# Software Engineering: Design Principles

*EBU6304 Software Engineering*
*2019/20*

*Dr Matthew Huntbach*
matthew.huntbach@qmul.ac.uk

## Introduction and overview

## What is Software Engineering?

All aspects of software development:

- Requirements analysis
- Design
- Implementation
- Testing

But also

- Management of human resources
- Business issues
- Use of CASE tools (Computer Aided Software Engineering)
- Modification of existing software
- Ethical issues

## Trade offs

Software MUST

- Meet customer requirements
- Be convenient to use
- Be safe
- Be efficient
- Be secure
- Meet legal requirements

But also

- Has to be produced within a time limit
- Has to be produced using available human resources
- Has to be produced using available software resources
- Has to be produced using available financial resources

## Software lifespan

Most software systems are under continuous development:

- Customer requirements change
- Experience with use suggests new features
- Errors or poor design features need to be corrected
- Advances in technology open up new possibilities

## Software scale

Most real world software systems are large scale:

- Millions of lines of code
- Divided into many classes
- Produced by many people
- Expanded by many revisions

## Design principles

- Attention to high quality design and implementation is essential if we are to manage the complexity of modern real world software systems
- We cannot approach software development in the casual way we may have used when we first learnt to program
- We have to think of software systems at a higher level than just code
- When we write code, it is not enough that it works correctly, it must also be written in a way that makes it easy to develop further

## The dangers of poor quality code

- We cannot be sure it works correctly
- If we change one part of it to meet a new requirement, we cannot be sure whether that will cause other parts to stop working correctly
- When we change one part to meet a new requirement, we find we have to make many changes elsewhere
- We have to understand every part of the code to understand how any part of it works
- Every change made causes all these problems to grow worse
- As modern software is long-lived and undergoes many revisions, a poor design decision made early on can have very expensive consequences later

## Security problems with code

- Programs in most cases are not self-contained, they interact with other programs
- Interaction can lead to security problems, where one program can gain access to data in another program that is meant to be secure, and make changes to it that should not be made
- A security problem may be unintentional, meaning it was not realised by the programmers that one piece of code would interact with another to make an unexpected change
- Security problems may be intentional: deliberate use of code to gain unauthorised access
- Simple examples of where code can lead to security problems are given later:
  - ➢ Exposing the representation
  - ➢ Liskov Substitution Principle

## Decomposition

To manage large scale software we have to divide a system into separate parts ("modules"), in a way that means as far as possible each part can be considered on its own. The aim is to give:

- Locality - the implementation of a module can be read or written without having to examine the implementation of any other module
- Modifiability - a module can be reimplemented without requiring changes to any module which makes use of it

## Specification and implementation

- We design a software system in terms of modules which are defined by the services they provide to other modules
- We implement a module so that it provides the services it is required to provide
- Specification is the link between these
- Implementing a module is done in the same way: consider how it could be split into separate parts ("top down" approach)
- To maintain locality and modifiability it is important that modules cannot interact in any way that is outside their specification

## Classes and Methods

- You could think of "module" here as meaning class, as the standard way of dividing code into parts in object-oriented programming is to divide it into separate classes
- The code for a class is itself divided into code for methods
- The way in which code interacts is for code in a method in one class to call a method from another class
- A method in one class can also call a method from the same class
- Object-oriented programming is based on the concept of calling a method on an object, with the code for the method being in the class of the object.

## Separation

- Treat independently WHAT an entity does from HOW it does it
- In object oriented programming, WHAT an object does is given by the specification of its methods
- HOW an object does it is given by the code for its methods and the variables that form its internal structure
- Separation allows us to design with objects at application level (using them) and implementation level (what makes them work)
- It is much easier if we can think of one without having to think of the other at the same time

## Splitting Methods and Classes

- Planning separate classes and methods before implementation is one aspect of keeping code well structured
- Splitting classes and methods as code becomes complex is another aspect of keeping code well structured
- This is particularly important with the Agile approach, which involves frequent modification of code to meet new requirements
- If the code for a method is long and complex, consider whether some of it can be put into separate methods which perform subtasks
- If several variables used are related, consider whether it makes sense to combine them in an object defined by a separate class

## Helper Methods and Classes

- Declaring variables in a class as `private` is needed so that code in other classes cannot access them directly, to keep to the principle that only specified interaction can occur
- If you have defined a separate method just to help split your code into parts, it is called a helper method
- A helper method should be declared as `private`, because code in other classes should not be able to call it directly, as that would also be outside the specification
- Java allows a class to be declared inside another class (a nested class), and declared as `private` so it can only be used directly by code in the class it is in

## Reuse

Instead of writing new code for a module to provide some required service, we may make use of existing code. This has the advantage of:

- Saving the effort of writing new code
- The existing code may already be well developed and tested for errors
- The existing code may be a well-understood standard
- Overall code size is smaller if code is reused in several places (this is not “cut and paste” reuse!)
- When re-used code is updated, all the code that uses it shares in the update

## Code for Reuse

Code for reuse may be:

- Provided as standard with our programming language
- Provided by a “third party” supplier
- Developed as an “in house” code library
- Adapted from other parts of the project
- Developed intentionally early on in the project as it is anticipated such a service will be required (“bottom up” approach)

## Abstraction and generalisation

- Abstraction is identifying the essential aspects of some situation in order to model it in software
- Generalisation is identifying common elements in software requirements so we can provide them in reusable code
- Ability and confidence with abstraction and generalisation are key skills for expert programmers
- The growing emphasis on reusability in code makes abstraction and generalisation particularly important
- Code written in an abstract style can be hard to follow, do not over-use to “show off”

## UML diagrams v. Java code

- It is important to see software at a higher level than code, this is a form of abstraction
- UML diagrams are a standard way of expressing the design of software at a higher level than code
- Object oriented programming languages have become the standard for general purpose programming
- Object oriented programming language code relates closely to UML class diagrams
- Java is currently the most popular object oriented programming language, C++ and C# are similar to it

## UML class diagrams

- UML class diagrams correspond particularly closely to Java code (and to code in similar languages)
- The nodes in UML class diagrams correspond to Java classes and interface types
- The nodes are annotated with information corresponding to the fields and methods of a class
- Arcs represent relationships between classes, IS-A relationships are inheritance, HAS-A relationships are where a class has a field of a particular type
- UML enables documentation on intended HAS-A relationships between classes beyond what can be expressed in Java

## Static v. Dynamic

- Static means those aspects of code that are to do with its permanent structure
- Dynamic means those aspects of code that are to do with when the code is run
- Class diagrams in UML match the static view of code, sequence diagrams match the dynamic view
- In an object oriented programming language, a class is a static aspect, an object is a dynamic aspect
- Another use of the word “static” is seen in the Java keyword `static`

## Classes v. Objects

- A class may be considered a “recipe” or “blueprint” or “description” of a type of object
- Any number of objects may be produced to the description given in a class
- We describe an object as an instance of its class
- Each object of a particular class has its own variables of the names and types declared in the class
- A similar relationship can be considered between a method (static concept) and a method call (dynamic concept)

## Objects and method calls

- An object contains data in its own instance variables
- Method calls have their own parameter and local variables
- So a method call on an object has access to the instance variables of that object and parameter and local variables created just for that method call
- Method calls may:
  - Return values (primitive or objects) <u>or</u> throw exceptions
  - Change the values of the instance variables of the object they are called on
  - Have an external effect (input/output)
  - Call methods on objects referred to by the instance variables
  - Call methods on `this`
  - Call methods on objects referred to by parameter variables
  - Call methods on objects constructed by the method call

## Java's keyword `static`

- If a method is declared as `static` in Java, it means the method is called on its own rather than called on an object
- A `static` method from another class can be called attached to its class, so long as it is not declared as a `private` method
- If a variable is declared as `static`, it means instead of every object having its own variable of that name, there is just one variable of that name shared by all objects of its class
- If one object changes the value of a `static` variable then another object makes use of its value, that is another form of object interaction
- Interaction by using shared variables makes code hard to follow, so is generally not a good idea

# Variables and referencing

- Variables in Java refer to objects
- A method call attached to a variable in the code means it is called on the object that the variable refers to at that point
- More than one variable may refer to an object, this is called aliasing
- A variable may change which object it refers to, so if `var1` and `var2` are variables, the assignment `var2=var1` means "`var2` stops referring to what it used to refer to and starts referring to what `var1` refers to".
- If `var2` and `var3` are aliases, `var2=var1` does not change what `var3` refers to, so it causes `var2` to stop being an alias to `var3` and start being an alias to `var1`
- Change through an alias is shared
- This can be tricky to understand, diagrams may help:

