SOCIALIZE VOCABULARY ABOUT THE READING TEXT "REAL-WORLD EXAMPLES OF PATTERN-BASED DESIGN."

BUILDER PATTERN

Meaning:

A design pattern used when an object requires the construction of multiple parameters, allowing the separation of mandatory and optional fields. The construction logic is moved to a separate builder class, enhancing flexibility and maintainability.

FACTORY METHOD

Meaning:

A design pattern that provides an interface for creating instances of a superclass, with multiple subclasses implementing the interface. It allows a method to be delegated to the subclasses, creating objects based on certain conditions.

ABSTRACT FACTORY

Meaning:

A design pattern that defines a super-factory for creating families of related or dependent objects without specifying their concrete classes. It encapsulates the creation of multiple object types, providing a unified interface.

PROTOTYPE PATTERN

Meaning:

A design pattern where objects are cloned rather than created with a constructor, improving performance and minimizing complexity in object creation. Useful when object creation is time-consuming.

FLYWEIGHT PATTERN

Meaning:

A design pattern that focuses on sharing objects to improve space efficiency. Objects are shared for efficiency and consistency, particularly when a large number of objects need to be created.

PROXY PATTERN

Meaning:

A design pattern that acts as a surrogate or placeholder for another object, controlling access to it. It can be used for various purposes, such as representing a large object that should be loaded on demand or adding a layer of authorization.

DECORATOR PATTERN

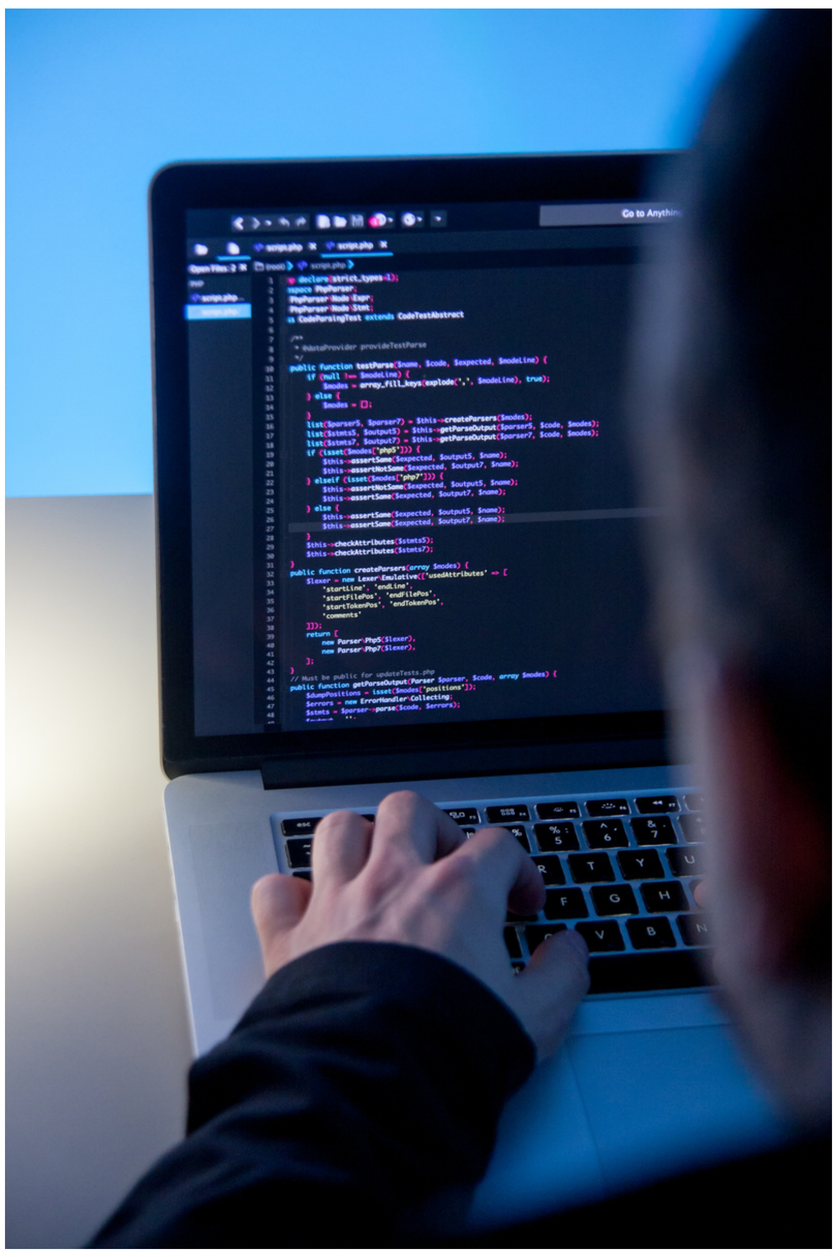
Meaning:

A design pattern that allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class. It requires the decorator object's interface to be identical to the decorated object.

