INFO1113 Object-Oriented Programming

Week 11A: Idioms and patterns

Copyright Warning

Topics

- Defensive Checking (s. 4)
- Casting and packing objects (s. 14)
- Builder Pattern (s. 27)
- State Pattern (s. 40)

Idioms

All programming languages have regular patterns that arise from usage. Typical structures in our loops, iterators and bounds check are common idioms in the java programming language.

Defensive Checking

Let's consider this code:

```
public void changeFrom(int[] array, int i, int v) {
    if(i >= 0 || i < array.length) {
        for(int j = i; j < array.length; j++) {
            array[j] = v;
        }
    } else {
        System.err.println("Unable to set from " + i + ": Out of bounds");
    }
}
```

Defensive Checking

Let's consider this code:

```
public void changeFrom(int[] array, int i, int v) {
    if(i >= 0 || i < array.length) {
        for(int j = i; j < array.length; j++) {
            array[j] = v;
        }
    } else {
        System.err.println("Unable to set from " + i + ": Out of bounds");
    }
}
```

We check the index and see if it is within the bounds of an array.

Defensive Checking

Let's consider this code:

```
public void changeFrom(int[] array, int i, int v) {
    if(i >= 0 || i < array.length) {
        for(int j = i; j < array.length; j++) {
            array[j] = v;
        }
    } else {
        System.err.println("Unable to set from " + i + ": Out of bounds");
    }
}
```

If this fails, we are able to output to standard error a case where an attempt to access outside of the bounds of the array.

Why would we use standard error? What else could we use?

Defensive Checking

Let's consider this code:

```
public void changeFrom(int[] array, int i, int v) {
    if(i >= 0 || i < array.length) {
        for(int j = i; j < array.length; j++) {
            array[j] = v;
        }
    } else {
        throw new IndexOutOfBoundsException("ith value is out of bounds");
    }
}
```

Instead of standard error, we are able to throw an exception where the index is out of the bounds.

What are we missing from this? What are we not checking?

Defensive Checking

Let's consider this code:

```
public void changeFrom(int[] array, int i, int v) {
    if(array == null) {
        throw new NullPointerException("Unable to use the array");
    } else if(i < 0 || i >= array.length) {
        throw new IndexOutOfBoundsException("ith value is out of bounds");
    } else {
        for(int j = i; j < array.length; j++) {
            array[j] = v;
        }
    }
}
```

We have checked for the index but we have not checked if the array is null.

Defensive Checking

Let's consider this code:

```
public void changeFrom(int[] array, int i, int v) {
    if(array == null) {
        throw new NullPointerException("Unable to use the array");
    } else if(i < 0 || i >= array.length) {
        throw new IndexOutOfBoundsException("ith value is out of bounds");
    } else {
        for(int j = i; j < array.length; j++) {
            array[j] = v;
        }
    }
}
```

We check if it is null and if it is we are able to throw out own exception.

However, we may not want to always throw an exception, what else could we do?

Shallow and deep copy

Occasionally we may want to duplicate objects. However objects themselves may contain other objects as instance fields. How much of an object should we copy?

- Shallow-copy: Copy all immediate fields that are related to an object. This will likely alias any reference type fields and share them with the copy

- Deep-copy: Copy all immediate fields and any other objects that it refers to. For it to more than a shallow copy, it will create a copy of any reference type fields.

  An object may form a graph and potentially have a cyclic relationship with other objects, a deep copy may be undesirable or would need to consider these constraints.

So how would each copy method look like?

Casting and packing

You may want to utilise and write methods that will allow any type of object to be passed to it. This can be a form of packing and unpacking.

There is a clear advantage to being able to pass any object to the method but there are a few issues we need to consider.

Time to live on the edge

Could we write a complete general method?

Builder Pattern

Let’s consider the following:

```
public class PackingExample {

    public static void unpack(Object packedData) {
        Integer[] myInts = (Integer[]) packedData;
        System.out.println(myInts[0]);
        System.out.println(myInts[1]);

        System.out.println(myInts[0] + myInts[1]);
    }

    public static void main(String[] args) {
        unpack(new Integer[] {1, 2});
    }

}
```

Builder Pattern

Let's consider the following:

```
public class PackingExample {

    public static void unpack(Object packedData) {
        Integer[] myInts = (Integer[]) packedData;
        System.out.println(myInts[0]);
        System.out.println(myInts[1]);

        System.out.println(myInts[0] + myInts[1]);
    }

    public static void main(String[] args) {
        unpack(new Integer[] {1, 2});
    }

}
```

We create a method that allows an object variable to be passed to it. Since all types inherit from Object we can pass anything to it.