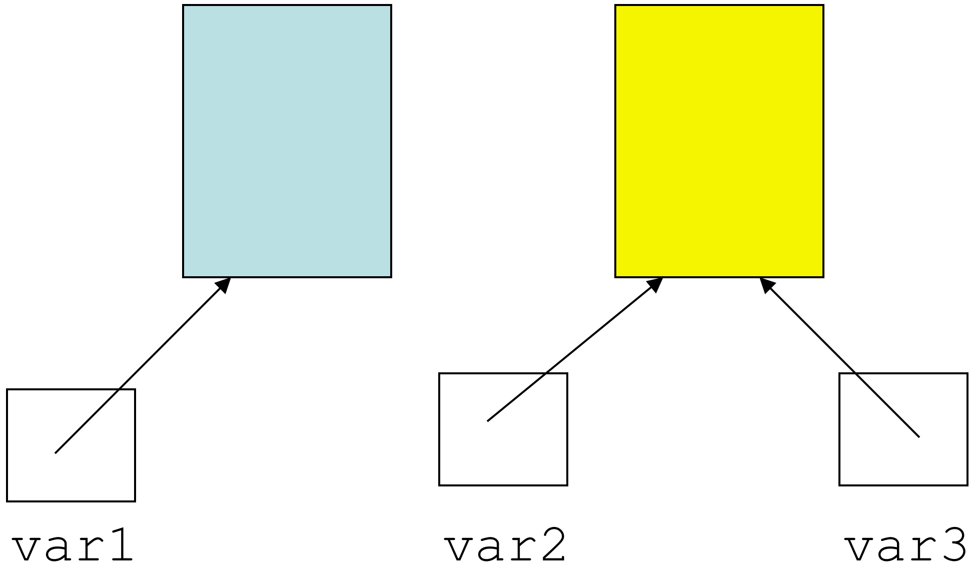
## Cells and Pointers diagrams

- Can be used to show dynamic structure of object oriented programs at variable and object level
- A small box represents a variable
- A large box, or other shape, represents an objects
- The representation of objects may contain boxes representing the variables inside them
- Arrows represent variables referring to objects
- So an arrow comes from inside a box representing a variable and points to the outside of a box representing an object
- Only one arrow can come from a box
- More than one arrow can point to a box

## Aliasing

*before*

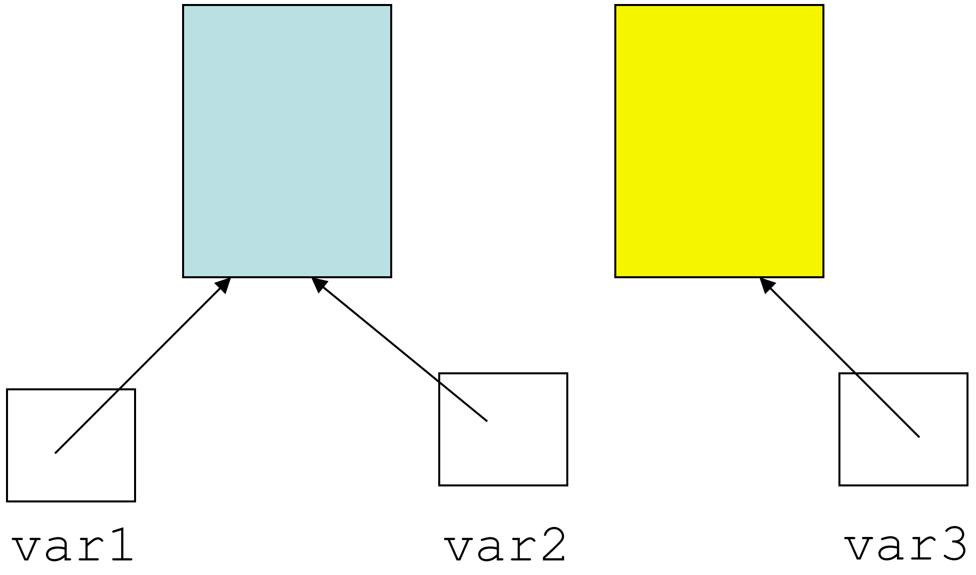
```
var2=var1;
```



## Aliasing

*after*

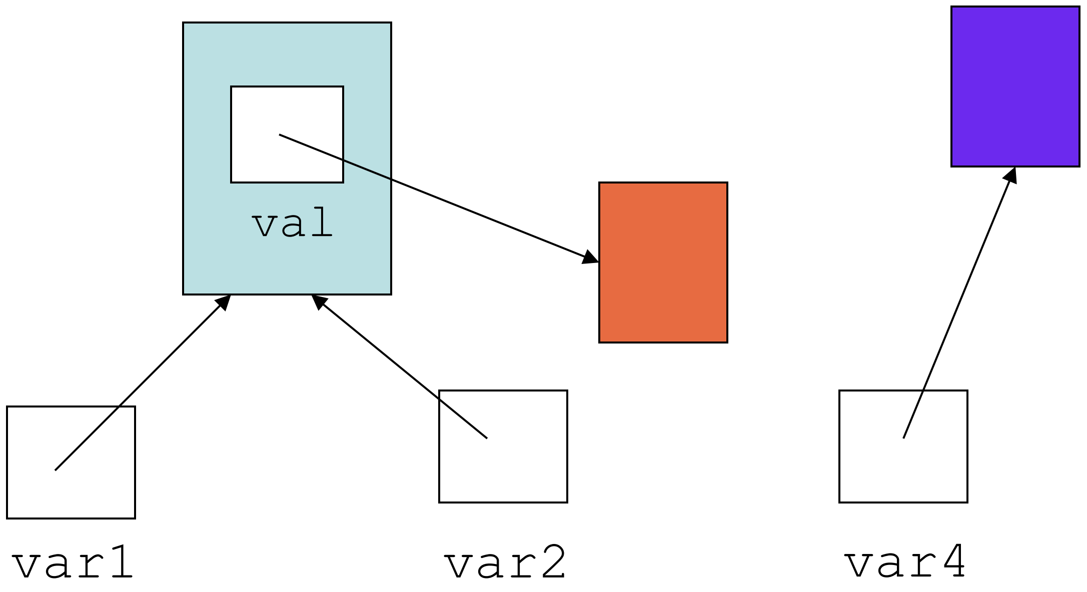
```
var2=var1;
```



## Aliasing: side effects

*before*

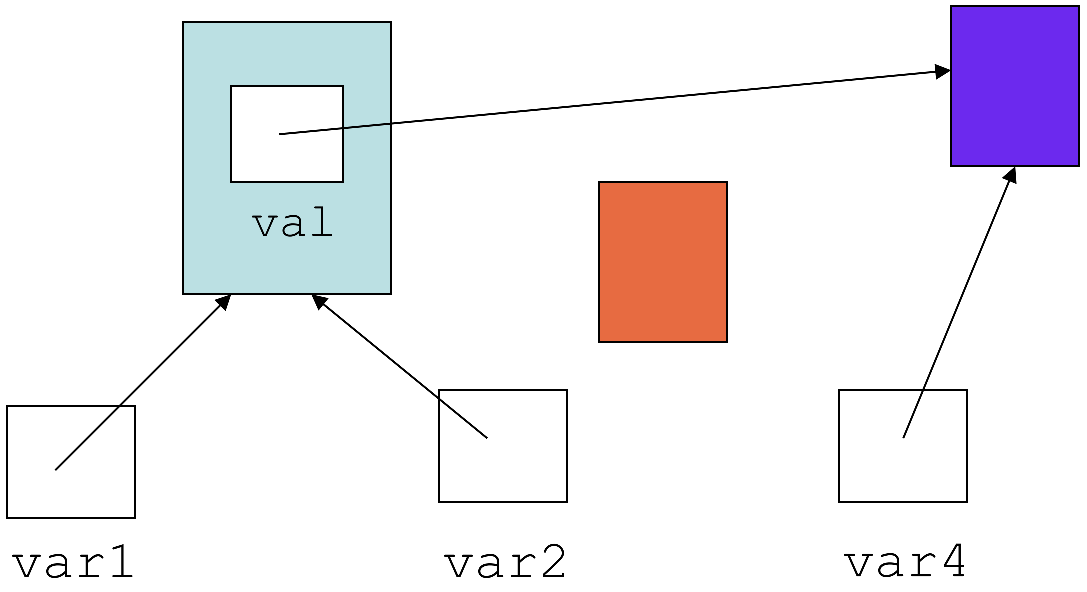
```
var2.setVal(var4);
```



## Aliasing: side effects

*after*

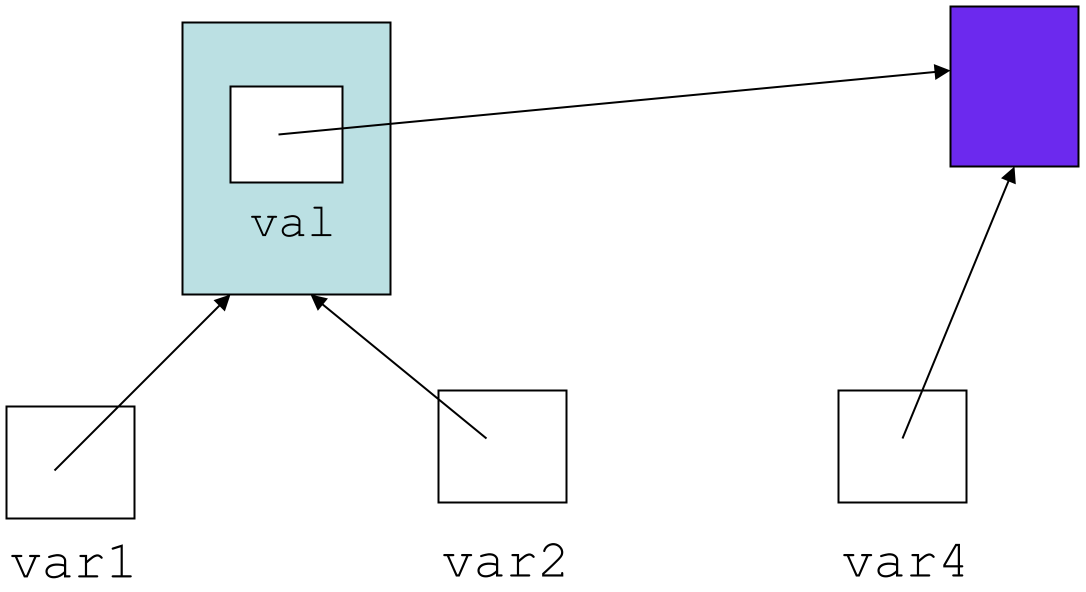
```
var2.setVal(var4);
```



## Aliasing: side effects

*after*

```
var2.setVal(var4);
```



## Scope and garbage collection

- In Java, an object, with all its instance variables, remains in existence so long as there is at least one variable which refers to it
- A parameter variable remains in existence until the method call which brought it into existence finishes
- A local variable remains in existence until execution reaches the end of the block of code it is in (the matching } to the first { before it was declared)
- If there is no variable referring to an object it is automatically “garbage collected” (in some programming languages the programmer must write code to do this)
- An object can be created in a method call but remain in existence after the method call, because it is returned by that method call or an instance variable is set to refer to it

