21 Facade

The facade belongs to the structure patterns; it describes how you can access a complicated or complex subsystem in an uncomplicated way.

21.1 An Example Outside IT

Imagine that you have a complex or complicated system – for example, an SLR camera. When you take a portrait, you want the depth of field to cover only a small area. You open the aperture as wide as possible, so choose a small f-number. This allows more light to fall on the film or the processor. To prevent the image from becoming too bright, you need to reduce the exposure time.

Maybe you don't want to take a portrait later, but a landscape. There, the depth of field should be as large as possible; you select a large f-stop number and thereby close the aperture. Now there is less light reaching the processor, so you have to increase the exposure time. However, the exposure time cannot be increased indefinitely; experience shows that a picture can only be taken handheld if the exposure time is shorter than 1/focal length. If the exposure time is longer, you run the risk of blurring the image. So you need to increase the film speed, the ISO number.

Does that sound complicated? I think it is! And most camera manufacturers see it the same way. Modern compact cameras, but also (digital) SLR cameras, come with scene programs. All you have to do is tell the camera: "I want to take a portrait!" or: "I want to take landscapes!". The camera automatically sets the aperture and exposure time so that the result is optimal.

This actually explains the principle of the facade: You have a complicated or complex system. To make it easier for you to access the system, a facade is created. As a user or

O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_21

photographer, you no longer need to concern yourself with the details. It should be enough that you tell the facade – the subject program "Portrait" – what you want to have. The "how" is realized by the facade. You will still have access to the individual components of the system: you do not have to use the facade, you can still set the aperture and exposure time manually.

In the following section, after this short excursion, you will again be introduced to an example from IT.

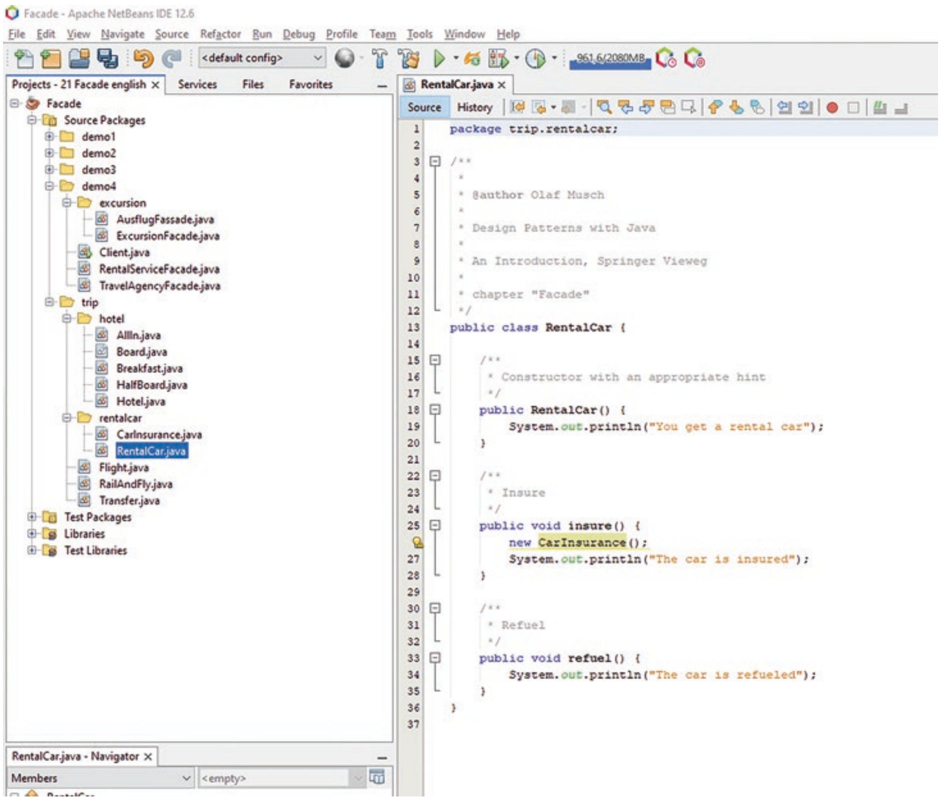
21.2 The Facade in a Java Example

Take a look at the example project facade. It consists of the packages demo1 to demo4 and the package trip. In this example, you will create a journey. For a trip, you must first take the train to the airport. From there, you fly to your vacation destination. After you arrive at the far airport, the transfer service will take you to the hotel. At the hotel, you will be provided with either all-inclusive or half board or breakfast only. Optionally, you can take a rental car, which must be insured and refueled. In the package trip all necessary classes can be found. You can see how this project looks in the development environment in Fig. 21.1.

