Software Engineering: Design Patterns

EBU5304 Software Engineering 2018/19

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

A problem

- Suppose we have a method which takes a `Quackable` object:

  ```
      public Value process(…,Quackable q,…)
  ```

  a variable of type `Quackable`, set to an object:

  ```
      Quackable  q1 = new Duck("Donald");
  ```

  and a call to the method with that variable as an argument:

  ```
      Value v = t.process(…,q1,…);
  ```

- The method is in a class called `Thing`, it returns a `Value`
- `Thing` and `Value` are otherwise unknown types
- How can we count the number of calls of method `quack()` on the `Quackable` object when this method call takes place without changing any code in class `Thing` or class `Duck` ? Remember the OCP

A solution

```
Quackable  q1 = new Duck("Donald");
QuackCounter q2 = new QuackCounter(q1);
Value v = t.process(…,q2,…);
int quackcount = q2.getCount();
```

- In order for `q2` to replace `q1` as an argument, it must also implement `Quackable`
- The code for the method `process` then works as before, but calls to `quack()` are on `q2` rather than `q1`
- Each call to `quack()` on `q2` calls `quack()` on `q1` and increases a count in `q2`
- The call `getCount()` on `q2` returns the value of that count

```
class QuackCounter implements Quackable {
 private Quackable  myQuacker;
 private int count;

 public QuackCounter(Quackable q) {
  myQuacker=q;
 }

 public int getCount() {
   return count;
 }

 public void quack()
 {
  count++;
  myQuacker.quack();
 }
}
```

The Decorator Design pattern

- This is an example of the Decorator design pattern
- The Decorator design pattern involves
  - A class which implements an interface
  - Inside it a variable of the same interface type whose value is set in the constructor
  - Each method from the interface has code which calls the same method with the same arguments on that variable, plus some extra work
- In this example, the extra work is to increase the value of the count by one
- The effect is to “decorate” the methods of an object of the interface type with the code that does the extra work

Wrapper design patterns

- The Decorator design pattern is an example of a general sort of pattern called a wrapper pattern
- A wrapper pattern is any pattern where an object (called the inner object) is referred to by a variable inside another object (the outer object), and each method call on the outer object results in a method call on the inner object
- So the outer object “wraps” the inner object
- In our example, `QuackDecorator` was the outer object wrapping an inner object which could be any object of type `Quackable`
- In the Decorator pattern, the inner object and outer object must both be of a class which implements the same interface, in this case `Quackable`
- Another wrapper pattern is the Adapter pattern, which involves an outer object which implements an interface wrapping an inner object which is of a class that does not implement the interface

The Adapter design pattern

- The Adapter design pattern "adapts" an object of one class to fit into an interface when the object is <u>not</u> of a type which implements the interface
- It works by "wrapping" the object as an inner object in an outer object which is of a class that does implement the interface
- Each method in the interface has to be implemented by code which performs what is regarded as the "equivalent" in the class of the inner object
- Example: class `Goose` has a method

```
    public void honk() …
```

- which is judged equivalent to the `quack()` method of interface `Quackable`

Fitting a class into an interface

```
class Goose
{
 private String myName;
 public Goose(String theName)
 {
   myName=theName;
 }
 public void honk()
 {
  System.out.println(myName+": Honk! ");
 }
}
…

Goose g = new Goose("Gertrude");
Quackable  q3 = new GooseAdapter(g);
Value v = t.process(…,q3,…);
```

A simple example of an adapter

```
  class GooseAdapter implements Quackable
  {
   private Goose myGoose;
   public GooseAdapter(Goose aGoose)
   {
     myGoose=aGoose;
   }
   public void quack()
   {
    myGoose.honk();
   }
  }
```

…

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

Why not just change the code?

