Design Pattern Review

# Understanding Design Principles

A *design principle* is an established idea or best practice that facilitates the software design process. In general, following good design principles leads to:

* More logical code
* Code that is easier to understand
* Classes that are easier to reuse in other relationships and applications
* Code that is easier to maintain and that adapts more readily to changes in the application requirements

## Encapsulating Data

One fundamental principle of object-oriented design is the concept of encapsulating data. In software development, *encapsulation* is the idea of combining fields and methods in a class such that the methods operate on the data, as opposed to the users of the class accessing the fields directly. In Java, it is commonly implemented with private instance members that have public methods to retrieve or modify the data, commonly referred to as getters and setters.

## Using Inheritance

In objet-oriented design, we describe the property of an object being an instance of a data type as having an *is-a relationship****.*** The is-a relationship is also known as the inheritance test.

The fundamental result of the is-a principle is that if B is-a A, then any instance of B can be treated like an instance of A. This holds true for a child that is a subclass of any parent, be it a direct subclass or a distant child.

**Using instanceof**

In the expression obj1 instanceof A, the expression returns true if the reference obj1 is an instance of the class A, a subclass of A, or a class that implements the A interface.

Note, the compiler checks if there is any way for the object reference to be A and will not let the code compile if they are completely unrelated classes. For example given classes:

class HeavyAnimal { } class Hippo extends HeavyAnimal { } class Elephant extends HeavyAnimal { }

The below code is allowed,

HeavyAnimal hipp = new Hippo();

Boolean b1 = hippo instanceof Hippo; // true

Boolean b2 = hippo instanceof HeavyAnimal // true

Boolean b3 = hippo instanceof Elephant; // false

However, the below code will not compile.

HeavyAnimal hippo = new Hippo()

Boolean b4 = hippo instanceof Number; // compiler error

## Using Composition

In object-oriented design, we refer to object composition as the property of constructing a class using references to other classes in order to reuse the functionality of the other class. In particular, the class contains the other classes in the has-a sencse and may delegate methods to the other classes.

Object composition should be thought of as an alternative to inheritance and is often used to simulate polymorphic behavior that cannot be achieved via single inheritance. For example, imagine that we have the following two classes:

public class Flippers {

public void flap() {

System.out.println(“Flap flap”);

}

}

public class WebbedFeet {

public void kick {

System.out.println(“kick kick”);

}

}

Trying to relate these objects using inheritance does not make sense, as WebbedFeet are not the same as Flippers. Instead, we can compose a new class that contains both of these objects and delegates its methods to them, such as in the following code:

public class Penguin {

private final Flippers flippers

private final WebbedFeet webbedFeet;

public Penguin() {

this.flippers = new Flippers();

this.webbedFeet = new WebbedFeet();

}

public void flap() { flippers.flap() }

public void kick() {

this.webbedFeet.kick();

}

}

One of the advantages of object composition over inheritance is that it tends to promote greater code reuse. By using object composition, you gain access to other classes and methods that would be difficult to obtain via Java’s single-inheritance model.

Object composition may seem more attractive than inheritance because of its reusable nature, but bear in mind that one of the strengths of Java is its powerful inheritance model. Object composition still requires you to explicitly expose the underlying methods and values manually, whereas inheritance includes protected and public members automatically. Also, using method overloading to determine dynamically which method to select at runtime is an extremely powerful tool for building intelligent classes. In other words, both object composition and inheritance have their proper place in developing good code.

# Working with Design Patterns

A *design pattern* is an established general solution to a commonly occurring software development problem. The purpose of a design pattern is to leverage the wealth of knowledge of developers who have come before you in order to easily solve problems that you may encounter. It also gives developers a common vocabulary in which they can discuss common problems and solutions. For example, if you say that you wrote getters/setters or implemented the singletin pattern, most developers will understand the structure of your code without you having to get into the low-level details.

Gangs of Four Design Patterns is the collection of 23 design patterns from the book “Design Patterns: Elements of Reusable Object-Oriented Software”. This book was first published in 1994 and it’s one of the most popular books to learn design patterns. The book was authored by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. It got nicknamed as Gangs of Four design patterns because of four authors.

Design Patterns are divided into three categories:

1. Creational: The design patterns that deal with the creation of an object.
2. Structural: The design patterns in this category deals with the class structure such as Inheritance and Composition.
3. Behavioral: This type of design patterns provide solution for the better interaction between objects, how to provide lose coupling, and flexibility to extend easily in future.

The 3 design patterns in the creational design patterns category you should know are:.

**Singleton**:

The singleton pattern restricts the initialization of a class to ensure that only one instance of the class can be created.

**Immutable:**

The immutable constructs an object with all fields final thus unchangeable once created.

**Factory**:

The factory pattern takes out the responsibility of instantiating a object from the class to a Factory class.

**Builder**:

Creating an object step by step and a method to finally get the object instance.

## Applying the Singleton Pattern

is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

**Problem**

The Singleton pattern solves two problems at the same time, violating the Single Responsibility Principle:

1. Ensure that a class has just a single instance.
   1. Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file.
2. Provide a global access point to that instance.
   1. Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. It also protects that instance from being overwritten by other code.
   2. There’s another side to this problem: you don’t want the code that solves problem #1 to be scattered all over your program. It’s much better to have it within one class, especially if the rest of your code already depends on it.

**Solution**

All implementations of the Singleton have these steps in common:

1. Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
2. Make a private static property to hold the single instance of the class
3. Make a public static method to access the instance of the class.

**Lazy Initialization**

To delay creation of the instance until it is needed, make sure the property is not marked final so it initializes to null when the class is loaded. Then when the access method is first called it will check

if (instance == null)

And then either call the private constructor or return the existing instance.

Lazy initialization reduces memory usage and imporoves performance whan an application nstarts up.

Thread safety is an issue that singletons being accessed by multiple threads have to deal with. We will talk more about this next week.

## Creating immutable objects

Immutable classes are the classes that did not change once initialized, which means once an object is created for an immutable class then you will not able to change anything in that class. Due to this non-changing behavior, developers did not have to worry about thread safety, and good to cache as their values are not changing. If you want to change anything in the object you have to create a new object.

**How to create an immutable class?**

There are a few steps that you have to follow let’s see those steps and understand them.

1. Create all fields private and final. (By making fields private and final, you are making sure that no one will able to change the field value once an object is created)
2. Make class final (By making a class final you are making sure no one can extend that class, so it can change the value by overriding it)
3. There are only getter methods in your class no setter methods, as you did not want to change any field value.
4. All fields are initialized using a constructor, so values can be assigned at the time of object creation.
5. Last but most important point. If you have any nested objects (nor a primitive type nither a string), in that case, two more sub-steps are added.
6. Return a clone of that object in the getter method. (As that object is also a reference and this is mutable in nature so any changes in this object's values make the whole class mutable)
7. While creating an object when you are given any nested object, you have to deep copy that parameter and then assign it to the field. Otherwise, if you assign the copy which also contains any nested object make it mutable in nature.

## Using the Builder Pattern

*Builder* is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

**Problem**

Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. Or even worse: scattered all over the client code.

The problem of a constructor growing too large actually has a name, referred to as the telescoping constructor anti-pattern. An anti-pattern is a common solution to a reoccurring problem that tends to lead to unmanageable or difficult-to-use code. Anti-atterns often appear in complex systems as time goes on, when developers implement series of successive changes without considering the olong-term effects of their actions.

For example, wht the telescoping constructor ant-pattern, the class may start of fwith only two parameters in the constructor. Another developer may con in and add another parameter, thinking it’s only one more. A third developer may udpadt the class and addd afourth paramenter and so on and so forth until the class has dozens of parameters and is in depsperate need of refactoring.

**Solution**

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builders.

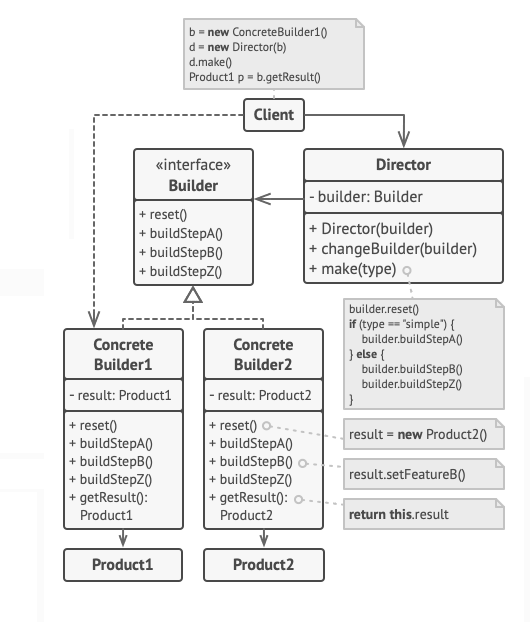
The pattern organizes object construction into a set of steps (buildWalls, buildDoor, etc.). To create an object, you execute a series of these steps on a builder object. The important part is that you don’t need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object.

Some of the construction steps might require different implementation when you need to build various representations of the product. For example, walls of a cabin may be built of wood, but the castle walls must be built with stone.

In this case, you can create several different builder classes that implement the same set of building steps, but in a different manner. Then you can use these builders in the construction process (i.e., an ordered set of calls to the building steps) to produce different kinds of objects.

**Director**

You can go further and extract a series of calls to the builder steps you use to construct a product into a separate class called director. The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps. Having a director class in your program isn’t strictly necessary. You can always call the building steps in a specific order directly from the client code. However, the director class might be a good place to put various construction routines so you can reuse them across your program.



Creating Objects with the Factory Pattern

Problem: How do we write code that creates objects in which the precise type of the object may not be known until runtime?

Solution

The factory pattern sometimes referred to as the factory method pattern, is a creational pattern based on the idea of using a factory class to produce instances of objects based on a set of input parameters. It is like the builder pattern, although it focuses on supporting class polymorphism.

Factory patterns are often, although not always, implemented using static methods that return objects and do not require a pointer to an instance of the factory class. It is also good coding practice to postfix the class name with the word Factory.

replace direct object construction calls (using the new operator) with calls to a special factory method. Don’t worry: the objects are still created via the new operator, but it’s being called from within the factory method. Objects returned by a factory method are often referred to as products.

At first glance, this change may look pointless: we just moved the constructor call from one part of the program to another. However, consider this: now you can override the factory method in a subclass and change the class of products being created by the method.

There’s a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. Also, the factory method in the base class should have its return type declared as this interface.

The code that uses the factory method (often called the client code) doesn’t see a difference between the actual products returned by various subclasses. The client treats all the products as abstract product class instances. The client knows that all these objects are supposed to have the same set of methods declared by the interface, but exactly how it works isn’t important to the client.

