Design pattern: singleton, prototype and builder

In my previous article, I spoke about the factory patterns. These patterns are part of creational patterns. In this post we'll focus on the rest of the creational patterns: Singleton, Builder and Prototype.

In my opinion, these patterns are less important than factories. Yet, it's still useful to know them. Following the same logic as my previous article, I'll provide an UML description, simple java examples (so that even if you don't know java you can understand) and present real examples from famous Java frameworks or APIs.

I'll sometimes use factories so read my previous article if you don't feel comfortable with factory patterns.

Creational Patterns

Creational patterns are design patterns that deal with object initialization and overcome the limitations of constructors. The **Gang of Four** in their book "*Design Patterns: Elements of Reusable Object-Oriented Software*" described five of them:

- Singleton,
- Builder,
- Prototype,
- Abstract Factory,
- Factory pattern.

Since this book was released (in 1994), many creational patterns have been invented:

- other type of factories (like the static one),

- pool pattern,
- lazy initialization,
- dependency injection,
- service locator,
- …

In this post, we'll only focus on the rest of the GoF's creational patterns I haven't already described. As I said in the introduction, they are less important than factories because you can live without them (whereas factories are the backbone of many applications and frameworks). But, they are useful and unlike factories they don't make the code much more difficult to read.

Singleton Pattern

This pattern is the most famous. During the last decades, it was over-used but its popularity has decreased since. I personally avoid using it since it makes the code **more difficult to unit test** and creates a **tight coupling**. I prefer to use a factory (like the Spring container) that deals with the number of authorized instances of a class, we'll speak about this approach. I think you should avoid the singleton pattern. In fact, the most important use of this pattern is to be able to answer an interviewer when he asks "what is a singleton?". This pattern is very controversial and there are still people in favor of it.

That being said, according to the GoF a singleton aims to:

> *"Ensure a class only has one instance, and provide a global point of access to it"*

So, there are 2 requirements for a class to be a singleton:

- Having a unique instance
- Being accessible from anywhere

Some people only think about the first requirement (like me few years ago). In this case, the class is only a **single instance**.

Let's look how to do a singleton in UML

In this UML diagram, the Singleton class has 3 items:

- a class attribute (instance): this attribute contains the unique instance of the singleton class.
- a class public method (getInstance()) : it provides the only way to get the unique instance of the class Singleton. The method can be called from anywhere since it's a class method (and not an instance method).
- a private constructor (Singleton()) : it prevents anyone to instantiate a Singleton with a constructor.

In this example, a developer that needs an instance of Singleton will call the Singleton.getInstance() class method.

The singleton instance inside the Singleton class can be:

- pre-initialized (which means it is instantiated before someone call getInstance())
- lazy-initialized (which means it is instantiated during the first call of getInstance())

Of course a real singleton has other methods and attributes to do its business logic.

Java implementations

Here is a very simple way to create a singleton in Java using the pre-instantiated approach.

Using this way, the singleton instance is created only once when the class is loaded by the classloader. If the class is never used in your code, the instance won't be instantiated (because the classloader of the JVM won't load it) and therefore waste memory. But if the class appears and you don't use it (for example if it's only used in a very very rare condition), the singleton will be

initialized for nothing. Unless your singleton takes a huge amount of memory, you should use this way.

Still, if you need to create your singleton only when it's really used (the lazy initialization), here is a way to do it in a multithreaded environment. This part is a bit tricky since it involves thread coherency.

As I said, it's really more difficult to read (this is why the pre-instanciated way is better). This singleton involves a lock to avoid that 2 threads calling the getInstance() at the same time create 2 instances. Since the lock is costly, there is first a test without a lock then a test with the lock (it's a double-checked locking) so that when the instance already exists the lock is not used.

Other particularity, the instance has to be volatile to ensure that its state is the same on the different processor cores when it is created.

When do you need to use a singleton?

- When you need only one resource (a database connection, a socket connection …)
- To avoid multiple instances of a stateless class to avoid memory waste
- For business reasons

You shouldn't use singleton for sharing variables/data between different objects since it produces a very tight coupling!

Why shouldn't you use a singleton?

