

# Chapter 19

## Facade

### 19.1 The Problem

Throughout the pay station story, I have only relied on my test cases for verifying my code's behavior. In practice it is also important to test the software in its proper context using the real hardware where users can enter coins, push buttons, and inspect receipts. This system testing level often also finds some defects, defects that are best reproduced as automatic test cases and next corrected.

In our case, however, setting up the hardware is tedious so a second best approach is to equip the pay station with a graphical user interface that mimics a real pay station. I have developed a user interface as shown in Figure 19.1<sup>1</sup>. Buttons, marked with coin values, replace inserting coins, and receipts appear as separate windows with the proper text. You can find the source code on the website in folder *chapter/facade*.

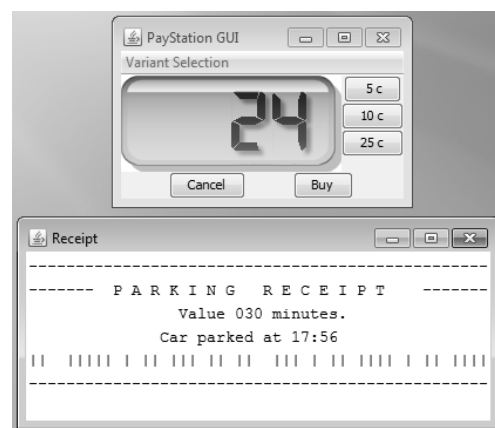


Figure 19.1: A graphical user interface for the pay station.

<sup>1</sup>The digital number display used by permission by SoftCollection.

Originally, I defined the protocol between the hardware and the pay station code in a sequence diagram in Chapter 4. The question is whether the graphical user interface (GUI) forces me to make changes in the `PayStation` interface or in its complex design? The answer is fortunately *no*. Looking over the sequence diagram I see that the GUI needs no further methods to add coins, read the display, nor receive receipts. All interaction is via the methods defined in the original `PayStation` interface. Thus all the complexity of rate strategies, receipts, factories, etc., is nicely encapsulated behind this interface. If you inspect the folder structure of the production code, you will see that I have added another package, `paystation.view`, that contains the Swing based GUI. If you inspect the class implementing the GUI you will furthermore discover that it is only coupled with the two interfaces of the paystation, `PayStation` and `Receipt`, as outlined in Figure 19.2.

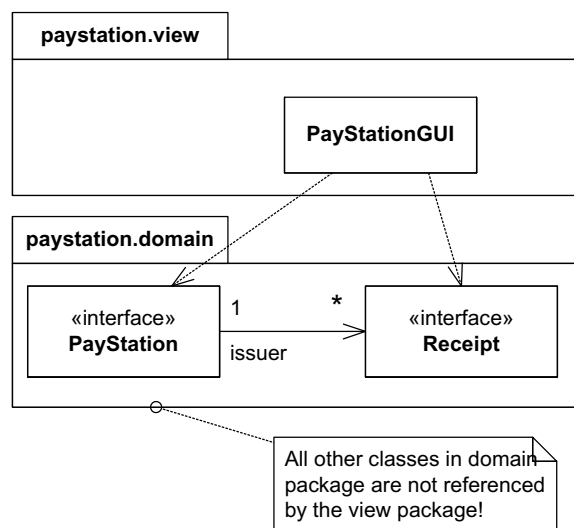


Figure 19.2: `PayStation` and `Receipt` acting as a FACADE.


**Exercise 19.1:** Actually, the GUI code also refers to some concrete classes from the `paystation.domain` package as well. Review the production code and find these classes. How could you remove this coupling?

This is an example of the FACADE design pattern, outlined in pattern box 19.1 on page 10. The pattern defines two central roles: the **client** that interacts with a complex subsystem, and the **facade** that shields the client from the subsystem's complexity by defining a simple interface. A good example of a FACADE pattern is a compiler. A compiler is a vastly complex piece of software with a lot of substructure, yet the interface is basically very simple: you invoke it with the name of the source file you want to be compiled; or press a single key in your integrated development environment.

My pay station design has thus embodied a facade pattern from the very beginning. Was this accidental? No. I did not spend much time discussing my original design proposal in Chapter 4. At that time it was postulated as a premise for our continued process. However, this design was also created based on the ③-①-② process.

- ③ *I identified some behavior that varied.* The hardware may have to drive different kinds of pay station domain implementations: I envisioned that Alphatown would not be the only customer.
- ① *I stated the pay station domain responsibility in an interface.* The `PayStation` interface encapsulates all behavior related to what pay stations must be able to do.
- ② *I composed the full behavior by letting the hardware delegating concrete pay station behavior to a subordinate object.* Instead of integrating the pay station domain behavior like calculating rates, keeping a sum of entered amount, etc., directly in the hardware near code (or in the user interface code) I delegated all these aspects to the object that implements the `PayStation` interface.

**Exercise 19.2:** Actually the `PayStation` interface also acts as facade for client code beside the hardware and the GUI. Which client code is that?

 Study the source code provided in folder `chapter/facade` on the web site.

## 19.2 The Facade Pattern

The key aspect of the FACADE (design pattern box 19.1, page 10) pattern is encapsulation behind an interface (thus the ① part of the ③-①-② process). Its intent is

*Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.*

Facades shield clients from a subsystem's complexity and as subsystems tend to get more complex as they age and evolve, this is important. Most patterns result in more and smaller classes which makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don't need to customize it.