READING: "REAL-WORLD EXAMPLES OF PATTERN-BASED DESIGN."

From my experience as a Senior Software Engineer in a big software company, many times developers concentrate mostly on design of specific tasks and not enough effort is done on thinking about the changes and additional features that may be added in the future. And there are always changes and extensions of existing features in the software industry! With the poor design, such code becomes very hard for maintenance and each change may be time consuming and very difficult for implementation. Good OO designs should be reusable, extensible and maintainable and Design Patterns could be very helpful in that.

On other hand, sometimes developers overuse Design Patterns which only adds complexity to the code and it is exactly the opposite of what is needed!



In this article, I`ll try to describe in high level the most useful Design Patterns by providing the real day to day examples of when they should be used. I believe this will help developers to use Design Patterns in the right situations.

1. BUILDER

Did it happen to you, that you had to use a constructor with 5 or even more parameters? For example, to create a Pizza object, you need to call the following constructor:

```
Pizza(Size size, Boolean onion, Boolean cheese, Boolean olives,
Boolean tomato, Boolean corn, Boolean mushroom, Sauce sauceType);
```

In addition to the fact that it's annoying, the parameters may be easily mixed up by the developers. Usually, most of the parameters are not even mandatory, but in this constructor, the user is forced to set value for each of the parameters. What will happen when new ingredients are added? Should this constructor be extended with even more parameters?

Exactly for those cases, when a lot of parameters needed to build the object — Builder design pattern is used! The main idea is to separate required fields from an optional and to move the construction logic out of the object class to a separate static inner class referred to as a builder class. That Builder class has a constructor only for mandatory parameters and setter methods for all the optional parameters. In addition, there is a build() method which glues everything together and returns an immutable complete object. All the builder setter methods return the builder itself, so the invocations can be chained.

So, our Pizza object creation, after applying a builder pattern will look like this:

```
Pizza pizza = new
Pizza.Builder(Size.medium).onion(true).olives(true).build();
```

Note, that Pizza class should not have any public constructor at all and the objects will be created only using the Builder class.

Consider using this when Object contains a lot of attributes

USAGE EXAMPLES

Builder patterns may be very useful while writing Unit Tests. In order to construct the object under the test, you need to pass a lot of parameters to the constructor and some of these parameters are completely irrelevant for the specific test. Builder class creation with separate methods for each parameter that should be tested, which returns by the end complete object under the test will help to write many UTs effectively, without duplicating the code.

Building an XML document with HTML elements (<html>, <h1>,<h2>, <body>,<p> and etc)

Building a smartphone object with attributes like RAM, size, resolution, OS, waterproof and so on.

2. FACTORY METHOD

Factory pattern is one of the most used design patterns and it's easy to provide examples of cases in which it should be used. Consider building a Logger Framework where the log messages may be written into the log file (represented by FileLogger class) or displayed in the console (represented by ConsoleLogger class). Depending on some logic (for example the variable “logger.logToFile=true” stored in some properties file), an appropriate Logger implementer needs to be used to log messages . The Logger Framework may be used by many different clients, therefore it would be a great idea to keep all the logic of creation and instantiation of the objects away from the clients. In this way, the client objects will not have to repeat the same logic again and again and it will be totally isolated from the future changes (like extension of Logger Framework by adding XmlLogger).

The clients will be using the following lines in order to write to the log, even without knowing if the log will be written to the file or to the console:

