AP(IT): Design Patterns

Simon Rogers

23rd Feb 2015

Introduction

What are patterns?

- Sets of rules that, if followed, allow easy code understanding and re-use
- Separation of tasks: e.g. *iterators* decouple algorithms from data objects
- Can operate at many levels:
  - Whole applications (e.g. Model-View-Controller)
  - Small parts of an application (e.g. iterators)

Some useful patterns

Iterators

- Many algorithms require the ability to move through a collection of objects
  - Finding, sorting, etc
- We'll concentrate on finding - i.e. we'd like to find the position of some object in the collection
  - Can we make a general finder that will work with *any* collection of objects?
  - Yes: we just need to define various operations that the collection should be able to perform (*Design Pattern*)
    - What do we need?

Java's `Iterator` interface

- For an object to be iterable in Java, it must implement the `Iterator` interface
- To do so, it must implement the methods:
  - `hasNext()` (boolean - returns true if there is at least one more element in the iterator)
  - `next()` (type `T` - returns a reference to the next object in the iterator)
  - e.g. `ItArray` and `Finder`

Java's `for` loops

- If you want to be able to put your type in the new `for` loop: `for(Integer a: ItArray){}`
- Your object must also implement `Iterable` and its only method:
  - `Iterator<E> iterator()`
- e.g. `ItCollection` and `Finder2` (or `ItArray2` and `Finder3`)
  - Note it's common to wrap your object in another that implements `Iterable`

List iterators

- `Iterator` has a sub-interface: `ListIterator` with more methods:
  - See ListIterator
  - See ArrayList for an object that implements `ListIterator` and `Iterator`

The composite pattern

- In some applications we might need to perform the same operation on objects or groups of objects. e.g.
  - file system question in lab exam
  - items and groups of items in an online shop
- Taking the shop example: imagine all items in the shop have a price. Items can be purchased individually or in multi-packs (i.e. groups of items at once). Any particular customer has a percentage discount that needs to be applied when the price is computed. Not all items are discounted.

The composite pattern - definition

- There are three interfaces in the composite pattern
  - `component`: this is the highest level of abstraction. It defines all of the methods we want to be able to invoke on objects or groups of objects. Normally an interface.
  - `leaf`: individual objects in the system (items that can be purchased). Implements everything in `component`
  - `composite': class for groups of objects. Implements everything incomponentas well asaddandremove‘` method for adding and removing objects.

Composite pattern - shop example - component

- There is one method we want to be able to invoke on objects or composites: `compPrice(Double discount);`
- `component` is therefore:

```
public interface ShopComponent {
    public Double compPrice(Double discount);
}
```

Composite pattern - shop example - leaf

- Each item needs a name, a base price and a boolean that says whether it can be discounted or not:

```
public class ShopLeaf implements ShopComponent {
    private Double basePrice;
    private Boolean canBeDiscounted;
    public ShopLeaf(Double base,Boolean disc) {
        basePrice = base;
        canBeDiscounted = disc;
    }
    public Double compPrice(Double discount) {
        if(canBeDiscounted) {
            return basePrice*(1.0-(discount)/100.0);
        }else {
            return basePrice;
        }
    }
}
```

Composite pattern - shop example - `composite`

- A composite needs a structure to hold its children (leaves) as well as additional methods for adding or removing a child.
- See `ShopComposite.java`
- `CompositeExample.java` gives an example

Composite pattern - summary

- Final implementation of methods is usually deferred to the leaves.
- Useful in any application where objects can be in a hierarchy.
- What would we need to do to allow composites of composites?

The factory pattern (by popular request)

- Note that factories blur the line of what is and isn't a pattern, depending on their use. Doesn't matter though, they can be useful however you use them!
- Most objects are created via a call to new, which calls the `constructor`
- Constructors can be overloaded (defined in multiple-forms)
- Not always possible to get what you want...