If a client wants to create a trip, it must create all components. Look at the client code in the package demo1.

```
public class Client {
    public static void main(String[] args) {
        System.out.println("A client:");
        var railAndFly = new RailAndFly();
        railAndFly.board();
        var flight = new Flight();
        flight.checkBaggage();
        flight.identify();
        var transfer = new Transfer();
        transfer.loadBaggage();
        var hotel = new Hotel();
        hotel.rentSafe();
        var halfboard = new HalfBoard();
        halfboard.orderBeer();
        // … . abridged
    }
}
```

If the customer also needs a rental car, they must also first create a rental car object and then call the `insure()` and `refuel()` methods on it. So on the one hand, you have many components that you have to serve. On the other hand, you have different configuration options – in one case it is sufficient to create objects, in the other case you have to call methods on the objects in addition. Does this look like a complex system to you?



```
package trip.rentalcar;

/**
 *
 * @author Olaf Musch
 *
 * Design Patterns with Java
 *
 * An Introduction, Springer Vieweg
 *
 * chapter "Facade"
 */
public class RentalCar {

    /**
     * Constructor with an appropriate hint
     */
    public RentalCar() {
        System.out.println("You get a rental car");
    }

    /**
     * Insure
     */
    public void insure() {
        new CarInsurance();
        System.out.println("The car is insured");
    }

    /**
     * Refuel
     */
    public void refuel() {
        System.out.println("The car is refueled");
    }
}
```

Fig. 21.1 The “Facade” project in the NetBeans development environment

```
public class Client {
    public static void main(String[] args) {

        // … abridged

        System.out.println("\nAnother Client:");
        railAndFly = new RailAndFly();
        transfer = new Transfer();
        hotel = new Hotel();
        halfBoard = new HalfBoard();
        var rentalCar = new RentalCar();
        rentalCar.insure();
        rentalCar.refuel();
    }
}
```

You must have noticed that the flight was forgotten in the upper listing. But maybe you know the problem from practice: Errors in extensive configurations can only be found after an even more extensive test procedure. So the code from above is error-prone – in the next section you will learn a solution to this dilemma.

21.2.1 Introduction of a Facade

You introduce a facade whose only task is to provide a simplified handling for the client on the one hand, but on the other hand to master the complex interrelationships of the system. In the package demo2, you will find the class `TravelAgencyFacade`, which offers the method `bookTrip()`. You give this method a truth value as a parameter that specifies whether a car should be rented.

```
public final class TravelAgencyFacade {
    public void bookTrip(boolean withRentalCar) {
        var railAndFly = new RailAndFly();
        var flight = new Flight();
        var transfer = new Transfer();
        var hotel = new Hotel();
        var halfboard = new HalfBoard();
        if (withRentalCar) {
            var rentalCar = new RentalCar();
            rentalCar.insure();
            rentalCar.refuel();
        }
    }
}
```

The client class in the package demo2 can now call the method very easily.

```
public class Client {
    public static void main(String[] args) {
        System.out.println("A Client:");
        new TravelAgencyFacade().bookTrip(false);

        System.out.println("\nAnother Client:");
        new TravelAgencyFacade().bookTrip(true);
}   }
```

It is now much easier for the client to book a trip. Let me point out one thing: In practice, you will just call the class `TravelAgency`. I only added the addition `TravelAgencyFacade` for clarification.

21.2.2 A Closer Look at the Term "System"

The term "system" is important for the facade: They provide a simplified access to a subsystem. The term system or subsystem is to be interpreted broadly. It can mean an arbitrarily large unit; but it can also mean access to a single class. In the package demo3, I have considered the car rental as a separate system and moved it to a separate package. I also defined the Car Rental facade, which has a single static method that can be used to rent a car: Access to this (sub)system is now only possible through the facade.

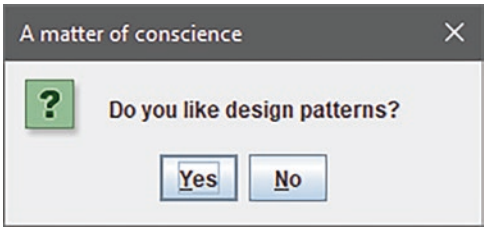
```
public class CarRentalFacade {
    public static void rentCar() {
        var rentalCar = new Rentalcar();
        rentalCar.insure();
        rentalCar.refuel();
    }
}
```

The travel agency now has simplified access to this subsystem.

```
public class TravelAgencyFacade {
    public void bookTrip(boolean withRentalCar) {
        var railAndFly = new RailAndFly();
        var flight = new Flight();
        var transfer = new Transfer();
        var hotel = new Hotel();
        var halfboard = new HalfBoard();
        if (withRentalCar)
            CarRentalFacade.rentCar();
    }
}
```

The client can either rent a car from the travel agency or directly from the car rental company. By the way, in the implementation I've presented to you here, it's not a problem that the client still accesses the individual elements of the subsystem. The facade does not restrict, nor does it introduce any new logic – it merely simplifies access to a subsystem.

