Theory: Facade

14 minutes 0 / 5 problems solved

Skip this topic Start practicing

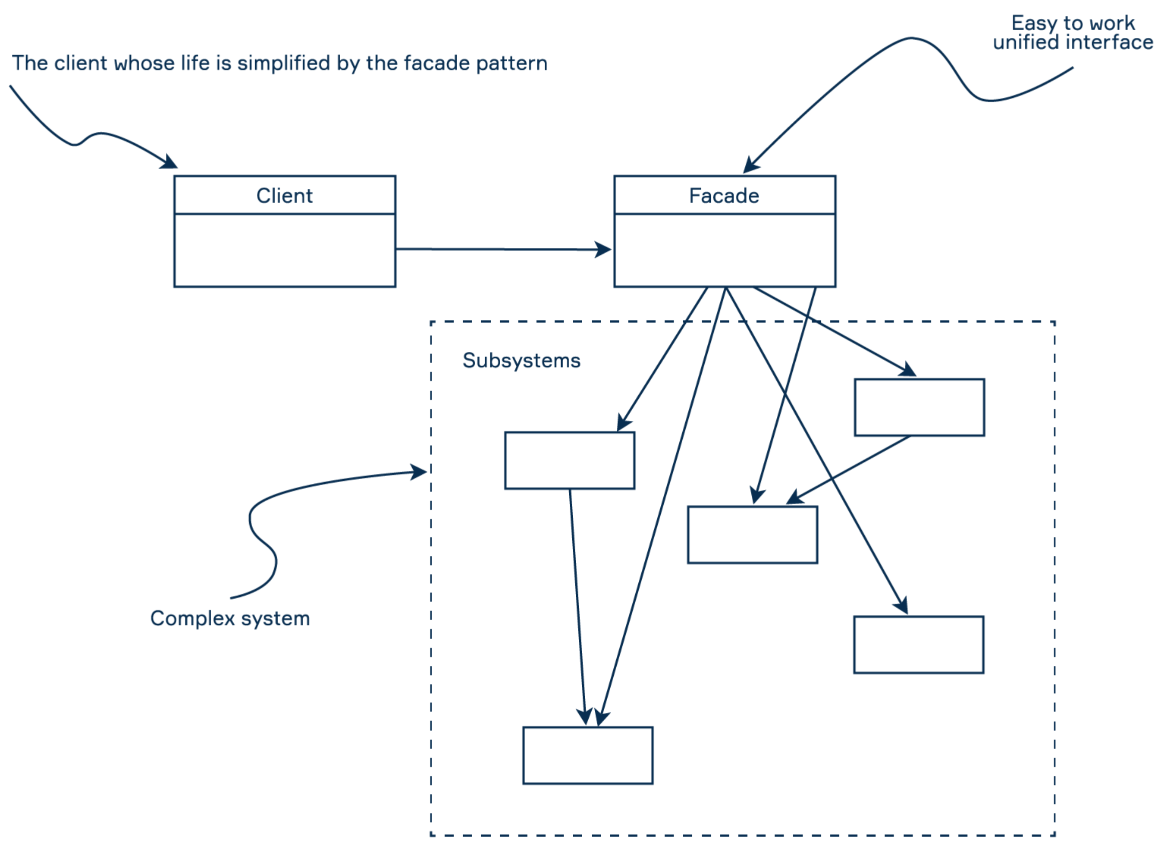
685 users solved this topic. Latest completion was about 17 hours ago.

Imagine you have a complex system with a large number of different subsystems. In order to start using your system, you activate all subsystems every day. Wouldn't it be awesome if you had some sort of a controller with the big red button that will do the algorithm for you? Well, this is what the **Facade** structural pattern is about.

§1. The Facade pattern

Facade provides a unified interface to a group of subsystem interfaces for you as a client and makes using the main system easier. Think of a car: when you insert a key and start it, there are subsystems that should be activated — the engine, the heated seats and your stereo system. When you arrive at your destination, you turn off the car and thus turn off the engine, the heated seats and the stereo.

The following diagram shows the relation between entities using Facade:


The client whose life is simplified by the facade pattern
Easy to work unified interface
Client
Facade
Subsystems
Complex system

§2. Example: CarFacade

This pattern is pretty easy to implement and most likely you did it in your projects.

First of all, we should define our subsystems: *Engine, StereoSystem,* and *HeatedSeats*.

