name actions) in descriptions of a system's proposed behaviour
- An entity will relate to other entities represented by object references in the object which represents it
- An entity will relate to other entities represented by object references as the argument to methods called on the object which represents it
- An object does not know which object called a method on it, the relationship is one way
- An object which calls a method on an object of another class has a "dependency" on that class, if it passes references to objects of another class but does not call methods on them, it has a "weak dependency"

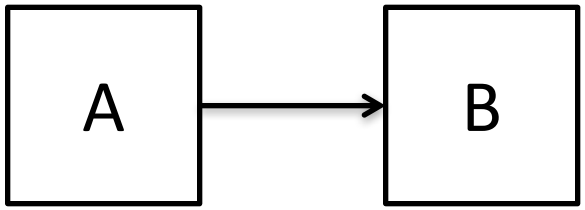
Programming as more than coding

- We build the entities in our model as “objects” according to templates called “classes”
- We set up the initial situation, and leave the entities to communicate with each other and with the human user (and possibly with other external devices)
- Action between the entities may cause new entities to be created, new collaborator links between entities to be established
- The important thing is to view our system first in terms of these entities, and only after that in terms of the code which makes them
- Object oriented programming languages help us think like that
- It can be hard to think like that when we are only dealing with small-scale code
- Skilled programmers, who can carry on thinking of programs as one computer manipulating one big block of memory using procedural commands when the scale grows larger, may find it particularly hard to adapt to this way of thinking

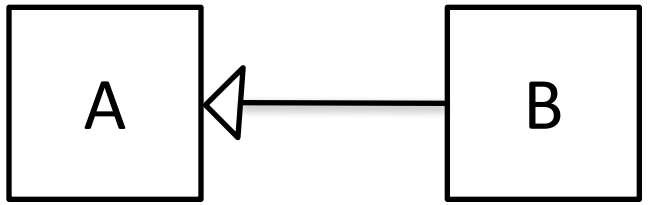
UML diagrams

- A common approach is to show the structure of a proposed program and its execution in diagrammatic form before the code is written
- UML (Unified Modelling Language) gives a standard format for several types of diagram
- The most common type of UML diagram is a “class diagram”, where classes are represented by boxes, and arrows between them represent relationships between objects of those classes – this summarises proposed code structure
- Another type of UML diagram is a “sequence diagram” which shows the pattern of interaction between objects over time – this summarises proposed behaviour of code when executed

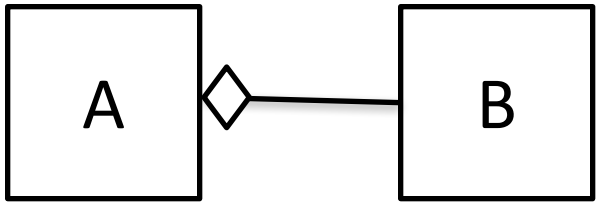
UML class relationships (1)



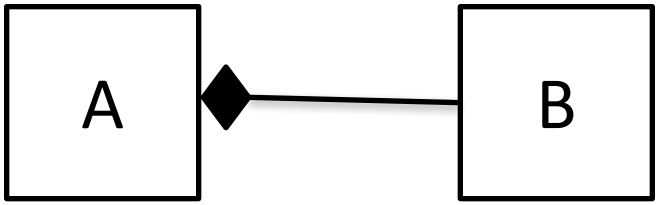
B has an association with A
(class A makes use of class B)



B is a subclass of A
(class B `extends` class A)

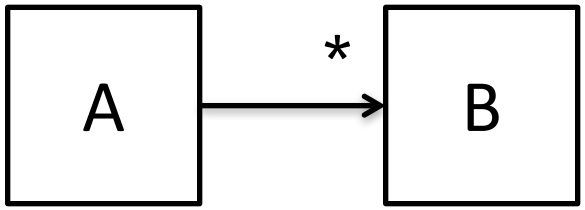


A is an aggregation of B
(class A contains objects of
class B which may be shared
by other objects)