## Care with Aliasing

- Problems can be caused when objects whose state can be changed by method calls are aliased
- When writing code that makes use of an object, we need to take into account any possibility that some other code that aliases it may also make changes to it
- When testing code that has two object parameters of the same type, we need to test that it behaves correctly when they are aliases
- It is good if possible to make objects immutable (no state changes possible) as there are then no aliasing problems
- Take particular care when returning references to internal objects, or incorporating references to external objects

## Exposing the Representation (1)

```
class MyClass {
  private int[] arr;
 …
  public int[] getArr() {
   return arr;
  }
…
}
```

- Now if we have `m` of type `MyClass` and the call

```
int[] b = m.getArr();
```

variable `b` aliases the internal variable `arr` in the object referred to by `m`

## Exposing the Representation (2)

```
class MyClass {
  private int[] arr;
 …
  public MyClass(int[] a) {
   arr=a;
  }
…
}
```

- Now if we have `b` of type `int[]` and the call

```
MyClass m = new MyClass(b);
```

variable `b` aliases the internal variable `arr` in the object referred to by `m`

## Object orientation

- A programming language concept originally designed for the special purpose of simulation works more generally because much of program design can be considered "constructing a model of the real world"
- Objects have both an internal structure (the variables inside them) and capabilities (the methods that can be called on them)
- We can think of objects as real things even people ("anthropomorphism") which communicate with each other by making method calls
- Objects "decide" for themselves how  a method call made on them is interpreted under the principle of "dynamic binding"

## Computers are not intelligent!

- It helps to talk about objects as intelligent entities which “know”, “decide”, “communicate” with each other
- Always remember though that computer programs will only do what they are programmed to do
- An object can be thought of as an “actor” who follows a script
- Programming languages are precisely defined, computers do not “guess” what is meant if an instruction is incorrect
- Computers have no “common sense”
- We can reason about computer programs and be sure they will work as we want <u>because</u> computers are not intelligent

## Standard Java programming is sequential

- When the code in an object makes a method call on another object, it halts and waits for the method call to finish
- The object on which the method call is made starts executing the code, it could call a method on another object and so on
- Sequence diagrams in UML illustrate this
- Code for a method call executes in its own environment: parameter and local variables and instance variables of the object it is called on
- When a method call terminates, code execution returns to the previous environment
- Aliasing means separate environments can share objects
- Concurrent programming using “threads” is a specialist topic, but one of increasing importance

## The most basic design principles

- Classes should be written so the objects they define are things we can have an image of in our minds
- The methods in objects should be defined so they represent clear operations on these things
- Remember the terms “locality” and “modifiability”
- Code should be written so that objects can only interact through defined method calls

## The Client-Contractor model of software

- One way of thinking of software structure is that an object which makes use of another object by calling a method on it and getting a return value or a side-effect is that it is like a client making use of a contractor
- If someone has a job that needs doing, they may hire an expert who has the skills to do that job rather than do it themselves
- Modern society is able to achieve much more than a primitive society because most people have particular jobs and rely on others for other aspects of life
- Similarly, a complex software system works because it is divided into parts, each of which has its own role, and interacts with others performing their roles
- If I hire someone to do a job for me, we will agree to a contract
- The equivalent in a software system is the specification of a class or method

## Obligations and Benefits (1)

- If I hire someone to provide me with a product or service, we agree on the payment, I get the benefit of the product or service, the supplier gets the benefit of the payment
- I have the obligation to make the payment, the supplier has the obligation to provide the product or service
- For simple transactions, the benefits and obligations are laid down by state law, or by the internal regulations of the organisation we are working within
- For more complex transactions, we may draw up an individual contract
- When a method is called on an object in a software system, it has the obligation to return the specified result, it also has the obligation not to do anything damaging, such as change a data structure passed to it when that change was not in the specification

## Obligations and Benefits (2)

- If I write code for someone else to use, I have the obligation to make sure it meets its requirements, and perhaps the benefit of being paid for writing it
- If I am writing complex code, it may help for me to think of it as being broken down into separate pieces (classes) which describe how workers work to provide their obligations, and rely on other workers they use providing their obligations
- A clear establishment of the separate roles of each class, described by their specification, and an understanding of why classes should not do anything destructive outside these roles is key to good software development which is sustainable as the software grows more complex
- The principles for good software design help us decide how to break down software into separate parts with clearly defined roles

## Design by Contract™

- The phrase and idea of Design by Contract was developed (and trademarked) by the French Computer Scientist, Bertrand Meyer
- Meyer considered that client code has an obligation to meet any pre-conditions specified by the contractor code (as a simple example it may be specified that an `int` parameter should be set to a non-negative value)
- From this, Meyer opposed the idea of defensive programming, which means writing your code in a way that makes sure it can handle all possible inputs
- Meyer proposed that methods given invalid input should terminate in a special way – which was influential in the development of Java's exception mechanism
- Meyer proposed that tests for whether methods have met their post-conditions be incorporated into the program itself as assertions which are now an aspect of Java

# Design Principles (1)

Single Responsibility Principle (SRP)

*dependency, association, aggregation, composition*

Open-Closed Principle (OCP)

Don't Repeat Yourself (DRY)

*abstraction and generalisation*

## Single Responsibility Principle (SRP)

- Every object in a system should have a single responsibility, and all the object’s services should be focused on carrying out that single responsibility
- Leads to highly cohesive software

Also expressed as:

- “A class should have only one reason to change”

## Cohesion and Coupling

As covered in Week 2:

- Cohesion: the extent to which code inside a module works with other code inside the same module
- Coupling: the extent to which code inside a module works with other code in another module

Good quality code has high cohesion and loose coupling

So:

- If a class has methods and variables which do not make use of each other, can it be split into separate classes?
- If code in one class makes a lot of calls of methods on objects of another class, can the classes be re-arranged to reduce the number of method calls?

## SRP Example

- Consider the following class:

```
class Employee // Example of code which breaks the SRP
{ …
 public Money calculatePay(…) { … }
 public void saveToDatabase(…) { … }
 public String reportHours(…) { … }
 …
}
```

- This means that class `Employee` has its own code to
  - Produce an object of type `Money` representing the payment to an employee
  - Save its details to a database
  - Produce a `String` which represents a recording of the employee's worked hours
- Class `Employee` will need to be changed if the pay roll calculation requirements change, if the database storage requirements change, if the format for `String`s for reporting hours worked changes.

## Poor Cohesion and Coupling

- Consider the following class:

```
class Employee
{ …
 public Money calculatePay(…) { … }
 public void saveToDatabase(…) { … }
 public String reportHours(…) { … }
 …
}
```

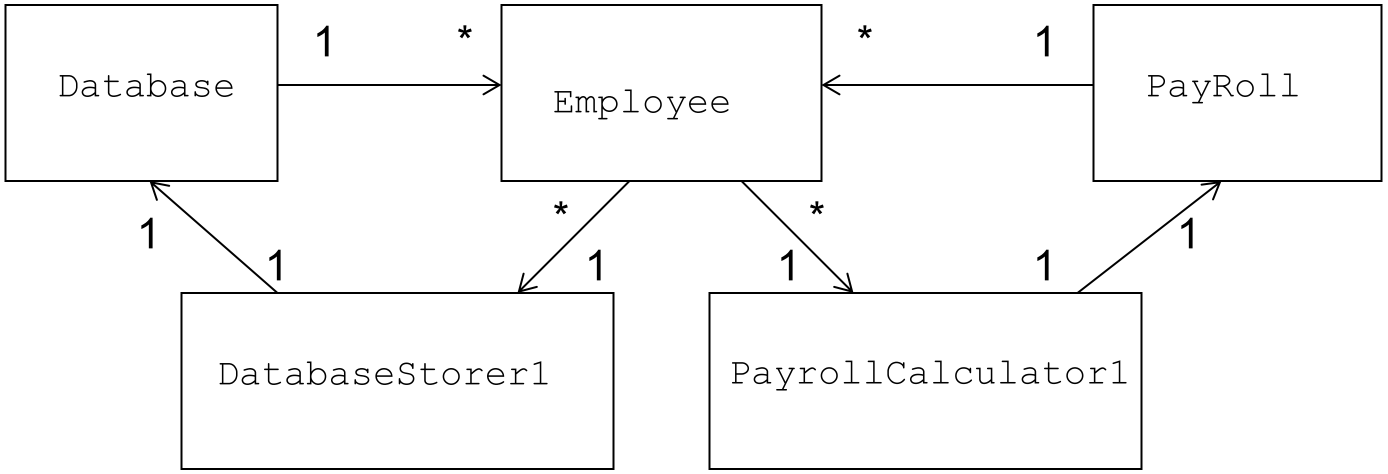
- The code in the methods `calculatePay`, `saveToDatabase` and `reportHours` will have a lot of aspects that do not relate closely to each other, giving low cohesion
- The code in the method `calculatePay` will need to interact extensively with a `PayRoll` object, the code in the method `saveToDatabase` will need to interact extensively with a `Database` object, giving tight coupling