- Other code may already make use of class `Duck`, class `Goose`, or method `process`, we cannot change the code for them without having to check whether the change causes that code to stop working properly (OCP)
- If we added code to count the number of calls to the method `quack` made in the call to the method `process`, we are making that method do two separate tasks. The `QuackCounter` wrapper separates out those two separate tasks (SRP)
- If we wanted to count the number of calls to the method `quack` made in a call to another method which takes an argument of type `Quackable`, we can just use the wrapper we already have (DRY)
- We may not want to pass full access to `Goose` objects to other code, passing the `GooseAdapter` object passes only the ability to call the method `honk` on a `Goose` object (DIP)

Adapter using Inheritance

```
class QuackingGoose extends Goose implements Quackable
{
   public QuackingGoose(String name)
   {
     super(name);
    }

   public void quack()
   {
    honk();
    }
}
```

- This is another sort of Adapter, this time using inheritance rather than delegation. Here a `QuackingGoose` object IS-A `Goose` object, whereas previously a `GooseAdapter` HAS-A `Goose` object
- Passing a `QuackingGoose` object to other code passes access to all `Goose` methods, whereas passing a `GooseAdapter` object only passes access to `honk` called indirectly through `quack`.

Design patterns

- These are just two examples of a “design pattern”, there are many more established design patterns
- Design patterns solve a particular problem, here:
  - How can we add an additional task to an existing method without changing the code of that method?
  - How can we fit a class into an interface which it is not declared as implementing without changing the code of that class?
- Design patterns are general, we can re-use the same pattern in many different circumstances
- Design patterns go beyond what is provided directly in the programming language
- Design patterns often involve a use of code which a novice programmer probably would not think of
- Design patterns provide us with a “vocabulary”, here we can talk to other programmers about using a “decorator” or an “adapter” and they will know what we mean

The Composite Design Pattern

Another example of a wrapper pattern, this time a collection of objects of a type is wrapped and given the behaviour of a single object of that type:

```
class QuackComposite implements Quackable
{
 private List<Quackable> quackers;
 private int count;

 public QuackComposite()
 {
  quackers = new ArrayList<Quackable>();
 }
 public void addQuacker(Quackable q)
 {
   quackers.add(q);
 }
 public void quack()
 {
  for(Quackable aQuacker : quackers)
     aQuacker.quack();
 }
}
```

Immutable View

- We have seen previously how problems can be caused by aliasing: two different variables that refer to the same object,
- A method call made on one of the variables that causes a change of state to the object it refers to means the object the other variable refers to also changes state as it is the same object
- For example, exposing the representation was when a reference was passed that meant an internal variable of an object was aliased by another variable outside that object
- This problem can be avoided by passing an “immutable view” of an object rather than a direct reference
- This is another example of the general principle of a wrapper design pattern. It involves calling the same method on the inner object if it does not make a change, but for methods that do make changes throwing an exception instead

```
class ImmutableRectangle implements Rectangle {
 private Rectangle myRectangle;

 public ImmutableRectangle(Rectangle r) {
  myRectangle=r;
 }

 public int getWidth() {
   return myRectangle.getWidth();
 }

 public int getHeight() {
   return myRectangle.getHeight();
 }

 public int area() {
  return myRectangle.area();
 }

 public void setWidth(int w) {
   throw new UnsupportedOperationException();
 }

 public void setHeight(int h) {
   throw new UnsupportedOperationException();
 }
}
```

The Observer Pattern

- The example given earlier of counting the number of calls of the method `quack` could also be covered by the Observer design pattern
- However, to cover it properly, we will start with a different example
- The Observer design pattern is intended to cover cases where there is a link between two classes such that an action on an object of one of the classes is “observed” by objects of the other class
- Its most common use is to connect objects which store data or perform actions to other objects which deal with “input/output”
- Here “input/output” could mean writing data to a file or database, or displaying on a graphical user interface

