Structural Patterns

Wednesday, November 23, 2022 9:14 AM

The Structural Design Patterns mainly concerns with the **structure of the classes** and **relationships between entities**. It simplifies the structure by identifying the relationships. These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

Adapter

It gets the interface that you want from the other interface that you were given by the system. Make the **existing classes work** with **others without modifying** the source **code**.
The adapter pattern convert the interface of a class into another interface clients expect. Adapter **lets classes work together that couldn't otherwise because of incompatible interfaces**.

When to implement?

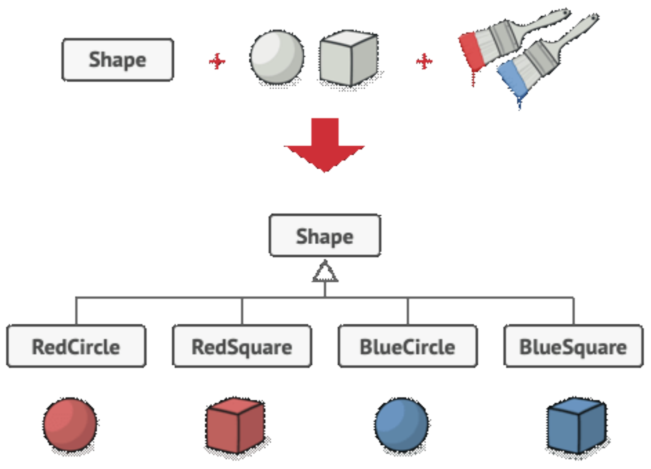
In different countries, electrical devices have different power requirements such as the voltage or the socket/plug type. So, if you are travelling to another country, an **adapter is needed for your devices if the destination has different type of power requirements**.
The same is applied in programming. There are many existing applications and **some of them need to adapt to others**.

Bridge

It is essential to "**bridge**" the **Abstraction** and the **Implementation** of a **class**. The **Implementation** is an **interface** that will describe a set of **specific behavior** that can be used by any Object in our codebase. It can have multiple concrete implementations that adhere to the contract defined in the interface. The **Abstraction** is the **object that will provide an API** that will make **use** of an **underlying Implementation**. It acts as a **layer over the top of the Implementation** that can be further **refined** through inheritance if required.
Bridge lets you **split a large class** or a set of closely related classes into **two separate hierarchies**- abstraction and implementation- which can be developed independently of each other.

When to implement?

Say you have a geometric Shape class with a pair of subclasses: Circle and Square. You want to extend this class hierarchy to incorporate colors, so you plan to create Red and Blue shape subclasses. However, since you already have two subclasses, you'll need to create four class combinations such as BlueCircle and RedSquare.



Adding new shape types and color to the hierarchy will grow it exponentially. For example, to add a triangle shape you'd need to introduce two subclasses, one for each color. And after that, adding a new color would require creating three subclasses, one for each shape type.

Composite

It has the peculiar goal of **allowing us to treat individual components** and **aggregate objects** in the same manner. The intent of a composite is to **"compose" objects into tree structures** to represent part-whole **hierarchies**.
Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures an then work with these structures as if they were individual objects.

When to implement?

When the **core model** of your application has a **tree structure**, it's time to consider the Composite Design Pattern.

There are many examples of a tree structure in applications. For instance, if you have a e-commerce website which utilizes an ordering system that uses Product and Box classes. One order can contain one or more boxes. Also, a box can contain one or more smaller boxes or products. In the smaller boxes, they can also contain other boxes or products as well.

Decorator

The Decorator Design Pattern helps developers to **add new functionality** to a class **without modifying it's existing content** or structure.
It lets you **attach new behaviors** to objects by placing these objects inside **special wrapper objects** that contain the behaviors.

When to implement?

When a system or a class need to be **enhances** by adding **new functionality**, it's time to consider this design pattern. Especially, the existing **classes are not editable** due to Open-Closed Design Principle or other business reasons.

Facade

The Facade Design Pattern provides a **unified interface to a set of interfaces** in a **subsystem**. Facade

defines a high level interface that makes the subsystem easier to use.
Facade is a structural design pattern that **provides a simplified interface** to a library, a framework, or any other complex set of classes.

When to implement?

This design pattern may be used unknowingly in your project even you are not aware of this. This is **one of the most useful design pattern** which helps your project architecture better if you understand it thoroughly.

You may want to implement the Facade Design Pattern when:

- You want to provide a **simple interface to a complex subsystem**. Subsystems often get more complex as they evolve.
- These are **many dependencies between clients and the implementation classes** of an abstraction.
- You want to **layer the subsystems**. Use a facade to define an entry point to each subsystem level.

Flyweight

The Flyweight Design Pattern helps to reduce memory usage and is used to create a large number of similar objects.
It lets you fit more objects into the available amount of RAM by sharing common parts of a state between multiple objects instead of keeping all of the data in each object.

When to implement?

It is somehow complicated and is not popular to use.

Proxy

The Proxy acts like a **getaway or a substitute** which **controls the access to the other object** and allows to perform actions either before or after the requests get through that object.
The proxy pattern is a software design pattern. A proxy, in its most general form, is a **class functioning as an interface** to something else.

When to implement?

The Proxy is used when we need a getaway to the main object's complexity from the client. By doing so, when the object is needed, we just need to initialize the proxy to access the object. Example: a credit card is a proxy for a bank account.