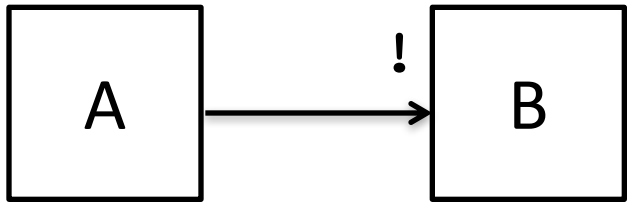


A is a composition of B
(class A contains objects of
class B which are not shared
by other objects)

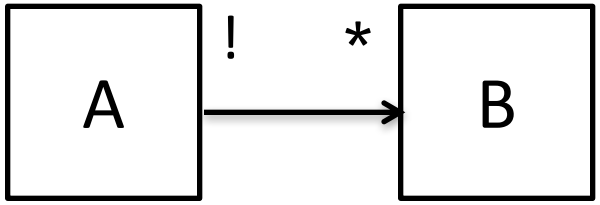
UML class relationships (2)



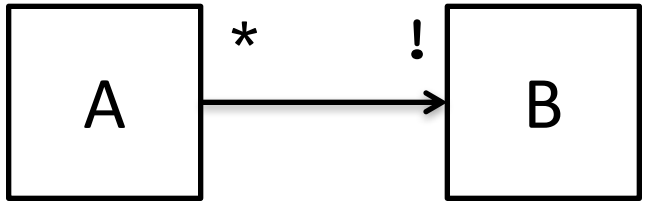
An A object makes use of <u>any</u> number (including 0) of B objects



An A object makes use of exactly 1 B object



Every B object is used by exactly one A object, although the A object may also use other B objects



The B object which an A object uses may be used by any other number of A objects, not every B is used by an A object

The CRC Card design technique

- CRC stands for “Class, Responsibilities, Collaborators”
- The CRC Card design technique was introduced as a simple way to encourage thinking in object-oriented terms, it is described in a short paper available from: `http://c2.com/doc/oopsla89/paper.html`
- It can be considered a form of “brainstorming”, an activity undertaken by a team at the start of a project to encourage thinking about the project
- The original idea was that classes would be represented physically by small cards, listing their responsibilities (what objects of that class are meant to do) and their collaborators (the classes of other objects which objects of that class have relationships with)
- Use cases are acted out, with each card held by a person who must fulfill responsibilities if necessary by asking others, but only those holding the cards of collaborators
- If necessary, new responsibilities and collaborators are added to a card
- The use of small cards is deliberate: if there is too much information to fit on one card, it should be split into several cards, representing implementing a class by breaking its work down into several classes

Reducing Dependency (1)

- The use of small cards in the CRC card technique indicates a belief that it is better to have many small classes than few large and complex classes: classes are not expensive, don't be afraid to introduce new one of your own design as and when needed
- Keeping the number of collaborators down is also a way of reducing size and complexity of classes, this is known as "reducing dependency" or "de-coupling"
- The less interaction there is between objects, the easier it is to understand and modify code
- Splitting responsibilities between classes should be done in a way to minimise the amount of communication between objects
- If you can't remove a dependency altogether, see if you can change it to a weak dependency

Reducing Dependency (2)

- A good principle: *"Don't ask for the data to perform some task, ask the object which has the data to perform the task"*
- Good attention to modelling will help: classes that define objects with a clear identity, methods that define operations with a clear purpose
- Another good principle: *"If you can't think of a good descriptive name for your class or method, you should consider redesigning your code so it no longer exists"*
- A third good principle: "*Don't have superman classes*"
  - ✧ If a class models a real-world entity, avoid giving it methods that would be modelling actions impossible in the real world
  - ✧ If you introduce some code which invokes "super-powers" to intervene in other code and solve some tricky problem, it will probably cause more problems later on
- The reason for this is that code which *"breaks the rules"* means the possibility of the rules being broken by that code in other places, and so special extra code to deal with it then has to be introduced

Law of Demeter

- A design guideline for keeping dependency low, also called the *"Principle of Least Knowledge"*
- In Java terms it is expressed as if we have a method called `meth`, then further method calls in the code for `meth` should only be on:
  - ✧ The object which the method call `meth` was made on (which is called `this` inside the code, a method call not attached to an object is by default taken to be on `this`)
  - ✧ Objects passed as arguments directly to `meth` through the parameter variables in its header
  - ✧ Objects created directly in `meth`
  - ✧ Objects which are held in instance variables of `this`
- So the code for `meth` should not contain a method call which returns an object, and then a method call on that object, unless it is clear that method call is creating a new object (a method which functions like a constructor in this way is called a "factory method")

