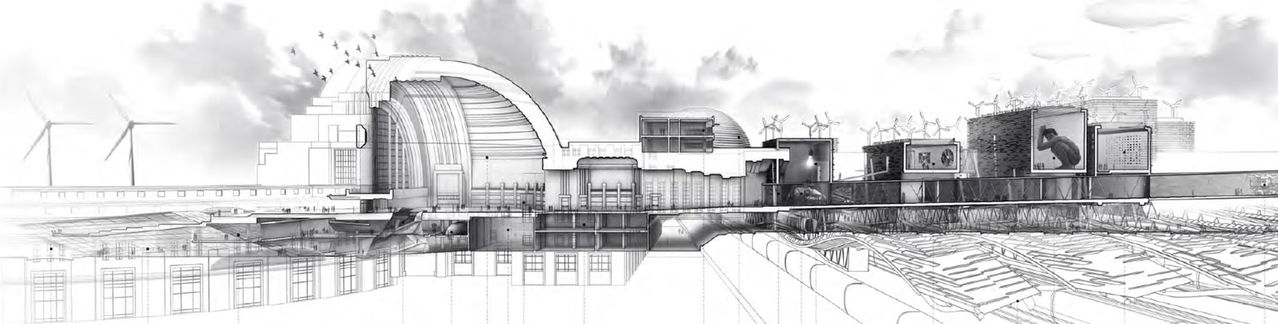
**[8 design patterns that every developer should know](http://www.thedevpiece.com/design-patterns-that-every-developer-should-know/" \o "8 design patterns that every developer should know)**

[**Gabriel Francisco**](http://www.thedevpiece.com/)

*Software Engineer at UOL, Bachelor of Computer Science and MBA in Service-oriented Software Engineering.*



As a developer, you are constantly **resolving problems**. Many of these problems probably were **already solved by other developers**, so, why do we need to solve them again?   
You have a problem, specifically a **common problem**, so, you try to find out if anyone already did the job solving, before you try to resolve it, right? You probably don't want to **reinvent the wheel**, do ya? Design patterns are there for these situations. Unfortunately there are a lot of developers who still refuse to use a few patterns, mostly because they just don't know them or even don't know how to fit those patterns into some problems.

The main question that every developer ask before using design patterns is: **are design patterns really that important?** Someone could make a big explanation about this, but, I'll be succinct: **yes, they are.** And I'm going to list three main reasons:

* It saves your time. You don't need to use your brain to create a possible solution from nowhere and lose lots of time;
* Design patterns are well-known. When you say: *"I used a factory to create that object"* everyone will understand what you are talking about;
* Most of these patterns are easy to understand. Probably your solution will be not as elegant and easy as a design pattern.

Okay, so I'm going to talk about the most useful design patterns and explain in which situations you should use them. :)

Before continuing, let's be clear: **there is no silver bullet.** You need to adapt the design pattern to your problem and not try to adapt your problem to the design pattern. If you need to do it, probably it is not the right design pattern for this problem.

Well, here we go!

**Singleton**

This is the most used pattern. A lot of framework already implement this pattern, such as Spring, CDI (via @ApplicationScoped) or EJBs (using @Singleton). Still, it is nice to know how to implement it the old way. ;)

public class SingletonSample {

private static SingletonSample instance = null;

private SingletonSample() {

}

public static SingletonSample getInstance() {

if(instance == null) {

instance = new SingletonSample();

}

return instance;

}

}

There is no secret. You need an object that only needs to be instantiate once, so, you can use a singleton. The class needs to declare a private constructor to prevent people to instantiate it from outside the class. Also, you need to declare a static field of the type of the singleton. The method getInstance() assures that only one instance of this class is created at runtime.

**Initialization on Demand Holder**

This pattern is much like the Singleton, but the Initialization on Demand Holder has critical advantage over the Singleton: It is thread safe.   
The getInstance() method from the Singleton pattern is not thread safe, not if you don't make it synchronized, so, it can be instantiate more than once. If you do make your method synchronized, you are making your getInstance() method slower than it could be if it were not.

public class SingletonSample {

private SingletonSample() {

}

public static SingletonSample getInstance() {

return SingletonSampleHolder.INSTANCE;

}

private static class SingletonSampleHolder {

private static final SingletonSample INSTANCE = new SingletonSample();

}

}

As the name says, this type of Singleton does not initialize the instance until the getInstance() is called, with the advantage that it is thread safe.   
Since the JVM does not load the SingletonSampleHolder at startup because there is no static reference of this class anywhere in the code, the instance is not created until the first call of the *getInstance()* method.

