Strategy Pattern

* Strategy pattern claims if you have class that includes various strategy implemented. You should seperate the strategies to different classes. Then inject an interface to main class to implement various strategies. Otherwise the main class will grow and hard to maintain. As an example if you have map application. You have multiple route strategies such as walking strategy, biking strategy. İn the future if you decide add another feature like public transport strategy you are going to have to write in the main class. This is a risky situtation.

A diagram of a strategy

Description automatically generated

Builder Pattern

* Lets assume that we want to create a House object. There would be lots of types of house right?

For example a house with swimming pool, a house wtih garage etc.

I bet first design comes to mind is to create an House object that has essential parts than have subclasses of this house object. But that could lead to having too many subclasses that is going to be hard to manage it.

Another way would be have an House object has huge constructor that is going to take too many parameter. Such as the properties of pool and garage. Lets say ill have a house with pool but not garage. Then parameters related to garage will be useless. That is not gonna look good. Both designs leads to clients take a part of process. Client should put effort to carry out the process.

Here builder comes handy. Since creating a house is a process that has a lot of steps we create a class called House Builder which includes all steps. To make easier user interaction we create another class called Director. Director will be injected with interface of Builder. Director will contact with builder to prepare product. By implementing Builder design client is not gonna know the process of building a house. Client doesnt need to be responsible of building process.

<https://refactoring.guru/design-patterns/builder>

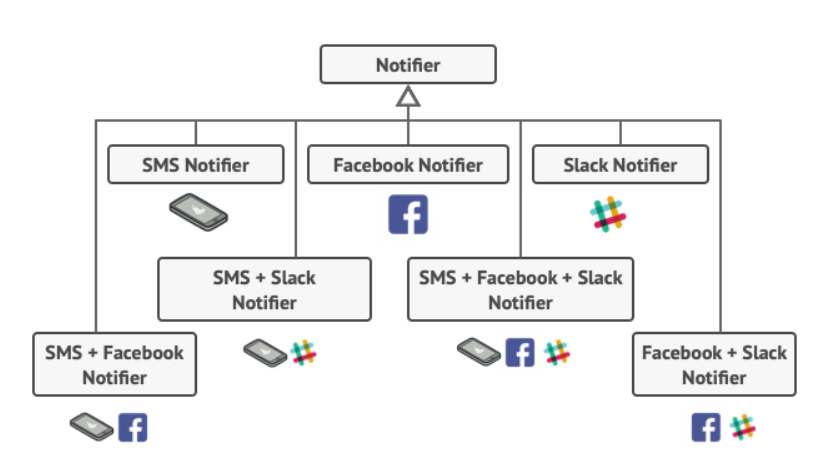
A diagram of a company

Description automatically generated

Decorator

* Imagine you created an notifier class that sends email to subscribers. Now it is asked to extend the notifier to send whatsapp mesagges also. First solution which comes to mind is to create another class that implements notifier. Then they demanded to send messages via facebook. Another class is created naturally. Then they asked to have both whatsapp and facebook. Then another class called whatsapp and facebook notifier. That can go on more and it will be caotic.

Solution is composition over inheritance. Composition will let us to meet needs with more sustainable and easy to implement code. Firstly BaseDecorator class will be created this class will implement same interface as notifier. Then concrete classes can be created as they are implementing the base decorator. Decorator will take another decorator as constructor argument(composition) then trigger the same send method of that decorator. This structure looks like a stack.



A diagram of a component

Description automatically generated

A computer screen shot of a computer

Description automatically generated

1. **Factory Method Pattern (FMP)**

FMP is one of the creational design patterns but what creational design is. Basically, creational design means dealing with object creation mechanisms, trying to create objects in a manner suitable to the situation.

FMP provides an interface for creating instances of a class but allows subclasses to alter the type of objects that will be created. We can talk about three components of FMP, which are client, product, and creator.

Client component is the application code that depends on an interface to complete its task.

Product component is the interface defined.

Creator component decides which concrete implementation to use.

Please have a look at code examples. That’s way it’s easier to understand it

[For more details.](https://realpython.com/factory-method-python/)

1. **Observer Pattern**

The observer design pattern enables a subscriber to register with and receive notifications from a provider. The pattern defines a provider (also known as a subject or an observable) and zero, one, or more observers.

Observers register with the provider, and whenever a predefined condition, event, or state change occurs, the provider automatically notifies all observers by invoking a delegate.

The observer design pattern is suitable for distributed push-based notifications, because it supports a clean separation between two different components or application layers, such as a data source (business logic) layer and a user interface (display) layer.

1. **Singleton Pattern**

Singleton Pattern is another one of the creational design patterns. Singleton ensures a class has ONLY one instance and provides a global point of access to this instance. It is commonly used when exactly one object is needed to coordinate actions across the system, such as a single database connection or a logging service.

Private Constructor: The class has a private constructor to prevent direct instantiation from external classes.

Private Static Instance: The class contains a private static instance of itself.

Static Method (Getter): There is a public static method, often named getInstance(), that provides the global point of access to the singleton instance. If an instance does not exist, this method creates one; otherwise, it returns the existing instance.