Builder Pattern

Let’s consider the following:

```
public class PackingExample {

    public static void unpack(Object packedData) {
        Integer[] myInts = (Integer[]) packedData;
        System.out.println(myInts[0]);
        System.out.println(myInts[1]);

        System.out.println(myInts[0] + myInts[1]);
    }

    public static void main(String[] args) {
        unpack(new Integer[] {1, 2});
    }

}
```

We have created an array of Integer types and given the values 1 and 2.

Builder Pattern

Let’s consider the following:

```
public class PackingExample {

    public static void unpack(Object packedData) {
        Integer[] myInts = (Integer[]) packedData;
        System.out.println(myInts[0]);
        System.out.println(myInts[1]);

        System.out.println(myInts[0] + myInts[1]);
    }

    public static void main(String[] args) {
        unpack(new Integer[] {1, 2});
    }

}
```

Once passed we can convert it back to what it was through casting.

Builder Pattern

Let’s consider the following:

```
public class PackingExample {

    public static void unpack(Object packedData) {
        Integer[] myInts = (Integer[]) packedData;
        System.out.println(myInts[0]);
        System.out.println(myInts[1]);

        System.out.println(myInts[0] + myInts[1]);
    }

    public static void main(String[] args) {
        unpack(new Integer[] {1, 2});
    }

}
```

Afterwards we can use it for our own purposes.

What have we seen before that address this issue?

Patterns

The difference between an idiom and a pattern for software development is that an idiom is a pattern that is afforded by the language and is typically language specific.

Design patterns allow for greater flexibility, better control of constraints, robust structure and better maintainability.

Over the course of development, our code can be trickier to maintain and expand. The desired outcome is to reduce the amount of refactoring of code.

Refactoring isn’t evil!

Refactoring is the act of restructuring our code. We identify a segment in our code that is causing conflict with our requirements or is simply structure poorly and we resolve this problem by restructuring.

Small programs don’t typically encounter much refactoring but large applications that have been worked on for a long period of time will be refactored many times.

Builder Pattern

Let’s consider this code:

```
public class Computer {
    private Part cpu;
    private Part motherboard;
    private ArrayList<Part> hdds;
    private ArrayList<Part> ram;
    private Enclosure compCase;

    public Computer(Enclosure c, Part cpu, Part motherboard, Part psu,
        ArrayList<Part> hdds, ArrayList<Part> ram) {
        this.cpu = cpu;
        this.motherboard = motherboard;
        this.psu = psu;
        this.hdds = hdds;
        this.ram = ram;
        this.compCase = c;
    }

}
```

Builder Pattern

Let's consider this code:

```
public class Computer {
    private Part cpu;
    private Part motherboard;
    private ArrayList<Part> hdds;
    private ArrayList<Part> ram;
    private Enclosure compCase;

    public Computer(Enclosure c, Part cpu, Part motherboard, Part psu,
        ArrayList<Part> hdds, ArrayList<Part> ram) {
        this.cpu = cpu;
        this.motherboard = motherboard;
        this.psu = psu;
        this.hdds = hdds;
        this.ram = ram;
        this.compCase = c;
    }

}
```

Hrmm… this constructor is doing a lot. Let's try use this constructor.

Builder Pattern

Let's consider this code:

Wow! That is a big constructor! What if we don't want to include certain objects?

```
public class Computer {
    private Part cpu;
    private Part motherboard;
    private ArrayList<Part> hdds;
    private ArrayList<Part> ram;
    private Enclosure compCase;

    public Computer(Enclosure c, Part cpu, Part motherboard,
 Part psu, ArrayList<Part> hdds, ArrayList<Part> ram) {
        this.cpu = cpu;
        this.motherboard = motherboard;
        this.psu = psu;
        this.hdds = hdds;
        this.ram = ram;
        this.compCase = c;
    }

}
```

```
Computer c = new Computer(
    new Enclosure("CoolerMaster R88"),
    new CPU("AMD Ryzen 7"),
    new Motherboard("Gigabyte"),
    Arrays.<Part>asList(
        new Part("SSD"),
        new Part("HDD")),
    Arrays.<Part>asList(
        new Part("4GB DDR"),
        new Part("4GB DDR"),
        new Part("4GB DDR"),
        new Part("4GB DDR")
    )
);
```

Builder Pattern

Let's consider this code:

It could be hard to remember all those parameters!

```
public class Computer {
    private Part cpu;
    private Part motherboard;
    private ArrayList<Part> hdds;
    private ArrayList<Part> ram;
    private Enclosure compCase;

    public Computer(Enclosure c, Part cpu, Part motherboard,
 Part psu, ArrayList<Part> hdds, ArrayList<Part> ram) {
        this.cpu = cpu;
        this.motherboard = motherboard;
        this.psu = psu;
        this.hdds = hdds;
        this.ram = ram;
        this.compCase = c;
    }

}
```

```
Computer c = new Computer(
    null,
    new CPU("AMD Ryzen 7"),
    new Motherboard("Gigabyte"),
    Arrays.<Part>asList(
        new Part("SSD"),
        new Part("HDD")),
    null
);
```

Builder Pattern

Let’s consider this code:

```
public class Computer {
    private Part cpu;
    private Part motherboard;
    private ArrayList<Part> hdds;
    private ArrayList<Part> ram;
    private Enclosure compCase;

    public Computer(Enclosure c, Part cpu, Part motherboard, Part psu,
        ArrayList<Part> hdds, ArrayList<Part> ram) {
        this.cpu = cpu;
        this.motherboard = motherboard;
        this.psu = psu;
        this.hdds = hdds;
        this.ram = ram;
        this.compCase = c;
    }

    public Computer(Part cpu, Part motherboard, Part psu,
        ArrayList<Part> hdds, ArrayList<Part> ram) {
        this.cpu = cpu;
        this.motherboard = motherboard;
        this.psu = psu;
        this.hdds = hdds;
        this.ram = ram;
        this.compCase = c;
    }

    public Computer(Part cpu, Part motherboard, ArrayList<Part> hdds) {
        super(null, cpu, motherboard, null, hdds, null);
    }

    public Computer(Part cpu, Part motherboard) {
        super(null, cpu, motherboard, null, null, null);
    }

    public Computer() {
        super(null, null, null, null, null, null);
    }

}
```

Hrmm… how could we resolve this?

Builder Pattern

Let’s consider this segment of code <builder pattern>

```
public class ComputerBuilder {
    private Part cpu;
    private Part motherboard;
    private ArrayList<Part> hdds;
    private ArrayList<Part> ram;
    private Enclosure enclosure;

    public ComputerBuilder() {
        cpu = null;
        motherboard = null;
        PSU = null;
        hdds = new ArrayList<Part>();;
        ram = new ArrayList<Part>();
        enclosure = null;
    }

    public ComputerBuilder setMotherboard(Part p) {
        motherboard = p;
        return this;
    }

    public ComputerBuilder setCPU(Part p) {
        cpu = p;
    //Motherboard check
        return this;
    }

    public ComputerBuilder addHDD(Part p) {
        hdds.add(p);
        return this;
    }

    public ComputerBuilder addRAM(Part p) {
        ram.add(p);
        return this;
    }

    public Computer build() {
        return new Computer(enclosure, cpu, motherboard, psu, hdds, ram);
    }
}
```

Builder Pattern

Let's consider this segment of code <builder pattern>

```
public class ComputerBuilder {
    private Part cpu;
    private Part motherboard;
    private ArrayList<Part> hdds;
    private ArrayList<Part> ram;
    private Enclosure enclosure;

    public ComputerBuilder() {
        cpu = null;
        motherboard = null;
        PSU = null;
        hdds = new ArrayList<Part>();;
        ram = new ArrayList<Part>();
        enclosure = null;
    }

    public ComputerBuilder setMotherboard(Part p) {
        motherboard = p;
        return this;
    }

    public ComputerBuilder setCPU(Part p) {
        cpu = p;
    //Motherboard check
        return this;
    }

    public ComputerBuilder addHDD(Part p) {
        hdds.add(p);
        return this;
    }

    public ComputerBuilder addRAM(Part p) {
        ram.add(p);
        return this;
    }

    public Computer build() {
        return new Computer(enclosure, cpu, motherboard, psu, hdds, ram);
    }
}
```

Let's introduce a builder, it provides functionality to allow us to build. We have our constructor.

Builder Pattern

Let’s consider this segment of code <builder pattern>

```
public class ComputerBuilder {
    private Part cpu;
    private Part motherboard;
    private ArrayList<Part> hdds;
    private ArrayList<Part> ram;
    private Enclosure enclosure;

    public ComputerBuilder() {
        cpu = null;
        motherboard = null;
        PSU = null;
        hdds = new ArrayList<Part>();;
        ram = new ArrayList<Part>();
        enclosure = null;
    }

    public ComputerBuilder setMotherboard(Part p) {
        motherboard = p;
        return this;
    }

    public ComputerBuilder setCPU(Part p) {
        cpu = p;
    //Motherboard check
        return this;
    }

    public ComputerBuilder addHDD(Part p) {
        hdds.add(p);
        return this;
    }

    public ComputerBuilder addRAM(Part p) {
        ram.add(p);
        return this;
    }

    public Computer build() {
        return new Computer(enclosure, cpu, motherboard, psu, hdds, ram);
    }
}
```

We have setters and getters which will set different properties of the computer.

