Bridge

25

The Bridge Pattern literally bridges the gap between the implementation of a functionality and its application. It separates the abstraction from the implementation. This allows both sides to be developed independently of each other. At the same time, the implementation remains hidden from the client. Various examples of this can also be found in the Java class library.

25.1 Two Definitions

I want to define two terms before I get to the Bridge Pattern. If you are reading this section, you may find my execution trivial. However, a precise definition of the terms abstraction and implementation is important for understanding the Bridge Pattern.

25.1.1 What Is an Abstraction?

The first concept I want to address is abstraction. When you develop a class, you always have to decide what part of the real world you want to represent and with what accuracy. If you're writing software that a college uses to manage its students, you're going to pick out very specific relevant characteristics of a student and include those in your classes. For example, date of birth, address, and exams passed are relevant to the university. Irrelevant are likely to be hair color, weight, number of siblings, and many other characteristics. In your university software, you **reduce the** student to the relevant characteristics; you create a **model**, you **abstract**. For example, another abstraction is a calculator, which will be relevant to the sample project in a moment. When you look at the calculator on your desk,

O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_25

it has certain features: a specific weight, a color, different arithmetic operations, and so on. When you develop the class Calculator, you ignore numerous features and capabilities; you reduce the real object to the features and capabilities that are relevant to the given context.

The next listing from the sample project Calculator shows such an abstraction. I only print the interface here, because it is important for the client. I leave out the implementation behind it to your own analysis.

```
public class Calculator {
    public double add(double summand_1,
                                    double summand_2) {
        // Implementation
    }

    public double subtract(double minuend,
                                    double subtrahend) {
        // Implementation
    }

    public double multiply(double factor_1,
                                    double factor_2) {
        // Implementation
    }

    public double divide(double dividend,
                                        double divisor) {
        // Implementation
    }
}
```

When a client creates an object of this class, it calls the methods that the interface provides. The abstraction specifies what the object can do.

25.1.2 What Is an Implementation?

If the client relies on the abstraction, it does not have to worry about the implementation behind it. The details of the implementation may even remain hidden from him. I've never been interested in how the candy machine in the office turns a euro piece into a candy bar. Nor do I care how the calculator adds two numbers. For me as a user, the implementation, the how of a method, is not important.

Let's look at the implementation in the following step. In the first approach, you could simply implement the school methods.

```
public class Calculator {
    public double add(double summand_1,
                                    double summand_2) {
           return summand_1 + summand_2;
    }

    public double subtract(double minuend,
                                    double subtrahend) {
        return minuend - subtrahend;
    }

    public double multiply(double factor_1,
                                    double factor_2) {
        return factor_1 * factor_2;
    }

    public double divide(double dividend,
                                        double divisor) {
        return dividend / divisor;
    }
}
```

You now want to change the implementation. There are alternative multiplication methods that – especially for long numbers – are much more performant than the multiplication algorithm taught in school. If you want to change the implementation, you can simply create a subclass that overrides the multiply() method. If you want to multiply two numbers using the Fast Fourier Transform (FFT), use the following class.

```
public class FFT extends calculator {
    @Override
    public double multiply(double factor_1,
                                    double factor_2) {
        // not printed - FFT multiplication
    }
}
```

A client can now choose whether to use the simple multiplication algorithm or the FFT. Accordingly, it can start an instance of the calculator either with.

```
Calculator calculator = new calculator();
```

or with.

```
Calculator calculator = new FFT();
```

By the way, I find the different multiplication methods quite exciting. If you want to deal with FFT multiplication, I recommend the following site: http://www.inf.fh-flensburg.de/lang/algorithmen/fft/fft.htm. You can also find the article by Prof. Dr. Lang deposited there as "Transformations.pdf" in the directory of sample projects for this chapter.

There are many other multiplication algorithms, such as the peasant multiplication (PM). A few of these algorithms are highly performant, others impress with elegance.