Current topic:

Facade

Topic depends on:

✓ The concept of patterns

✓ Access modifiers Stage 6

Table of contents:

```java
class Engine {
    private String description;

    Engine() {
        this.description = "Engine";
    }

    void on() {
        System.out.println(description + " is on");
    }

    void off() {
        System.out.println(description + " is off");
    }
}
```

The engine subsystem can be simply turned on and off. We don't want to make it complex, so we generally follow the *KISS* rule.

```java
class HeatedSeats {
    private String description;
    private int heatLevel;

    HeatedSeats() {
        this.description = "HeatedSeats System";
    }

    void on() {
        heatLevel = 1;
        System.out.println(description + " is on");
    }

    void off() {
        heatLevel = 0;
        System.out.println(description + " is off");
    }

    void increaseHeatLevel() {
        if (heatLevel == 0) {
            on();
        } else if (heatLevel == 1 || heatLevel == 2) {
            heatLevel++;
        } else {
            off();
        }
    }
}
```

The HeatedSeats subsystem will set heat level to 1 when it's on and to 0 when it's off. Also, we provide *increaseHeatLevel()* method that implements the feature of increasing the heat level with the maximum value of 2 and the minimum value:

```java
public class StereoSystem {
    private String description;
    private String trackTitle;
    private int volume;

    StereoSystem() {
        this.description = "Stereo system";
        this.volume = 50;
    }



    void on() { System.out.println(description + " is turning on"); }



    void off() { System.out.println(description + " is turning off"); }



    void playTrack(String title) {

        this.trackTitle = title;

        System.out.println(title + " is playing");

    }



    public void pause() {

        System.out.println("Track: \"" + trackTitle + "\" were paused");

    }



    public String getTrackTitle() {

        return ("The current track is: \"" + trackTitle + "\"");

    }



    public void setVolume(int volume) {

        if (volume > 100) {

            this.volume = 100;

        } else {

            this.volume = volume;

        }

    }



    public int getVolume() {

        return volume;

    }

}
```

The StereoSystem is the last subsystem that we will add to the CarFacade. It can also be turned on and off. The StereoSystem can play the track you wish using ‘playTrack()’ method, pause it with ‘pause()’ method and set the volume.

Now let's create the CarFacade. The Facade should contain all the subsystems mentioned above and implement the desired algorithm. In our example, we decided to turn on *Engine, HeatedSeats* and then *StereoSystem* with a default track. Try other features of the subsystems and let only your imagination be the limit.

```
class CarFacade {
    private Engine engine;
    private HeatedSeats heatedSeats;
    private StereoSystem stereoSystem;

    CarFacade(Engine engine, HeatedSeats heatedSeats, StereoSystem stereoSystem) {
        this.engine = engine;
        this.heatedSeats = heatedSeats;
        this.stereoSystem = stereoSystem;
    }

    public void turnOnCar() {
        engine.on();
        heatedSeats.on();
        stereoSystem.on();
        stereoSystem.playTrack("Queen - I'm in love with my car");
    }

    public void turnOffCar() {
        engine.off();
        heatedSeats.off();
        stereoSystem.off();
    }

    public void increaseHeatedSeats() {
        heatedSeats.increaseHeatLevel();
    }

    public void playTrack(String title) {
        stereoSystem.playTrack(title);
    }
}
```

As you can see, we have implemented the Facade design pattern with three subsystems using composition. Composition, unlike inheritance, can make the architecture of your system more flexible. However, this statement deserves an individual topic.

Finally, after combining the main system and the subsystems, we can proceed to the test drive!

```
Engine engine = new Engine();
StereoSystem stereoSystem = new StereoSystem();
HeatedSeats heatedSeats = new HeatedSeats();

CarFacade carFacade = new CarFacade(engine, heatedSeats, stereoSystem);

carFacade.turnOnCar();
System.out.println();
   for (int i = 0; i < 5; i++) {

      Thread.sleep(1500);

      System.out.println("Driving to work... " + i + "km");

      switch (i) {

         case 1:

            Thread.sleep(500);

            carFacade.playTrack("Queen - Bohemian Rhapsody");

            break;

         case 2:

            Thread.sleep(500);

            carFacade.playTrack("Queen - I want to break free");

            break;

         case 3:

            Thread.sleep(500);

            carFacade.playTrack("Queen - Another one bites the dust");

            break;

         case 4:

            Thread.sleep(500);

            carFacade.playTrack("Queen - Scandal");

            break;

      }

   }

System.out.println("\nWe have arrived");

Thread.sleep(1000);

carFacade.turnOffCar();
```

And the final output is looking good:

```
Engine is on
HeatedSeats System is on
Stereo system is turning on
Queen - I'm in love with my car is playing

Driving to work... 0km
Driving to work... 1km
Queen - Bohemian Rhapsody is playing
Driving to work... 2km
Queen - I want to break free is playing
Driving to work... 3km
Queen - Who wants to live forever? is playing
Driving to work... 4km
Queen - Scandal is playing

We have arrived
Engine is off
HeatedSeats System is off
Stereo system is turning off
```

§3. Conclusion

1) Use the Facade when you want to provide a simple interface to a complex system. The Facade offers some kind of default system that suits most customers.

2) The Facade will allow separating the subsystem both from the customers and from other subsystems, which, in turn, increases independence and portability.

3) The Facade does not prevent applications from directly accessing the subsystem classes, if necessary.

Report a typo

83 users liked this theory. 0 didn't like it. **What about you?**

Start practicing

Comments (1) Hints (0) Useful links (0) Show discussion