Builder Pattern

Let's consider this segment of code <builder pattern>

```
public class ComputerBuilder {
    private Part cpu;
    private Part motherboard;
    private ArrayList<Part> hdds;
    private ArrayList<Part> ram;
    private Enclosure enclosure;

    public ComputerBuilder() {
        cpu = null;
        motherboard = null;
        PSU = null;
        hdds = new ArrayList<Part>();;
        ram = new ArrayList<Part>();
        enclosure = null;
    }

    public ComputerBuilder setMotherboard(Part p) {
        motherboard = p;
        return this;
    }

    public ComputerBuilder setCPU(Part p) {
        cpu = p;
    //Motherboard check
        return this;
    }

    public ComputerBuilder addHDD(Part p) {
        hdds.add(p);
        return this;
    }

    public ComputerBuilder addRAM(Part p) {
        ram.add(p);
        return this;
    }

    public Computer build() {
        return new Computer(enclosure, cpu, motherboard, psu, hdds, ram);
    }
}
```

```
Computer c = new ComputerBuilder()
             .setMotherboard(new Motherboard("Gigabyte"))
             .setCPU(new CPU("AMD Ryzen 7")),
             .addHDD(new Part("SSD"))
             .addHDD(new Part("HDD"))
             .build();
```

We utilise the builder like so, specifying what different components without needing to know the order of parameters for the constructors.

Builder Pattern

Let's consider this segment of code <builder pattern>

```
public class ComputerBuilder {
    private Part cpu;
    private Part motherboard;
    private ArrayList<Part> hdds;
    private ArrayList<Part> ram;
    private Enclosure enclosure;

    public ComputerBuilder() {
        cpu = null;
        motherboard = null;
        PSU = null;
        hdds = new ArrayList<Part>();;
        ram = new ArrayList<Part>();
        enclosure = null;
    }

    public ComputerBuilder setMotherboard(Part p) {
        motherboard = p;
        return this;
    }

    public ComputerBuilder setCPU(Part p) {
        cpu = p;
    //Motherboard check
        return this;
    }

    public ComputerBuilder addHDD(Part p) {
        hdds.add(p);
        return this;
    }

    public ComputerBuilder addRAM(Part p) {
        ram.add(p);
        return this;
    }

    public Computer build() {
        return new Computer(enclosure, cpu, motherboard, psu, hdds, ram);
    }
}
```

```
Computer c = new ComputerBuilder()
            .setMotherboard(new Motherboard("Gigabyte"))
            .setCPU(new CPU("AMD Ryzen 7")),
            .addHDD(new Part("SSD"))
            .addHDD(new Part("HDD"))
            .build();
```

After the builder has set the properties required we call **build** which will return a **Computer** object.

Demo

Where have we seen this before?

Builder Pattern

We have seen it within the Java API, through the classes **StringBuilder** and **Stream.**

We are able to assemble a class and return the object to itself, chaining the methods to construct it.

State Pattern

State Pattern is a behavioural pattern that allows us to utilise and constrain behaviour based on the state the object is in. It is a common pattern to enforce constraints on an object in certain states.

For example

- A person could be in a state of Running, Jumping and Swimming.

  A person cannot jump again if they are already in mid air or in the state of Jumping.

How could we represent state for an object?

How would you decide which action to take?

State Pattern

```
public class Telephone {

    private TelephoneState state;

    public Telephone() {
        state = new LineWaiting();
    }

    public void dial(String phonenumber) {
        state = state.dial(phonenumber);
    }

    public void hangup() {
        state = state.hangup();
    }

    public static void main(String[] args) {
        Telephone phone = new Telephone();
        phone.dial("12341234");
        phone.hangup();
    }

}
```

We can see that it is invalid for the phone state to change when the state is waiting and **hangup** is called.