25.1.3 A Problem Begins to Mature

Right after you have published your calculator, extension requests come in. The calculator should also support quadrature. Since you want to distribute this feature separately, you develop the abstraction, the interface, further. Since squaring means multiplying a number by itself, you can use the existing methods.

```
public class CalculatorDeluxe extends Calculator {
    public double square(double number) {
        return multiply(number, number);
    }
}
```

As a result, you have the class diagram in Fig. 25.1.

This solution has a catch. They had further developed the implementation up front and implemented high performance multiplication algorithms. But when a customer buys the CalculatorDeluxe, he only gets the standard implementation of the multiplication method. He can't use FFT multiplication any more than he can use peasant multiplication. Maybe the cow is off the ice when the CalculatorDeluxe doesn't inherit from Calculator but from FFT.

```
public class CalculatorDeluxeFFT extends FFT {
    public double square(double number) {
        // Implementation
    }
}
```

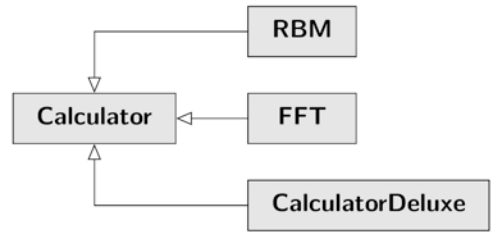

Fig. 25.1 Incomplete class diagram of the project calculator

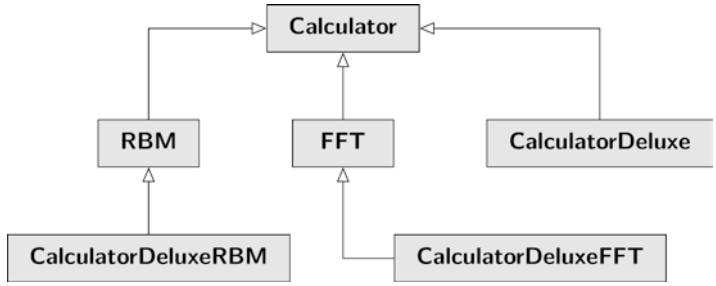

Fig. 25.2 "Advanced" class diagram of the project calculator

However, this solution leaves out the peasant multiplication and the school multiplication implemented by default. You cannot avoid extending the inheritance hierarchy according to these two algorithms accordingly (Fig. 25.2).

It becomes clear that the inheritance hierarchy is too broad and too inflexible. If either the abstraction is extended or an algorithm is added to the implementation, the number of classes grows rapidly. Assume that division is to be replaced. The algorithm you know from school asks how many times the divisor fits in the dividend. An alternative approach might be to multiply by the reciprocal of the divisor. You approach the reciprocal of a number iteratively using a formula from Newton. If you were to try to implement the division algorithm by inheritance, you would have to consider that you have two abstractions – the Calculator and the CalculatorDeluxe. Both abstractions would want to optionally multiply using FFT, the PM, or the school method, and optionally divide using the school method or Newton's method.

Conclusion: The problem is that abstraction and implementation depend too much on each other and influence each other. The Bridge Pattern will solve this problem.

25.2 The Bridge Pattern in Use

The Bridge Pattern separates the abstraction from the implementation. How does this work? You may remember the State Pattern and the Strategy Pattern! There you defined the behavior of an object in its own classes. You have specified an interface in your abstraction that the implementation of the classes that define the behavior must match.