At the beginning, I said that you shouldn't use singletons because of the way you get the singleton. It's based on a class function that can be called anywhere in the code. I read an excellent answer on stackoverflow that gives 4 reasons why it's bad:

- With singletons, you hide the dependencies between the classes instead of

exposing them through the interfaces. This means you need to read the code of each method to know if a class is using another class.
- They violate the single responsibility principle: they control their own creation and lifecycle (using lazy initialization, the Singleton chose when it is created). A class should only focus on what it is meant to do. If you have a Singleton that manages people, it should only manage people and not how/when it is created.
- They inherently cause code to be tightly coupled. This makes faking or mocking them for unit testing very difficult.
- They carry states around for the lifetime of the application (for stateful singletons).
    - It makes unit testing difficult since you can end up with a situation where tests need to be ordered which is a piece of nonsense. By definition, each unit test should be independent from each other.
    - Moreover, it makes the code less predictable.

Ok, so singleton are bad. But what should you use instead?

Use a single instance instead of a singleton

A singleton is just a specific type of single instance that can be getting anywhere with its class method. If you remove this second requirement, you remove many problems. But how can you deal with single instances?

A possible way is to manage single instances with a factory and Dependency Injection (it will be the subject of a future post).

Let's take an example to understand:

- You have a PersonBusiness class that needs a unique DatabaseConnection instance.
- Instead of using a singleton to get this connection, the PersonBusiness will have a DatabaseConnection attribute.
- This attribute will be injected at the instantiation of PersonBusiness by its

constructor. Of course, you can inject any type of DatabaseConnection:
    - A MysqlDatabaseConnection for your development environment
    - A OracleDatabaseConnection for the production environment
    - A MockDatabaseConnection for the unit tests
- At this stage, nothing prevents the DatabaseConnection to be unique. This is where the factory is useful. You delegate the creation of PersonBusiness to a factory and this factory also takes care of the creation of DatabaseConnection:
    - It chooses which kind of connection to create (for example using a property file that specify the type of connection)
    - It ensures that the DatabaseConnection is unique.

If you didn't understand what I've just said, look the next java example then re-read this part again, it should by more comprehensive. Otherwise, feel free to tell me.

Here is an example in Java where the factory creates a MysqlDatabaseConnection but you could imagine a more complex factory that decides the type of connection according to a property file or an environment variable.

This is not a good example since the PersonBusiness could have a single instance since it has no state. But you could imagine that there are a ContractBusiness and a HouseBusiness that also needs that unique DatabaseConnection.

Still, I hope you see that using dependency injection + a factory you end up with a single instance of DatabaseConnection in your business classes as if you used a singleton. But this time, it's a loose coupling, which means instead of using a MysqlDatabaseConnection, you can easily use a MockDatabaseConnection for testing only the PersonBusiness class.

Moreover, it's easy to know that PersonBusiness is using a DatabaseConnection. You just have to look at the attributes of the class and not one of the 2000 lines of code of the class (ok, imagine this class has many functions and the overall takes 2000 lines of code).

This approach is used by most Java framework (Spring, Hibernate…) and Java

containers (EJB containers). It's not a real singleton since you can instantiate the class multiple times if you want to and you can't get the instance from everywhere. But if you only create your instances through the factory/container, you'll end up with a unique instance of the class in your code.

Note: I think the Spring Framework is very confusing because its "singleton" scope is only a single instance. It took me some time to understand that it wasn't a real GoF's singleton.

Some thoughts

The single instance has the same drawback than the singleton when it comes to global states. **You should avoid using a single instance to share data between different classes!** The only exception I see is caching:

- Imagine you have a trading application that makes hundreds of calls per seconds and it only needs to have the stock prices from the last minutes. You could use a single instance (StockPriceManager) shared among the trading business classes, and every function that needs the prices would get it from the Cache. If the price is outdated, the cache would refresh it. In this situation, the drawbacks of tight coupling are worth the gain in performance. But when you need to understand a bug in production because of this global state you cry (I've been there and it wasn't funny).

I told you to use the single instance approach instead of the singleton but sometimes it is worth using the real singleton when you need this object in all of the classes. For example when you need to log:

- Each class needs to log and this log class is often unique (because the logs are written in the same file). Since all classes use the log class, you know that every class has an implicit dependency to this log class. Moreover, it's not a business needs so it's "less important" to unit test the logs (shame on me).

Writing a singleton is easier than writing a single instance using Dependency

Injection. For a quick and dirty solution I'd use a singleton. For a long and durable solution I'd use a single instance. Since most applications are based on frameworks, the implementation of the single instance is easier than from scratch (assuming you know how to use the framework).

If you want to know more about singletons:

- there is a good article on dzone about the singleton pattern.
- There is also this very good answer on stackexchange about the cons of the singleton pattern.