**The Strategy and The Factory pattern**

Both are well-known design patterns. For sure two of the most useful design patterns, specially using them together.   
When both are combined, you can create objects from a given qualifier. The example is right below:

public interface Building {

String getType();

}

public class House implements Building {

public String getType(){

return "house"

}

}

public class Edifice implements Building {

public String getType(){

return "edifice"

}

}

public class BuildingFactory {

private static Map<String, Building> instances;

static {

instances = new HashMap<>();

instances.put("house", new House());

instances.put("edifice", new Edifice());

}

public static <T extends Building> T getBuilding(String type){

return (T) instances.get(type);

}

}

Building building = BuildingFactory.getBuilding("house");

If you need a specific building, you need just to give a building type and one will be returned, or null there is no instance for this type. Very useful, besides, it gives us a chance to use the most of polymorphism.

**Fluent Builder**

Some objects require lots of parameters to be created. In this case, either using the constructor to create this object or using the setters will make our code ugly and hard to understand.

The builder pattern can help us in this case.

public class Product {

private String id;

private String name;

private String description;

private Double value;

private Product(Builder builder) {

setId(builder.id);

setName(builder.name);

setDescription(builder.description);

setValue(builder.value);

}

public static Builder newProduct() {

return new Builder();

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

public Double getValue() {

return value;

}

public void setValue(Double value) {

this.value = value;

}

public static final class Builder {

private String id;

private String name;

private String description;

private Double value;

private Builder() {

}

public Builder id(String id) {

this.id = id;

return this;

}

public Builder name(String name) {

this.name = name;

return this;

}

public Builder description(String description) {

this.description = description;

return this;

}

public Builder value(Double value) {

this.value = value;

return this;

}

public Product build() {

return new Product(this);

}

}

}

Product product = Product.newProduct()

.id(1l)

.description("TV 46'")

.value(2000.00)

.name("TV 46'")

.build();

Okay, this class is not that big and it does not have a lot of fields, but, this is just for an example, so... :P

**Chain of Responsibility**

You always need to build applications which require a lot of business logic. Behind this much of logic, there is always high complexity. This high complexity makes our code harder to understand, as well as harder to track, to log and so on.   
The CoR pattern makes us break our code into little pieces and organize them into sequential steps.

public interface Command<T>{

boolean execute(T context);

}

public class FirstCommand implements Command<Map<String, Object>>{

public boolean execute(Map<String, Object> context){

//doing something in here

}

}

public class SecondCommand implements Command<Map<String, Object>>{

public boolean execute(Map<String, Object> context){

//doing something in here

}

}

public class Chain {

public List<Command> commands;

public Chain(Command... commands){

this.commands = Arrays.asList(commands);

}

public void start(Object context){

for(Command command : commands){

boolean shouldStop = command.execute(context);

if(shouldStop){

return;

}

}

}

}

Chain chain = new Chain(new FirstCommand(), new SecondCommand());

Map<String, Object> context = new HashMap<>();

context.put("some parameter", "some value");

chain.start(context);

Now we can break our code into Commands and separate each logic in just one place. Also, we can reorganize as we want, making our code more decoupled, thus solving our complexity problem.

**Template method**

This pattern defines a skeleton in a method for a operation. It is very useful when your have common method calls but different behaviors. This pattern is totally based in polymorphism.

public abstract class Animal {

public abstract void makeSound();

public abstract void eatFood();

public abstract void sleep();

public void doEveryday(){

makeSound();

eatFood();

sleep();

}

}

public class Dog extends Animal {

public void makeSound(){

//bark!

}

public void eatFood(){

//eat dog food

}

public void sleep(){

//sleep a lot!

}

}

public class Cat extends Animal {

public void makeSound(){

//meow!

}

public void eatFood(){

//eat cat food

}

public void sleep(){

//sleep just a little bit

}

}

**State pattern**

A lot of objects have states. For example, a radio. A radio have basically two states: on and off. Can we represent this in object oriented programming? Well, yes:

public class Radio {

private boolean on;

private RadioState state;

public Radio(RadioState state){

this.state = state;

}

public void execute(){

state.execute(this);

}

public void setState(RadioState state){

this.state = state;

}

public void setOn(boolean on){

this.on = on;

}

public boolean isOn(){

return on;

}

public boolean isOff(){

return !on;

}

}

public interface RadioState {

void execute(Radio radio);

}

public class OnRadioState implements RadioState {

public void execute(Radio radio){

//throws exception if radio is already on

radio.setOn(true);

}

}

public class OffRadioState implements RadioState {

public void execute(Radio radio){

//throws exception if radio is already off

radio.setOn(false);

}

}

Radio radio = new Radio(new OffRadioState()); //initial status

radio.setState(new OnRadioState());

radio.execute(); //radio on

radio.setState(new OffRadioState());

radio.execute(); //radio off

This is just a simple example, but, when we talk about an object that have a lot of states, this control can help a lot. You can define rules to create final states and states that requires an previous state to be executed. For example, in this radio scenario, you can only turn a radio off if the radio is on, otherwise an exception will be thrown. Basically you can do whatever the business tell you to do.

**Finalizing...**

Of course there are a lot of others useful design patterns, maybe I'm going to talk about them in the future, but, for now I think those are sufficient for you design your app better. ;)

I hope I could help you! Any question or any compliment, please, leave a message. =]