An alternative procedure is demonstrated by the package demo4. Here there is the subsystem (subpackage) excursion. This system is represented in a simplified way by the class `Excursion`. The class and its methods are package-private, so that access is only possible through the facade. Please analyze demo4 from this point of view. It is also conceivable that the facade not only mediates between client and subsystem, but that it defines its own logic.


A matter of conscience

Do you like design patterns?

Fig. 21.2 Message generated by "JOptionPane"

The advantage of the facade is obvious: access is simplified, the dependency of client and subsystem is loosened. This avoids client code breaking when the subsystem is changed or replaced. It is even possible to replace a complete subsystem.

In the next section, let's look at where the facade can be found in the Java class library.

21.3 The Facade in the Class Library

I would like to show you an example from the Java class library. You have created a graphical user interface and want to display a message to the user. You could now design a `JDialog` that has a `BorderLayout.` On `BorderLayout.WEST,` you place an icon that displays a warning exclamation point. On BorderLayout.`CENTER`, display the message. BorderLayout.`SOUTH` contains a `JButton` labeled "Ok." When the user clicks on it, the EventListener will cause the `JDialog` to close.

Alternatively, you could make do with the facade of the `JOptionPane` class, which brings various static methods that you can pass different parameters to in order to configure the dialog. Consider the following lines of code.

```
String question = "Do you like design patterns?";
String title = "A matter of conscience";
javax.swing.JOptionPane.showConfirmDialog(null, question,
            title, javax.swing.JOptionPane.YES_NO_OPTION,
            javax.swing.JOptionPane.QUESTION_MESSAGE);
```

When you run this code, you will be presented with a message, as shown in Fig. 21.2.

The facade requires a large number of classes that it calls. To give you an idea of the large number of classes involved, I am printing an overview in Fig. 21.3, which I have borrowed (and slightly adapted in layout) from Philipp Hauer.[1]

I think this example makes the meaning of the phrase "simplified access to a subsystem" pretty clear.

[1] https://www.philipphauer.de/study/se/design-pattern/facade.php

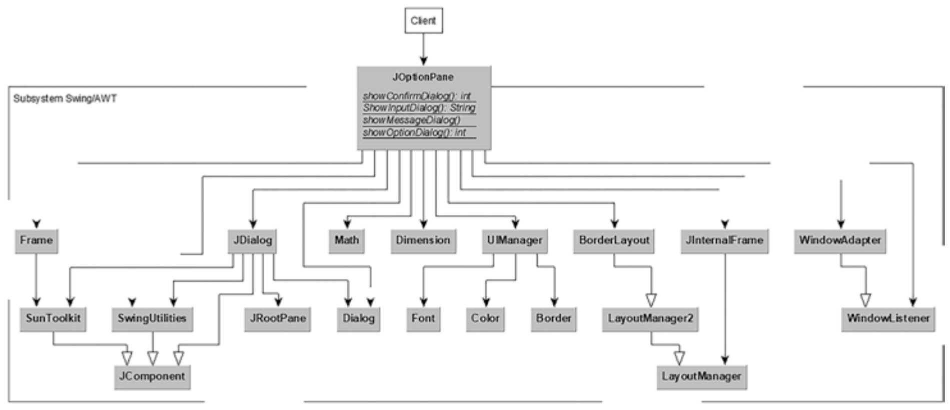


Fig. 21.3 Extract from the API Doc of the JOptionPane Class

21.4 The “Law of Demeter”

In Chap. 2 I showed you some design principles. Now I will address another design principle. It is about the “Law of Demeter”,[2] which says – in short – that objects should limit their communication to close friends. The facade shows you one way to make this happen as easily as possible. So what exactly does Demeter’s Law say? Objects should only communicate with close friends. Close friends are:

- Attributes and methods of its own object, that is, everything called `this`,
- Methods of objects passed as parameters,
- Methods of objects that the method itself creates,
- Global objects.

If you look at the travel example from above, you can see that the client only needs to access one friend – the facade. The facade gives the client the option of not having to deal with strangers, i.e. additional classes. The facade is an illustrative example of the realization of the principle. I found a detailed description of the LoD on Matthias Kleine’s blog. If you want to look further into object-oriented design principles and principles of software engineering, this page is a good starting point for your research:

http://prinzipien-der-softwaretechnik.blogspot.com/2013/06/das-gesetz-von-demeter.html

You can also find the (german) text as “LoD.pdf” in the directory of sample projects for this chapter.

[2] Demeter in Greek mythology is the mother of Persephone and the unwilling mother-in-law of Hades. She provides seasons, fertility and growth.