Real examples

The single instance pattern uses a factory. If you use an instantiable factory, you might need to ensure that this factory is unique. And more broadly, when you use a factory you might want it to be unique in order to avoid that 2 factory instances mess with each other. You could use a "meta-factory" to build the unique factory but you'll end up with the same problem for the "meta-factory". So, the only way to do that is to create the factory with a singleton.

It's the case for java.awt.Toolkit in the old graphical library AWT. This class provides a getDefaultToolkit() method that gives the unique Toolkit instance and it's the only way to get one. Using this toolkit (which is a factory), you can create a windows, a button, a checkbox …

But you can also encounters singletons for other concerns. When you need to monitor a system in Java, you have to use the class java.lang.Runtime. I guess this class has to be unique because it represents the global state (environment variables) of the process. If I quote the java API:

> *"Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the* ***getRuntime*** *method."*

Prototype Pattern

I've used prototypes through Spring but I never had the need to use my own prototypes. This pattern is meant to build objects by copy instead of constructors. Here is the definition given by the GoF:

"Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype."

As often with the GoF, I don't understand their sentences (is this because English is not my native language?). If you're like me, here is another explanation: If you don't want or can't use the constructor of a class, the prototype pattern lets you create new instances of this class by duplicating an already existing instance.

Let's look at the formal definition using a UML diagram:

In this diagram

- the prototype is a an interface that defines a function clone()
- A real prototype has to implement this interface and implement the clone() function to return an copy of itself.

A developer will have to instantiate the ConcretePrototype once. Then, he will be able to create new instances of ConcretePrototype by:

- duplicating the first instance using the clone() function
- or creating a ConcretePrototype using (again) the constructor.

When to use prototypes?

According to the Gof, the prototype should be used:

- when a system should be independent of how its products are created, composed, and represented
- when the classes to instantiate are specified at run-time, for example, by

dynamic loading

- to avoid building a class hierarchy of factories that parallels the class hierarchy of products
- when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

The dynamic loading of an unknown class is a very rare case, even more if the dynamically loaded instance needs to be duplicated.

This book was written in 1994. Now, you can "avoid building a class hierarchy of factories" by using dependency injection (again, I'm going to present this wonderful pattern in a future article).

In my opinion the most common case is where creating a stateful instance is way more expensive than copying an existing instance and you need to create lots of this object. For example if the creation needs to:

- get data from a database connection,
- get data from the system (with system calls) or the filesystem,
- get data from another server (with sockets, web services or whatever),
- compute a large amount of data (for example if it needs to sort data),
- do anything that takes time.

The object must be stateful because if it has no state, a Singleton (or a single instance) will do the trick.

There is also another use case. If you have an instance that is mutable and you want to give it to another part of the code, for security reasons you might want to give a duplicate instead of the real instance because this instance can be modified by the client code and have an impact on other parts of the code that use it.

Java implementation

Let’s look at a simple example in Java:

- We have a CarComparator business class. This class contains a function that compares 2 cars.
- To instanciate a CarComparator, the constructor needs to load a default configuration from a database to configure the car comparison algorithm (for example to put more weight on the fuel consumption than the speed or the price).
- This class cannot be a singleton because the configuration can be modified by each user (therefore each user needs its own instance).
- This is why we create only once a instance using the costly constructor.
- Then when a client needs an instance of CarComparator he gets a duplicate of the first instance.

If you look at the next part, you’ll see that I could have made a simpler code using the right Java interface but I wanted you to understand a prototype.

In this example, at start-up, the prototype will be created using the default configuration in the database and each client will get a copy of this instance using the getCarComparator() method of the factory.

Real example

The Java API provides a prototype interfaces called Cloneable. This interface define a clone() function that a concrete prototype needs to implements. Many Java classes from the Java APIs implement this interface, for example the collections from the collection API. Using an ArrayList, I can clone it and get a new ArrayList that contains the same data as the original one:

The result of this code is:

> *content of the set [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]*
> *content of the duplicated set [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]*

Builder Pattern

The Builder pattern is very useful to factorize code. According to the GoF, this pattern:

“Separate the construction of a complex object from its representation so that the same construction process can create different representations.”

Its aim has changed through time and it’s most of the time used to avoid creating a lot of constructors that differs only by the number of arguments. It’s a way to avoid the **telescoping constructor anti-pattern**.