Factory methods

- Methods that create new objects
- Nice example from Wikipedia
- Complex numbers can be represented as a real and complex co-ordinate or as an angle and a radius. We'd like to store as co-ordinates but should allow creation either way.
- Both ways require two doubles so we can't overload constructors.
- Answer: create factory methods:

Factory methods - complex numbers

```
class Complex {
    public static Complex fromCart(double r,double i) {
        return new Complex(r,i);
    }
    public static Complex fromPolar(double r, double a) {
        return new complex(r*cos(a),r*sin(a));
    }
    private Complex(double r,i) {
        ...
        // Note private constructor!
    }
}
```

Factory pattern

- Separates use of objects from their creation
- E.g. We have some abstract class or interface (e.g. shape)
- We build a program to manipulate these objects
- Don't want to change it if new objects are added
- All creation is done via a separate factory class

Factory pattern - example

- Drawing program
- Users type in a shape type to draw
- Program passes string to factory
- Factory creates object and returns it (as abstract)
- Program manipulates object
- Can also add and remove class types from factory without changing factory code
  - But this gets complex

Factory pattern - example 1

```
public Interface Shape {
    public draw();
}
public class Rectangle implements Shape {
    public draw() {
        System.out.println("Drawing a rectangle");
    }
}
public class Circle implements Shape {
    public draw() {
        System.out.println("Drawing a circle");
    }
}
```

Factory pattern - example 2

```
public class ShapeFactory {
    public static Shape makeShape(String shapeType) {
        if(shapeType.equals("RECTANGLE")) {
            return new Rectangle();
        }else if(shapeType.equals("CIRCLE")) {
            return new Circle();
        }
    }
}
```

Factory pattern - example 3

```
public static void main(String[] args) {
    System.out.println("Enter type of shape:");
    String shape = System.console.readLine();
    Shape newShape = ShapeFactory.makeShape(shape);
    newShape.draw();
}
```

Factory pattern - summary

- The drawing program `FactoryTest` wouldn't need to be changed at all if further shapes were added.
- This is particularly useful in complex systems where changes would otherwise be required in lots of places
- Note that it assumes that in the drawing program we would only use the methods declared in `Shape` and not new methods declared in the different concrete objects (there weren't any in my example)

Visitor pattern

- Some times is is useful to keep some methods away from our nice neat class hierarchies
  - e.g. methods that are platform/device specific that would require multiple definitions in each class
  - methods that span unrelated classes
  - or perhaps we want to make new methods without modifying the classes themselves
- The `visitor` pattern allows us to do this
- Running example: we have a set of (unrelated) objects (e.g. of types human, dog), each of which has an age-related attribute (e.g. age, or date of birth). In another part of our program we require the ages of all of these objects in days. We don't want to change the definitions of `human` and `dog` so we use the `visitor` pattern.

Visitor pattern definitions

- The visitor pattern defines two interfaces:
  - The `Element` interface: each of our original types must implement this, it has one method: `accept(Visitor visitor)`
  - The `Visitor` interface: classes implementing our new methods implement this. We must define a `visit` method for each of the original types.
  - Once we force our original obejcts to implement `MyElement` we can add as many visitors (doing different things) as we like.

Visitor pattern - diagram

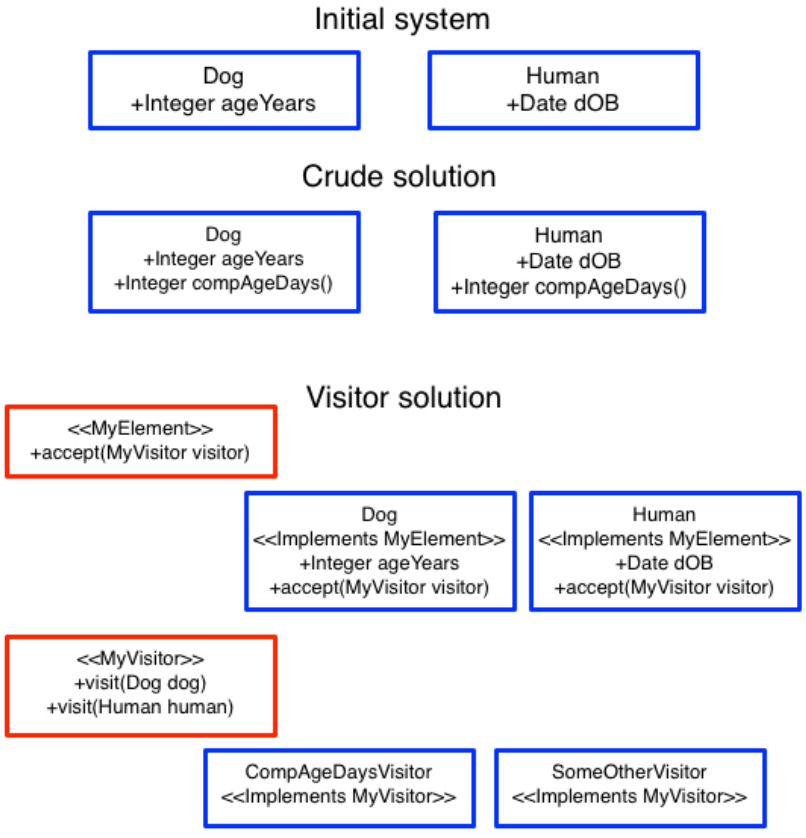

Figure 1:

