TASK

Object-Oriented Design and Design Patterns

Visit our website

Introduction

WELCOME TO THE OBJECT-ORIENTED DESIGN PATTERNS TASK!

This task will provide a brief overview of what design patterns are in the realm of software development. We will also take a closer look at a few specific design patterns.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact your mentor to get support on any aspect of your course.

The best way to get help is to login to **www.hyperiondev.com/portal** to start a chat with your mentor. You can also schedule a call or get support via email.

Your mentor is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

WHAT ARE DESIGN PATTERNS?

There are very few times, if any, where it would be necessary to reinvent the wheel. That is why design patterns can be so useful. Design patterns describe neat and efficient ways of solving problems in code. These solutions are tried, tested and honed for efficiency. It is possible to code for many years and never use design patterns (many programmers do). However, if you ignore design patterns you may either waste time coming up with solutions that already exist or you could end up writing less efficient methods. Design patterns are considered best practice for Object-Oriented Programmers and it is where the *engineering* part of *software engineering* really comes into play!

The Hillside Group (**http://hillside.net**), which is dedicated to maintaining information about patterns, describes design patterns as follows:

"Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience."

Design patterns have made a huge impact on object-oriented software design. Not only are they well-tested solutions to common problems, but they also have provided a vocabulary for talking about design. Therefore, you can explain your design by describing the patterns that you have used.

BENEFITS OF USING DESIGN PATTERNS

There are many benefits to using design patterns. Some benefits include:

- Using them can speed up the development process.
- Reusing design patterns helps to prevent subtle issues that may only become visible later in the implementation, which can cause major problems.
- It improves code readability for Coders and Architects familiar with the patterns.
- Design patterns provide general solutions, documented in a format that does not require specifics tied to a particular problem.
- They allow developers to communicate using well-known, well-understood names for software interactions.
- Design patterns are constantly improved over time, making them more robust than ad-hoc designs.

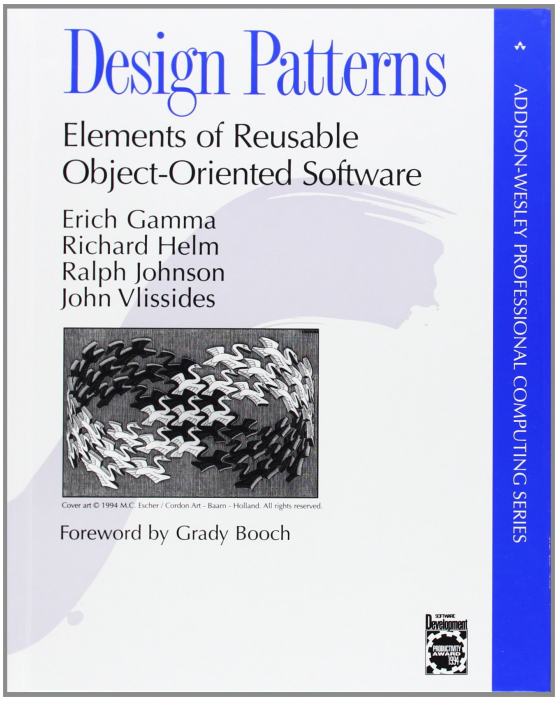
- They are language-neutral and so can be applied to any Object-Oriented Programming language.

THE GANG OF FOUR

The best-known design patterns are described in a book entitled: Design Patterns: Elements of Reusable Object-Oriented Software. This book initiated the concept of Design Pattern in Software development. It was written by four authors, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, who collectively became known as the “Gang of Four” (GOF).

