AP(IT): Design Patterns

Simon Rogers

2nd Jan 2019

Contents

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)

Why are we covering them?

- Useful things to be aware of
- Great examples of the benefits of inheritance and polymorphism

Some useful patterns

Iterators

- Many algorithms require the ability to move through a collection of objects
  - Finding, sorting, etc
- Java's `Iterator` interface provides a standard way to allow other code to move through the items in a particular collection.
- Many inbuilt Java classes already have the ability to provide iterators
- The `Iterator` interface defines three methods: `hasNext()`, `next()` and `remove()` (the final one is often not implemented).

```java
import java.util.ArrayList;
import java.util.Iterator;
public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("Simon");
        al.add("Ella");
        Iterator i = al.iterator();
        while(i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

In this example, we use the inbuilt `iterator()` method for `ArrayList` to provide us with an iterator over the list. Note that it is the `Iterator` that has the `hasNext()` and `next()` methods, and not the `ArrayList` itself. This separation makes sense as it allows us to have multiple simultaneous iterators over the same object.

- To make your own iterator, simply implement the `next()`, `hasNext()`, and `remove()` methods.
- e.g. `Counter.java` - an Iterator that iterates over the integers from 0 to 9.

```java
import java.util.Iterator;
public class Counter implements Iterator<Integer>{
    int pos;
    public Counter() {
        pos = 0; // Start at 0
    }
    public Integer next() {
        return pos++;
    }
    public boolean hasNext() {
        if(pos < 10) {
            return true;
        }else {
            return false;
        }
    }
    public void remove() {
        throw new UnsupportedOperationException();
    }
    public static void main(String[] args) {
        Counter c = new Counter();
        while(c.hasNext()) {
            System.out.println(c.next());
        }
    }
}
```

Iterable

- In the `ArrayList` example, the `ArrayList` object provided us an `Iterator`
- This is because it implements an interface called `Iterable`.
- `Iterable` defines a single method `iterator()` that returns an `Iterator`.
- `TenRandoms.java` is an example of how to do this. It is a class that creates an array of ten random numbers and then provides `Iterators` over the array. Note it implements `Iterable` and not `Iterator`
- `Iterable` also allows you to use Java's concise for loop syntax

```java
import java.util.Random;
import java.util.Iterator;
public class TenRandoms implements Iterable<Double> {
    private Random r;
    private Double[] theNumbers;
    public TenRandoms(int howMany) {
        theNumbers = new Double[howMany];
        r = new Random();
        for(int i=0;i<10;i++) {
            theNumbers[i] = r.nextDouble();
        }
    }
    public Iterator<Double> iterator() {
        Iterator<Double> it = new Iterator<Double>() {
```

```java
            private int pos = 0;
            public boolean hasNext() {
                if(pos < theNumbers.length) {
                    return true;
                }
                return false;
            }
            public Double next() {
                return theNumbers[pos++];
            }
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
        return it;
    }


    public static void main(String[] args) {
        TenRandoms tr = new TenRandoms(10);
        Iterator it = tr.iterator();
        while(it.hasNext()) {
            System.out.println(it.next());
        }
        /*
         Iterable also allows you to use Java's concise
         for loop syntax
        */
        System.out.println();
        for(Double r : tr) {
            System.out.println(r);
        }
    }
}
```

- `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 systems: computing the size of files and folders of files
  - 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 in`component`as well as`add`and`remove` 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?

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

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);
```