Model-View-Controller

- It is generally agreed that one of the most basic divisions of responsibility in a software system is between those aspects which interact with a human user - the “view” - and those aspects which deal with the details of what is happening internally - the “model”. The “controller” is the link between these two parts
- For example, in a computer game, one part of the system will hold the details of the pieces and characters of the game, and make them change according to the rules of the game, another part will produce the screen picture that the human player of the game sees
- If a single object held the internal details of characters in the game, and also dealt with displaying their representation on the screen, that would be breaking the Single Responsibility Principle
- It would lead to code that is hard to understand and change
- It would make it difficult to change the screen appearance of the game as the code that produces the screen appearance is mixed up with the code that deals with the other aspects of the game

```
class DogBot {
 protected int hungry,tired;
 private String name;

 public DogBot(String nm,
               int h,int t) {
  name=nm;
  hungry=h;
  tired=t;
 }

 public boolean eat() {
  if(hungry>6) {
      hungry-=3;
      return true;
  }
  else
     return false;
 }

// continued …
```

```
public void rest() {
  hungry++;
  tired-=2;
 }

 public void play() {
  hungry+=2;
  tired+=3;
 }

 public String getName() {
  return name;
 }

 public String noise() {
  if(hungry>8&&tired<11)
     return name+": whine ";
  else if(tired>7&&tired>hungry)
     return name+": snore ";
  else
     return name+": woof ";
 }
}
```

Java’s support for the Observer Pattern

- Java provides API code to give direct support for the Observer pattern
- The abstract class `Observable` provides code for methods for objects to maintain a list of observers and notify them:

```
public void addObserver(Observer o)
public void deleteObserver(Observer o)
public void notifyObservers(Object arg)
protected void setChanged()
```

- The interface `Observer` provides a method which is called when the method `notifyObservers` is called on an `Observable` object which has had `setChanged` called on it - it is called on all `Observer`s in its list:

```
public void update(Observable obj, Object action)
```

```
class DogBot
extends Observable{

 // as before

 public boolean eat() {
  if(hungry>6) {
   hungry-=3;
   setChanged();
   notifyObservers("eat");
   return true;
  }
  else
   return false;
 }

// continued …
```

```
public void rest() {
  hungry++;
  tired-=2;
  setChanged();
  notifyObservers("rest");
 }

 public void play() {
  hungry+=2;
  tired+=3;
  setChanged();
  notifyObservers("play");
 }

// as before
```

```
class DogWatcher implements Observer {
 private String name;

 DogWatcher(String nm) {
  name=nm;
 }

 public void update(Observable obj, Object action) {
  if(obj instanceof DogBot) {
      DogBot dog = (DogBot) obj;
      System.out.print("** "+name+" observes ");
      System.out.println(dog.getName()+" "+action);
     }
 }

 public String toString() {
  return "Dog Watcher: "+name;
 }
}
```

```java
DogBot patch = new DogBot("Patch",4,2);
DogBot rover = new DogBot("Rover",9,6);
DogWatcher jim = new DogWatcher("Jim");
DogWatcher fred = new DogWatcher("Fred");
rover.addObserver(fred);
patch.addObserver(jim);
System.out.println("Patch plays");
patch.play();
System.out.println("Rover plays");
rover.play();
patch.addObserver(fred);
patch.play();
rover.deleteObserver(fred);
rover.play();
if(patch.eat())
   System.out.println(" Patch eats");
else
   System.out.println(" Patch decides not to eat");
```

Notes on Java's Observer code

- We do not have to write any code to maintain the list of observers, it is inherited from `Observable`
- It is flexible, an `Observable` object can have any number of `Observer` objects, an `Observer` object can observe any number of `Observable` objects
- The code for the `Observable` object passes whatever information it likes to the `Observer` objects ("push")
- The code for the `Observer` object can get additional information by calling public methods on the `Observable` objects ("pull")
- The code which extends `Observable` cannot access the `Observer` objects directly, an object of a class which extends `Observable` does not "know" what its observers are