Clear Responsibility

- Methods should work by taking arguments and returning values, you should avoid other forms of interaction such as:
  - ✧ Setting class variables (variables shared by all objects of the class) to values which other methods use
  - ✧ Setting instance variables (variables which are declared outside methods) which exist only for temporary holding of values for other methods to use
  - ✧ Writing to files, or modifying some shared data structure for other method calls to pick up
  - ✧ Using instance variables declared as `public`
- You should avoid “side effects”, a method which seems to exist to do one thing, but also does another, for example returns a value but also changes the state of the object it is called on
- Rules of style like this can be broken if you are certain it makes sense in terms of modelling, it reflects the way the world you are modelling works – but not because it solves a coding problem that would have taken just a little longer to solve in a cleaner way

Summary of Good Quality Object-Oriented Programming

- When you are writing a program, think of it in terms of objects that interact with each other
- Have separate objects that perform the task of interacting with the human user of the program
- An object interacts with another object by having a reference to it, and calling a method on it
- So, design classes that implement objects in terms of the methods that can be called on them
- Define what the methods should do in terms of interaction
- Only after that consider what needs to be inside objects of the class to make the methods work
- Ensure no interaction can take place except that which is part of the design

Design Principles

Design Principles

- The previous consideration of what is needed for good design has led to the establishment of a number of commonly accepted rules
- These are oriented towards code structure, there are many issues regarding good practice in design of human computer interface and interaction with the customer during development which are important to overall software design, but are outside the scope of this module
- Attention is given to non-coding aspects of design in Software Engineering and Systems Analysis modules

The Single Responsibility Principle (SRP)

- Also expressed as *"There should never be more than one reason for a class to change"*
- An object should have one overall role. There may be several methods that can be called on the object, but they are all aspects of its one underlying role
- A class which seems to describe objects that have two different roles is likely to be poorly designed, you should see if it could be divided into two classes, one for each role
- A class which has several responsibilities has the danger that if a change is made to its code because of changes in one of those responsibilities, it will have an effect on the way it performs its other responsibilities, resulting in further code change being necessary to correct any problems introduced

The Open Closed Principle (OCP)

- Also expressed as *"Software entities should be open for extension but closed for modification"*
- The idea is that if you want to make use of a class or method in a new way, you should be able to do so by using the existing code without changing it, just using different parameters settings (arguments to methods, arguments to constructors for classes), possibly also extending classes by writing subclasses
- Once a class or method has an established use, it should rarely be necessary to make changes to its code because you want to use it in a different way – making such changes can be difficult and dangerous because it could have unintended effect on the existing code which uses the code you are changing
- This can be achieved by careful attention to writing code with a clear general role, seeing if there are any aspects that can be set by parameters rather than fixed to particular values

The Don’t Repeat Yourself Principle (DRY)

- Also expressed as *“Every piece of knowledge must have a single, unambiguous, authoritative representation within a system”*
- The idea is that if you find you are writing code very similar to code you have written somewhere else, you ought to see if both pieces of code could be replaced by a use of one class or method, perhaps adapted for circumstances by the setting of parameters
- A “cut and paste” approach to coding is a bad habit – by this is meant seeing some existing code which looks similar to what you want, copying the code to the new place where you want it, and modifying it manually
- If a similar pattern of code is manually copied to many places, should any changes need to be made to it, they will need to be made to each place where it was copied
- If there is one piece of code used many times by calling it in many places, then if it needs to be changed, only that one piece of code has to be changed
- A “cut and paste” approach is also often a sign of poor understanding, you end with code that works but you don’t know why, which means there’s a much higher chance of errors than in code you have thought through

The Liskov Substitution Principle (LSP)

- Named after Barbara Liskov, a pioneering computer scientist who first described this problem
- The problem comes from inheritance – a powerful way of enabling code to be re-used, so keeping to the OCP and DRY principles, but a feature that can cause problems if not used with caution
- For a method to override a method in a superclass, all that is needed is for it to have the same signature, it could otherwise behave in a completely different way to the original method
- The dynamic binding principle means that code written with parameters of a particular class type will work when those parameters are set to objects whose actual class is a subclass,
- The LSP states that methods should only be overridden by code which adds extra aspects to the original specification, but does not take anything away, so that any logical reasoning made about code expressed using the superclass will still apply when the code is working with objects of a subclass