The decoupling goes both ways. The subsystem is also shielded from change in the clients. In the pay station case, the pay station domain code is decoupled from the concrete kind of client: GUI or hardware. Thus, facade supports a many-to-many relation between clients and subsystems.

In the presentation of FACADE in the design pattern box, the client only interacts with a single interface. In reality, the subsystem designer has to consider how data and objects are exported out to the clients over the facade interface. Several options exist.

- *Make the facade opaque.* Clients only communicate in simple data types over the facade—no object references to objects created within the subsystem must be returned to clients. Alternatively, you may define “dumb” data objects (objects containing public instance variables but no methods) that are passed to clients. The facade must then typically convert internal objects into data objects by extracting their information and copy it into the data objects before returning these objects to the client.

- *Make the facade export read-only objects.* This way the facade in reality consists of multiple interfaces and is allowed to export references to objects created within the subsystem. However, exported objects must only be known to the client by a read-only interface, i.e. an interface that contains accessor methods but no mutator methods. Obviously if the exported object had mutator methods, the client would be able to change state in the subsystem by these and thus bypass the facade. The exported objects appear immutable to the client, but can of course be mutable by subsystem objects behind the facade that knows their concrete class.

**Exercise 19.3:** Classify the pay station facade. Which of the two types of facade is it?

FACADE does not require *all* clients to use it. For instance a complex subsystem could provide a facade to provide easily understandable access to the most common uses of the subsystem but not access to all functionality. If a client is required to access all functionality it can bypass the facade but developers must then be more careful and understand the subsystem in considerably more detail.

## 19.3 Selected Solutions

### Discussion of Exercise 19.1:

The GUI class also refers to `PayStationImpl` and one of the `ABSTRACT FACTORY` classes. This is because I have made the GUI class responsible for creating and configuring the concrete variant of pay station to use: in the constructor and in the menu handling code. However, all other methods in the GUI are independent of concrete classes from the pay station domain package. It is thus a relatively simple exercise to split the present code into a *configuration* part (that couples to concrete classes) and a *graphical user interface* part (that only refers to interfaces).

### Discussion of Exercise 19.2:

The JUnit integration test cases act as a third type of client as they only interact with the pay station through the facade.

### Discussion of Exercise 19.3:

The pay station exports receipt objects which are a direct reference to an object created by the subsystem—it is thus of the latter, non-opaque, type. The `Receipt` interface only contains accessor methods, so the client cannot change it.

## 19.4 Review Questions

Describe the FACADE pattern. What problem does it solve? What is its structure and what is the protocol? What roles and responsibilities are defined? What are the benefits and liabilities?

## 19.5 Further Exercises

### Exercise 19.4:

It seems that FACADE is very much like the *interface* language construct found in Java and C#. So—what is it that makes facade more than just stating that you should *program to an interface*? As an example, the Receipt interface also encapsulates the receipt behavior. But, why is this not a facade pattern?

### Exercise 19.5. Source code directory:

`exercise/tdd/breakthrough`

The Breakthrough interface can be viewed as a FACADE to a Breakthrough game. Argue in favor of this interpretation.

### Exercise 19.6:

Consider a software based media player system that can play CDs, DVDs, MP3 files, and other media. Consider that the system is divided into a domain part and a graphical user interface part. Sketch a FACADE for the domain part.

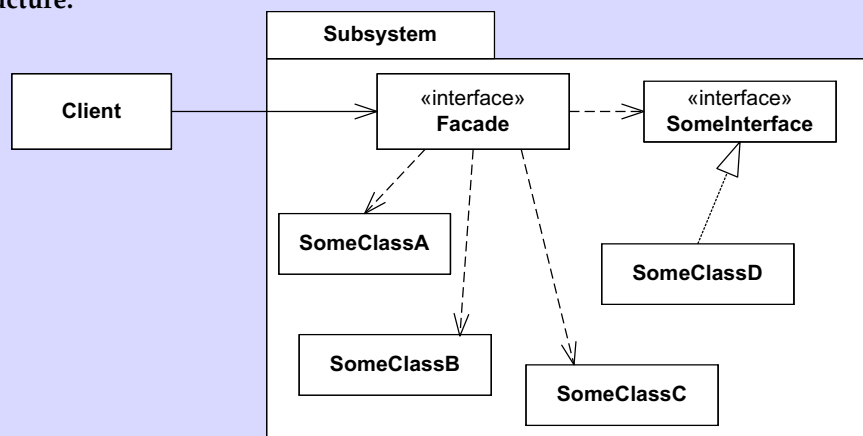
## [19.1] Design Pattern: Facade

**Intent** Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

**Problem** The complexity of a subsystem should not be exposed to clients.

**Solution** Define an interface (the facade) that provides simple access to a complex subsystem. Clients that use the facade do not have to access the subsystem objects directly.

**Structure:**



**Roles** **Facade** defines the simple interface to a subsystem. **Clients** only access the subsystem via the **Facade**.

**Cost - Benefit** The benefits are that it *shields clients from subsystem objects*. It *promotes weak coupling* between the client and the subsystem objects and thus lets you change these more easily without affecting the clients. A liability is that a **Facade** can bloat with a large set of methods in order for clients to access all aspects of the subsystem.