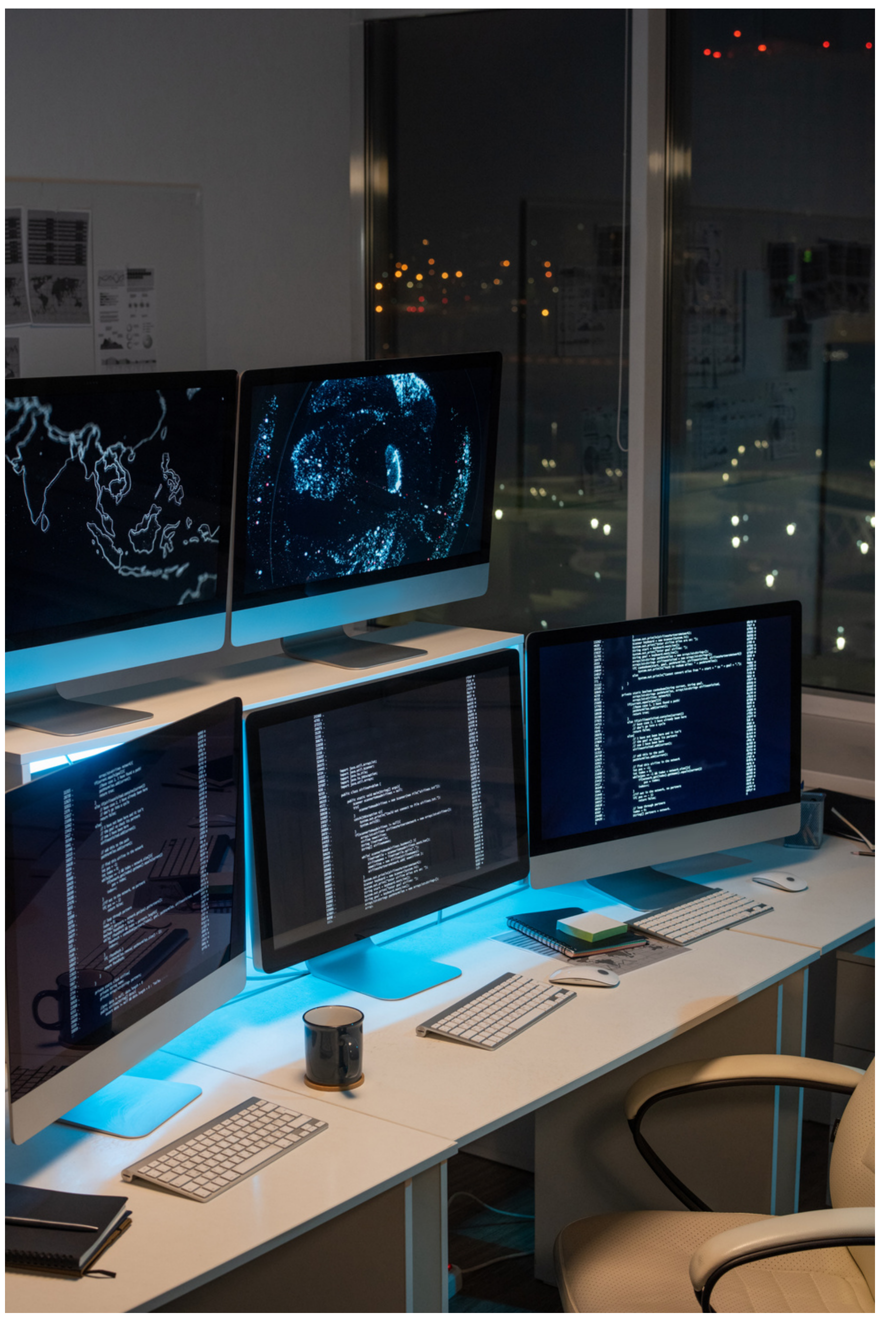
```
Logger logger = LoggerFactory.getLogger();
logger.log("write some message to the log");
```

All the logic will be encapsulated within the LoggerFactory class.

Consider using this when: A superclass has multiple sub-classes and based on input, needs to return one of the sub-class.

USAGE EXAMPLES

Java JDK is widely using the Factory pattern, for example the valueOf() method in wrapper classes like String, Boolean and etc.



different databases maybe supported: Oracle, SQLServer, H2

Each time when we have family of different kind of objects that created according to some logic:

different kind of employees: developer, tester, manager

3. ABSTRACT FACTORY

This pattern captures how to create families of related product objects, without instantiating classes directly. It`s a super-factory which creates other factories.

Consider using this when: there is a family of factories and you need a super factory for related factories

USAGE EXAMPLES

Application may have the Abstract DeviceProviderFactory which will be able to detect if the device is local or remote (from Amazon Device Farm), and return the corresponding factory accordingly: LocalDeviceProviderFactory or RemoteDeviceProviderFactory . Each such factory knows how to create device provider per OS type of device: AndroidDeviceProvider, iOSDeviceProvider, WindowsPhoneDeviceProvider

4. PROTOTYPE

When using prototype patterns, the objects are cloned instead of creating the new ones with a constructor, which improves the performance. In addition, the prototype pattern helps to minimize complexity in object creation.

Consider using this when: Creation of the object is very time consuming

USAGE EXAMPLES:

Cover letters: no need to create the Cover letter for each organization from scratch. Instead, one cover letter will be created in the most appealing format and for others only a copy will be created with a personalized organization name.