The problem it solves

Let’s look at the problem this pattern solves. Imagine a class Person with 5 attributes:

- age
- weight
- height
- id
- name

We want to be able to construct a person knowing:

- only this age,
- or only this age and weight,
- or only this age, weight and height,
- or only this age, weight, height and id
- or only this age, weight, height, id and name

In java, we could write something like that

In order to deal with this simple need, we've just created 5 constructors which a lot of code. I know java is a very verbose language (troll inside) but what if there was a cleaner way?

Moreover, using this telescopic approach, the code is hard to read. For example, if you read the following code, can you easily understand what the parameters are? Are they age, id or height?

Next problem: imagine you now want to be able to create a Person with every possible part of information you get, for example:

- an age
- a weight
- an age and a weight
- an age and an id,
- an age, a weight and a name,
- …

With the constructor approach, you'll end up with 120 constructors (5! = 120). And you'll have another problem, how can you deal with different constructors using the same types? For example how can you have both:

- a constructor for the age and the weigth (which are 2 int) and
- a constructor for the age and the id (which are also 2 int)?

You could use a static factory method but it would still require 120 static factory methods.

This is where the builder comes into play!

The idea of this pattern is to simulate **named optional arguments**. These types of arguments are natively available in some languages like python.

Since the UML version is very complicated (I think), we'll start with a simple java example and end with the UML formal definition.

A simple java example

In this example, I have a person but this time the id field is mandatory and the other fields are optional.

I create a builder so that a developer can use the optional fields if he wants to.

In this example, I suppose the classes Person and PersonBuilder are in the same package, which allows the builder to use the Person constructor and the classes outside the package will have to use the PersonBuilder to create a Person.

This PersonBuilder has 2 kinds of methods, one for **building a part** of a person and one for **creating** a person. All the properties of a person can only be modified by classes in the same package. I should have used getter and setter but I wanted to have a short example. You see that the part th uses the builder is easy to read, we know that we are creating

- a person named Robert whose age is 18 and weight 80,
- another person named Jennifer whose length is 170.

Another advantage of this technic is that you can still create **immutable objects**. In my example, if I don't add public setters in the Person class, a Person instance is immutable since no class outside the package can modify its attributes.

The formal definition

Now let's look at the UML:

This diagram is really abstract, a GoF's builder has:

- a builder interface that specify functions for creating parts of a Product object. In my diagram, there is just one method, buildPart().

- a ConcreteBuilder that constructs and assembles parts of the product by implementing the Builder interface.
- a Director : it constructs a product using the Builder interface.

According to the GoF, this pattern is useful when:

> *• the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.*
> *• the construction process must allow different representations for the object that's constructed.*

The example given by the GoF was a TextConverter builder that has 3 implementations to build: an ASCIIText or a TeXText or a TextWidget. The 3 builder implemenations (ASCIIConverter, TeXConverter and TextWidgetConverter) have the same functions except the createObject() function that differs (this is why this function is not in the interface of this pattern). Using this pattern, the code that converts a text (the Director) uses the builder interface so it can easily switch from ASCII to TeX or TextWidget. Moreover, you can add a new converter without modify the rest of the code. In a way, this pattern is very close to the State pattern.

But this problem is a rare case.

Another use of this pattern was popularized by Joshua Bloch, a Java developper who led the construction of many Java APIs. He wrote in his book *"Effective Java"*:

> *"Consider a builder when faced with many constructor parameters"*

Most of the time the pattern is used for this use case. You don't need the builder interface nor multiple builder implementations nor a director for this problem. In my java example and most of the time you will find just a concrete builder.

The UML then becomes easier:

In this diagram the ConcreteBuilder has multiple functions that create each part of the product (but I just put one, buildPart(), because I'm lazy). These functions return the ConcreteBuilder so that you can chain the function calls, for example: builder.buildPart1().buildPart7().createObject(). The builder has a createObject() method to create the product when you don't need to add more parts.

To sum up, the builder pattern is a good choice when you have a class with many optional parameters and you don't want to end up with to many constructors. Though this pattern was not designed for this problem, it's most of the time used for that (at least in Java).

Real example

The most common example in the Java APIs is the StringBuilder. Using it, you can create a temporary string, append new strings to it and when you're finished you can create a real String object (that is immutable).

Conclusion

You should now have a better overview of the creational patterns. If you need to remember one thing it's to use single instances instead of singletons. Keep in mind the builder pattern (Joshua Bloch's version), it might be useful if you're dealing with optional parameters.