- It would be better if `Employee` objects did not need to "know" about all the detail that would be required to perform the various operations
- Here is one way to resolve it (using delegation):

```
class Employee
{ …
 private PayrollCalculator1 pc;
 private DatabaseStorer1 ds;
 private HoursFormatter1 hf;
  …

 public Money calculatePay(…)
  { return pc.calculatePay(this,…) }

 public void saveToDatabase(…)
  { ds.saveToDatabase(this,…) }

 public String reportHours(…)
   { return hf.reportHours(this,…) }
 …
}
```

- With this structure, there are separate classes to perform the operations

- In this case there would be one `PayrollCalculator1`, `DatabaseStorer1` and `HoursFormatter1` object which manage the connections between `Employee` objects and the `PayRoll`, `Database` and `Display` objects
- The diagram here is when the `Database` and `PayRoll` objects call methods directly on `Employee` object



*Plus an `HoursFormatter1` class similarly linked to the `Display` class*

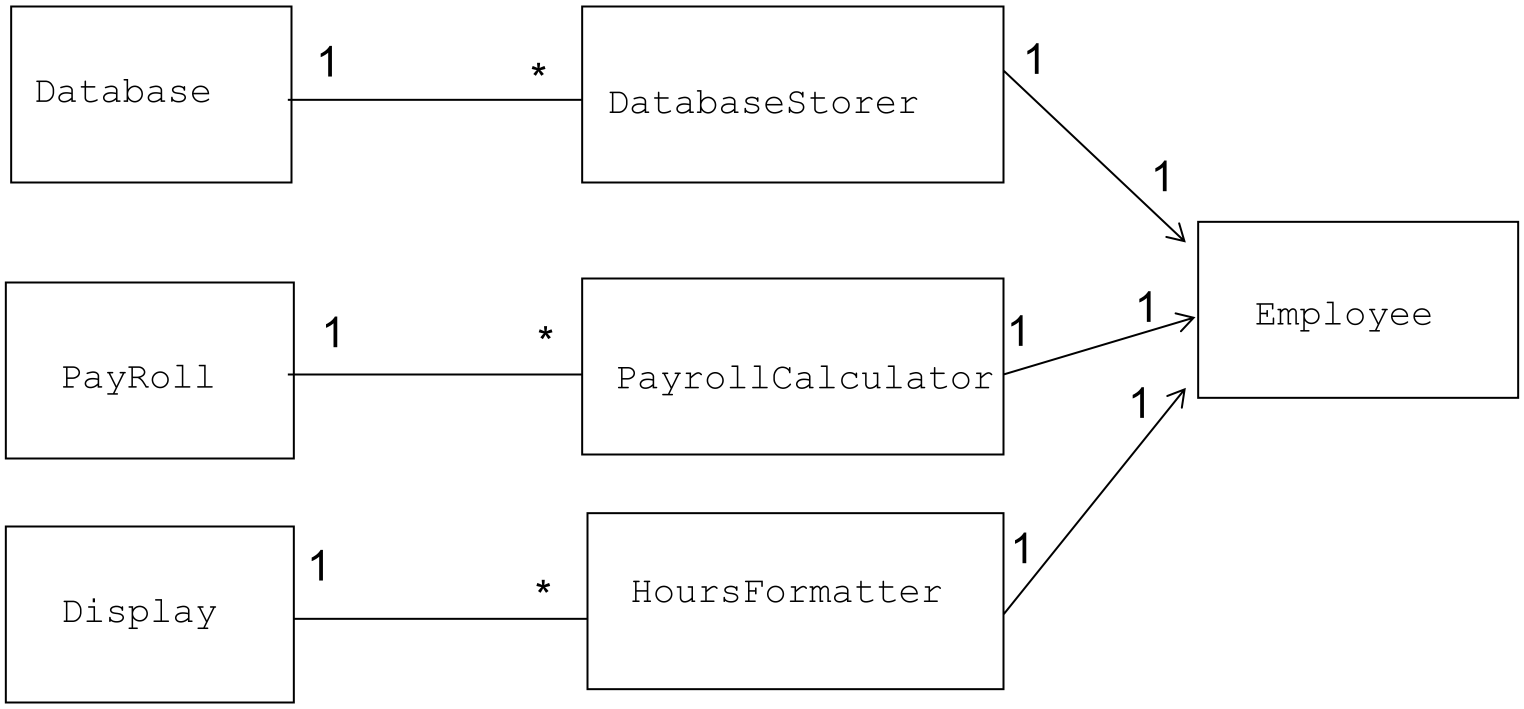
- This is not ideal, as the `PayRoll`, `Database` and `Display` objects need to "know" about `Employee` objects, which are passed to them through methods `calculatePay`, `saveToDatabase`, and `reportHours` using the argument `this`

- A better way to resolve it is for the delegation to be the other way round:

```
class PayrollCalculator
{ …
 private Employee emp;
 private PayRoll pr;
   …
 public Money calculatePay(…) { … }
}

class DataBaseStorer
{ …
 private Employee emp;
 private Database db;
   …
 public void saveToDatabase(…) { … }
}

// Similar for HoursFormatter
```

- Now the `PayRoll`, `DataBase` and `Display` objects interact with `Employee` objects indirectly through objects which have the single responsibility of managing that interaction

- In this case there would be many `PayrollCalculator`, `DatabaseStorer` and `HoursFormatter` objects, one for each `Employee` object. A `PayrollCalculator`, `DatabaseStorer` or `HoursFormatter` object manages the connections between an `Employee` object and the `PayRoll`, `Database` or `Display` objects.



## Dependency

- We wrote of one class "knowing" another, but there are degrees of linkage between classes
- In our first design, an `Employee` object HAS-A `PayrollCalculator1`, a `DatabaseStorer1` and an `HoursFormatter1` object, possibly shared by other `Employee` objects. `PayrollCalculator1` etc objects do not have an `Employee` object as part of their structure, but they have a method which takes an `Employee` object and calls methods on it
- In our second design, a `PayrollCalculator` object has an `Employee` object and a `PayRoll` object, the same `Employee` object will also be referenced as part of the structure of a `DataBaseStorer` object and an `HoursFormatter` object
- This is not an IS-A relationship (inheritance)
- Any linkage between classes is a "dependency", a "weak dependency" is when a class takes object of another class as method arguments, but does not call methods on them

## Association, Aggregation and Composition

- An “association” is when an object of one class HAS-A link to an object of another class through an instance variable
- An “aggregation” is a form of association when an object of one class is made up of objects of other classes - in our second design a `DatabaseStorer` object is an aggregation of an `Employee` object and a `Database` object
- A “composition” is a form of aggregation where the objects joined together exist only to make the aggregation - this is NOT the case with a `DatabaseStorer` object as it shares its `Employee` and `Database` objects with other objects which are also aggregations such as the `PayrollStorer` object for that `Employee` object
- Be careful - these terms are not strictly defined, different authors may use them with slightly different meanings

## Open-Closed Principle (OCP)

Software modules (classes, methods, etc) should be

- "open for extension"
- "closed for modification"

Meaning

- We can make a module behave in new and different ways as requirements change or to meet new needs
- But we should be able to do this in a way that does not require changing the code of the module

Abstraction is the key

## Abstraction

Consider:

- Sort an array of `String`s of length 100 into alphabetical order
- Sort an array of `String`s of any length into alphabetical order
- Sort a `List<String>` object into alphabetical order
- Sort a `List<T>` object in the ordering defined by type `T` (using `compareTo`, for `String` it is alphabetical)
- Sort a `List<T>` object in the ordering defined by a `Comparator<T>` object

Each of these is more abstract than the previous one, the last two have no dependency on the `String` class

## Forms of Abstraction

- By delegation - leave an array to know its own size, a `String` to know its own way of ordering itself with other `String`s using the method `compareTo`
- By generalisation - using type hierarchy, Java's type `List` is an interface type which covers `Vector`, `ArrayList`, `LinkedList`
- By parameterisation - a `Comparator` object has its own method `compare` which works similar to `compareTo`, but having a separate `Comparator` object enables us to separate out the comparison operation, for example comparing `String`s by length rather than alphabetically
- By type parameterisation - Java (since Java 5) has "type variables", a `List<String>` is a list of Strings, but if `T` has been declared as a type variable a `List<T>` parameter matches a list of any particular type.
- We will look further at some of these concepts later

## Writing Abstract Code

- Instead of modifying code when we want to use it for a different purpose we should write our code in a way where we can make use of it as it is written
- So we should try to design and write code in a general way so that we can make use of it in a more specialist way by passing in parameters or through inheritance
- Learning to think in an abstract way is one of the key skills for a good programmer
- Abstraction means identifying what are the essentials
- The essentials for sorting are a list of objects (a collection where objects have a position), and an operation that takes two of those objects and determines if they are equal or which is the greatest in some consistent order