25.2.1 First Step

The Bridge Pattern takes a similar approach. You define the abstraction with all the methods it should offer. The execution is delegated to an object of type Implementor. It is allowed that the methods of the abstraction and the implementor have different identifiers. Have a look at the sample project CalculatorBridge.

```
public class Calculator {
    private Implementor implementor;
    public Calculator(Implementor implementor) {
        this.implementor = implementor;
    }

    public void setImplementor(Implementor implementor) {
        this.implementor = implementor;
    }

    public double add(double summand_1,
                                  double summand_2) {
        return implementor.add(summand_1, summand_2);
    }

    public double subtract(double minuend,
                                  double subtrahend) {
        return implementor.subtract(minuend, subtrahend);
    }

    public double multiply(double factor_1,
                                  double factor_2) {
        return implementor.multiply(factor_1, factor_2);
    }

    public double divide(double dividend,
                                     double divisor) {
        return implementor.divide(dividend, divisor);
    }
}
```

The implementor can be either an interface or an (abstract) class. In the current project, I equipped the implementor with the computational methods I learned in school.

```
public class Implementor {
    double add(double summand_1, double summand_2) {
        return summand_1 + summand_2;
    }

    double subtract(double minuend, double subtrahend) {
        return minuend - subtrahend;
    }

    double multiply(double factor_1, double factor_2) {
        return factor_1 * factor_2;
    }
```

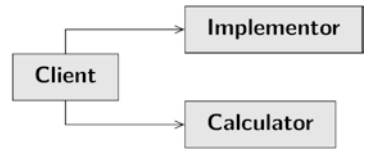

Fig. 25.3 Class diagram of the project CalculatorBridge

```
    double divide(double dividend, double divisor) {
        return dividend / divisor;
    }
}
```

The client creates an instance of the implementation and parameterizes the abstraction with it:

```
Implementor implementor = new Implementor();
Calculator calculator = new calculator(implementor);
System.out.println("30 * 12 = " + calculator.multiply(30, 12));
```

The class diagram looks as follows at this stage of development (Fig. 25.3).

The implementor and the abstraction are connected via an aggregation – thus bridging the gap between the two.

25.2.2 Second Step

Up to this point, it all looks harmless. But let us convince you that with this solution you have an incredibly powerful tool in your hands.

25.2.2.1 Extending the Abstraction

First, let's expand the abstraction. As in the first example, the client wants to be able to square a number. Since the square of a number is the number times itself, the implementation is completely unaffected.

```
public class CalculatorDeluxe extends Calculator {
    public CalculatorDeluxe(Implementor implementor) {
        super(implementor);
    }

    public double square(double number) {
        return multiply(number, number);
    }
}
```

Further abstractions would be conceivable. If you want to calculate the square root of a number, you can fall back on the implementation already defined. The square root of y is the multiplication of y by the reciprocal of the square root of y. To calculate the reciprocal of the square root of a number, there is an iterative procedure that goes back to Newton. Starting from an approximated value, the respective consequent value is calculated – the consequent value is calculated by addition, multiplication, difference and division alone.

```
public class CalculatorNewton extends CalculatorDeluxe {
    public CalculatorNewton(Implementor implementor) {
        super(implementor);
    }

    public double square(double number) {
        // iterative scheme according to Newton
    }
}
```

The client can now choose the abstraction it wants to have and parameterize it with the implementor. It is important that only the abstraction changes or is extended.

25.2.3 Extending the Implementation

Regardless of the abstraction, the inheritance hierarchy can be extended according to the Implementor class. The Implementor itself provides only the simple implementation of the basic arithmetic operations, which are then used in the calculator. As in the previous project, I defined the classes FFT and RBM, which override the multiplication algorithm – i.e. do not add any fundamentally new functionality. I did not implement the respective methods. But you are welcome to try that yourself.

```
public class RBM extends Implementor {
    @Override
    double multiply(double factor_1, double factor_2) {
        System.out.println("\tPeasant multiplication");
        return factor_1 * factor_2;
    }
}
```

The client can now choose its abstraction and a suitable implementation at will.

```
implementor = new PM();
calculator = new calculator(implementor);
System.out.println("30 * 12 = " + calculator.multiply(30, 12));
```

The Implementor can be extended at runtime.

```
calculator.setImplementor(new FFT());
System.out.println("30 * 12 = " + calculator.multiply(30, 12));
```

When you run the code, it prints to the console:

```
Peasant multiplication
30 * 12 = 360.0
FFT multiplication
30 * 12 = 360.0
```

It is important – and the class diagram in Fig. 25.4 should make this clear once again – that the implementation can now evolve independently of the abstraction.