Visitor pattern - examples

```
public interface MyElement {
    public void accept(MyVisitor visitor);
}
```

- `MyElement` only implements the `accept` method and this just calls the `visit` method of `MyVisitor`

Visitor pattern - examples

```
public interface MyVisitor {
    public void visit(Dog dog);
    public void visit(Human human);
}
```

- `MyVisitor` forces subclasses to implement methods for each of the original objects

Visitor pattern - examples

```
import java.util.Calendar;
public class Human implements MyElement {
    public Calendar dOB;
    public Human(Calendar d) {
        dOB = d;
    }
    public void accept(MyVisitor visitor) {
        visitor.visit(this);
    }
}
```

- accept is always the same...

Visitor pattern - examples

```
public class Dog implements MyElement{
    public Integer ageYears;
    public Dog(Integer a) {
        ageYears = a;
    }

    public void accept(MyVisitor visitor) {
        visitor.visit(this);
    }
}
```

Visitor pattern - examples

```
import java.util.GregorianCalendar;
public class CompAgeDaysVisitor implements MyVisitor {
    public void visit(Human human) {
        // Converting dates to differences in days
        GregorianCalendar today = new GregorianCalendar();
        long diffSeconds = (today.getTimeInMillis()
            - human.dOB.getTimeInMillis())/1000;
        Integer ageDays = (int)diffSeconds/(60*60*24);
        System.out.println("This human is " + ageDays + " d
    }
    public void visit(Dog dog) {
        Integer ageDays = dog.ageYears * 365;
        System.out.println("This dog is " + ageDays + " day
    }
}
```

Visitor pattern - examples

```
import java.util.*;
public class TestVisitor {
    public static void main(String[] args) {
        Dog d = new Dog(5);
        Calendar cal = new GregorianCalendar();
        cal.set(1995,5,12);
        Human h = new Human(cal);
        CompAgeDaysVisitor c = new CompAgeDaysVisitor();
        d.accept(c);
        h.accept(c);
    }
}
```

- Method is invoked by calling accept on the original objects

Visitor pattern - summary

- In our example, we call the method by invoking `accept`
- This calls the relevant `visit` method for the object of interest
- We could now write more visitors for these objects without changing them at all
- e.g. `DisplayStuffVisitor` can be called via:

```
DisplayStuffVisitor dS = new DisplayStuffVisitor();
d.accept(dS);
h.accept(dS);
// Or e.g.
d.accept(new DisplayStuffVisitor());
```

Visitor pattern - DisplayStuffVisitor

```
public class DisplayStuffVisitor implements MyVisitor {
    public void visit(Dog dog) {
        System.out.println("This is some stuff about dogs.
    }
    public void visit(Human human) {
        System.out.println("This is some stuff about humans
    }
}
```

The Proxy pattern

- The proxy pattern defines wrapper classes that act as interfaces (in a general sense) to other classes
- Useful for e.g.
  - *Simplicity:* can make a simpler API for complex objects (and standardise for different objects)
  - *Remote resources:* make remote objects look like local objects (e.g. `Socket`)
  - *Streamlining*: can stop big computations happening if they are unnecessary
  - *Security*: adding security checks to an object

Proxy - example

- From wikipedia
- Class `RealImage` loads and displays images
- Assume that loading is time and memory consuming and therefore we only load if we definitely need to display
  - Displaying might not always be necessary
- `ProxyImage` wraps `RealImage` and only creates the `RealImage` option when `display` is called

```
public interface Image {
    public void displayImage();
}
```

Proxy - example

- RealImage:

```
public class RealImage implements Image{
    private String filename = null;
    public RealImage(final String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }
    private void loadImageFromDisk() {
        System.out.println("Loading " + filename);
    }
    public void displayImage() {
        System.out.println("Displaying " + filename);
    }
}
```

Proxy - example

- ProxyImage:

```
public class ProxyImage implements Image{
    private String filename = null;
    private RealImage image = null;
    public ProxyImage(final String filename) {
        this.filename = filename;
    }
    public void displayImage() {
        if (image == null) {
            image = new RealImage(filename);
        }
        image.displayImage();
    }
}
```