According to the Gang of Four, design patterns are primarily based on the following principles of object orientated design:

- "Program to an 'interface', not an 'implementation'." (Gamma et al., 1994, p. 18)
- Composition over inheritance: "Favor 'object composition' over 'class inheritance'." (Gamma et al., 1994, p. 20)



The cover of Design Patterns: Elements of Reusable Object-Oriented Software

TYPES OF DESIGN PATTERNS

According to the book Design Patterns: Elements of Reusable Object-Oriented Software, design patterns can be classified into three categories:

1. Creational
2. Structural
3. Behavioural

Creational Design Patterns
Creational design patterns provide a way to create objects while hiding the creation logic, instead of instantiating objects directly using the new operator. This gives programs more flexibility in deciding which objects need to be created for a given use case.

Structural Design Patterns
Structural design patterns are all about class and object composition. Interfaces are composed using the concept of inheritance. The concept of inheritance is also used to define ways to compose objects to obtain new functionalities.

Behavioural Design Patterns
Behavioural design patterns are concerned with communication between objects.

We will now take a closer look at a few popular design patterns.

SINGLETON DESIGN PATTERN

The Singleton pattern is a **creational** pattern. It ensures that **only one object of a class is created** by making the constructor `private`, making a `getInstance` method that checks if an instance already exists, and making it `static` so we can call it without creating objects of the class.

As you can see from the definition above, you use the Singleton pattern when you must make sure that an application can only create one object of a particular class. An example of when a Singleton pattern may be used is when you create a `Logger` class that logs application errors to an external file. Different classes and threads of your application may want to log an error at once. If each of these tried to write errors to a single log file at once, it would cause errors. However, having a single `Logger` object that writes different errors to the log file would be a good solution.
As you may remember from the OOP task, we can create multiple instantiations of an object from a single class (we made Lion and Cheetah objects from the Animal class). In our example, we create the `Logger` class, and instantiate two `Logger` objects below.

Printer.java:

```
public class Logger {
   public Logger() {
   }
}
```

Main.java:

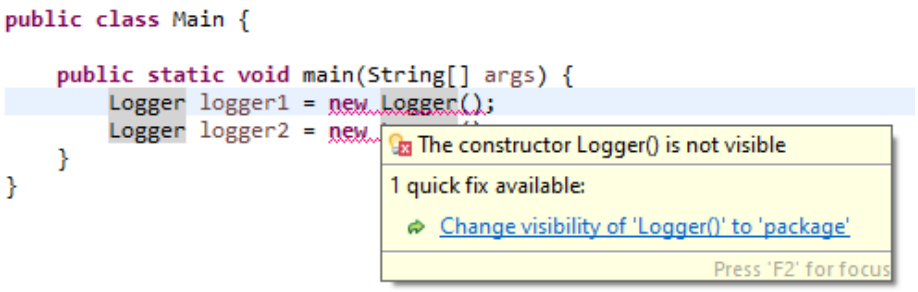
```
public class Main {

   public static void main(String[] args) {
      Logger logger1 = new Logger();
      Logger logger2 = new Logger();
   }
}
```

We have called the **Logger** class twice to create two new **Logger** objects. That works, but as stated previously, it will cause errors if the two objects try to write to the log file at the same time. Therefore, we do the following:

1. Firstly, we make the constructor **private**, so that the constructor method can no longer be accessed outside the **Logger** class.

   Printer.java:

```
public class Logger {
   // private means it cannot be accessed outside this class
   private Logger() {
   }
}
```



   As you can see in the image above, we will now not be able to create an object from the **Logger** class because the **Logger** constructor method is **private**. But that's a problem. We still want a **Logger** object so that we can log errors — we still need one instantiation of the **Logger** object.

2. To do this, we:

a. Create a **static** variable called **logWriter** (the name of this variable can be anything you choose). A static variable is a variable that belongs to the class instead of a specific instance of that class (or object). You will use this variable in the **getInstance** (see next step) method.
b. Add a **static getInstance** method into the **Logger** class. This method works as follows: if the **logWriter** variable (or whatever you called the variable created in the previous step) does not have a value, call the constructor method of the **Logger** class to create a new **Logger** object. But, if the **logWriter** variable has already been assigned a value, return **logWriter**. With this code, the same object is always returned.

Printer.java:

```
public class Logger {

   private static Logger logWriter;

   // private means it cannot be accessed outside this class
   public static Logger getInstance() {
      if (logWriter == null) {
              logWriter = new Logger();
      }
       return logWriter;
  }
}
```

c. Call the **getInstance()** method to access a **Logger** object from the Main method.

Remember that the **getInstance** method is **static**. This allows us to call it outside the **Logger** class without creating an object. It is important to note that a **static** method can only call other **static** methods or data, so we amend the Main method as shown below:

Main.java:

```
public class Main {

   public static void main(String[] args) {
      Logger logger1 = Logger.getInstance();
      Logger logger2 = Logger.getInstance();

      if (logger1 == logger2) {
         System.out.println("Only one Logger object created.");
      }
```

```
        else {
            System.out.println("More than one logger object created");
        }
    }
}
```

To prove that the two objects are identical, we can use a direct comparison.

In summary: the Singleton pattern is a creational pattern. It ensures that only one object of a class is created by making the constructor `private`, making a `getInstance` method that checks if an instance already exists, and making it `static` so we can call it without creating objects of the class. Therefore, in the code above, `logger1` and `logger2` are exactly the same object. This is because if you look back to the *if statement* in the `Logger` class, the first time it runs (for `logger1`), the `Logger` object is created. The second time it runs (for `logger2`), it will simply call the object that was just created.

FACTORY DESIGN PATTERN

If you think of a factory, you might think of a huge warehouse that churns out vast quantities of the same (or similar type of) product. That is the basic purpose of the Factory Pattern: it is a method that **returns one of several possible classes** that share a common superclass. It is particularly useful when you don't know beforehand what class you need and the decision can only be made at run time.

The superclass that the Factory pattern uses is usually implemented as an interface. An interface is a special type of class that acts as a blueprint of classes to come. It is an abstract type that specifies the behaviour of future classes. An interface will contain method headings but the methods in an interface don't contain any logic. The logic for the methods specified in the interface is written in the classes that implement the interface. In essence, an interface shows a class what to do, and not how to do it.

To see an example of working with the factory pattern, let's go back to our `Animal` class from the Inheritance task with subclasses `Cheetah`, `Lion` and `Kangaroo`.
To start, create the `Animal` interface that contains the `create` method.

Animal.java

```
public interface Animal {
    void create();
}
```

Next, create **Lion**, **Cheetah** and **Kangaroo** classes that implement the **Animal** interface. Since each of these classes implements **Animal** interface, they must all implement the **create** method specified in the **Animal** interface.

Lion.java

```
public class Lion implements Animal {

   @Override
   public void create() {
      System.out.println("You've just created a lion!");
   }
}
```

Looking at the above code, you will notice something that might be unfamiliar: **@Override**. Method overriding is used when you are overriding a method of a parent class. In this case, you are overriding the *create* method from the implemented **Animal** interface that has the same method name. It means that *Lion's* version of the method will execute, not **Animal's**. You will notice that **@Override** is used for the **Cheetah** and **Kangaroo** classes too.

Cheetah.java

```
public class Cheetah implements Animal {

   @Override
   public void create() {
      System.out.println("You've just created a cheetah!");
   }
}
```

Kangaroo.java

```
public class Kangaroo implements Animal {
   @Override
   public void create() {
      System.out.println("You've just created a kangaroo!");
   }
}
```

The next step is to create the **AnimalFactory** class. Here, we use the **getAnimal** method to return an **Animal** object depending on what input is received.

AnimalFactory.java

```
public class AnimalFactory {

   //use getAnimal method to get an object of type Animal
```

```
   public Animal getAnimal(String animalType){
      if(animalType == null){
         return null;
      }
      if(animalType.equalsIgnoreCase("Kangaroo")){
         return new Kangaroo();

      } else if(animalType.equalsIgnoreCase("Cheetah")){
         return new Cheetah();

      } else if(animalType.equalsIgnoreCase("Lion")){
         return new Lion();
      }

      return null;
   }
}
```

Finally, we use our Main class to create **animal1**, **animal2** and **animal3**. Note how the argument given for **getAnimal** is all lowercase, but the code still executes. This is because of the **.equalsIgnoreCase** method in **AnimalFactory.java**.

Main.java

```
public class Main {

   public static void main(String[] args) {
      AnimalFactory animalFactory = new AnimalFactory();

      //get an object of Kangaroo and call its create method.
      Animal animal1 = animalFactory.getAnimal("kangaroo");
      animal1.create();

      //get an object of Cheetah and call its create method.
      Animal animal2 = animalFactory.getAnimal("cheetah");
      animal2.create();

      //get an object of Lion and call its create method.
      Animal animal3 = animalFactory.getAnimal("lion");
      animal3.create();
   }
}
```

Output

```
You've just created a kangaroo!
You've just created a cheetah!
You've just created a lion!
```

In summary: The Factory Pattern is a creational pattern that allows you to combine the construction of multiple classes with a safe level of separation from the rest of the code.

ADAPTER DESIGN PATTERN

Say you were lost in a foreign country. You come across a sign that you think may give you an idea of where you are, but it is written in a language you can't understand. Someone walks by and you make (somewhat obscure) hand gestures indicating that you need help reading the sign. The person looks at the sign, then translates it in her head and writes down the English translation for you. Now not only have you found your way, but you also found yourself an adaptor. Like your helpful translator, the Adaptor pattern bridges two originally incompatible interfaces. This is done by creating an `Adaptor` class. The Adaptor is also known as a wrapper because it lets you "wrap" things into a class and reuse these actions in the correct situations. The Adapter Pattern is a **structural pattern** that wraps a class to be able to connect it to an otherwise incompatible interface and so is particularly useful when **integrating old objects into new interfaces**.

In the example below, we have an old object and an old interface that needs to be compatible with a new interface. Have a look at the code below:

OldInterface.java

```
public interface OldInterface {
    public void oldVoidMethod();
    public int oldIntMethod(boolean rinse);
}
```

As you can see above, the old interface has two methods — one that is `void` and one that returns a `boolean`. These two methods are expanded on in the `OldObject` below:

OldObject.java

```
public class OldObject implements OldInterface {

    @Override
    public void oldVoidMethod() {
        System.out.println("This is oldVoidMethod, an old interface
method.");
    }

    @Override
```

```
    public int oldIntMethod(boolean rinse) {
        System.out.println("This is oldIntMethod, an old interface
method.");
        if (rinse) {
            return 0;
        } else {
            return 1;
        }
    }
}
```

Next, we have the new interface that also has two methods — one `void`, and one that returns a `boolean`.

NewInterface.java:

```
public interface NewInterface{
    public void newVoidMethod();
    public int newIntMethod(boolean rinse);
}
```

Next is the adaptor itself. Notice how it starts with an instance of `OldObject`. It then has methods with the `NewInterface` names that call the `OldInterface` methods.

Adapter.java

```
public class Adapter implements NewInterface{
    private OldInterface adapterTarget = new OldObject();

    public Adapter(OldInterface adapterTarget) {
        this.adapterTarget = adapterTarget;
    }

    @Override
    public void newVoidMethod() {
        adapterTarget.oldVoidMethod();
    }

    @Override
    public int newIntMethod(boolean rinse) {
        return adapterTarget.oldIntMethod(rinse);
    }
}
```

Next is the Main method that executes the `Adaptor`.

Main.java

```
public class Main{
    public static void main(String[] args) {
        OldInterface anOldObject = new OldObject();
        NewInterface adaptedOldObject = new Adapter(anOldObject);

        System.out.println("Both these methods are in an object implementing
NewInterface!");

        System.out.println("newVoidMethod:");
        adaptedOldObject.newVoidMethod();

        System.out.println("newIntMethod:");
        System.out.println(adaptedOldObject.newIntMethod(true));
    }
}
```

Output:

```
Both these methods are in an object implementing NewInterface!
newVoidMethod:
This is oldVoidMethod, an old interface method.
newIntMethod:
This is oldIntMethod, an old interface method.
0
```

In summary: the Adapter Pattern is a structural pattern that wraps a class to be able to connect it to an otherwise-incompatible interface and so is particularly useful when integrating old objects into new interfaces.

DECORATOR DESIGN PATTERN

Much like a bow on a birthday present, the Decorator pattern adds something extra (the bow) to an object that already exists (the present). Like the Adapter Pattern, the Decorator Pattern is a **structural pattern** that also incorporates wrapping. However, unlike the Adapter Pattern, it **wraps an object,** not a class, in order **to add functionality or features to the object**. The purpose of the pattern is to make it easy to add or remove functions if needed. This is done by wrapping a single object (not class) to add new behaviour. It has an attribute for that object and has all the same methods as the wrapped object.

To begin with, we start with the interface.

Shape.java

```
public interface Shape {
```

```
    String draw();
}
```

Next, we have a `Rectangle` class that implements the `Shape` interface.

Rectangle.java

```
public class Rectangle implements Shape {

    @Override
    public String draw() {
        return "This rectangle:\n";
    }
}
```

Next, we create a class that will decorate the Rectangle object, `ShapeDecorator`, that also implements the `Shape` interface.

ShapeDecorator.java

```
public abstract class ShapeDecorator implements Shape {
    protected Shape decoratedShape;

    public ShapeDecorator(Shape decoratedShape){
        this.decoratedShape = decoratedShape;
    }

    @Override
    public String draw() {
        return decoratedShape.draw();
    }
}
```

Now, we create a class that will decorate the rectangle with a colour, `ColourRectangle` This class extends the `ShapeDecorator` class.

ColourRectangle.java

```
public class ColourRectangle extends ShapeDecorator {

    public ColourRectangle(Shape decoratedShape) {
        super(decoratedShape);
    }

    @Override
    public String draw() {
        return super.draw() + decorateWithColour();
    }
```

```
    private String decorateWithColour() {
        return "- is red\n";
    }
}
```

Again, we create a class, `BorderRectangle` that also extends the `ShapeDecorator` class. This class adds a border to the shape object.

BorderRectangle.java

```
public class BorderRectangle extends ShapeDecorator {
   public BorderRectangle(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public String draw() {
      return super.draw() + decorateWithBorder();
   }
    private String decorateWithBorder() {
        return "- has a thick black border\n";
    }
}
```

Finally, we have our Main class to show how the Decorator Pattern has worked. Notice how you can stack the decorators.

Main.java

```
public class Main {

   public static void main(String[] args) {

        Shape rectangleObject1 = new Rectangle();
        Shape rectangleObject2 = new BorderRectangle(rectangleObject1);
        Shape rectangleObject3 = new ColourRectangle(rectangleObject2);

        System.out.println(rectangleObject1.draw());
        System.out.println(rectangleObject2.draw());
        System.out.println(rectangleObject3.draw());
   }
}
```

Output:

```
This rectangle:

```

```
This rectangle:
- has a thick black border

This rectangle:
- has a thick black border
- is red
```

In summary: Like the Adapter Pattern, the Decorator Pattern is a structural pattern that also incorporates wrapping. However, unlike the Adapter Pattern, it wraps an object, not a class, in order to add functionality or features to the object. This makes it easy to add or remove functions from an object if needed.

STRATEGY DESIGN PATTERN

The final pattern we cover in this task is the Strategy Design Pattern. Imagine you were creating a character in a videogame that needed to fight an enemy character. Now imagine that your character could have different armour and abilities based on the opponent that it faces. Pretty cool, right? That is similar to the function of a Strategy design pattern: a behaviour is chosen based on other information that is given.