The Dependency Inversion Principle (DIP)

- Also expressed as *"High level modules should not depend on low level modules"* or "*Abstraction should not depend upon details"*
- The idea is that the behaviour of code should as far as possible be self-contained, it should not be dependent for a proper explanation of what it does on the details of other code it uses
- In practice, this principle can often be interpreted as advice to use interface types as parameters rather than class types
- If a class constructor or a method in a class has parameters of an interface type rather than of another class type, it means the class does not have a dependency on the details of another class. The interface type just gives the public aspects of the methods it needs to use, which would be from any class which implements the interface

The Interface Segregation Principle (ISP)

- Also expressed as *"Clients should not be forced to depend on methods they do not use"*
- The idea is that interface types should provide only a small number of methods that are closely related to each other
- An interface which provides a large number of methods is called a "fat" interface, such an interface might be suggested so it could be used in many different ways
- Every class that implements an interface type must implement every method in that interface type, with a "fat" interface it may have to implement methods that are not relevant to it
- A class can only extend one class, but it can implement any number of interfaces, so it is better to divide a fat interface into several interfaces and then decide for each class which of them it should implement
- Smaller interfaces with classes implementing multiple interfaces means fewer overall changes if one interface has its methods changed

General Points on the Design Principles

- These principles with these names were put together by Robert C. Martin ("Uncle Bob"):

  `https://davesquared.net/2009/01/introduction-to-solid-principles-of-oo.html`

- Various authors have suggested various sets of fundamental design principles in software design, but the ones given here are the most widely accepted
- These principles have been developed through practical experience by respected software engineers working in "real world" development
- The terms are used in discussion when comparing various ways of solving a program design problem, they provide a "vocabulary", enabling software designers to talk on a higher level than just in terms of code syntax
- At this stage, they may seem rather vague, they make more sense when you have more experience with large scale programs

Long Term Software

- An important aspect of the design principles is that they are based on the idea that code will remain in existence for a long time and be subject to frequent modification
- Large scale software systems develop incrementally: versions are released and practical experience with them leads to ideas for additional features being suggested
- As noted previously, “Agile” software development techniques are based on the idea of software systems being developed through modification rather than through large scale “up front” design
- The design principles are based around the idea of keeping code well-structured so it can be easily modified to meet new requirements
- Code which is written in a generalised way is good because individual components can be dealt with in isolation and it can be re-used to meet new needs, but novice programmers often find it hard to adapt to the abstract way of thinking it requires

Design Patterns

- The “Design Patterns” idea also provides a vocabulary for program design
- The design principles are general guidelines for breaking code into individual components
- Design patterns are named patterns of putting together code for solving particular problems
- As with the design principles, there is a common set associated with a particular author or group of authors which is recognised as the standard
- The authors are Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides known as the Gang of Four (GoF):

  `http://c2.com/cgi/wiki?GangOfFour`
- A common feature in the established design principles and design patterns is the importance of abstraction expressed through interface types
- More coverage of design patterns is in the 2nd year Software Engineering module and 3rd year Further Object Oriented Programming module

Clean Code

Clean Code

- `http://www.cleancoder.com/` follow link to *Clean Code* book
- Focuses on very basic aspects of code with the idea of keeping code easy to read and understand, which also means easy to modify:
  - ✧ Meaningful names for variables, methods, classes
  - ✧ Careful attention to division of work between functions (methods)
  - ✧ Careful use of comments
  - ✧ Careful attention to code layout
  - ✧ Distinguishing “data structures” from objects
  - ✧ Good error handling techniques
  - ✧ Well-designed classes
  - ✧ Well-designed tests

The Total Cost of Owning a Mess (1)

(quoted from "*Clean Code*")

"If you have been a programmer for more than two or three years, you have probably been significantly slowed down by someone else's messy code. The degree of slowdown can be significant. Over the span of a year or two, teams that were moving fast at the beginning of a project can find themselves moving at a snail's pace. Every addition or modification to the system requires that the tangles, twists and knots be 'understood' so that more tangles, twists and knots can be added. Over time the mess becomes so big and so deep and so tall, they cannot clean it up. There is no way at all.