However, some tasks can no longer be accomplished with the basic arithmetic already defined. For example, if the client needs a random number generator, you need to extend both the abstraction interface and the implementor.

25.3 Discussion of the Bridge Pattern

In this section, I will conclude by showing where you can find the Bridge in the class library. I will also distinguish the Bridge from other patterns.

25.3.1 Bridge in the Wild

When you deal with AWT and peer classes, you dig deep into the primordial slime of Java's evolution. An AWT component is not drawn by Java itself, but by the operating system. For example, the developer has a `Button` class that inherits from `Component`. `Component` is the top of the inheritance hierarchy of abstraction. On the other side of the bridge is the interface `ComponentPeer`, from which the implementation, for example, the class `ButtonPeer`, is derived. As a programmer, you have no access to this implementation (at least in theory).

You can also find the Bridge Pattern in a completely different context – database programming. Take a look at how you work with JDBC. You load a database driver with `Class.forName(<driver name>)`. Then you let it give you the connection to the database:

```
Connection connection =
   DriverManager.getConnection(<url>, <user>, <password>)
```

From this connection, get an object of type `Statement`:

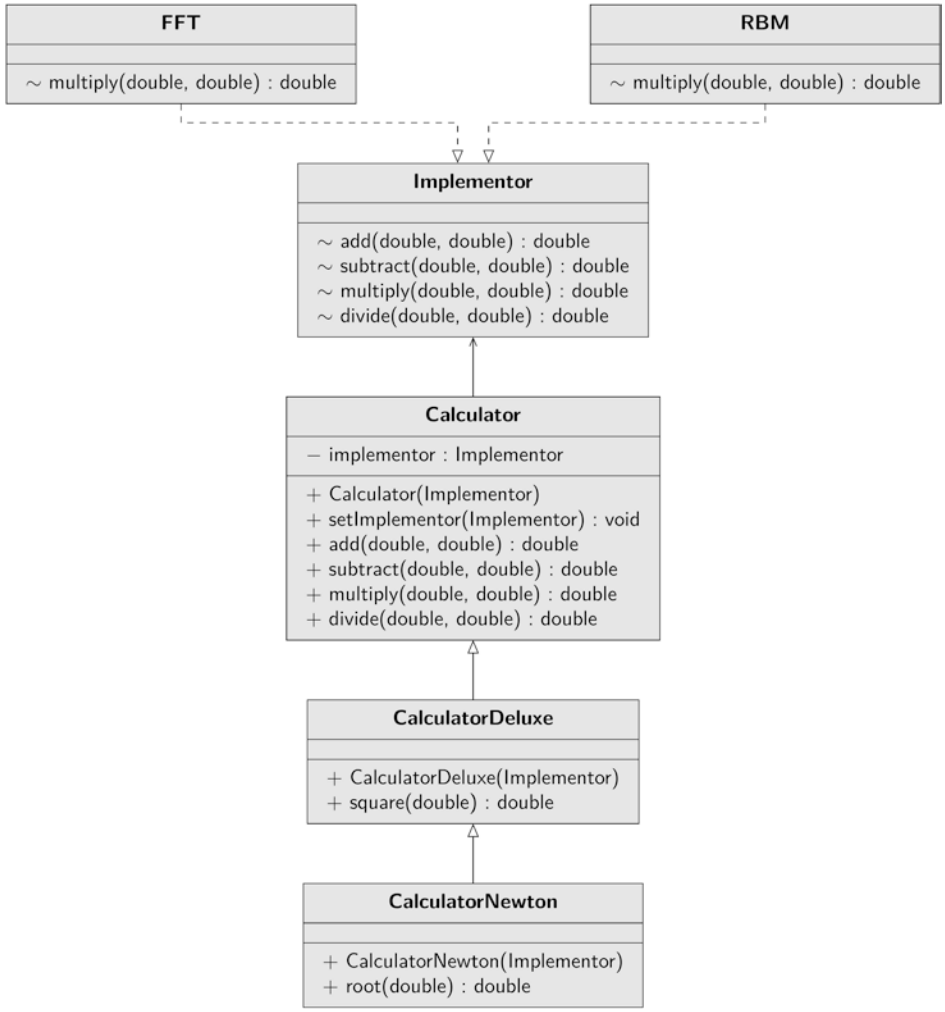

Fig. 25.4 UML diagram of the Bridge Pattern (sample project CalculatorBridge)

```
Statement state = connection.createStatement();
```

You place your SQL command on this statement object:

```
ResultSet result = state.executeQuery( … command … );
```

You iterate over the returned ResultSet to get the data you want. If you initialize the abstraction – your application – with a given driver, you can choose to access an Access database or an Oracle database or any other database. You always work with the given

Connection, Statement, and ResultSet components. The question of how your query is processed is not relevant to you, i.e.: you have no access to the implementation running in the background.

Abstraction and implementation evolve independently of each other. Both can evolve and be reused independently.

25.3.2 Distinction from Other Patterns

Facade, Adapter, Proxy, Decorator and Bridge are very similar. They use composition to wrap another object. A method call is delegated to the wrapped object.

The task of the adapter is to relate two interfaces of different types in such a way that they can work together. Since multiple objects can be adapted, this compensates for the lack of multiple inheritance in Java. Adapters are typically very lean because their only task is to relate two systems; own intelligence might provide, for example, that data is converted.

