Design Pattern Review

# Understanding Design Principles

A *design principle* is \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. In general, following good design principles leads to:

* More \_\_\_\_\_ code
* Code that is easier \_\_\_\_\_\_\_\_\_
* Classes that are easier to \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* Code that is easier to \_\_\_\_\_\_\_\_\_ 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 \_\_\_\_\_\_\_\_\_\_\_\_ 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 \_\_\_\_\_\_\_\_ instance members that have \_\_\_\_\_\_ methods to retrieve or modify the data, commonly referred to as \_\_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_\_\_\_.

## Using Inheritance

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

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

**Using instanceof**

In the expression obj1 instanceof A, the expression returns true if the reference obj1 is \_\_\_\_\_\_\_\_\_\_ of the class A, a \_\_\_\_\_\_\_\_\_ of A, or a class that \_\_\_\_\_\_\_ 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 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. 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 \_\_\_\_\_\_\_\_\_\_\_\_ to other classes in order to \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. In particular, the class contains the other classes in the \_\_\_\_\_\_\_ sense and may \_\_\_\_\_\_\_\_\_ methods to the other classes.

Object composition should be thought of as an alternative to \_\_\_\_\_\_\_\_\_\_\_\_ and is often used to simulate polymorphic behavior that cannot be achieved via \_\_\_\_\_\_\_\_\_\_\_\_\_\_. 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 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. By using object composition, you gain \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ that would be difficult to obtain via Java’s \_\_\_\_\_\_-inheritance model.

Object composition may seem more attractive than inheritance because \_\_\_\_\_\_\_\_\_\_\_\_\_, 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 \_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_ members automatically. Also, using method \_\_\_\_\_\_\_ 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 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. 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 singleton 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 \_\_ 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. \_\_\_\_\_\_\_\_\_\_\_\_\_
2. \_\_\_\_\_\_\_\_\_\_\_\_\_
3. \_\_\_\_\_\_\_\_\_\_\_\_\_

The 4 design patterns in the creational design patterns category you should know are:

**Singleton**:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Immutable:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Factory**:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Builder**:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## 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 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
   1. Why would anyone want to control how many instances a class has? The most common reason for this is \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_—for example, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.
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 \_\_\_\_\_\_\_\_\_\_\_
2. Make \_\_\_\_\_\_\_\_\_\_\_\_
3. Make \_\_\_\_\_\_\_\_\_\_\_\_

**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 (\_\_\_\_\_\_\_\_\_\_)

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 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, which means once an object is created for an immutable class then you will not \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

Due to this non-changing behavior, developers did not have to worry about \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, and good to cache as their values are not changing. If you want to change anything in the object you have \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**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. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
2. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
3. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
4. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
5. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
   1. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
   2. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Using the Builder Pattern

*Builder* is a creational design pattern that \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**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 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ anti-pattern. An anti-pattern is a common solution to a reoccurring problem that tends to lead to \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. Anti-atterns often appear in complex systems as time goes on, when developers implement series of successive changes without considering the long-term effects of their actions.

For example, what 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 update 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.

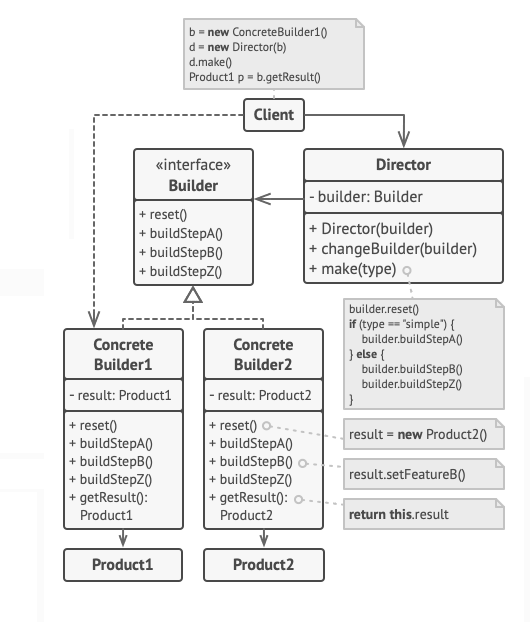
The pattern organizes object construction into a set of steps (buildWalls, buildDoor, etc.). To create an object, you execute a series of these steps on a builder object. The important part is that you don’t need to call all of the steps. You can call only those steps that are necessary for producing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