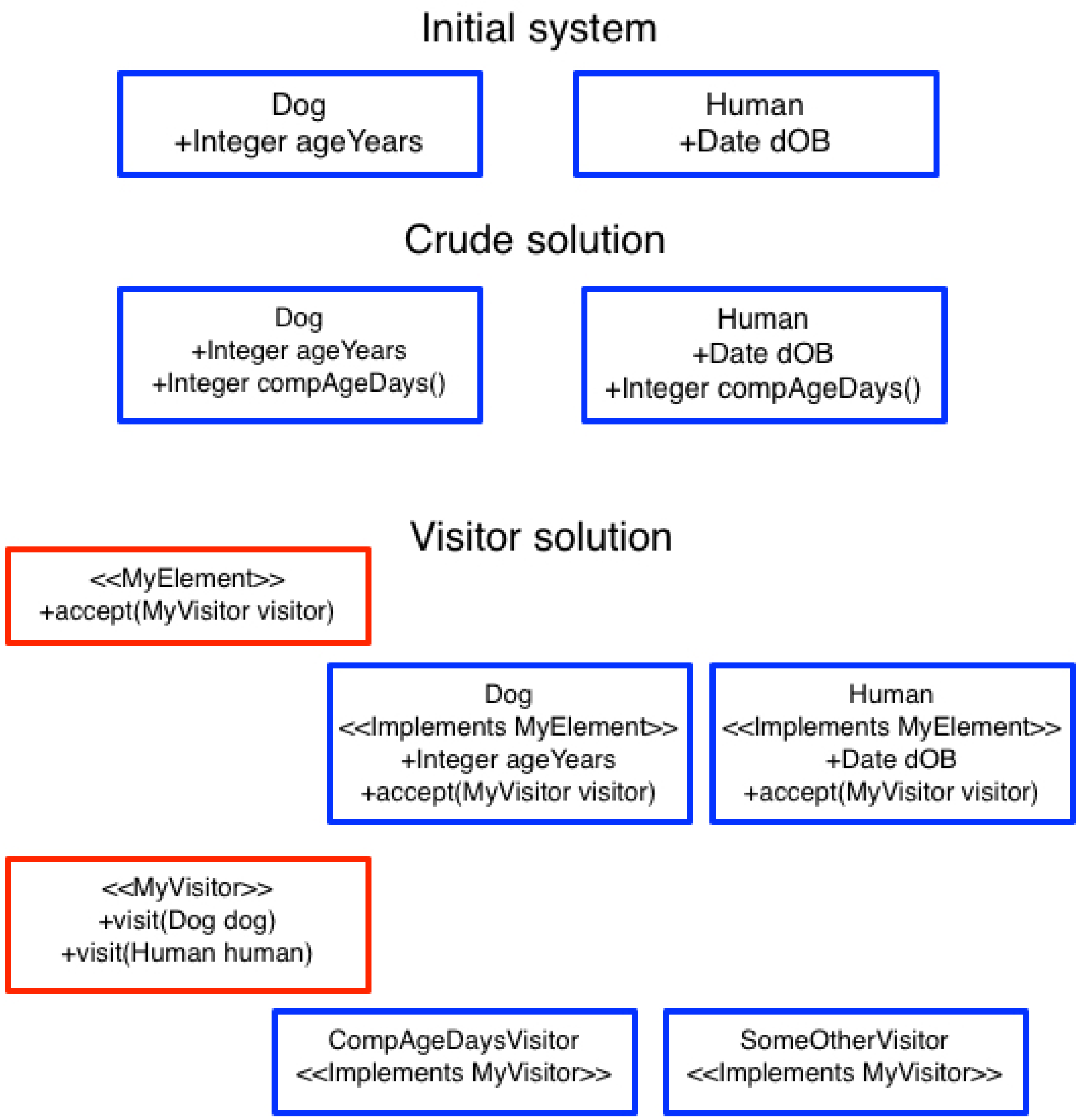
Initial system
Dog
+Integer ageYears
Human
+Date dOB
Crude solution
Dog
+Integer ageYears
+Integer compAgeDays()
Human
+Date dOB
+Integer compAgeDays()
Visitor solution
<<MyElement>>
+accept(MyVisitor visitor)
Dog
<<Implements MyElement>>
+Integer ageYears
+accept(MyVisitor visitor)
Human
<<Implements MyElement>>
+Date dOB
+accept(MyVisitor visitor)
<<MyVisitor>>
+visit(Dog dog)
+visit(Human human)
CompAgeDaysVisitor
<<Implements MyVisitor>>
SomeOtherVisitor
<<Implements MyVisitor>>

Figure 1:

```
    }
}
```

- `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 + " days old");
    }
    public void visit(Dog dog) {
        Integer ageDays = dog.ageYears * 365;
        System.out.println("This dog is " + ageDays + " days old");
    }
}
```

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. They have 4 legs.");
    }
    public void visit(Human human) {
        System.out.println("This is some stuff about humans. They have 2 legs.");
    }
}
```

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

```java
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

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

- Adds alloys to the basic car

The decorator pattern

```java
public class CDDecorator extends CarDecorator {
    public CDDecorator (Car decoratedCar) {
        super(decoratedCar); // Call the CarDecorator constructor
    }
    public Double getPrice() {
        return super.getPrice() + 150; // Add the price of alloys
    }
    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 observer 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<Observer>();
    private Double[] data;
    public void setData(Double[] data) {
        this.data = data;
        this.notifyAllObservers();
        }
    public void attach(Observer observer) { this.observer = 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] = 2.4;
    new ListDataObserver(s);
    new MeanDataObserver(s);
    s.setData(d);
    d[3] = 3.2;
    s.setData(d);
}
}
```