Proxy - example

- Test code:

```
public class ProxyImage implements Image{
    private String filename = null;
    private RealImage image = null;
    public ProxyImage(final String filename) {
        this.filename = filename;
    }
    public void displayImage() {
        if (image == null) {
            image = new RealImage(filename);
        }
        image.displayImage();
    }
}
```

The decorator pattern

- The `decorator` pattern allows us to add functionality to existing objects without having to add it to all objects of that class (as would be the case if we simply put the methods into the class definition)
- Consider the following `BasicCar` object to some instants of which, we'd like to add extras (CD player, alloys, etc):

```
public class BasicCar {
    public double getPrice() {
        return 10000;
    }
    public String getDescription() {
        return "The basic car"
    }
}
```

The decorator pattern

- The first step is to define an abstract class that both `BasicCar` and our decorators will extend:

```
public abstract class Car {
    public abstract double getPrice();
    public abstract String getDescription();
}
```

- `BasicCar` now `extends Car`
- Decorators will add functionality to this by implementing these methods slightly differently to `BasicCar`

The decorator pattern

```
public abstract class CarDecorator extends Car {
    protected Car decoratedCar;
    public CarDecorator(Car decoratedCar) {
        this.decoratedCar = decoratedCar;
    }
    public double getPrice() {
        return decoratedCar.getPrice();
    }
    public String getDescription() {
        return decoratedCar.getDescription();
    }
}
```

- By default, the methods just call the methods on the object coming in
- We can now build a concrete decorator

The decorator pattern

```
public class AlloyDecorator extends CarDecorator {
    public AlloyDecorator (Car decoratedCar) {
        super(decoratedCar); // Call the CarDecorator cons
    }
    public Double getPrice() {
        return super.getPrice() + 250; // Add the price of
    }
    public String getDescription() {
        return super.getDescription() + " + Alloys";
    }
}
```

- Adds alloys to the basic car

The decorator pattern

```
public class CDDecorator extends CarDecorator {
    public CDDecorator (Car decoratedCar) {
        super(decoratedCar); // Call the CarDecorator cons
    }
    public Double getPrice() {
        return super.getPrice() + 150; // Add the price of
    }
    public String getDescription() {
        return super.getDescription() + " + CD Player";
    }
}
```

- Adds a CD player

The decorator pattern

- See `DecoratorTest`

The observer pattern

- Our final pattern is the `observer`
- It is useful when our program has a class (the `Subject`) containing some kind of *state* that might be required by various other classes
- The `observer` ensures that they are all updated whenever the `Subject` is updated
- We define the following classes
  - The `Subject` – the class containing the state of the system
  - The `Observer` – an abstract class that will be extended by concrete observers
  - Concrete observers (potentially several)

The decorator pattern

- Example: our `Subject` class will contain an array of `Double` values
- We will create concrete `observers` that display all of the data, or the mean of the data, (or the max, or the min, . . .)

The Subject

```
public class Subject {
    private ArrayList<Observer> observers = new ArrayList<(
    private Double[] data;
    public void setData(Double[] data) {
        this.data = data;
        this.notifyAllObservers();
        }
    public void attach(Observer observer) { this.observer =
    public void notifyAllObservers() {
        for(Observer observer : observers) {
            observer.notifyMe();
            }
        }
}
```

Abstract Observer

```
public abstract class Observer {
    protected Subject subject;
    public abstract void notifyMe();
}
```

Concrete list data observer

```
public class ListDataObserver extends Observer {
    public ListDatabObserver(Subject subject) {
        this.subject = subject;
        this.subject.attach(this);
    }
    public void notifyMe() {
        Double[] data = subject.getData();
        for(int i=0;i<data.length;i++) {
            System.out.println(data[i]);
        }
    }
}
```

Concrete mean data observer

```
public class MeanDataObserver extends Observer {
    public ListDatabObserver(Subject subject) {
        this.subject = subject;
        this.subject.attach(this);
    }
    public void notifyMe() {
        Double[] data = subject.getData();
        Double mean = 0.0;
        for(int i=0;i<data.length;i++) {
            mean += data[i];
        }
        mean = mean / data.length;
        System.out.println("Mean: " + mean);
    }
}
```

Observer pattern

- ObserverTest.java

```
public class ObserverTest {
public static void main(String[] args) {
    Subject s = new Subject();
    Double[] d = new Double[5];
    d[0] = 1.0;d[1] = 1.2;d[2] = 1.4;d[3] = 1.7;d[4] =
    new ListDataObserver(s);
    new MeanDataObserver(s);
    s.setData(d);
    d[3] = 3.2;
    s.setData(d);
}
}
```