Chess game : may be used for chess board creation, which may be time consuming. Using the Prototype pattern, the board may be cloned, from the already existing board object.

5. FLYWEIGHT

The Flyweight pattern defines a structure for sharing objects. Objects are shared for at least two reasons: efficiency and consistency. Flyweight focuses on sharing for space efficiency. But objects can be shared only if they don't define context-dependent state. Flyweight objects have no such state. Any additional information they need to perform their task is passed to them when needed. With no context-dependent state, Flyweight objects may be shared freely.

Consider using this when

The number of Objects to be created by application should be huge.

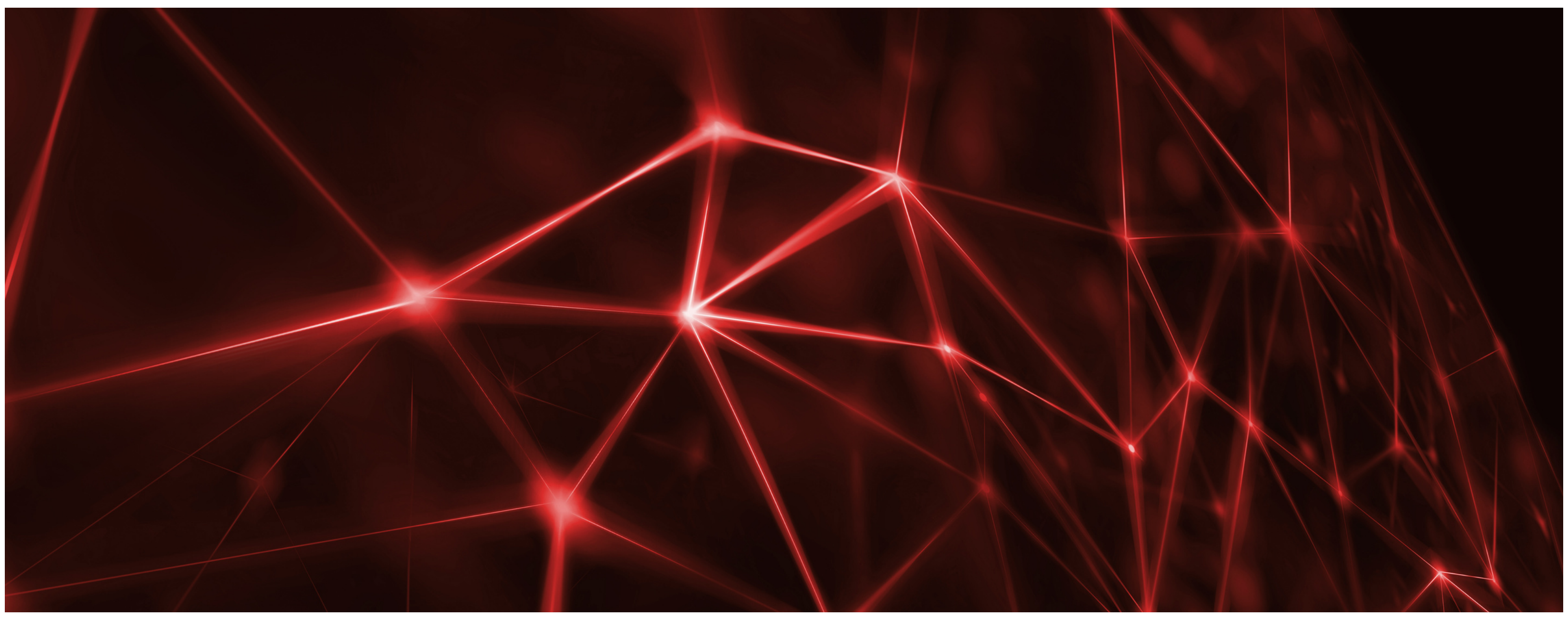
The object creation is heavy on memory and it can be time consuming too.

The object should be immutable

USAGE EXAMPLES:

May be used to represent the keyboard characters: one object for 'a', one for 'b' and so on.

When drawing a lot of shapes with different colors: one object for the red circle, one object for the blue circle and so on. In case the red circle was already created once, there is no need to create a new such object, since the same object may be reused.



6. PROXY

A proxy can be used in many ways. It can act as a local representative for an object in a remote address space. It can represent a large object that should be loaded on demand and avoids duplication of the same object. Without the concept of proxies, an application could be slow, and appear non-responsive. The proxy might protect access to a sensitive object.

USAGE EXAMPLES:

Image viewer program that lists and displays high resolution photos. The program has to show a list of all photos however it does not need to display the actual photo until the user selects an image item from a list.

