13 Flyweight

The Flyweight Pattern is a structural pattern; in order to understand it, the terms intrinsic and extrinsic state must be defined. Therefore, I will present two examples in this chapter. The first example clarifies the principle of the pattern, the second is closer to the example of the GoF and describes these two terms.

13.1 Task of the Pattern

The flyweight has the task of turning an elephant into a mosquito. Imagine programming an application that records all births in a year. Each record consists of an ID1, first name, last name, and birthday. Take a look at the very simple example project Birthrates.

```
public class Child {
    private final long id;
    private final String forename;
    private final String surname;
    private final Date dob;

    public Child(String forename, String surname, LocalDate dob) {
        this.forename = forename;
        this.surname  = surname;
        this.dob= dob;
        id = (long) (((Long.MAX_VALUE * Math.random())
                               * (forename.hashCode()
                               + surname.hashCode())
                               + dob.getTime()));
    }
}
```

A client creates a large number of children:

A critical look at the client code shows that two children are named Jack. Two others have the last name Miles. And all six were born on the same day. Do these repeating values meet your expectations?

```
public class ApplStart {
    public static void main(String[] args) {
        var child_1 =
            new Child("Jack", "Miles", LocalDate.now());
        var child_2 =
            new Child("Peter", "Smith", LocalDate.now());
        var child_3 =
            new Child("Jack", "Darby", LocalDate.now());
        var child_4 =
            new Child("Frank", "Miller", LocalDate.now());
        var child_5 =
            new Child("Peter", "Burke", LocalDate.now());
        var child_6 =
            new Child("John", "Miles", LocalDate.now());
    }
}
```

According to the Federal Statistical Office, just under 780,000 children were born in Germany in 2019. If we assume that the births are evenly distributed over the year, then over 2100 people have their birthday on the same day. According to the Gesellschaft für deutsche Sprache e.V. (Society for the German Language), about 2.52% of newborn girls had the first name Marie in 2019, which would therefore result in the same new first name being given about 54 times per day. And surname sameness is also certainly likely for newborns on a single day (In Germany, some common family names are Müller, Schmidt, Schneider, Fischer, …).

So the above approach would result in you having over 2100 identical Date objects; also, the previous programming would create thousands of strings with the same first name and same last name. And we're only talking about one birth cohort here. For the entire population in Germany, you would have to manage over 83 million first and last names individually.

In the simple example code shown above, the compiler is intelligent enough to detect this and optimize it, but if the names need to be read from external sources, the compiler cannot detect this in advance. It would not take long for the program to blow up the available memory.

The Flyweight Pattern now describes how objects of the smallest granularity can be reused. Let's look at what is meant by this in the next section.

13.2 The Realization

Take again as a negative example the project births. Let’s assume for the sake of simplicity that the majority of children get either fashionable names or traditional names. The number of different first names will be very small in proportion to the 780,000 children. I’m assuming there are 200 different first names given in 2019. It would make sense to not create 780,000 string objects that are almost all the same anyway. It would make more sense to create 200 different string objects. Children with the same first name can share a string object.

The example project Birthrates_Flyweight demonstrates this. First, it is noticeable that the child objects are not called or created directly by the client. The client calls the method `getChild()` on a factory and passes the data of the child to this method.

```
public static void main(String[] args) {
    var factory = new factory();
    var child_1 =
      factory.getChild("Jack", "Miles", LocalDate.now());
    var child_2 =
     factory.getChild("Peter", "Smith", LocalDate.now());
    var child_3 =
      factory.getChild("Jack", "Darby", LocalDate.now());
    var child_4 =
    factory.getChild("Frank", "Miller", LocalDate.now());
    var child_5 =
     factory.getChild("Peter", "Burke", LocalDate.now());
    var child_6 =
      factory.getChild("John", "Miles", LocalDate.now());
    factory.evaluate();
}
```

The factory keeps a HashSet in which all first names are stored. When a child is to be created, the call to the add method adds the name to the HashSet only if it is not already contained. Surnames and birthdays are handled in the same way.

```
public class Factory {
    private final HashSet<String> forenameSet =
                                     new HashSet<>();
    private final HashSet<String> surnameSet =
                                     new HashSet<>();
    private final HashSet<LocalDate> dobSet =
                                     new HashSet<>();
    Child getChild(String forename,
                           String surname, Date dob) {
        forenameSet.add(forename);
```

```
            surnameSet.add(surname);
            dobSet.add(dob);
            return new Child(forename, surname, dob);
        }
    }
```

Nothing changes in the child class itself. The check to see if an entry already exists in the set is handled here by the add method of the HashSet, so we don't have to worry about that at all. This makes the code very simple.

The advantage of this approach is that you have reduced a large number of identical objects by dividing the objects.

When objects are shared, you need to critically examine whether you define them mutable or better immutable. If a child is renamed from Peter to Paul, the change of name would affect all births that refer to it. However, since strings are immutable, this problem is unnecessary here. With birthdays, the situation would be more problematic; a Date object could be set to a different value with `date.setTime()`. Thus, all child objects referencing the date would suddenly have a different birthday.