## Don't Repeat Yourself (DRY)

- Suppose you find yourself writing very similar code many times
- You might use "cut and paste", copying the code and then modifying it for where it is used each time
- This seems to save time, but is poor practice, for example:
  - Suppose you have code for sorting an array of integers
  - You find you need to sort an array of `String`s, so you cut and paste that code, changing `n1<n2` to `str1.compareTo(str2)<0` and so on
  - Now suppose you find a mistake in your sort code, or you discover a more efficient sorting algorithm
  - You will have to go through all the places where you used "cut and paste" to get the sort code, and change it to the new version
  - It is better to have one generalised sort method

## Generalisation

- The “Don’t Repeat Yourself” (DRY) principle is the “Open-Closed Principle” (OCP) looked at the other way round
- Designing and writing abstract code is hard, so we often start off writing more specific code
- When we find we are repeating ourselves in code, we should see if we could replace the repeated code by writing and using more abstract code
- So, generalisation is identifying repeated aspects and from this moving to write more abstract code

## Inheritance as a way of meeting the OCP

- When we declare one class as a subclass of another (its superclass), we only have to give the way it differs from the superclass
- The subclass can add extra features (methods and instance variables that they use)
- The subclass can change the way the methods of the superclass work (overriding)
- So writing a subclass is a way of modifying the way the code of the superclass works without changing that actual code

## Two ways inheritance meets the OCP

- Another way in which inheritance meets the OCP is that a variable of a class type can be set to refer to an object whose type is a subclass
- So an existing method with a parameter of a particular class will work with an object of a subclass of that class as an argument without any need to make changes to it
- We will consider this further when we look at interface types and the Dependency Inversion Principle, and issues with inheritance in the the Liskov Substitution Principle
- The Liskov Substitution Principle suggests caution on using inheritance because of the way in which a method is overridden could result in code having an unexpected behaviour

## Example of parameterisation as a way of meeting the OCP

- Longest String in an ArrayList of Strings:

```
static String longest(ArrayList<String> list)
{
 String longestSoFar="";
 for(String str : list)
    if(str.length()>longestSofar.length())
       longestSofar=str;
 return longestSoFar;
}
```

- Generalised with a Comparator parameter:

```
static String most(ArrayList<String> list, Comparator<String> comp)
{
 String mostSoFar="";
 for(String str : list)
    if(comp.compare(str,mostSoFar)>0)
       mostSofar=str;
 return mostSoFar;
}
```

## Parameterisation as a way of meeting the OCP

- The way the method `most` works can be modified without changing its code by passing different `Comparator<String>` objects as its second argument
- Here is the definition of a `Comparator<String>` that would make it return the longest `String`:

```
class LongComp implements Comparator<String>
{
 public int compare(String str1, String str2)
  {
   return str1.length()-str2.length();
  }
}
```

## Parameterisation as a way of meeting the OCP

- A `Comparator<String>` that would make `most` return the `String` with the most occurrences of a particular character:

```
class MostOccurrences implements Comparator<String>
{
 private char myChar;

 public MostOccurrences(char ch)
 {
  myChar=ch;
 }

 public int compare(String str1, String str2)
 {
  return countChar(str1)-countChar(str2);
 }

 private int countChar(String str)
 {
  int count=0;
  for(int i=0; i<str.length(); i++)
     if(str.charAt(i)==myChar)
        count++;
  return count;
 }
}
```

# Design Principles (2)

Dependency Inversion Principle (DIP)

Interface Segregation Principle (ISP)

*Interface types (polymorphism)*

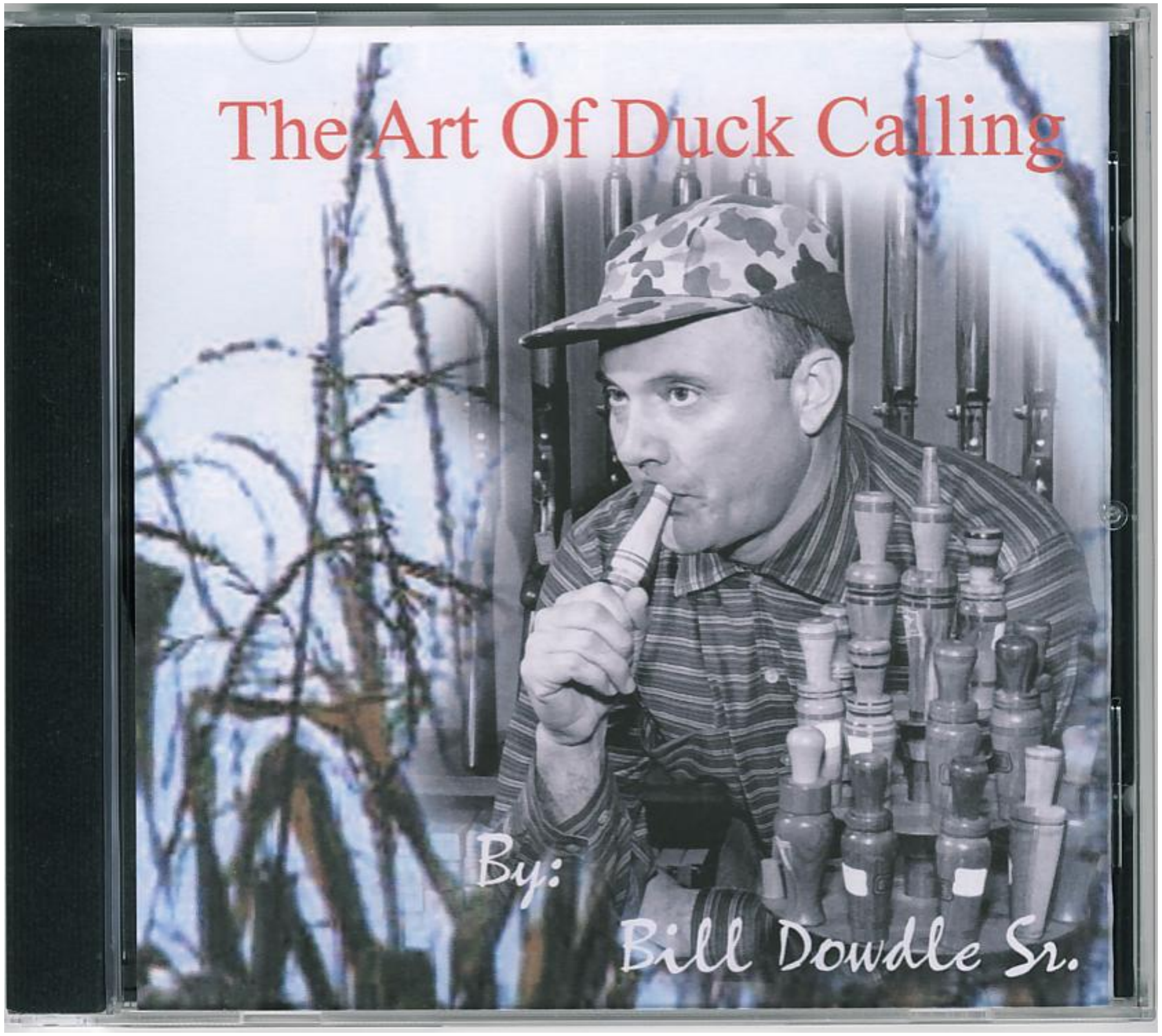
Liskov Substitution Principle (LSP)

*Inheritance and overriding*

## Generalisation

```
class Duck {
 private String myName;
 public Duck(String theName) {
   myName=theName;
  }
 public void quack() {
  System.out.println(myName+": Quack! ");
 }
}

class DuckCall {
 private String myName;
 public DuckCall(String theName) {
   myName=theName;
  }
 public void quack() {
  System.out.println(myName+": Fake Quack! ");
 }
}
```

- Using a print statement rather than returning a `String` is poor practice, as in general user interface code should be kept separate, but it is done here to give a simple example


The Art Of Duck Calling
By:
Bill Dowdle Sr.

## Interface types

- Both classes have a method with the same header:

```
public void quack()
```

- Can we write code that would work for both?

```
public static void nQuacks(int n, Duck q)
// What about DuckCall ?
{
 for(int i=0; i<n; i++)
    q.quack();
}
```

- Java’s `interface` mechanism is the way to do this

-
  ```
  interface Quackable {
    public void quack();
  }
  ```

-
  ```
  class Duck implements Quackable
  // rest as before
  ```

-
  ```
  class DuckCall implements Quackable
  // rest as before
  ```

-
  ```
  public static void nQuacks(int n, Quackable q){
   for(int i=0; i<n; i++)
      q.quack();
  }
  ```
- `Quackable` is "the type of objects on which `quack()` can be called"

## Types as capabilities

- Class types mix implementation with application, the actual class type of an object says WHAT it can do and HOW it does it
- An interface type says only WHAT an object can do: the methods that can be called on an object known to be of that type (or its “capabilities”)
- Using interface types is not only a way of generalisation, it helps maintain the principle of separation of WHAT from HOW

## Actual and apparent types

- A variable of type `Quackable` can refer to an object of type `Duck`, an object of type `DuckCall`, or an object of any other type which is declared as `implements Quackable`
- From this we can see an object may have more than one type, its actual type is the class of the constructor which created it
- But it also has the type of any interface its actual type implements
- The apparent type of an object referred to by a variable is the type of the variable