The Strategy Pattern is a **behavioural pattern** as it deals with object communication and interaction. In this pattern, a class behaviour or **algorithm can be changed at runtime**: we create objects which represent various strategies and a `Context` object whose behaviour varies as per its `Strategy` object. This allows us to change entire algorithms at will.

So, say we have an object, and we want it to be able to have a particular behaviour, but we only want to decide this behaviour at runtime. Instead of wrapping an object (as in the Decorator Pattern), we wrap a `Strategy` class with a `Context` object. That way, other classes can call the `Context` object without worrying about the `Strategy` class. This allows for entire algorithms to change quickly and easily.

Once again, we begin with an interface.

Strategy.java

```
public interface Strategy {
    public String makeChange(String word);
}
```

Then, create concrete classes that implement the interface. In this case, we have `LowerChange` (makes the word all lowercase), `UpperChange` (makes the word all uppercase) and `EchoChange` (repeats the original word).

LowerChange.java

```
public class LowerChange implements Strategy {
    @Override
    public String makeChange(String word) {
       return word.toLowerCase();
    }
 }
```

UpperChange.java

```
public class UpperChange implements Strategy {
    @Override
    public String makeChange(String word) {
       return word.toUpperCase();
    }
 }
```

EchoChange.java

```
public class EchoChange implements Strategy {
    @Override
    public String makeChange(String word) {
       return word + word;
    }
 }
```

Now create the `Context` class.

Context.java

```
public class Context {
    private Strategy strategy;

    public Context(Strategy strategy){
       this.strategy = strategy;
    }

    public String executeStrategy(String word){
       return strategy.makeChange(word);
    }
 }
```

Finally, create a Main method that uses the `Context` class to see the change in behaviour when it changes its `Strategy`.

Main.java

```
public class Main {
    public static void main(String[] args) {
       Context context = new Context(new LowerChange());
       System.out.println("Lowercase: " + context.executeStrategy("HeLlO"));

       context = new Context(new UpperChange());
       System.out.println("Uppercase: " + context.executeStrategy("HeLlO"));

       context = new Context(new EchoChange());
       System.out.println("Echo: " + context.executeStrategy("HeLlO"));
    }
 }
```

Output:

```
Lowercase: hello
Uppercase: HELLO
Echo: HeLlOHeLlO
```

In summary: The Strategy pattern is a behavioural pattern as it deals with object communication and interaction. In this pattern, a class behaviour or algorithm can be changed at runtime: we create objects which represent various strategies, and a `Context` object whose behaviour varies as per its `Strategy` object. This allows us to change entire algorithms at will.

Compulsory Task 1

Follow these steps:

- Download the **SingletonExercise.java** file.
- Change the *Deck* class into a *Singleton* class.
- Change the main method to get the *Deck* instance rather than creating a new object.
- The program should print all 52 cards in random order.

Compulsory Task 2

Follow these steps:

- Using one of the patterns described, write the code needed to create 3 student objects. Each student object should have a *describe()* method that returns a string that describes the student. Each student will be described using a full name and a student number. The rest of the description will be built up based on the student's activity at HyperionDev. For example, the description should explain which Bootcamps the student has registered for and which level of the Bootcamp has been completed.

  Below is sample output:

  Student 1: Susan Smith:
  - *Registered for the Software Engineering Bootcamp*
  - *Completed level 1*

  Student 2: Michael Jackson:
  - *Registered for the Web Development Bootcamp*
  - *Completed level 1*
  - *Completed level 2*
  - *Completed level 3*
  - *Registered for the Software Engineering Bootcamp*

  Student 3: Saoirse Ronan
  - *Registered for the Web Development Bootcamp*
  - *Completed level 1*
  - *Completed level 2*
  - *Completed level 3*
  - *Registered for the Software Engineering Bootcamp*
  - *Completed level 1*

Rate us
Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

References:

Gabriel, D. (2018). What are Patterns. Retrieved from The Hillside Group: https://hillside.net/patterns-definition

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns. Elements of Reusable Object-Oriented Software. Boston, Massachusetts: Addison-Wesley Professional.