…

The Total Cost of Owning a Mess (2)

…

As the mess builds up, the productivity of the team continues to decrease, asymptotically approaching zero. As productivity decreases, management does the only thing they can: they add more staff to the project in the hopes of increasing productivity. But the new staff are not versed in the design of the system. They don't know the difference between a change that matches the design intent and a change that thwarts the design intent. Furthermore, they, and everyone else on the team, are under horrific pressure to increase productivity. So they all make more and more messes, driving the productivity ever further towards zero."

Total Productive Maintenance

- A quality approach which emerged in Japan in the 1950s
- The “5 S” philosophy:
  - ✧ *Seiri*, or organisation: knowing where things are – using approaches such as sensible names is crucial
  - ✧ *Seiton*, or tidiness: a piece of code should be where you expect to find it
  - ✧ *Seiso*, or cleaning: keep the workplace free of hanging wires, grease, scraps, and waste, in programming terms remove anything that is redundant or added for temporary support reasons
  - ✧ *Seiketsu*, or organisation: the group agrees about how to keep the workplace clean, in programming terms observe a common set of standards
  - ✧ *Shutsuke*, or discipline: the discipline to follow the practices, to reflect frequently on one’s work, and be willing to change
- *… thinks* – this applies to lecturers as well!

Meaningful Names

- Names exist only at the human level of code – the compiler doesn't care what names you use
- At the human level, good choice of name can be very helpful in giving a quick intuitive understanding of the purpose of some code element and the structure of the code
- When learning to program, only you and possibly your tutor see the code, it is very short and thrown away when the exercise is finished, but code in "real world" use may have a long life time
- Code examples shown in lectures may use short names for convenience of display, but don't take this as general good practice
- Long names don't cost anything, so for class fields and method parameters always use names which describe the purpose of what they are naming
- Be especially careful not to use names which mislead

Short name conventions

- Short names are acceptable for local variables which have a limited temporary purpose with scope just a few lines of code
- Be aware of convention for short names
- `i` is used for an `int` whose purpose is to index an array or similar structure, typically incremented by 1 in a loop update to traverse it, use `j` for another such variable, do not use `i` and `j` for anything else
- `x` and `y` would generally be used for floating point numbers, `m` and `n` for integers
- `ch` (or `ch1`, `ch2` for two) might be used for a `char`, `str` for a String
- A variable storing a collection often has a name ending in `s`, for example do not use `str` for a variable of type `String[]` or some other collection of `Strings`, use `strs`

Case sensitivity

- Java is a “case sensitive” language – it always distinguishes between a lower case letter and its equivalent upper case letter
- All Java keywords are written in lower-case letters only, the Java compiler will not recognise them if they are written with some letters in upper-case
- A convention is that class and interface names always have initial upper-case letters, method and variable names always have initial lower-case letters
- The Java compiler does not insist on names obeying this convention, but it is observed in all Java library code
- Keeping to this convention makes a visually clear distinction between a static method called on a class and an instance method called on a variable, for example
- By convention, single upper case letters are used for type variables (see later)

Constants and “Magic Numbers”

- By convention, constants (`static final` variables) have names which are all upper case, with words separated by underscore characters
- If you are writing a number in some code, you should consider whether it is a value which will always be fixed, if not, make it a parameter
- If the value is always fixed, if it is used for a particular reason, it should be done through a named constant for example:

```
static final int TWO_TO_POWER_OF_FIVE_LESS_1 = 31;
static final int NUMBER_OF_DAYS_IN_MARCH = 31;
```

- If you just used `31`, then needed to change it to the number of days in April, you would have to go through all your code, to find cases where 31 is used to mean “the number of days in March”
- Numbers which appear in the code without clear explanation are called “magic numbers”, and should be avoided
- Keeping constant values in one place where they can be easily identified enables them to be easily changed if necessary

Functions

- Dividing code into “functions” is part of design, here think of “function” as meaning “helper method”, a method whose purpose is to provide a service to some other method rather than a public method which is part of the specification
- Methods should be small: do not be afraid to define a function which is called just to avoid a loop-within-a-loop or some other complex control structure
- Functions should do one thing, do not define a function which does one thing and then another unless they are clearly two aspects of one action; similarly do not define a method which does one thing or another depending on some special argument value
- Functions should not have side-effects, for example changing the value of variables in their class or changing the state of objects passed to them as arguments
- Functions should have a small number of arguments, avoid three or more if possible
- Functions should have names which describe their purpose

Comments

- We are often advised “always comment your code” when we are first taught to program, but that can lead to comments which obscure rather than help
- Comments should only be used where variable and method names and code structure are not enough to make the purpose of some code and the way it works obvious
- Never keep comments whose purpose is no longer relevant, especially comments which have become incorrect due to changes in code
- Never “cut and paste” code, and if you do, never keep comments from the old code that you don’t understand or don’t apply
- Never keep “commented out” code (modern environments will have source code control systems to allow return to previous versions of code if necessary)
- Don’t overload with information
- Make sure all your comments are written in a clear and unambiguous way

Layout

- Always indent your code to show its structure, that means statements which are inside a loop or if-statement should be shown with more spaces to the left than the loop or if header, code following it which is not in the loop or if-statement should be shown with the same number of spaces to the left as the loop of if header.
- Keep lines of code which are closely related close together, insert blank lines to distinguish separate components (for example, when one method ends and another starts
- Declare all the instance variables of a class in one place (usually at the top, though some experts recommend the bottom)
- Declare local variables (variables for temporary use inside methods) as close as possible to where they are used
- Keep closely related functions close to each other
- Don’t overload with information
- Make sure all your comments are written in a clear and unambiguous way

Objects and Data Structures

- It is good to make a distinction between “objects” and “data structures”:
  - ✧ Objects expose behaviour and hide data
  - ✧ Data structures expose data and have no significant behaviour
- This means that objects will have an internal structure which forms their representation, but is kept distinct from their public methods, so do not add getters and setters for the variables of this structure
- Data structures are just a collection of variables, in some cases we would want to put them into a separate object, which would then only have getters and setters (an array is actually a data structure object)
- We can ignore the Law of Demeter with data structure objects
- Separate data structure classes may be used as helper classes in the implementation of a class with a public use
- Java has the facility for declaring “nested classes” inside other classes, helper classes can be declared as `private static` nested classes (nested classes which are not static are called “inner classes” and raise additional issues)

Error handling and `null`

- As covered previously, use exceptions rather than special values of the return type to indicate an error has occurred with a method call
- Use exceptions which have a meaningful name in the code which catches them, if necessary catch a standard Java exception object such as `NullPointerException` and throw an exception of a type you have defined yourself which is relevant to the circumstances under which the `NullPointerException` was thrown
- Use the facility to create exception objects with `String` arguments which can be returned by a call of `getMessage()` to pass information on the cause of the error
- Avoid having code which returns `null` or passes `null` as an argument, `null` should not be used as a special value indicating an error or abnormal situation
- `null` does not mean an empty `String`, or an array or other collection of size `0`, do not use `null` when what you mean is a collection of size `0`

Cohesive classes

- As with functions, keep them small, do not have one class with a large number of responsibilities, divide classes up if necessary: this is similar to the advice we have seen with the CRC Card design technique, and the Single Responsibility Principle
- Classes are described as “cohesive” when the methods and variables of the class are co-dependent and hang together as a logical whole – you should aim for high cohesiveness in your design
- Keep to the principle of encapsulation, all variables in a class should be declared as `private`, the only case when that doesn’t apply is with data structure objects which are strictly for private use, then code which accesses them directly is neater than use of getter and setter methods

Refactoring

- Refactoring means changing your code so that it still gives the same results for the human user, but has a cleaner style inside
- Refactoring to improve the style of code is important, because, as explained, hard to understand code is hard to modify and more likely to contain errors
- It can be hard to see the importance of rules of style and good design when you are just writing small pieces of code for student exercises
- Understanding and appreciating good style in programming requires experience, but get used to the basic rules early so they become second nature
- Concentrate first on code that works, but do not be afraid to refactor – build a good test suite so you know refactoring has not stopped your code from working correctly

Defensive Programming

Defensive Programming

- Defensive programming means ensuring your code will work under all circumstances
- So, if you are asked to write a method that performs a particular task, always consider how your code would work if the arguments to the method do not meet the requirements of that task
- For example, if your method takes a list and an integer that is meant to be an index to that list, consider what it would do if the index is outside the range of the list
- Another example: what would your method do if a parameter variable that is meant to refer to an object is set to `null`
- The most important thing is to make sure it will not cause anything harmful to occur
- Throwing an exception would be one way of dealing with this

Security

- In a large system, code you have written may be used by other code written by someone else, so you cannot guarantee it will always be used in the way you suppose
- It may be used in a wrong way due to error in some other code
- Code may also be used in a way that was not intended deliberately in order to do things like obtain access to data that is meant to be secure
- In code for a financial system, that could be used to steal money
- Making sure instance variables in classes are declared as `private` is one aspect of dealing with issues like this
- Another is ensuring that references to objects are not returned if other code could then change the object, so for example return a copy of an object rather than a reference to the same object

Summary

Large scale software

- There has been a lot of material in covered in this section, but underlying it is the importance of an approach to designing and writing software that will work with large scale systems
- We started with perhaps the most important principle – that of “separation”, making a clear distinction between the public interface of a code module (what it does) from the code inside (how it does it), with the specification as the “contract” which links the two
- We moved to an object-oriented approach to software development, which consists of designing it and viewing it as interacting objects rather than lines of Java code

Loose Coupling

- We moved to the importance of reducing dependency in order to give “loosely coupled” code
- Dependency is any way in which one module of code (which would usually be a class in Java programming) relies on another
- The more dependency there is in a software system, the harder it is to take one piece of code, understand it, show it is correct, modify it to meet some new requirements, without having to consider the details of other pieces of code
- Dividing code into classes with clearly defined responsibilities helps reduce dependencies
- We need to be careful to avoid ways in which code in one class can cause change to objects of another class in a way that does not reflect their defined responsibilities

Design Principles

- The design principles are general rules for good practice when considering how to design and build a software system in an object-oriented way
- The design principles give names to aspects of software quality to do with ease of understanding and modifiability
- The design principles are based on the idea that software systems will be developed by continuous modification, so we need to design software in a way that means a change in requirements can be met by a small change in the code:
  - ✧ The effect of the change will be limited, it should not cause further problem in other parts of the code
  - ✧ The change can be made in one place, a small change in representation should not require large numbers of changes to code
- The use of interface types to reduce dependency and to write generalised code is a feature of several of the design principles

Clean Code

- Clean code consideration are at the level of actual code rather than the higher level of dividing responsibilities among objects
- As with the design principles, however, clean code is about aspects of software quality to do with ease of understanding and modifiability
- The clean code idea is about leaving the appearance of code in a way that makes it easier for others to pick it up and develop it further
- As with the design principles, these are issue to do with human efficiency, not machine efficiency
- The computer can execute poorly designed code efficiently
- Humans cannot modify poorly designed code efficiently
- If the code is well designed, but inefficient, it's easier to modify it and make it efficient than to modify poorly designed code to improve efficiency

Software Craftsmanship

- Manifesto for Software Craftmanship:
  `http://manifesto.softwarecraftsmanship.org`
- Views software development as a practical craft, like carpentry, plumbing, needlework, cooking
- Practical crafts are learnt by experience – it takes time to become a master (or mistress …) in them
- There are facts that need to be known, rules of practice that need to be respected, but there is also plenty of room for personal creativity and initiative
- A good craftsman is always learning
- A good craftsman takes pride in his or her work
- The goal is to provide customer satisfaction
- But a good craftsman knows that a good product or service requires careful attention to details which are not directly visible to or understood by the customer

Homework

- There is a lot of material in this section, you are not expected to grasp all the details in one lecture
- So take the opportunity to look more carefully through the slides in your own time later
- The material in these slides is given in detail because much of it is supplementary to what is in your text book
- The web links given make good extra reading if you are interested, but it is not necessary for assessment purposes
- The main purpose of this section is to introduce you to the idea that there is more to programming than just understanding how the programming language you are using works
- Thinking of software development in terms of design and analysis rather than just lines of code becomes essential as its scale increases beyond introductory programming exercises
- There will be more of this in second and third year modules