Design Patterns — A quick guide to Facade pattern.

medium.com/@andreaspoyias/design-patterns-a-quick-guide-to-facade-pattern-16e3d2f1bfb6

February 8, 2019

Facade pattern is often needed when there is a large number of interdependent classes or because parts of the code are unavailable. It is used as a camouflage to cover the complexities of a large system and therefore provides a simple interface to the client. In other words, it is a wrapper class used to hide the implementation details.

> **Facade pattern** is classified as a structural design pattern. This design patterns is all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality. [by Design Patterns explained simply]

The picture above is the perfect **example** of a Facade pattern. A customer in a restaurant orders food from the menu, which is probably described in half a line. The order goes to the kitchen and the food comes back after a while. **Simple!** The customer doesn't want to know who will cut the meat for how long will it be cooked and who is going to wash the dishes afterward. The customer just wants to eat a tasty meal that meets the expectations. Therefore, the menu serves as the facade to make it easier for the customer by avoiding the complexities coming from the kitchen or even the tasks that the waiter is assigned through this process.

Step 1 — Keywords

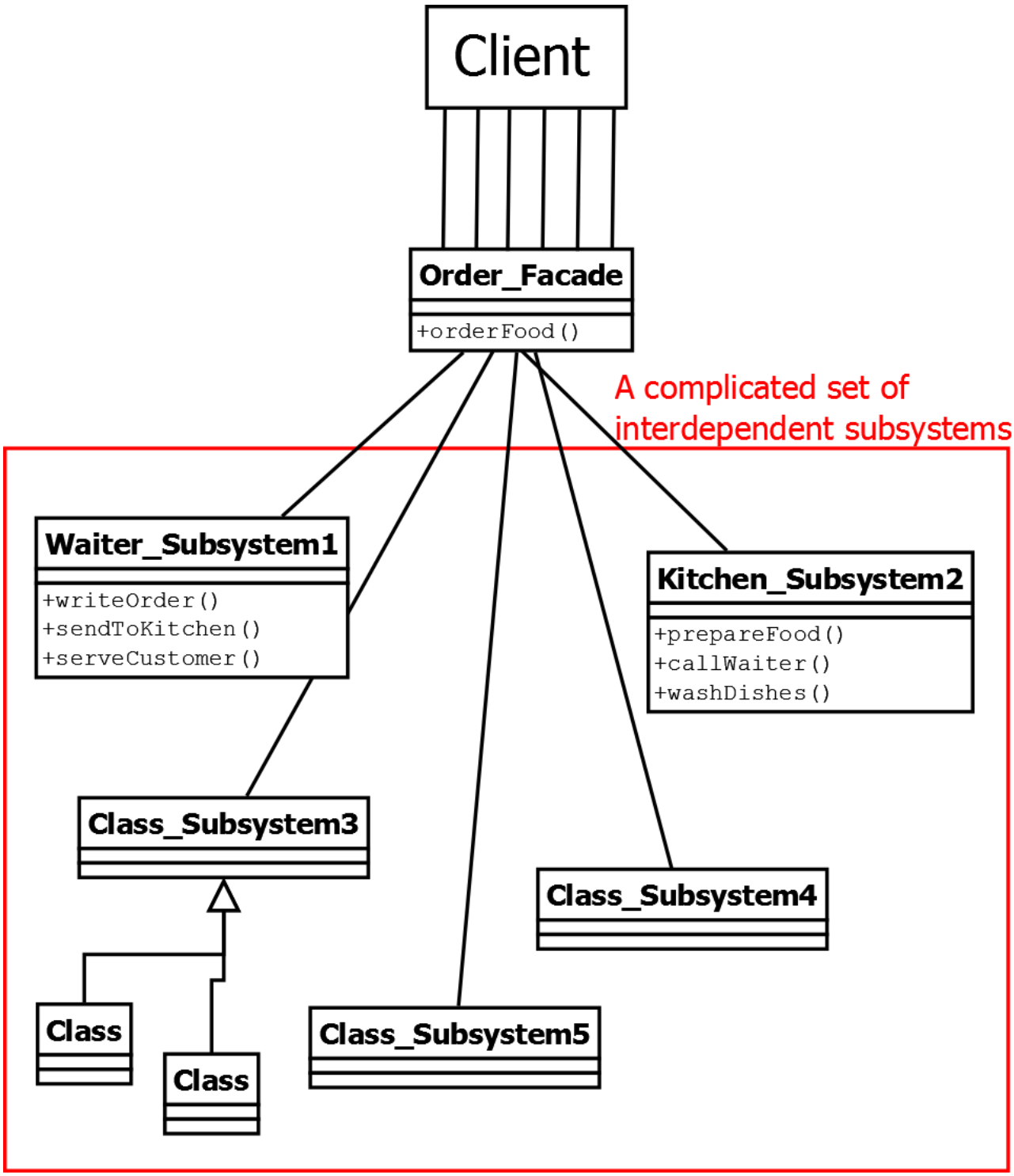
Defining **keywords** is the secret recipe in this series of quick-guides. This method helped me truly understand the design patterns, hardcode them in my mind and comprehend the differences among other design patterns.