## Subtypes and supertypes

- `Quackable` is a supertype of both `Duck` and `DuckCall`
- `Duck` and `DuckCall` are both a subtype of `Quackable`
- This is similar to the relationship between classes given by `extends`
- However, a class can only extend one other class, but it can implement any number of interfaces
- With `implements`, as interface types have only method headers there is no code to inherit

## Trying to generalise with complex classes

```
class Quacker // Example of POOR PRACTICE
{
 private String name;
 private boolean fake;

 public Quacker(String nm, boolean f) {
  name=nm;
  fake=f;
 }

 public void noise() {
  System.out.println("Quack!");
 }

 public String getName() {
  return name;
 }

 public boolean isFake() {
  return fake;
 }
}
```

- That was trying to use a single class called `Quacker` to represent both ducks and duck-calls
- With this, our `nQuacks` code might be:

```
public static void nQuacks1(int n, Quacker q) {
 for(int i=0; i<n; i++) {
    System.out.print(q.getName()+ ": ");
    if(q.isFake())
       System.out.print(" Fake ");
    q.noise();
   }
}
```

- The class containing this method has a much greater dependency on the `Quacker` class than with the previous version, it is also more complex
- The use of an interface type implemented by separate classes instead of the above means changes to code in the classes will not mean changes have to be made to the calling code
- It also fits the Open-Closed Principle

- Consider

```
class RubberDuck implements Quackable {
 private String myName;

 public RubberDuck(String theName) {
   myName=theName;
 }

 public void quack() {
  System.out.println(myName+": Squeak! ");
 }
}
```

- The `nQuacks` code does not have to be changed to deal with this
- The `Duck` and `DuckCall` code do not have to be changed
- But if we had to amend `Quacker` to introduce this new variation, so that a `Quacker` object could represent a duck, a duck-call or a rubber duck, we would also have to amend the `nQuacks1` code

## The Dependency-Inversion Principle (DIP)

- The Dependency Inversion Principle can be stated as:
  - ➢ High-level modules should not depend on low-level modules; instead, both should depend on abstractions
  - ➢ Abstractions should not depend on details; instead, details should depend on abstractions
- The attempt to generalise by creating a complex `Quacker` class was an example of a high-level module which is dependent on a low-level module, because the `nQuacks1` method which used the `Quacker` class objects needed to know the details of how those objects work
- The `nQuacks` method only depends on the abstraction `Quackable`, it leaves the details to the classes which implement that interface
- So the `nQuacks` method shows an example of keeping to the DIP, and the `nQuacks1` method shows an example of not keeping to the DIP
- A simple way of keeping to the DIP is always to use interface types for parameters to methods and variables in classes

## Dependency on abstractions

- The idea of abstraction is that we try to decide what are the important aspects of some type of object that we need to consider, and what are unimportant details which we can leave out of consideration
- An abstraction is a description of a type of object just in terms of those aspects which we need to know about it
- An important part of good program design is to work out what abstractions we want to define and use
- Code which refers to objects only through variables of an interface type is dependent only on the abstraction which that interface type defines
- In our example, the abstraction decided on was that we view an object just in terms of one method which makes a particular sort of noise, we do not consider the details of the different variations of that noise
- That means detailed decisions on what noise is made and how are left to the classes which implement the interface, so the details of these classes are dependent on the interfaces they implement

## The Interface-Segregation Principle (ISP)

- The ISP can be expressed as “Clients should not be forced to depend on methods they do not use”
- The DIP suggests the use of interface types, but the ISP goes further and stresses the importance of <u>small</u> interface types
- A small interface type is one which has few methods in it
- The ISP suggests if you have an interface type with a lot of methods in it (a “fat” interface), you should see if it could be broken up into several interfaces
- A class can implement more than one interface, so a class that provides all the methods from the original interface can be declared as implementing all the new small interfaces
- However, it means that new classes which provide some of the services only have to implement those small interfaces which are relevant to them
- Classes which use the services should refer to them through variables of interface types which cover only those methods they need

## Interface-Segregation Principle example

- As a simple example of the ISP, suppose instead of `Quackable` we had decided to have a general interface for duck-like objects:

```
interface DuckLike
{
 void quack();
 void fly();
}
```

- Ducks can fly, but duck-calls cannot, so if we had started off with this, we would have to give `DuckCall` a meaningless `fly` method
- Instead we can have a separate interface

```
interface Flyer
{
 void fly();
}
```

with

```
class Duck implements Quackable, Flyer
```

# Java's interface `Comparable<T>`

- As another example of the ISP, Java provides the following interface:

```
interface Comparable<T> {
   int compareTo(T obj)
}
```

- Two objects of a type which implements this interface can be compared by calling `obj1.compareTo(obj2)` which returns a negative integer if `obj1` is less than `obj2`, a positive integer if `obj1` is greater than `obj2` and 0 if they are equal in “natural order”
- This enables methods to be written such as:

```
<T extends Comparable<T>> T most(ArrayList<T> list) {
    T mostSoFar = list.get(0);
    for(int i=1; i<list.size(); i++) {
       T element=list.get(i);
       if(mostSoFar.compareTo(element)<0)
          mostSoFar=element;
     }
    return mostSoFar;
}
```

# Generalisation using `Comparable<T>`

In the method

```
<T extends Comparable<T>> T most(ArrayList<T> list) {
    T mostSoFar = list.get(0);
    for(int i=1; i<list.size(); i++) {
       T element=list.get(i);
       if(mostSoFar.compareTo(element)<0)
          mostSoFar=element;
     }
    return mostSoFar;
}
```

the use of an interface with just the method `compareTo` means the code is generalised and does not have dependency on any aspect of the elements of the `ArrayList` it takes, except that they have their own method `compareTo`.

- An additional generalisation technique here is the use of a type variable instead of a specific type of element for the `ArrayList`
- Java provides built-in methods using these generalisation techniques in its class `Collections`.

## Comparator<T> and Comparable<T>

- Note the distinction between:

```
interface Comparable<T> {
   int compareTo(T obj)
}
```

  and what we saw used previously, which is defined by:

```
interface Comparator<T> {
   int compare(T obj1, T obj2)
}
```

- Here is a further generalisation of the use of a `Comparator`, by using a type variable:

```
static <E> E most(ArrayList<E> list, Comparator<E> comp) {
 E mostSoFar=list.get(0);
 for(int i=1; i<list.size(); i++) {
    E element=list.get(i);
    if(comp.compare(element,mostSoFar)>0)
       mostSofar=element;
   }
 return mostSoFar;
}
```

# Generalising with Type Variables

- Taking the generalisation a bit further is:

```
static <E> E most(ArrayList<E> list, Comparator<? super E> comp)
{
 E mostSoFar=list.get(0);
 for(int i=1; i<list.size(); i++) {
    E element=list.get(i);
    if(comp.compare(element,mostSoFar)>0)
       mostSofar=element;
   }
 return mostSoFar;
}
```

- The reason for using `Comparator<? super E>` rather than `Comparator<E>` is that `E` is set to a particular type, but a `Comparator` of a supertype of `E` can compare objects of that type.
- For example, if we have type `BankAccount`, and a subtype of `BankAccount` such as `FeeAccount`, this enables an `ArrayList<FeeAccount>` to be processed with a `Comparator<BankAccount>`, with `E` set to `FeeAccount` indicating that the object returned must be of type `FeeAccount`.

## Generalising with Type Variables

- It could be put this way round:

```
static <E> E most(ArrayList<? extends E> list, Comparator<E> comp)
{
 E mostSoFar=list.get(0);
 for(int i=1; i<list.size(); i++) {
    E element=list.get(i);
    if(comp.compare(element,mostSoFar)>0)
       mostSofar=element;
   }
 return mostSoFar;
}
```

- However this means that if the arguments are an `ArrayList<FeeAccount>` and a `Comparator<BankAccount>`, then `E` is set to `BankAccount`, rather than `FeeAccount`.
- So that means the method call will be treated in a way that means it could return an object of type `BankAccount` that is not of the subtype `FeeAccount`.

## Inheritance and Generic Types

- A variable of type `BankAccount` can be set to refer to an object whose actual type is a subtype of `BankAccount`, such as `FeeAccount`.
- However, a variable of a generic type with the generic value set to `BankAccount` cannot be set to an object of the same generic type with the generic value set to a subtype.
- So a variable of type `ArrayList<BankAccount>` cannot be set to refer to an object of type `ArrayList<FeeAccount>`.
- This is to avoid something like `list.add(acc)` where `list` is a variable of type `ArrayList<BankAccount>` that refers to an object of type `ArrayList<FeeAccount>` and `acc` is a variable of type `BankAccount` or another subtype of `BankAccount` that refers to an object that is not of type `FeeAccount`.
- A variable of type `ArrayList<? extends BankAccount>` can be set to an `ArrayList<BankAccount>` object, or an `ArrayList` whose contents is a subclass of `BankAccount` such as `ArrayList<FeeAccount>`.

## Specialising Type Variables

- Alternatively, a type variable can be declared in a way that means it must be set to `BankAccount` or a subtype of `BankAccount`, such as `FeeAccount`.
- This is another way of generalising code, it means methods from type `BankAccount` can be called on variables whose type is the type variable.
- For example:

```
static <E extends BankAccount> E mostBalance(ArrayList<E> list)
{
 E mostSoFar=list.get(0);
 for(int i=1; i<list.size(); i++) {
    E account=list.get(i);
    if(account.getBalance()>mostSoFar.getBalance())
       mostSofar=account;
   }
 return mostSoFar;
}
```

- This means that if the call to the method `mostBalance` takes an `ArrayList<FeeAccount>` it is declared in a way that guarantees it returns a `FeeAccount` object.

## Polymorphism and Dynamic Binding

- The ability to write code that works for many different classes is known as polymorphism (in technical terms, “poly” indicates “many”, “morph” indicates “shape” or “appearance”, “ism” indicates an “idea” or “principle”; remember “anthropomorphism”?)
- The decision on which code to use when a method call is executed is made at run-time, it might depend on factors that are not known at compile time - this is called dynamic binding (sometimes also known as “late binding”)

## Mutable and immutable

```
class Rectangle {  // Is mutable
 private int height, width;

 public Rectangle(int h, int w) {
  height=h;
  width=w;
 }

 public int getHeight() {
   return height;
 }

 public void setHeight(int h) {
   height=h;
 }

 public int getWidth() {
   return height;
 }

 public void setWidth(int w) {
   width=w;
 }

 public int area() {
   return height*width;
 }
}
```

```
class Colour {  // Is immutable
 private int red, green, blue;

 public Colour(int r, int g, int b)
 {
  red=Math.max(0,Math.min(255,r));
  green=Math.max(0,Math.min(255,g));
  blue=Math.max(0,Math.min(255,b));
 }

 public int getRed() {
   return red;
 }

 public int getGreen() {
   return green;
 }

 public int getBlue() {
   return blue;
 }

 public Colour tint(double t){
   int r = Math.round(red*t);
   int g = Math.round(green*t);
   int b = Math.round(blue*t);
   return new Colour(r,g,b);
 }
}
```

## Inheritance

```
class ColouredRectangle extends Rectangle {
 private Colour colour;

 public ColouredRectangle(int h, int w, Colour c) {
  super(h,w);
  colour=c;
 }

 public Colour getColour() {
   return colour;
 }

 public void setColour(Colour c) {
   colour=c;
 }
}
```

- A `ColouredRectangle` IS-A `Rectangle`
- The class `ColouredRectangle` has all the methods of `Rectangle`, plus `getColour` and `setColour`
- It extends `Rectangle` without modifying its code (OCP)

## Polymorphism and subclasses

- Code written for `Rectangle` will work for objects of type `ColouredRectangle`, for example:

  ```
  static Rectangle bigger(Rectangle rect1, Rectangle rect2)
  {
   if(rect1.area()>rect2.area())
        return rect1;
   else
        return rect2;
  }
  ```

  could be called with either or both arguments of type `ColouredRectangle`
- `ColouredRectangle` is a subclass of `Rectangle`

## Actual and Apparent types

- Consider:

```
Rectangle rect1, rect2;
ColouredRectangle colrect;
```

… *some code which assigns values to* `rect1` *and* `colrect`

```
rect2=bigger(rect1, colrect);
```

- At this point we do not know whether the <u>actual</u> class of `rect2` is `Rectangle` or `ColouredRectangle`, it may depend on run-time factors; the <u>apparent</u> class of `rect2` is `Rectangle`
- We can call `area()` on `rect2` but we cannot call `getColour()` on `rect2`, even if `rect2` does alias `colrect`
- All classes are types, but interface types mean some types are not classes

## Assignment and casting

- With:

  ```
  Rectangle rect;
  ColouredRectangle colrect;
  ```

  *… some code which assigns values to* `rect` *and* `colrect`

- `rect = colrect` is allowed
- `colrect = rect` is <u>not</u> allowed
- `colrect = (ColouredRectangle) rect` is used, this is called casting
- If this casting was applied but `rect` did not refer to a `ColouredRectangle` object, a `ClassCastException` would be thrown

## `instanceof` testing

- We can test whether `rect` has been set to an object whose actual type is `ColouredRectangle` using `instanceof`, for example:

```
Colour myColour;
if(rect instanceof ColouredRectangle) {
    ColouredRectangle temp = (ColouredRectangle) rect;
    myColour = temp.getColour();
} else
    myColour = new Colour(0,0,0);
```

- Note casting still has to be done, although it could be expressed without declaring a separate variable:

```
     myColour = ((ColouredRectangle) rect).getColour();
```

## Avoiding `instanceof`

- Using `instanceof` breaks the Dependency Inversion Principle, it should be avoided if the decision can be done "invisibly" through dynamic binding
- For example, if `Rectangle` had the method

```
public Colour getColour() {
 return new Colour(0,0,0);
}
```

- then we could just use:

```
Colour myColour = rect.getColour();
```

- However, this breaks the Open-Closed Principle, because we have modified `Rectangle` to assist in its extension to `ColouredRectangle`

## Overriding

- The previous example showed overriding, this is when a subclass replaces the code of a method in its superclass
- It happens when the subclass has a method with the same signature as one in the superclass, here both `Rectangle` and `ColouredRectangle` had a method with signature:

```
    public Colour getColour()
```

- And we have a call on a variable whose type is the superclass, but which might refer to an object of the subclass, here:

```
    rect.getColour();
```

- Even though `rect` is of type `Rectangle`, if it refers to an object of type `ColouredRectangle`, the code from `ColouredRectangle` is used

## Dynamic Binding with Overriding

The rules are:

- When methods are called on variables, they must fit with the apparent type of the variable - this is checked at compile time (static checking)
- The code used for a method call comes from the actual class, it is decided at run-time
- If there is no overriding code in the actual class, the code from the superclass is used

## Inheritance hierarchy

Consider

```
class ShadedRectangle extends ColouredRectangle {
 private double shade;

 public ShadedRectangle(int h, int w, Colour c) {
  super(h,w,c);
  shade=0.0;
 }

 public Colour getColour() {
  return super.getColour().tint(1-shade);
 }

 public void setLight(double s) {
  shade=s;
 }
}
```

## Direct and indirect subclasses

- Now `ShadedRectangle` is a subclass of `ColouredRectangle` and `ColouredRectangle` is a subclass of `Rectangle`
- A variable of type `Rectangle` can also refer to an object of type `ShadedRectangle`
- So `ShadedRectangle` is a subclass of `Rectangle` as well as a subclass of `ColouredRectangle`
- To distinguish, we say `ShadedRectangle` is a direct subclass of `ColouredRectangle` and an indirect subclass of `Rectangle`
- Or `ColouredRectangle` is the direct superclass of `ShadedRectangle` and `Rectangle` is an indirect superclass of `ShadedRectangle`
- In Java a class may only have one direct superclass

## A Problem with `extends` Inheritance

A square is a rectangle whose height is the same as its width, so:

```
class Square extends Rectangle {
// Example of POOR PRACTICE
 public Square(int size) {
   super(size,size);
 }

 public void setWidth(int w) {
     super.setWidth(w);
     super.setHeight(w);
 }

 public void setHeight(int h) {
     super.setWidth(h);
     super.setHeight(h);
 }
}
```

It keeps the square property, but breaks the “Liskov Substitution Principle”

## The Liskov Substitution Principle (LSP)

- Consider we have `rect` of type `Rectangle` and:

```
int h = rect.getHeight();
int w = rect.getWidth();
rect.setWidth(w+1);
int area = rect.getArea();
```

- Here, `area` should have the value `(w+1)*h`
- But suppose `rect` refers to an object whose actual type is `Square`, then `area` will have the value `(w+1)*(h+1)`
- The problem is that a subclass can override a method with one which can do anything, so long as it has the same signature
- This can cause code to behave in an unexpected way if we did not realise a variable is actually set to refer to an object of a subclass
- The Liskov Substitution Principle says that methods should not be overridden in a way that changes assumptions about their behaviour

## Breaking the LSP

- The assumption broken is that calling `setHeight` on a `Rectangle` changes the value returned by `getHeight` but not the value returned by `getWidth`, and calling `setWidth` on a `Rectangle` changes the value returned by `getWidth` but not the value returned by `getHeight`
- We broke that by overriding `setHeight` and `setWidth` in `Square` to try and keep the property that in a square the height and width are always the same
- There are various ways of expressing the LSP such as “Subclasses must always be substitutable for the class they extend”
- It was named after Barbara Liskov (the first ever female Professor of Computer Science), who first identified the problem
- Another way of expressing it is “Inheritance should only weaken preconditions and strengthen postconditions”

## Coding to the Interface

- One way of avoiding breaking the LSP is not to use inheritance - at least not in the superclass-subclass way indicated by `extends` in Java
- There is less of a problem with the interface-implementation relationship, indicated by `implements` in Java, because that has no code to inherit, so no problem of code in a method being unexpectedly replaced by other code in a subclass
- This is another reason to favour “coding to the interface”
- However, interface types may have the idea of a general contract for their methods which it is specified all classes which implement them should keep to
- Java does not have a mechanism to enforce this
- One reason for keeping code inheritance is that code in a superclass is shared by all subclasses which inherit it - an example of the Do Not Repeat Yourself (DRY) principle
- This can be achieved by using abstract classes, where some methods have code and others have only signatures
- An abstract class is like a class in that any class which extends it cannot extend another class, but like an interface in that no object can have an abstract class as its actual class

## Composition

```
class ColouredRectangle {
  private Rectangle rect;
  private Colour colour;

 public ColouredRectangle(int h, int w, Colour c) {
  rect = new Rectangle(h,w);
  colour=c;
 }
…
 public int getWidth() {
   return rect.getWidth();
 }

 public void setWidth(int w) {
   rect.setWidth(w);
 }
…
}
```