The Open-Closed Principle

- Remember the Open-Closed Principle (OCP)?
- “Modules should be open for extension but closed for modification”
- We have not followed that principle here because we have modified the class `DogBot` in order to make `DogBot` objects observable
- Also, we could not make `DogBot` extend `Observable` if it already extended another class, because classes can only extend one other class
- We can get round this by using the Decorator pattern

```
class ObservableDogBot
extends Observable
{
 private DogBot myDogBot;

 public ObservableDogBot(DogBot d)
 {
  myDogBot = d;
 }

 public boolean eat()
 {
  if(myDogBot.eat()) {
      setChanged();
      notifyObservers("eat");
      return true;
  }
  else
     return false;
 }

// continued …
```

```
public void rest() {
  myDogBot.rest();
  setChanged();
  notifyObservers("rest");
 }

 public void play() {
  myDogBot.play();
  setChanged();
  notifyObservers("play");
 }

 public String getName() {
  return myDogBot.getName();
 }

 public String noise() {
  return myDogBot.noise();
 }
}
```

Programming to the interface

- We should have made `DogBot` an interface type:

```
    interface Dogbot {
        public boolean eat();
        public void rest();
        public void play();
        public String noise();
        public String getName();
    }
```

- with `class PlainDogBot implements DogBot` (code otherwise as for `DogBot`, but using name `PlainDogBot`)
- Then we could have

```
    class ObservableDogBot
    extends Observable implements DogBot
```

  and no other changes to code except in constructor calls

Separation of model from view

Using this technique to create observable `DogBot`s means:

- The actions of a `DogWatcher` object on observing a `DogBot` object are kept entirely separate from the `DogBot` code
- A `DogWatcher` object could keep a record of what it observes in a collection object
- It could write it to a file
- It could display it on a graphical user interface
- It could keep a count of the number of actions observed

```
class DogReporter implements Observer
{
  private List<String> list;

  DogReporter()
  {
   list = new ArrayList<String>();
  }

  public void update(Observable obj, Object action)
  {
   if(obj instanceof DogBot)
      {
       DogBot dog = (DogBot) obj;
       list.add(dog.getName()+" "+action);
      }
  }

  public List<String> report()
  {
   List<String> oldlist=list;
   list=new ArrayList<String>();
   return oldlist;
  }
}
```

```
ObservableDogBot patch =
   new ObservableDogBot(new PlainDogBot("Patch",4,2));
ObservableDogBot rover =
  new ObservableDogBot(new PlainDogBot("Rover",9,6));
DogReporter reporter1 = new DogReporter();
DogReporter reporter2 = new DogReporter();
rover.addObserver(reporter1);
patch.addObserver(reporter2);
patch.play();
rover.play();
patch.addObserver(reporter2);
patch.play();
rover.deleteObserver(reporter2);
rover.play();
…
List<String> report= reporter1.report();
System.out.println("Reporter 1 reports:");
for(String item : report)
   System.out.println("  "+item);
```

```
class Counter implements Observer
{
 private int count;

 Counter()
 {
  count=0;
 }

 public void update(Observable obj, Object action)
 {
   count++;
 }

 public int getCount()
 {
  return count;
 }

 public void reset()
 {
  count=0;
 }
}
```

```
ObservableDogBot patch =
   new ObservableDogBot(newPlainDogBot("Patch",4,2));
ObservableDogBot rover =
  new ObservableDogBot(new PlainDogBot("Rover",9,6));
Observer counter1 = new Counter();
Observer counter2 = new Counter();
rover.addObserver(counter1);
patch.addObserver(counter2);
patch.play();
rover.play();
```

Subclasses

```
class GreedyDogBot extends PlainDogBot
{
 public GreedyDogBot(String nm,int h,int t)
 {
   super(nm,h,t);
 }
 public boolean eat()
 {
  if(hungry>6)
    {
     hungry-=2;
     return true;
    }
  else if(hungry>3)
    {
     hungry-=1;
     return true;
    }
  else
     return false;
 }
}
```

Factory Methods

- We cannot write a constructor which returns either a `PlainDogBot` or a `GreedyDogBot`
- But we can write a method which does that:

```
public static DogBot makeDogBot(String nm,
   int h, int t, boolean greedy)
{
 if(greedy)
       return new GreedyDogBot(nm,h,t);
 else
       return new PlainDogBot(nm,h,t);
}
```