1. **Simplification**: This is the goal of this design pattern. Simplify a complicated system.

2. **Restriction**: Simplification often comes with a “sacred cost”, restriction. By simplifying the code, we restrict clients from unauthorized access. Therefore, it prevents them from doing mistakes that would be difficult to be perceived in a complicated subsystem.

There is a tradeoff between simplification and restriction. Over-simplifying a system means that the developer is over-restricted, therefore less freedom than necessary which not always a good thing. Under-simplifying Facade pattern means that there is too much freedom which makes the Facade pattern irrelevant. Finding the fine balance is what makes a good, useful and effective Facade pattern.

Step 2 — Diagram



The diagram is also based on the given example. To simplify this diagram we can separate into three pieces.

1. **Client:** The client in this example is the customer of a restaurant that wants to order food.
2. **Facade:** Its job is to be able to provide to the client more simplified access towards numerous interdependent subsystems that are considered complicated. In this example, a client's food-order would require a series of carefully sequenced method calls of two different subsystems (Waiter and Kitchen).

3. **Subsystems:** The subsystems are hidden from the client. They might also be not accessible to the client. The client cannot fiddle with any of the subsystems where a simple code-change may prove to be fatal or even break other unknown parts of the system itself. In this scenario, the waiter and the kitchen have to do a series of tasks. A subsystem' s task is sometimes dependent on the anothers task. For example, the kitchen cannot prepare the food if the waiter doesn't bring the order to the kitchen. The waiter cannot serve the customer if the food is not cooked.

Step 3 — Code by example

I would suggest to copy the code class by class from my **git repository "Andreas Poyias"** or the snippets below (in the order provided) and paste it in any of the available online C++ editors like c++shell, jdoodle , onlineGDB and run it to **observe the output**. Then read the comments or description below. Take your time, reading it thoroughly (that means one minute, not less and not more).

Subsystems

In this example, the two subsystems are the `Waiter_Subsystem1` and the `Kitchen_Subsystem2` . At a first glance, each subsystem seems to be independent as they can do certain tasks individually. But is this true?

```
#include <iostream>
using namespace std;class Waiter_Subsystem1
{
public:
  void writeOrder()   { cout << " Waiter writes client's order\n";}
  void sendToKitchen(){ cout << " Send order to kitchen\n";}
  void serveCustomer(){ cout << " Yeeei customer is served!!!\n";}
};class Kitchen_Subsystem2
{
public:
    void prepareFood(){ cout << " Cook food\n";}
    void callWaiter() { cout << " Call Waiter\n";}
    void washDishes() { cout << " Wash the dishes\n";}
};
```

Facade: In this example, the Facade class is about food-orders in the restaurant. To execute a food-order successfully, we rely on a specific sequence of method calls and one call is dependent on the previous one and so on. The kitchen, cannot prepare the food if the waiter doesn't write the order and send it to the kitchen. The Facade class provides the `orderFood` task to the client to simplify it and avoid any misuse due to the complexity that exists.

```
class Order_Facade
{
private:
   Waiter_Subsystem1waiter;
   Kitchen_Subsystem2 kitchen;
public:
   void orderFood()
   {
      cout << "A series of interdependent calls on various subsystems:\n";
      waiter.writeOrder();
      waiter.sendToKitchen();
      kitchen.prepareFood();
      kitchen.callWaiter();
      waiter.serveCustomer();
      kitchen.washDishes();
   }
};
```

Main function

The main function operates as the **client**(the same as the previous guides)**.** It is so easy for the client to be able to create a Facade instance and call a function to do his job.

```
int main(int argc, char *argv[])
{
   // Simple for the client
   // no need to know the order or the
   // dependencies among various subsystems.
   Order_Facade facade;
   facade.orderFood();return 0;
}// Output
// A series of interdependent calls on various subsystems:
//  Waiter writes client's order
//  Send order to kitchen
//  Cook food
//  Call Waiter
//  Yeeei customer is served!!!
//  Wash the dishes
```

There are a few **benefits** for the use of Facade pattern and a few points to be noted when Facade is to be approached.

- Facade defines a higher-level interface that makes the subsystem easier to use by wrapping a complicated subsystem.
- This reduces the learning curve necessary to successfully leverage the subsystem.
- It also promotes decoupling the subsystem from its potentially many clients.
- On the other hand, if the Facade is the only access point for the subsystem, it will limit the features and flexibility that “power users” may need.

The next blog will be a quick guide to the **Observer**design pattern. It is a behavioral pattern which is a must have to your knowledge repository. Don’t forget to like/clap my blog-post and follow my account. This is to give me the satisfaction that I helped some fellow developers and push me to keep on writing. If there is a specific design pattern

that you would like to learn about then let me know so I can provide it for you in the future.

Other quick-guides on design patterns: