Design patterns

Singleton design pattern

The singleton pattern is one of the simplest design patterns.

Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly.

Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.

**Definition:**   
*The singleton pattern is a design pattern that restricts the instantiation of a class to one object.*  
Let’s see various design options for implementing such a class. If you have a good handle on static class variables and access modifiers this should not be a difficult task.

Singleton is a part of **Gang of Four design pattern** and it is categorized under **creational** design patterns. In this article, we are going to take a deeper look into the usage of the Singleton pattern.

It is one of the most simple design patterns in terms of the modelling but on the other hand, this is one of the most controversial patterns in terms of complexity of usage.

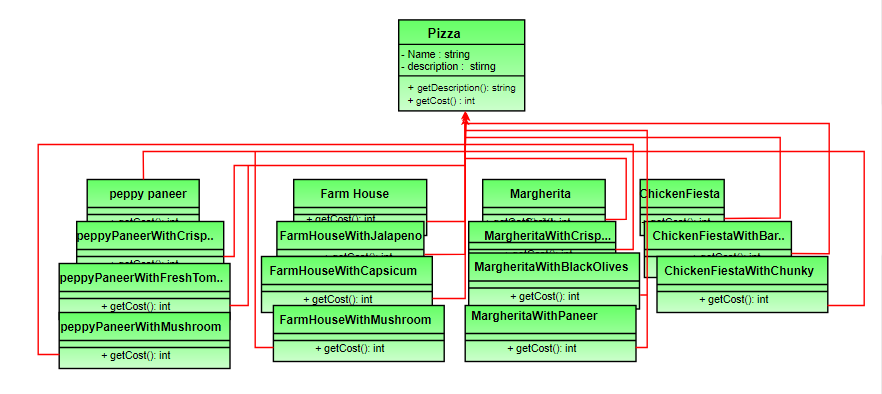
Singleton pattern is a design pattern which restricts a class to instantiate its multiple objects. It is nothing but a way of defining a class. Class is defined in such a way that only one instance of the class is created in the complete execution of a program or project. It is used where only a single instance of a class is required to control the action throughout the execution. A singleton class shouldn’t have multiple instances in any case and at any cost. Singleton classes are used for logging, driver objects, caching and thread pool, database connections.

Decorator pattern

To understand decorator pattern let us consider a scenario

inspired from the book “Head First Design Pattern”.  Suppose we are building an application for a pizza store and we need to model their pizza classes. Assume they offer four types of pizzas namely Peppy Paneer, Farmhouse, Margherita  and Chicken Fiesta. Initially we just use inheritance and abstract out the common functionality in a **Pizza**class.

Each pizza has different cost. We have overridden the getCost() in the subclasses to find the appropriate cost. Now suppose a new requirement, in addition to a pizza, customer can also ask for several toppings such as Fresh Tomato, Paneer, Jalapeno, Capsicum, Barbeque, etc. Let us think about for sometime that how do we accommodate changes in the above classes so that customer can choose pizza with toppings and we get the total cost of pizza and toppings the customer chooses. Let us look at various options. **Option 1** Create a new subclass for every topping with a pizza. The class diagram would look like:



This looks very complex. There are way too many classes and is a maintenance nightmare. Also if we want to add a new topping or pizza we have to add so many classes. This is obviously very bad design.

**Option 2:** Let’s add instance variables to pizza base class to represent whether or not each pizza has a topping. The class diagram would look like:

Adapter pattern

This pattern is easy to understand as the real world is full of adapters.   For example consider a USB to Ethernet adapter. We need this when we have an Ethernet interface on one end and USB on the other. Since they are incompatible with each other. we use an adapter that converts one to other. This example is pretty analogous to Object Oriented Adapters. In design, adapters are used when we have a class (Client) expecting some type of object and we have an object (Adaptee) offering the same features but exposing a different interface.

To use an adapter:

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates that request on the adaptee using the adaptee interface.
3. Client receive the results of the call and is unaware of adapter’s presence.

***Definition:***

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.