Decorators bring intelligence of their own – they are of the same type as the wrapped object and extend its behavior. A typical textbook example is the FileInputStream, which is wrapped into a BufferedInputStream. Both classes are of type `InputStream`.

The proxy is primarily a proxy for another object. The implementation of a proxy can be very similar to the decorator. The goal of a proxy can be to control access to the represented object.

The facade is most likely to be confused with the adapter. Its task is to simplify access to a (sub)system. Think of booking a holiday trip, which consists of many individual steps.

You can vary an implementation with many patterns: The Strategy Pattern lets you vary an algorithm. The State Pattern defines different behavior depending on the context. With the Adapter pattern, you access a different library. The Bridge Pattern differs from these patterns in that, in addition to the implementation, the abstraction can also be developed further. While you can implement an adapter or a facade at any time, the decision for a bridge must be made as early as possible.

After all, the task of the bridge is to separate an implementation – i.e. the arithmetic operations – from the abstraction – the calculator on which the user types in his calculation. The goal is to be able to develop behavior and abstraction independently of each other.

25.4 Bridge – The UML Diagram

The UML diagram from the sample project CalculatorBridge can be found in Fig. 25.4.

25.5 Summary

Go through the chapter again in key words:

- You model a section of the real world in a class.
- The class is an abstraction of reality.
- The abstraction represents the interface for the client.
- The interface is the contract between client and class.
- The client may rely on the provided methods to perform the defined behavior.
- The behavior is defined by the implementation behind it.
- Abstractions and implementation can change.
- One possible way to define new behavior or abstractions is inheritance.
- When abstraction and implementation both change, inheritance leads to a rapidly unmanageable system.
- The Bridge Pattern separates the abstraction from the implementation.
- The implementation is not defined in the class of the abstraction, but in its own.
- Abstraction and implementation are connected via aggregation.
- Abstraction and implementation form independent inheritance hierarchies.
- Implementation details remain hidden from the client.

25.6 Description of Purpose

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

> Decouple an abstraction from its implementation so that the two can be varied independently.