This approach is particularly interesting when these split objects are either very large and/or costly to create, for example by querying a database.

This example is extremely simple. In the following section, you will dive a little deeper into the matter.

13.3 A More Complex Project

Now let's look at an example that is a bit more complex. You program a software that takes orders for a pizzeria.

13.3.1 The First Approach

Your first approach might be to define a Pizza Order class that stores the name of the pizza and the table to which it should be delivered. Take a look at the Pizza sample project.

```
public class PizzaOrder {
    public final String name;
    public final int table;

    public PizzaOrder(String name, int table) {
        this.table = table;
        this.name = name;
        System.out.println("I'll make a " + name);
    }
}
```

The Client creates new order objects and stores them in a list. When each table has placed an order, the pizzas are served.

```
public class ApplStart {
    private static final List<PizzaOrder> orders =
                                new LinkedList<>();

    private static void takeOrder(int table,
                                     String pizza) {
        orders.add(new Pizza(pizza, table));
    }

    public static void main(String[] args) {
        takeOrder(1, "Pizza Hawaii");
        takeOrder(2, "Pizza Funghi");
        takeOrder(3, "Pizza Carbonara");
        // … abridged
        orders.forEach(pizza -> {
            System.out.
            println("Now serving " + pizza.name
                        + " to table " + pizza.table);
        });
    }
}
```

If you analyze the code, you will see that each pizza is ordered multiple times. Of course, in individual cases, it is also possible that two or more orders are placed at one table. Which attribute will you share – the pizza or the table number? It takes a lot of effort to make a pizza. So, since your goal is to save resources in your ordering software, you will share the pizza – but note: only in the software, of course, each guest will end up with the whole pizza of their choice. The number of the table the order comes from will not be shared – the client will be blamed for that right away. So you have two different states: the state that is shared; and the state that the client is held responsible for. **You call the shared state intrinsic; the other extrinsic.** Objects that define the intrinsic state are the flyweights. They are independent of the context in which they are used.

13.3.2 Intrinsic and Extrinsic State

The PizzaOrder class has two attributes: the description of the pizza and the number of the table to which it should be delivered. You decompose the PizzaOrder class into an intrinsic state and an extrinsic state, where the description of the pizza itself should be the intrinsic state. So, you pull the other attribute out of the class and move it into the context. What remains in the pizza class is only what is necessary to describe the pizza. This is not the baked pizza itself, but the note that it was ordered. In the example project PizzaFlyweight it should be sufficient to define the pizza by its name.

```
public class Pizza implements MenuItem {
    public final String name;

    public Pizza(String name) {
        this.name = name;
    }

    // … abridged
}
```

To allow for more dishes (salad, pasta, …) later on, I introduced the interface `MenuEntry` right away in this context, which is implemented by the class `Pizza`. I'll go into this in more detail in a moment; let's look at the client code first.

The client manages the different pizza objects and the table numbers. To enable a unique assignment, the orders are stored in a map; the key of this map is the table number, the value is the ordered menus. In order to represent the situation that a table can place several orders, the menus are stored as arrays.

```
public class ApplStart {
    private static final Map<Integer, MenuItem[]>
                               ORDERS = new HashMap<>();
    private static final MenuFactory
                         MENU_FACTORY = new MenuFactory();

    public static void takeOrder(int table,
                                    String... menue) {
        var order = MENU_FACTORY.getMenu(menu);
        ORDERS.put(table, order);
    }

    public static void main(String[] args) {
        takeOrder(1, "Pizza Hawaii");
        takeOrder(2, "Pizza Funghi");
        takeOrder(3, "Pizza Carbonara");
        takeOrder(4, "Pizza Calzone", "Pizza Carbonara");
        // … abridged
    }
}
```

The code of the factory, which keeps an overview of all ordered variants, is almost identical to the code you already know from the example project "Births". I do not want to go into it any further.

How is the pizza served to the table? The interface `menu` prescribes the method `serve()`, to which you pass the table as a parameter. The GoF describes this by saying, *"If operations depend on extrinsic state, they get it as a parameter."*

```
Public interface MenuItem {
    void serve(int table);
}
```

Now let's introduce the rest of the pizza class, the `serve()` method. It can be limited to outputting the message on the console that the pizza has been served to a specific table.

```
public class Pizza implements MenuItem {
    // … abridged
    @Override
    public void serve(int table) {
        System.out.println("" + name
                + " is served to table " + table + ".");
    }
}
```

Once the client has taken all the orders, he goes through the orders table by table and serves the pizzas.

```
public class ApplStart {
    // … abridged
    public static void main(String[] args) {
        // … abridged
        takeOrder(79, "Pizza Funghi");
        ORDERS.keySet().forEach(tempTable -> {
            var menus = ORDERS.get(tempTable);
            for (var tempMenu : menus)
                tempMenu.serve(tempTable);
        });
    }
}
```

In the following paragraph you can see where the flyweight is used in practice.