- If this is a static method in class `GreedyDogBot`, and we have a `boolean` variable called `val`, a call to it might look like:

```
DogBot myDog = GreedyDogBot.makeDogBot("Fatty",9,6,val);
```

Use of Factory Methods

- Factory methods are normal methods which work like constructors, they return new objects
- Constructor calls are preceded by the word `new`, factory methods are called like any other method
- Unlike constructors, factory methods can have an interface type as their return type:

```
public static Quackable makeQuacker(String nm, boolean fake)
{
  if(fake) return new DuckCall(nm);
  else return new Duck(nm);
}
```

- They are used when we want to determine the actual type of a new object at run time
- They can be used to <u>hide</u> the actual type of the new object

Factory Methods for Wrapping Objects

- A common use of factory methods is to put a wrapper around an object.
- A factory method could produce a new wrapped object, or an object which puts a wrapper around an existing object
- Here is a factory method for producing a new `Duck` object with a wrapper:

```
static Quackable makeCountedDuck(String name)
{
 Duck wrappedDuck = new Duck(name);
 return new GloballyCountedQuacker(wrappedDuck);
}
```

- The difference between `GloballyCountedQuacker` and the previous `QuackCounter` is that here there is just one count for all the calls of the method `quack()` on all the `Quackable` objects returned

```
class GloballyCountedQuacker implements Quackable
{
 private Quackable  myQuacker;
 private static int count=0;

 public GloballyCountedQuacker(Quackable q)
 {
  myQuacker=q;
 }

 public int getCount()
 {
   return count;
 }

 public void quack()
 {
  count++;
  myQuacker.quack();
 }
}
```

Factory objects

- Factory methods may be `static` methods, or they may be called on objects
- An object which is designed to have factory methods called on it is called a Factory Object
- A factory object is used when we want to delegate the construction of new objects
- Factory object enable us to write generalised code whose effects depend on the particular factory object passed into it
- Some of the aspects of the objects created by a factory method may be stored in the factory object
- A factory object may be used to hide the real type of the objects it returns

Consider:

```
interface DogFactory {
  public DogBot makeDogBot(String name);
}
```

with:

```
class PlainDogFactory implements DogFactory {
  private int hungry,tired;

  public PlainDogFactory(int h, int t) {
    hungry=h;
    tired=t;
  }

  public DogBot makeDogBot(String name) {
    return new PlainDogBot(name,hungry,tired);
  }
}
```

This is a class of objects which make `DogBot`s with fixed initial `hungry` and `tired` values:

```
DogFactory factory = new PlainDogFactory(4,2);
DogBot dog1 = factory.makeDogBot("Patch");
DogBot dog2 = factory.makeDogBot("Rover");
```

Now consider:

```
class SpyDogFactory extends PlainDogFactory {
  private Observer spy;

  public SpyDogFactory(int h, int t, Observer obs) {
    super(h,t);
    spy=obs;
  }

  public DogBot makeDogBot(String name) {
    DogBot d1 = super.makeDog(name);
    DogBot d2 = new ObservableDogBot(d1);
    d2.addObserver(spy);
    return d2;
  }
}
```

This is a class of objects which make `DogBot`s which are all observed by an `Observer` object set when the `SpyDogFactory` object is created:

```
Observer boss = new DogWatcher("Matthew");
DogFactory factory = new SpyDogFactory(4,2,boss);
DogBot dog1 = factory.makeDogBot("Patch");
DogBot dog2 = factory.makeDogBot("Rover");
```

- Here is something similar to the `GloballyCountedQuacker` example, but with a separate object keeping the count:

```
class CountedQuacker implements Quackable
{
 private Quackable myQuacker;
 private GroupQuackCounter myCounter;

 public CountedQuacker(Quackable q, GroupQuackCounter c)
 {
  myQuacker=q;
  myCounter=c;
 }

 public void quack()
 {
  myCounter.addCount();
  myQuacker.quack();
 }

 public int getCount()
 {
  return myCounter.getCount();
 }
}
```