The same for document editors that can embed graphical objects in a document. It isn't necessary to load all pictures when the document is opened, because not all of these objects will be visible at the same time.

The protective proxy acts as an authorization layer to verify if the actual user has access to appropriate content. An example can be thought about the proxy server which provides restrictive internet access in the office. Only the websites and contents which are valid will be allowed and the remaining ones will be blocked.

Maybe used also for adding a thread-safe feature to an existing class without changing the existing class's code.

7. DECORATOR

Decorator pattern allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class. Decorator require the interface of the Decorator object to be identical to the decorated object.

USAGE EXAMPLES:

Very useful when I need to measure the time it takes to handle some web service request. The class TimerDecorator with timer capabilities should be created, which will start the timer when request received and will stop the timer when response is sent back.

When needed to add to some shape component a border or a shadow functionality.

When need to add zooming, or scrolling functionalities to a page.

8. TEMPLATE METHOD

This pattern may be useful when we need to define an algorithm step by step. The implementation of some of the steps may be different, but the general flow and the order of the steps in the flow must remain unchanged. For example, if we have different types of phone devices (iOS and Android) which may be connected to the PC with USB, and we have an application that when device is connected it should perform the following steps: get device info, install an Agent on device and at the end report that device connected. We know that this flow is constant and want to be sure that it will stay the same also for the new device types in case they will be supported in the future (for exp. Windows Phone). Suppose we have an Abstract class — DeviceConnector and two subclasses IosDeviceConnector and AndroidDeviceConnector. We have the methods: getDeviceInfo(), installAgent() and reportDeviceConnected(). The first two methods will be implemented in the two different ways for Android and iOS, because we are using device specific libraries in order to do that. So, in Abstract Device class those two methods will be abstract and the sub-classes will be forced to add their specific implementation. The last method should have the same implementation for all device types, so it may be implemented in the Abstract Device class itself. But the main question here is, how will we be sure that all device types (also added by other developers in the future) will always implement those 3 methods in this specific order? The answer is simple; we will add a method to our Abstract class that will represent the steps of our flow (will include our three methods and in correct order). It's also important to define this method as final so that it could not be overridden and changed by its subclasses. In our example, the Abstract class will look like this:

```
public abstract class DeviceConnector {
     public final void connectDevice() {
           getDeviceInfo();
           installAgent();
           reportDeviceConnected();
    }
    protected abstract void getDeviceInfo();
    protected abstract void installAgent();
    public void reportDeviceConnected(){
          //Add implementation here
    }
}
```

USAGE EXAMPLES:

When implementing some general Parser, which loads the data from different sources (like CSV file, SQL Server database, some 3rd party tool), parses the data (data from different sources will be parsed in different ways and then it may be saved to some location. We will have template methods for load(), parse() and save() methods.

When implementing a credit card validator. For different kinds of credit cards (Visa, MasterCard and etc) the validation algorithm is the same: need to check expiration date, length of the credit card number, account status etc. But the actual implementation for each credit card type may be different.

9. OBSERVER (ALSO MAY BE CALLED PUBLISH/SUBSCRIBE PATTERN)

The Observer pattern defines and maintains a dependency between objects. The classic example of Observer is Model/View/Controller, where all views are Observers of the model, which is called Observable. All views are notified whenever the model's state changes. The main idea of the Observer pattern is that the Observable class will hold a list of Observers and whenever it wants to broadcast something, it just calls the method on all the observers, one after the other.

USAGE EXAMPLES:

Students and Board students should be notified when new messages appear on the board. Students are Observers and the board is Observable.

The chess game: the players are observers and they get notified when there is any change on the board.

Pub-Sub messaging. When messages are published on some channel — the clients that are listening on this channel will be notified that there is a new message.

10. STRATEGY

Strategy pattern helps to define a family of algorithms, to encapsulate each one of them and make them interchangeable and independent from the clients that use them. With this approach, our system becomes much more flexible and even the algorithm may be changed at runtime. The idea is to use an encapsulated family of algorithms as composition within the client’s class instead of inheritance.

USAGE EXAMPLES:

Application which should be able to choose a sorting algorithm at runtime (Bubble sort, QuickSort and so on).

In a game where we can have different characters and each character can have multiple weapons to attack but at a time can use only one weapon. The method attack() will have different implementations depending on which weapon is being used.

When implementing the shopping site: the user adds items to the basket and by the end of checkout, the user can choose the payment strategy at runtime: PayPal, Credit Card and so on.

Useful when a client may need to apply a different compression algorithm.

Adapted from: https://medium.com/@analempert/10-design-patterns-with-day-to-day-examples-e4f256d8439