13.4 Flyweight in Practice

Where can you find the flyweight pattern in the wild? When you create an integer object using the new operator, you compare the objects' references for equality using ==. Run the following code in a Java shell:

```
Integer value_1 = new Integer(1);
Integer value_2 = new Integer(1);
boolean equality = value_1 == value_2;
System.out.println("Objects are equal: " + equality);
```

The following is output on the console: `Objects are equal: false.`

But for a test, create the objects with `Integer.valueOf():`

Now output to the console: `Objects are equal: true.`

Obviously, the Integer class stores the values in the sense of the Flyweight pattern – at least if the objects are created using `valueOf`. And this is also the recommended method. The constructor `Integer(int)` is marked as "deprecated" since Java 9. It will not be available in future Java versions at some point.

Does this affect all numbers in the value range of integer? Let's just test this!

```
for (int i = -130; i < 130; i++) {
    Integer value_1 = Integer.valueOf(i);
    Integer value_2 = Integer.valueOf(i);
    boolean equality = value_1 == value_2;
    System.out.println(i + ": Objects are equal: "
                                        + equality);
}
```

If you run this code, you will see that all numbers from −128 to +127 are divided:

```
-130: Objects are equal: false
-129: Objects are equal: false
-128: Objects are equal: true
-127: Objects are equal: true
...
-2: Objects are equal: true
-1: Objects are equal: true
0: Objects are equal: true
1: Objects are equal: true
2: Objects are equal: true
...
126: Objects are equal: true
```

```
127: Objects are equal: true
128: Objects are equal: false
129: Objects are equal: false
```

What happens when you create objects via AutoBoxing, which is the automatic conversion of simple data types to the object-oriented types? Take the test:

```
Integer value_1 = 3;
Integer value_2 = 3;
boolean equality = value_1 == value_2;
System.out.println("Objects are equal when AutoBoxing: "
                                                + equality);
```

Output to the console:

```
Objects are the equal when AutoBoxing: true
```

So, you can work with autoboxing or valueOf.

13.5 Flyweight – The UML Diagram

The UML diagram from the PizzaFlyweight example project can be found in Fig. 13.1.

13.6 Summary

Go through the chapter again in key words:

- You have a large number of objects that are either large, costly to create, or simply consume a lot of system resources due to their number.
- You split these objects into a part that can be split and a part that is not split.
- The attributes that can be shared form the intrinsic state; they are flyweights.
- The extrinsic state is shifted to the context.
- The context manages the intrinsic and extrinsic state.
- Operations are performed by the context passing the extrinsic state as a parameter to a method of the intrinsic state.
- The intrinsic state is independent of its context.

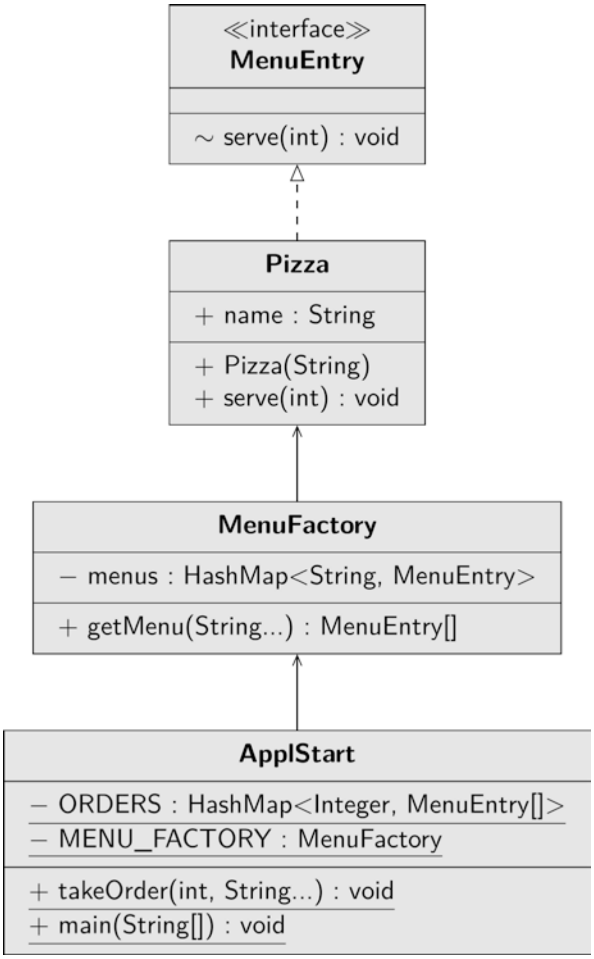
≪interface≫
MenuEntry
~ serve(int) : void
Pizza
+ name : String
+ Pizza(String)
+ serve(int) : void
MenuFactory
– menus : HashMap<String, MenuEntry>
+ getMenu(String...) : MenuEntry[]
ApplStart
– ORDERS : HashMap<Integer, MenuEntry[]>
– MENU_FACTORY : MenuFactory
+ takeOrder(int, String...) : void
+ main(String[]) : void

Fig. 13.1 UML diagram of the Flyweight Pattern. (Example project PizzaFlyweight)

13.7 Description of Purpose

The Gang of Four describes the purpose of the pattern "Flyweight" as follows:

> Use objects of smallest granularity together to be able to use large amounts of them efficiently.