In the chapter on object-oriented design patterns, I felt it was important to say that you, the programmer, are not there to satisfy the laws. Rather, the laws are there for you-they are there to make your job easier. Therefore, even with Demeter's law, *nulla regula sine exceptione*, or *No Rule Without Exception,* applies. You may break the law if you are clear about it and can justify the breach. If you want to print some text on standard output, keep coding `System.out.println()`. And if you want to know the name of a class, the statement `myObject.getClass(). getName()` is still allowed. However, be careful whenever you access objects that are not part of Java's standard class library.

I would like to offer you an introduction to design patterns with my book. Therefore, I limit myself to a few design principles. However, I hope that I can motivate you to deal with the topic in your own research.

21.5 Facade – The UML Diagram

Figure 21.4 shows the UML diagram for the example project Facade, package demo4.

21.6 Summary

Go through the chapter again in key words:

- The initial situation is a complicated system that you want to work with.
- Access to the system is simplified by a facade.
- The client accesses the facade and does not necessarily know the details of the system.
- The facade can either contain business logic or simply forward requests to the appropriate system object.
- The system becomes interchangeable.
- Dependencies between systems are simplified or dissolved.
- Optionally, a client can still access the system.
- The Demeter law is implemented.

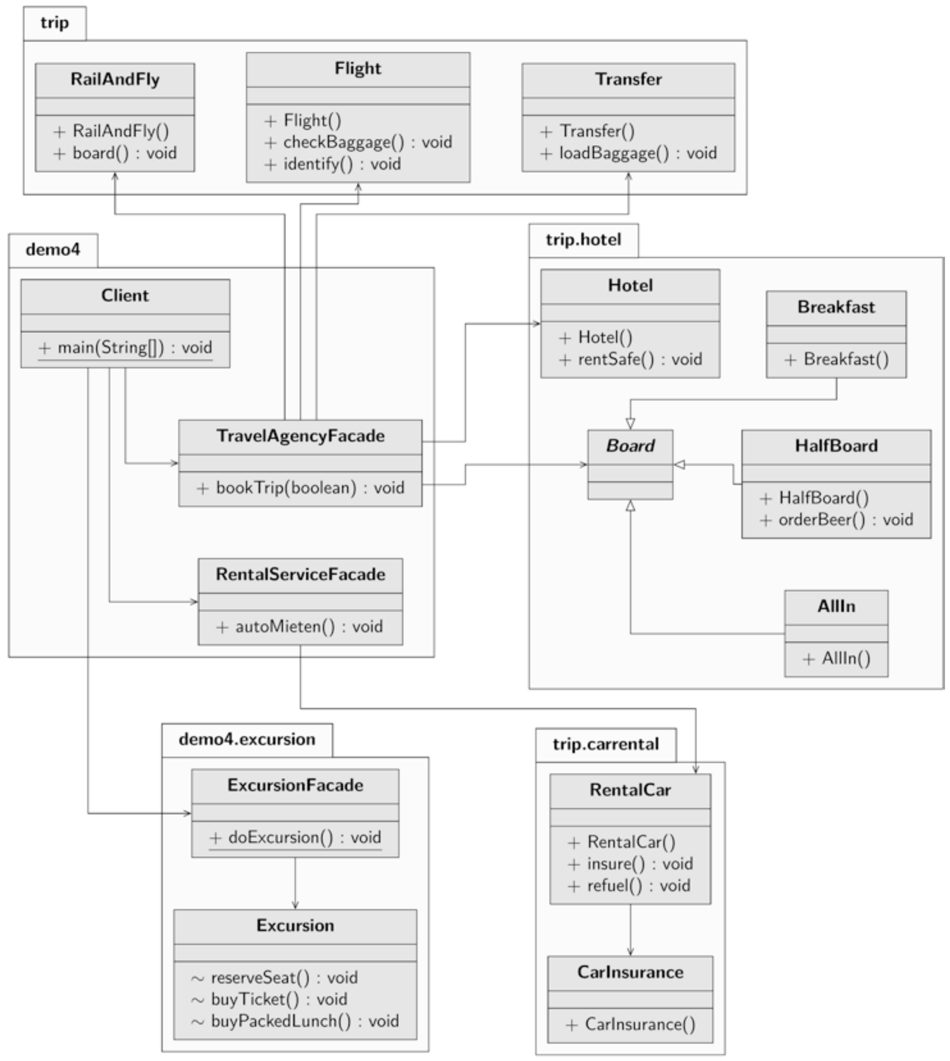


Fig. 21.4 UML diagram of the Facade Pattern (example project facade, package demo_4)

21.7 Description of Purpose

The Gang of Four describes the purpose of the “Facade” pattern as follows:

> Provide a unified interface to a set of interfaces of a subsystem. The facade class defines an abstract interface that simplifies the use of the subsystem.