- This can make new counted `Quackable` objects and wrap existing ones:

```
class GroupQuackCounter
{
 private int count=0;
 public CountedQuacker makeDuck(String name)
 {
  return wrap(new Duck(name));
 }
 public CountedQuacker wrap(Quackable q)
 {
  return new CountedQuacker(q,this);
 }
 public void addCount()
 {
  count++;
 }
 public int getCount()
 {
  return count;
 }
}
```

The Singleton Design Pattern

- Another reason for using factory methods is that a constructor must always return a new object, but factory methods can return an existing object instead of a new object. Here is an example:

```
class SingleDogFactory implements DogFactory
{
 private int hungry,tired;
 private DogBot theOnlyDogBot;

 public SingleDogFactory(int h,int t,String name)
 {
  theOnlyDogBot = new PlainDogBot(name,h,t);
 }

 public DogBot makeDogBot(String name)
 {
  return theOnlyDogBot;
 }
}
```

- Here is a sneaky way to hide the aliasing:

```
class AliasDogBot implements DogBot {
 private String name;
 private DogBot realDogBot;

 public AliasDogBot(String nm, DogBot dog) {
  name=nm;
  realDogBot=dog;
 }

 public String getName() {
  return name;
 }

 public boolean eat() {
  return realDogBot.eat();
 }

 public void rest(){
  realDogBot.rest();
 }

 public void play() {
  realDogBot.play();
 }

 public String noise() {
  return realDogBot.noise();
 }
}
```

- Replace `return theOnlyDogBot` in `SingleDogFactory` by `return new AliasDogBot(name,theOnlyDogBot);`

Liskov Substitution Principle (LSP)

- We can see here the effect of overriding in subclasses leading to unexpected behaviour, which is what the LSP is about.
- For a simpler example, consider:

```
DogBot dog = …
int count=0;
while(dog.eat())
   count++;
System.out.print(dog.getName());
System.out.println(" eats "+count+" times");
```

with the method `eat` overridden:

```
public boolean eat() {
 if(hungry>6) {
    hungry+=3;     // Eating does not reduce hunger here!
    return true;
   }
 else
    return false;
}
```

The Object Pool Design Pattern

- Object Pool (also called Intern), like Singleton, can return an object that was created before
- However, instead of one object it keeps a set of objects
- It can be used when the objects returned are immutable:

```
class ColourMaker {
  private Map<Colour,Colour> colours;

  public ColourMaker {
   colours = new HashMap<Colour,Colour>();
  }

  public Colour makeColour(int red, int green, int blue){
   Colour col1 = new Colour(red,green,blue);
   Colour col2 = colours.get(col1);
   if(col2==null) {
      colours.put(col1,col1);
      return col1;
   }
   else return col2;
  }
 }
```

The Strategy Design Pattern

- Suppose we want to call the various operations possible on a List of `DogBot`s. Instead of writing a separate method for each we can generalise by using the idea of a function object, which is an object whose only job is to carry out an action.
- Here is the interface type for such an object:

```
interface DogAction {
 public void action(DogBot d);
}
```

- Here is the generalised code that uses it:

```
void doIt(List<? extends DogBot> list, DogAction act)
{
 for(DogBot dog : list)
     act.action(dog)
}
```

- Generalising code by turning an action into a parameter is called the Strategy Design Pattern
- Note also the use of generic typing and the “for-each” loop here

Function Objects

- Here is the class for the function object for calling `play` on a `DogBot`:

```
class MakePlay implements DogAction
 {
   public void action(DogBot d)
    {
     d.play();
    }
 }
```

- Here is the class for the function object for calling `eat` on a `DogBot`:

```
class MakeEat implements DogAction
 {
   public void action(DogBot d)
    {
     d.eat();  // Ignores the return value
    }
 }
```

- Here is a call which results in `play` being called on every object in the list `dogList`:

```
doIt(dogList, new MakePlay())
```