- This is a version of `ColouredRectangle` which is <u>not</u> a subclass of `Rectangle`
- Composition is sometimes more appropriate to use than inheritance: another design decision

# Overriding `toString`

- Every class in Java is a subclass, directly or indirectly, of the class `Object`
- So every class has the method `toString` and `equals`, as these are in class `Object`
- It is common practice to override these methods, for example:

```
class Duck implements Quackable {
 private String myName;
 public Duck(String theName) {
  myName=theName;
 }
 public void quack() {
  System.out.println(myName+": Quack!");
 }
 public String toString() {
  return "Duck: "+myName;
 }
}
```

- Then if `d` refers to a `Duck` object with name `"Donald"`, the method call `System.out.println(d)` will print `Duck: Donald`

## Overriding `equals`

- Taking care in overriding `equals` is an example of keeping to the Liskov Substitution Principle
- If `equals` is not overridden in the class of `obj1`, then `obj1.equals(obj2)` will only return `true` if `obj1` and `obj2` are aliases (that is, they refer to the same object)
- It is common practice to override `equals` so that `obj1.equals(obj2)` will return `true` if `obj1` and `obj2` refer to different object with the same content
- The method `equals` is used in general code, for example if `list` is of type `ArrayList<E>` and `obj` is of type `E` then:
    - `list.indexOf(obj)` returns the lowest value of `i` where `list.get(i).equals(obj)` returns `true`
    - `list.remove(obj)` removes the element at the lowest value of `i` where `list.get(i).equals(obj)` returns `true`
- So, if in a class `SomeThing` the method `equals` was overridden in a way that did not fit in with the general assumption about how `equals` works, that would cause an `ArrayList<SomeThing>` to work in an unexpected way

## Assumptions for `equals`

- One general assumption for overriding `equals` is that `obj1.equals(obj2)` will return the same as `obj2.equals(obj1)`
- Another assumption is that if `obj1.equals(obj2)` returns `true` and `obj2.equals(obj3)` returns `true` then `obj1.equals(obj3)` should return `true`
- However, suppose `obj2` refers to an object of a subclass of `obj1` then dynamic binding means that `obj2.equals(obj1)` would not use the same code as `obj1.equals(obj2)` if `equals` was overridden in the class of the object that `obj2` refers to
- This would be an issue in overriding the method `equals` in the classes `Rectangle` and `ColouredRectangle` given previously

## Issues with overriding `equals`

- Code for overriding `equals` in `Rectangle`:

```
    class Rectangle {
     private int height,width;
     …
     public boolean equals(Object obj)  {
      if(obj instanceof Rectangle ) {
       Rectangle rect = (Rectangle) obj;
       return height==rect.height&&width==rect.width;
      }
      else return false;
    }
```

- The parameter type is `Object` rather than `Rectangle` in order to properly override the `equals` method inherited from `Object`
- Another issue is that `rect1.equals(rect2)` would change its value after a call of `setHeight` or `setWidth`
- It is up to the designer to decide whether `equals` should be a permanent or temporary property with mutable objects

## How to study this material

*Correct use of examples*

*Different people have different styles*

*Practise, practise, practise!*

## Remember what the example was FOR

- You have seen many examples here, but you need to remember what the examples were <u>for</u>
- This is abstraction and generalisation applied to the human mind
- Human beings find abstraction hard, we often find it easier to think about something when we have a specific example
- The design principles here are general principles, they apply to many different situations, on their own they are very abstract
- They have been illustrated with examples, small pieces of code which show in a very simple way one case where a particular design principle applies
- You should remember the design principles, with the examples helping you to understand them
- Don’t try to memorise the examples while forgetting what the principles were!

## Use other sources

- A good way to understand principles you are taught, especially if you are unsure, is to see how other people have explained them
- So it is good to use other sources - textbooks, websites and so on
- If you see the same principle illustrated by a different example, that may help you generalise from the examples towards the principle
- The design principles here are general principles, they apply to many different situations, on their own they are very abstract
- The textbook *Head First Object-Oriented Analysis and Design* shows the same principles illustrated by different examples

## What style works for you?

- Sometimes a text-book which one person finds helpful another person finds not very useful
- This is another reason for trying other sources
- *Head First Object-Oriented Analysis and Design* has a "fun" style, it tries to illustrate the principles in an amusing way, but underneath it is serious
- Some people like an amusing style, others prefer something more plain and "to the point"
- A book I have found very useful for more detail on Design Principles is *Agile Software Development: Principles, Patterns and Practices* by Robert C. Martin (published by Pearson)
- Find what best works for YOU - it may not be what best works for someone else

## Be persistent!

- With abstract material, you can hit a mental barrier - you reach a point where it seems impossible to understand
- With technical material, a lot of what follows depends on what has been covered before, so once you have lost understanding it's hard to catch up
- If this happens, be persistent, don't give up!
- Try to look at the material and understand what it is saying underneath
- Try to avoid a shallow "memorisation" approach
- Learning should be about developing  an understanding, not memorising
- This is especially the case with technical material where the actual facts are few, but the principles are deep and abstract

## Opinions vary

- Experts do not always agree, so you will sometimes find what one expert says is “good practice” another does not think so useful
- Matters of good programming style often generate discussion with opposing “sides”
- However, almost all experts agree on the most basic principles
- Almost all experts agree that code should be broken into small pieces which can be understood on their own, they may differ on how this is best done
- The Agile programming movement came about when a group of expert programmers disagreed with what was then believed by most to be best practice - a strong emphasis on “up front” design
- To what extent should we design before we code is still an open question

## Terminology varies

- Software development is a very new field - it has reached its current point within the space of one human lifetime
- It is fast developing:
  - Computers move from machines used only by experts to machines in everyone's house
  - Object-oriented programming becomes the main paradigm for software development
  - Computers move from mainly "stand-alone" machines towards machines oriented to web access
  - Less "hands on" programming, more use of support tools
- Each change requires a new vocabulary
- It takes a while for the vocabulary to settle down
- So be aware that some terms are used differently by different people

## Programmers

- A novice programmer does not know the rules of good programming
- A good programmer knows the rules of good programming and keeps to them
- A poor programmer knows the rules of good programming but does not keep to them
- An expert programmer knows the rules of good programming, and knows when to break them

## Becoming an expert

- An expert programmer understands the purpose of the rules of good programming, so keeps to those rules when they make sense according to their purpose, breaks them when there are other conflicting reasons
- Becoming an expert takes time and experience
- The best way to learn to program well is practice, practice, practice!
- When you have had some practice, go back and read again what the experts say - it will make more sense
- This is like many skills - playing a musical instrument, speaking a foreign language, photography etc - it requires a mixture of theory and practice

## Summary

*What do I really need to know?*

## Java Revision

- Some of the material in these slides is on material you should have covered in your previous module on Java programming
- It is included because it is an important foundation on the new material, so we need to make sure you have a good understanding of it for you to get a good understanding of the new material
- Make sure you know the basics of object-oriented programming in Java
- You may be asked to write small pieces of code in the exam, and the best way to become confident in doing that is to practice by writing and running your own code to try out different concepts
- Software Engineering is about all the aspects of building large scale software systems, coding is just a small part of it
- Recent trends in Software Engineering (particularly the Agile movement), however, have placed a greater emphasis on coding

## Aliasing and mutability

- Although based on what you should already know as basic aspects of Java, understanding aliasing and its consequences feeds into general program design
- Did you understand what is meant by “mutable” and “immutable” and why aliasing makes these an aspect that needs to be considered in design?
- Did you understand what is meant by “overriding `equals`” and how this fits in with aliasing and mutability?

## Object Oriented Programming

- Object oriented programming languages introduce new complexities into coding which have been found useful because they fit closely into design concepts
- Did you understand the idea of program design as about building a model of communicating entities, where communication is actually done by method calls, and entities are represented by code objects?
- Did you understand how the structure of object oriented programming helps you break a program into parts which can be considered independently?
- This idea is captured in the terms “locality” and “modifiability”

## Type Hierarchy

- Did you understand the idea of abstraction and generalisation as key aspects of program design?
- Did you understand how Java's type hierarchy helps support abstraction and generalisation?
- Did you understand how to change the way code works by using inheritance rather than changing the code in an actual class?
- Did you understand the use of interface types in Java, in particular the way they lead to types being seen in terms of capabilities rather than implementation?
- Much of this is summarised in the idea that you should "code to the interface"
- Do you understand the idea of "dynamic binding" in object-oriented programming languages, and how this leads to the Liskov Substitution Principle?

## Six Design Principles

- Here are the six named Design Principles that were covered:

|  |  |
|---|---|
| Single Responsibility Principle | (SRP) |
| Open-Closed Principle | (OCP) |
| Do not Repeat Yourself principle | (DRY) |
| Liskov Substitution Principle | (LSP) |
| Interface-Segregation Principle | (ISP) |
| Dependency Inversion Principle | (DIP) |

- Do you understand each of these to the point where you could write a brief explanation if asked?
- Do you understand each of these to the point where you could write your own example of the principle being kept to or the principle being broken, if asked?

## SOLID Design Principles

- The “Do not Repeat Yourself” principle is really just another way of expressing the “Open Closed” principle
- This leaves five principles, and from the initials of each, they are often called the SOLID design principles
- So for more explanation of them, and more examples to demonstrate them, try searching for “SOLID design principles”