The Facade Design Pattern

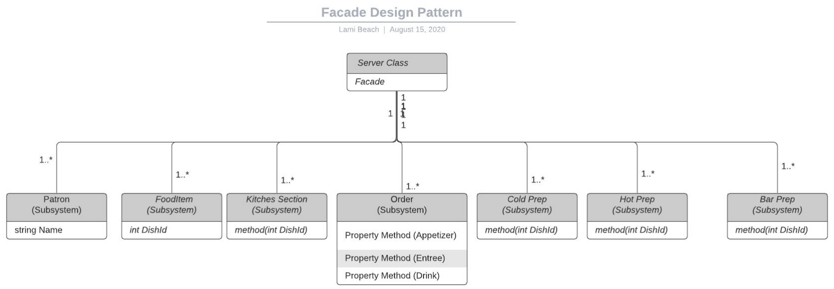
By: Lami Beach

What is it?

The idea is that if you don't want other code accessing the complex bits of a class or process, you hide those bits by covering them with a Facade.

The Facade pattern is a simple structure laid over a more complex structure.

UML Diagram



- The Subsystems are any classes or objects which implement functionality but can be "wrapped" or "covered" by the Facade to simplify an interface.

- The Facade is the layer of abstraction above the Subsystems, and knows which Subsystem to delegate appropriate work to.

Advantages

- Good: Hiding complexity which cannot be refactored away.

- The facade pattern is so general that it applies to almost every major application, especially those where refactoring or modifying pieces of applications are needed for various reasons.

- More Good: It provides a unified interface to a set of interfaces in a subsystem. However, it does not ab ovo prevent clients from using the subsystem interfaces directly. So you are not forced to make any unwanted compromises with it. It is a win-win situation.

Disadvantages

There is really no known **drawbacks** to **Facade**.

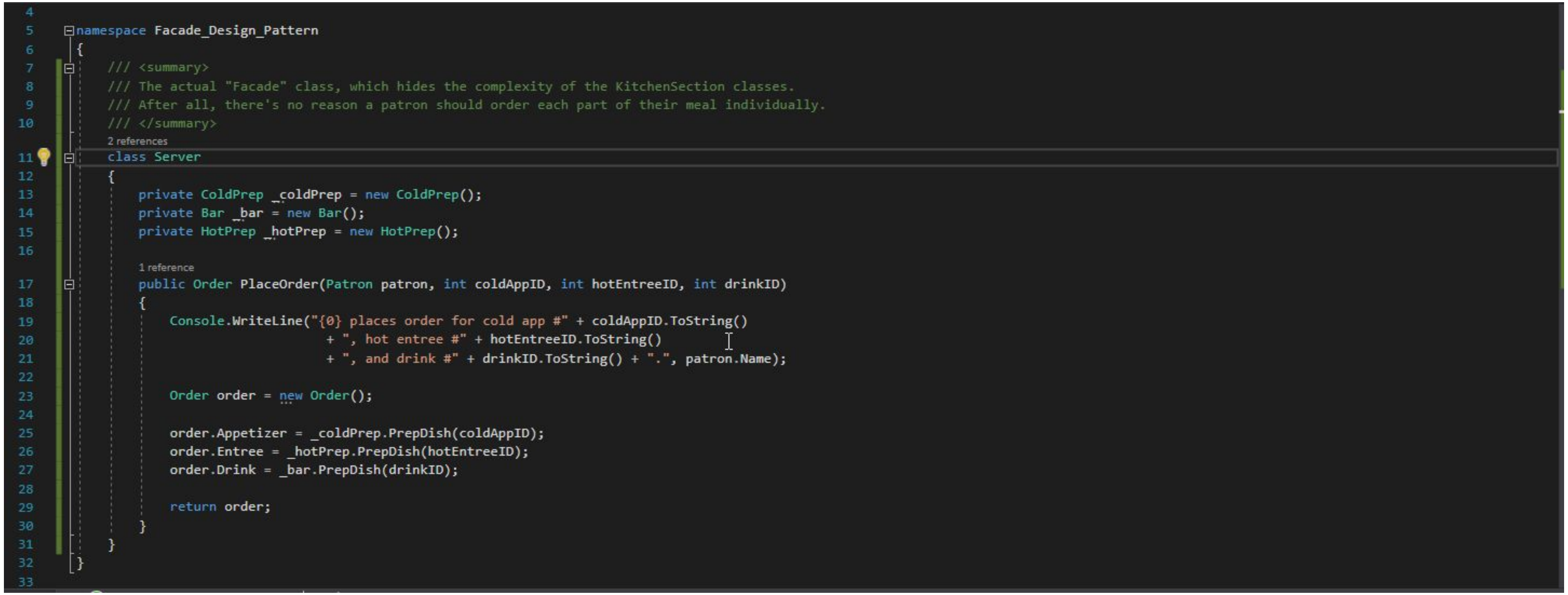
Real World?

- To demonstrate how we use the Facade pattern, let's think about a restaurant.

- In most kitchens, the work area is divided into sections. For this post, we'll assume that our kitchen is divided into three areas: Hot Prep, where hot dishes like meats and pasta are made; Cold Prep, where cold dishes like salads and desserts are made; and the Bar, where drinks are prepared.

- If you are a patron at this restaurant and you sit down at a booth, do you care what part of your meal is made at what section of the restaurant? Of course not. There is a natural layer of abstraction in place: the server.

Code Snippet



```
namespace Facade_Design_Pattern
{
    /// <summary>
    /// The actual "Facade" class, which hides the complexity of the KitchenSection classes.
    /// After all, there's no reason a patron should order each part of their meal individually.
    /// </summary>
    2 references
    class Server
    {
        private ColdPrep _coldPrep = new ColdPrep();
        private Bar _bar = new Bar();
        private HotPrep _hotPrep = new HotPrep();

        1 reference
        public Order PlaceOrder(Patron patron, int coldAppID, int hotEntreeID, int drinkID)
        {
            Console.WriteLine("{0} places order for cold app #" + coldAppID.ToString()
                              + ", hot entree #" + hotEntreeID.ToString()
                              + ", and drink #" + drinkID.ToString() + ".", patron.Name);

            Order order = new Order();

            order.Appetizer = _coldPrep.PrepDish(coldAppID);
            order.Entree = _hotPrep.PrepDish(hotEntreeID);
            order.Drink = _bar.PrepDish(drinkID);

            return order;
        }
    }
}
```

Overview

The Facade pattern is a simple (or at least simpler) overlay on top of a group of more complex subsystems. The Facade knows which Subsystem to direct different kinds of work toward. And it is really, really common, so it's one of the patterns we should know thoroughly.

Difference between Factory and Facade pattern:

The main difference between factory and facade design pattern is that factory is a creational design pattern that defines an interface or an abstract class to create an object, while the facade is a structural design pattern that provides a simplified interface to represent a set of interfaces in a subsystem to hide

OOP

Inheritance: The preparations inherit from Kitchen selection.

Abstraction: The patron is the only information that changes. However the concept of patron stays the same.

Encapsulation: Server encapsulates all other subsystems.

Polymorphism: Orders can change based on patron.

Questions?

Citations: Matthew Jones - The Facade Design Pattern(https://www.exceptionnotfound.net/facade-pattern-in-csharp/)