Function Objects with State

- Here is a class which describes a function object which repeats an action:

```
class MultiAction implements DogAction
{
 private DogAction myAction;
 private int ntimes;

 public MultiAction(DogAction act, int n)
 {
  myAction=act;
  ntimes=n;
 }

 public action(DogBot d)
 {
  for(int i=0; i<ntimes; i++)
      myAction.action(d);
 }
}
```

- So `act = new MultiAction(new MakePlay(),3)` sets `act` to refer to a function object where `act.action(dog)` will cause the method `play` to be called on the `DogBot` referenced by `dog` three times

Comparator Objects

- As covered in Week 3, Java has an interface

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

- If `comp` is of type `Comparator<Thing>` and `t1` and `t2` are of type `Thing`, then `comp.compare(t1,t2)` return an integer similar to `t1.compareTo(t2)` when `Thing` implements `Comparable<Thing>`
- We can use `Comparator` objects to order in a different way than natural order, for example `String`s ordered by length rather than alphabetically
- This is another example of the Strategy design pattern

Comparator Objects

- If we have:

```
class LengthComparer implements Comparator<String>
{
 public LengthComparer()
  {
  }

 public int compare(String str1, String str2)
  {
   return str1.length()-str2.length();
  }
}
```

- and we have `list` of type `List<String>` then:

```
Collections.sort(list);
```

sorts `list` in alphabetical order

```
Collections.sort(list, new LengthComparer());
```

sorts `list` in order of length of `String`

The State Design Pattern

- The State design pattern is used when we want to change the way we represent a type of object dynamically
- Consider the idea of a set, it is a collection in which each possible element is either in the collection or it is not, there is no concept of elements occurring multiple numbers of times or having a position in the collection.
- A mutable set is one where elements can be added or removed. Here is an interface type for a mutable set of integers:

```
interface IntSet
{
 public void add(int n);
 public void remove(int n);
 public boolean contains(int n);
}
```

- The set remains unchanged if an integer is added when it is already a member, or if it is removed when it is not a member - the definition in this interface does not report an error

- An efficient implementation for sets of integers which have a fixed range of possible members uses an array of `boolean`s:

```
class BoolIntSet implements IntSet {
 private boolean[] arr;

 public BoolIntSet(int range) {
  arr = new boolean[range];
 }

 public void add(int n) {
  arr[n]=true;
 }

 public void remove(int n) {
  arr[n]=false;
 }

 public boolean contains(int n) {
  return arr[n];
 }

 public int getRange() {
  return arr.length;
 }
}
```

- Suppose we have another implementation of `IntSet` called `OtherIntSet` which can cope with integers in any range, then we can have an implementation which switches:

```
class FlexibleIntSet implements IntSet {
 private boolean usesbool;
 private IntSet state;
 protected int limit, origrange;

 public FlexibleIntSet(int range) {
  state = new BoolIntSet(range);
  usesbool=true;
  limit=range;
  origrange=range;
 }

 public void add(int n) {
  if(n>limit)
     limit=n;
  if(usesbool&&n==limit) {
     IntSet state1 = new OtherIntSet();
     for(int i=0; i<limit; i++)
        if(state.contains(i))
           state1.add(i);
     state = state1;
     usesbool=false;
  }
  state.add(n);
 }

 public void remove(int n) {
  state.remove(n);
 }

 public boolean contains(int n) {
  return state.contains(n);
 }
}
```

```
class OtherIntSet implements IntSet {
 private int[] array;
 private int count;
 private static final int INITSIZE=1000;

 public OtherIntSet() {
  array = new int[INITSIZE];
 }

 public void add(int n) {
  if(!contains(n)) {
      if(count==array.length) {
          int[] replacearray = new int[array.length*2];
          for(int i=0; i<count; i++)
             replacearray[i]=array[i];
          array=replacearray;
         }
      array[count++]=n;
     }
 }

 public void remove(int n) {
  for(int i=0; i<count; i++)
      if(array[i]==n) {
          array[i]=array[--count];
          break;
         }
 }

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

The Bridge Design Pattern

- Suppose `FlexibleIntSet` had subclasses:

```
class ExtendedIntSet extends FlexibleIntSet
{
  public ExtendedIntSet(int range)
 {
  super(range);
  }

 … // Extra operations

}
```

- This is an example of the Bridge design pattern
- This is where there are two separate “crosscutting” class hierarchies
- If there was just one hierarchy, there would need to be two separate classes for `ExtendedIntSet`, one for where it is implemented using an array of `boolean`s, the other for the other set implementation

```
class ExtendedIntSet extends FlexibleIntSet
{

 …

 IntSet intersection(IntSet set)
 {
  IntSet result = new FlexibleIntSet(origrange);
  for(int i=0 i<limit; i++)
     if(set.contains(i)&&this.contains(i))
        result.add(i);
  return result;
 }

 …


}
```

The Flyweight Pattern

- The Flyweight design pattern is where objects have an immutable part which can be shared. The `ColouredRectangle` example we saw previously would be an example of the Flyweight pattern if the `Colour` part of a `ColouredRectangle` was produced using a `ColourMaker` object as given previously:

```
class ColouredRectangle extends PlainRectangle {
 private static ColourMaker maker = new ColourMaker();
 private Colour colour;

 public ColouredRectangle(int h, int w, int red, int green, int blue)
 {
  super(h,w);
  colour=make.makeColour(red,blue,green);
 }

 public Colour getColour() {
   return colour;
 }

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

Design Pattern Issues

Why are design patterns important?

Summary

Design Patterns Represent Experience

- Design patterns are “tricks” which experienced programmers have come to use
- Each design pattern is a way of putting together code to solve a particular sort of problem which it has been noted is common enough to set it down as a general technique
- Design patterns are applicable in any programming language, but the idea developed in the field of object-oriented programming, and the common design patterns are most easily implemented in object-oriented languages
- Design patterns are often a way to help keep programs in line with main principles of good program design
- Learning the main design patterns is a “fast track” to becoming an experienced programmer

Design Patterns Structure Code

- Design patterns give ways of structuring code into parts with clear separate responsibilities, in line with the Single Responsibility Principle (SRP)
- For example, we have seen how various tasks associated with a method call can be divided into parts using wrapper techniques so each wrapper is responsible for one task
- Design patterns enable code to be written in a generalised way, in line with the Open Closed Principle (OCP)
- For example, we have seen how aspects of a method can be delegated to a separate object passed in through a parameter, with different objects giving different effects, we saw this with factory objects and function objects

Design Patterns Reduce Dependency

- Design patterns patterns enable details connected to implementation to be hidden from code which uses objects, in line with Dependency Inversion Principle (DIP)
- For example, with the Object Pool and Flyweight design patterns, the code which uses objects does not have to manage when to allow aliasing to save space
- With the State design pattern, the code which uses objects does not have to manage the transition between one representation and another when for efficiency reasons it is a good idea to change
- With the Observer design pattern the code which uses objects does not have to manage the details of communicating the effects of method calls on objects to code which displays or reports those effects

Design Patterns as a Vocabulary

- It is hard to think and talk about something if we do not have a name for it
- Design patterns give a vocabulary which experienced programmers can use to communicate with each other when designing programs
- For example, if I say “Pass objects of type A into this method which requires objects of type B by using an adapter”, or “Create new objects of type C using a factory object which sets their size”, you should now have a good idea of the sort of program design I am suggesting
- If I did not have words and phrases like “adapter” and “factory method”, it would be <u>much</u> harder for me to explain what I meant

Be Cautious with Design Patterns

- Do not “over-design” your code: write code which is suitable for the situation you have now, do not make it more complicated than necessary
- Over-use of design pattern techniques can lead to code which is more complex than necessary
- It is often better to start with simple code and “refactor” it using one of the design patterns when more requirements are added later
- Good attention to the principles of good design means we will naturally produce code which is easy to refactor later on

The Design Patterns we have covered

- Decorator
- Adapter
- Composite
- Immutable View
- Observer
- Factory method
- Factory object
- Singleton
- Object Pool
- Strategy
- State
- Bridge
- Flyweight