1. **Decorator pattern**
2. A real-world problem: Battle Royale video game PUBG’s problem

PlayerUnknown's Battlegrounds (also known as PUBG: Battlegrounds) is an online multiplayer battle royale game developed and published by PUBG Corporation, a subsidiary of Bluehole.

In the game, we have multiple weapons, but we specifically care about Guns. A powerful gun is according to which accessories it takes in its specific situations. Each of the guns and their accessories have the price. The players should do research about their gun to be confident in the battlefield. For instance, in a small town with many houses, the assault rifle works best with 4x zoom and, while we are on it, we would also need compensator and suppressor. It will reduce recoil and reduce firing sound as well as echo. In case of the PUBG developers, how can we implement this feature?

We will allow players to buy their gun and their accessories. Player can buy the gun alone, with some or all the accessories and they would be charged accordingly. One of the ideas the developers produced was the naive approach below:

1. Naïve solution: Use inheritance with subclasses for the guns’ accessories.

Diagram

Description automatically generated

1. Problems with the naïve solution:

+ For each combination of accessories, we must build a separated class, making the code repetitive, lengthy, and therefore, excessive. Taking our game into account, we would need 3! = 6 different classes just to equip 3 accessories on SCAR-L!

+ It is also extremely rigid to do individual changes. For example, if the price of 4x zoom decided to rise (because it is an especially useful piece), there would be a tedious process of changing the cost of 6 mentioned classes!

1. Introduction to decorator pattern:

Then some questions might appear in your head:

"What if they want to add a Flash Hider? Will we need to make 6 more classes?"

"And how about the other guns? We will need to add like 20-30 more classes for them? That’s too much!"

So how can we optimize this? We will need a method for this situation. Seems like the best way is using the Decorator Pattern. So, what is Decorator Pattern?

"Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors." - refactoring.guru

From the definition, we have a critical changing in our thought that: "What if I just start with a simple gun, (pick a specific gun based on its name) and add the accessories as the players buy it?"

Let's see how Decorator Pattern is applied here:

- Suppose someone wants to buy SCAR-L (an assault rifle gun) with all three accessories mentioned above.

Step 1. Take an object of SCAR-L

Step 2. Decorate (or add) the SCAR-L with 4x zoom object

Step 3. Decorate the SCAR-L with suppressor object

Step 4. Decorate the SCAR-L with compressor object

Step 5. Call the cost method and let each object delegate to add on the cost using cost method of accessories

Wow, because of that we can design the class diagram using the Decorator Pattern.

1. General diagram:
2. Class diagram for the mentioned problem:

Diagram

Description automatically generated

1. Implementation of the pattern:
2. Pros and cons of our pattern:

Pros:

+ Decorators provide a flexible alternative to subclassing for extending functionality

+ Decorators allow behavior modification at runtime rather than making a new subclass

+ Decorators are a nice solution to combine several behaviors by wrapping an object into multiple decorators.

+ You can add or remove responsibilities from an object at runtime

+ The decorator pattern supports the principle that classes should be open for extension but closed for modification

Cons:

\_ Decorators can result in many small objects in our design, and overuse can be complex

\_ Decorators can cause issues if the client relies heavily on the components concrete type

\_ Decorators can complicate the process of instantiating the component because you not only have to instantiate the component but wrap it in a number of decorators

\_ It can be complicated to have decorators keep track of other decorators because to look back into multiple layers of the decorator chain starts to push the decorator pattern beyond its true intent

\_ It’s hard to implement a decorator in such a way that its behavior doesn’t depend on the order in the decorators stack.

1. Some other real-world problems:
   * A milk tea shop offers a variety of milk tea as well as toppings, so customers can order a certain type of milk tea according to their preferences. If we are using inheritance in OOP, we have to create a lot of classes corresponding to each type of milk tea with different toppings, which will be difficult. The decorator pattern can solve that problem.
   * Similar to the problem above, we can have a lot of problems in stores that sell other items like pizza, cake…
2. Quizzes!
3. **Strategy pattern**
4. A real-world problem: MS Paint toolbox

When we use drawing software like MS Paint, we usually want to switch between tools (pencil, fill, eraser…) Using a strategy pattern for each tool can make the design procedure become clearer.

1. Naïve solution: Use inheritance with subclasses for each of the tools.
2. Problems with the naïve solution:

+ The number of overridden functions will be the same as the number of tools. There would be certain features of the tools that are similar, like pen tip sizes, colors... but to avoid a number of exceptions that do not have those features, we still have to implement them independently, making the code repetitive, lengthy, and therefore, excessive.

+ Making a change to the common features will also be tedious when there are many tools, while having to also watch out for the exceptional tools.

1. Introduction to decorator pattern:
2. General diagram:
3. Class diagram for the mentioned problem:
4. Implementation of the pattern:
5. Pros and cons of our pattern:

Pros:

+ Prevents the conditional statements (switch, if, else…)

+ The algorithms are loosely coupled with the context entity. They can be changed/replaced without changing the context entity.

+ Very easily extendable.

+ Open/Closed Principle. You can introduce new strategies without having to change the context

Cons:

\_ Clients must be aware of the differences between strategies to be able to select a proper one.

\_ It increases the number of objects in the application.

\_ A lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions. Then you could use these functions exactly as you would have used the strategy objects, but without bloating your code with extra classes and interfaces.

1. Some other real-world problems:
   * Our program provides many different sorting algorithms: quick sort, merge sort, selection sort, heap sort, bubble sort.... Depending on the data type, the number of elements… the user can choose a suitable sorting algorithm.
2. Quizzes!