State Pattern

```
public abstract class TelephoneState {

    protected String numberDialed;

    public abstract TelephoneState dial(String phonenumber);

    public abstract TelephoneState hangup();

}
```

```
public class LineBusy extends TelephoneState {

    public LineBusy(String number) {
        super();
        numberDialed = number;
    }

    public TelephoneState dial(String phonenumber) {
        throw new InvalidPhoneState();
    }

    public TelephoneState hangup() {
        System.out.println("Hanging up: " + numberDialed);
        return new LineWaiting();
    }

}
```

```
public class LineWaiting extends TelephoneState {

    public TelephoneState dial(String phonenumber) {
        System.out.println("Dialing: " + phonenumber);
        return new LineBusy(phonenumber);
    }

    public TelephoneState hangup() {
        throw new InvalidPhoneState();
    }

}
```

State Pattern

```java
public abstract class TelephoneState {

    protected String numberDialed;

    public abstract TelephoneState dial(String phonenumber);

    public abstract TelephoneState hangup();

}
```

We create an abstract class (or interface) which will represent the state. Defines the actions that may happen in each state.

```java
public class LineBusy extends TelephoneState {

    public LineBusy(String number) {
        super();
        numberDialed = number;
    }

    public TelephoneState dial(String phonenumber) {
        throw new InvalidPhoneState();
    }

    public TelephoneState hangup() {
        System.out.println("Hanging up: " + numberDialed);
        return new LineWaiting();
    }

}
```

```java
public class LineWaiting extends TelephoneState {

    public TelephoneState dial(String phonenumber) {
        System.out.println("Dialing: " + phonenumber);
        return new LineBusy(phonenumber);
    }

    public TelephoneState hangup() {
        throw new InvalidPhoneState();
    }

}
```

State Pattern

```
public abstract class TelephoneState {

    protected String numberDialed;

    public abstract TelephoneState dial(String phonenumber);

    public abstract TelephoneState hangup();

}
```

```
public class LineBusy extends TelephoneState {

    public LineBusy(String number) {
        super();
        numberDialed = number;
    }

    public TelephoneState dial(String phonenumber) {
        throw new InvalidPhoneState();
    }

    public TelephoneState hangup() {
        System.out.println("Hanging up: " + numberDialed);
        return new LineWaiting();
    }

}
```

This class represents the **LineWaiting** state. Specifying the the actions when the line is waiting.

```
public class LineWaiting extends TelephoneState {

    public TelephoneState dial(String phonenumber) {
        System.out.println("Dialing: " + phonenumber);
        return new LineBusy(phonenumber);
    }

    public TelephoneState hangup() {
        throw new InvalidPhoneState();
    }

}
```

State Pattern

This class represents the **LineBusy** state. There is a large deviance between the two states.

```
public abstract class TelephoneState {

    protected String numberDialed;

    public abstract TelephoneState dial(String phonenumber);

    public abstract TelephoneState hangup();

}
```

```
public class LineBusy extends TelephoneState {

    public LineBusy(String number) {
        super();
        numberDialed = number;
    }

    public TelephoneState dial(String phonenumber) {
        throw new InvalidPhoneState();
    }

    public TelephoneState hangup() {
        System.out.println("Hanging up: " + numberDialed);
        return new LineWaiting();
    }

}
```

```
public class LineWaiting extends TelephoneState {

    public TelephoneState dial(String phonenumber) {
        System.out.println("Dialing: " + phonenumber);
        return new LineBusy(phonenumber);
    }

    public TelephoneState hangup() {
        throw new InvalidPhoneState();
    }

}
```

State Pattern

```
public abstract class TelephoneState {

    protected String numberDialed;

    public abstract TelephoneState dial(String phonenumber);

    public abstract TelephoneState hangup();

}
```

We can see that it is invalid for the phone state to change when the state is waiting and **hangup** is called.

```
public class LineBusy extends TelephoneState {

    public LineBusy(String number) {
        super();
        numberDialed = number;
    }

    public TelephoneState dial(String phonenumber) {
        throw new InvalidPhoneState();
    }

    public TelephoneState hangup() {
        System.out.println("Hanging up: " + numberDialed);
        return new LineWaiting();
    }

}
```

```
public class LineWaiting extends TelephoneState {

    public TelephoneState dial(String phonenumber) {
        System.out.println("Dialing: " + phonenumber);
        return new LineBusy(phonenumber);
    }

    public TelephoneState hangup() {
        throw new InvalidPhoneState();
    }

}
```

State Pattern

```
public class Telephone {

    private TelephoneState state;

    public Telephone() {
        state = new LineWaiting();
    }

    public void dial(String phonenumber) {
        state = state.dial(phonenumber);
    }

    public void hangup() {
        state = state.hangup();
    }

    public static void main(String[] args) {
        Telephone phone = new Telephone();
        phone.dial("12341234");
        phone.hangup();
    }

}
```

When an action such as dial or hangup occurs, the state of the telephone will change and therefore it is reflected as part of its return value.

Let’s give this a demo

See you next time!