Strategy 10

The Strategy Pattern is a behavioral pattern. You will always fall back on it when you can solve a task with different strategies; the term strategy in this context is a synonym for “algorithm” or for “behavior”. For example, you have a vacation photo and you want to save it in either jpg format or bmp format. You want to decide at runtime in which format to save the image. The Strategy Pattern solves the problem for you that you can implement one and the same task with different algorithms – to save a picture in different formats. You can easily add new algorithms – strategies.

10.1 A First Approach

To get into the subject, you will sort an array. There are very many sorting algorithms that have their advantages in different areas. In this chapter, I will introduce three algorithms: the SelectionSort, the MergeSort, and the QuickSort. Let’s start with a very naive approach. You have a class in which all the algorithms are defined in different methods. You specify at runtime which algorithm you want to call. Depending on the input, the relevant method is called. You can find the source code in the example project BadApproach. I didn’t implement the sorting algorithms themselves; what matters here is the structure of the application, not the implementation of sorting algorithms.

```
public class BadApproach {
    // … abridged
    BadApproach() {
        this.choose();
    }
```

O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_10

```
        void choose() {
            var question = "How should the data be sorted?";
            Object return = JOptionPane.showInputDialog(null,
                                question, "selection sort");
            var response = (String) return;
            switch (response) {
                case "selection sort" ->
                                    sortWithSelectionSort();
                case "merge sort" -> sortWithMergeSort();
                case "quick sort" -> sortWithQuickSort();
                default ->
                    System.out.println("Unknown selection");
            }
        }

        private void sortWithSelectionSort() {
            // ... the Selection Sort Algorithm
        }
        private void sortWithMergeSort() {
            // ... the Merge Sort Algorithm
        }
        private void sortWithQuickSort() {
            // ... the Quick Sort Algorithm
        }
    }
```

Note that in the switch statement of the `choose()` method, I use the "arrow syntax" from the switch expressions we learned about in Sect. 5.7. This eliminates the need for a `break` in the switch statement. The code becomes shorter and easier to read. You can modify this approach by using an if statement instead of the switch statement. However, you still can't avoid making case distinctions. So, it's very messy to implement new algorithms. Also, you have all the algorithms defined in one class. Therefore, the already large class contains a lot of code that you may never use. You realize that the class source code is very inflexible; it is every maintenance programmer's nightmare. So, let's look at the Strategy Pattern, which fixes these drawbacks.

10.2 Strategy in Action – Sorting Algorithms

The principle of the Strategy Pattern is that you can encapsulate algorithms and make them interchangeable. How could this be done? Define three classes that define the three sorting algorithms. If these three classes are of the same data type, the context can interchange them as desired. Figure 10.1 shows the class diagram of the Sort sample project we will discuss in this section.

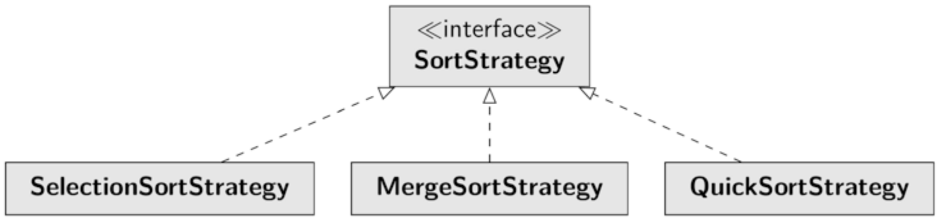

Fig. 10.1 Class diagram of the Sort sample project

10.2.1 The Common Interface

I have provided you with three classes with the sorting algorithms mentioned in the Sort sample project. They all implement the interface `SortStrategy`. The interface prescribes the method `sort()`. The int array to be sorted is passed to this method.

```
public interface SortStrategy {
    void sort(int[] numbers);
}
```

Let us now take a brief look at the logic of the sorting algorithms.

10.2.2 The Selection Sort

The Selection Sort searches an array element by element and finds the smallest value. This is written to the beginning of the still unsorted rest array and then this is sorted.

As an example, consider the sequence of numbers below:

In the first step, the zeroth position, i.e., the value 17, is assumed to be the smallest element. This zeroth position is compared with every other following position. In doing so, you notice that there is another position that contains the smallest value in the array, namely the 3 at the fourth position. The two positions swap their value. So, in the second step, you have the following array:

The smallest of all values is now at position zero. You continue at position 1. The value 45 is compared with all the following values, and you notice that the 17 at the fourth position is the smallest value. So, you swap positions 1 and 4. Now the array looks like this:

Now compare the second position, the value 21, with all the following positions and swap the 20 for the 21 and so on. You continue until the array is sorted altogether.

The code of the `SelectionSortStrategy` class encapsulates this algorithm:

```
public class SelectionSortStrategy
                            implements SortStrategy {
    @Override
```

```
        public void sort(int[] numbers) {
            for (var i = 0; i < numbers.length - 1; i++)
                for (var j = i + 1; j < numbers.length; j++)
                    if (numbers[i] > numbers[j]) {
                        var temp = numbers[i];
                        numbers[i] = numbers[j];
                        numbers[j] = temp;
                    }
        }
    }
```

Note that this example implementation always brings every smaller number to the front, not just the smallest. So, there are significantly more swap operations due than really necessary. There is certainly something that can be optimized. When searching for the smallest number, you can also first go through the entire field before actually swapping. This makes it possible to speed up this procedure by more than a factor of 3. I have already included the necessary adjustments (commented out) in the context. You will see that it is really only three lines: Re-cloning the number field, assigning the strategy, and executing it. You are now left with implementing the `Selection2SortStrategy` class based on a copy of the `SelectionSortStrategy` class. You will also find this copy already as a finished class – but still unchanged from the original. Have fun with this task.

10.2.3 The Merge Sort

The Merge Sort works on the principle of "divide and conquer". The array to be sorted is divided into two parts. Let's take the following unsorted array:

If you divide this array, you have two parts, the values 17, 45, 21, 99 on the left and 2, 20, 15, 12 on the right. If you divide these two halves again, you have the following four parts:

Obviously, it doesn't make sense to keep dividing the individual lists, so now sort them individually:

Now the sorted sublists are merged in pairs, making sure that the new lists are sorted correctly:

And these lists are now sorted and put together again to the final result.

The algorithm for this can be found in the `MergeSortStrategy` class.

10.2.4 The Quick Sort

The Quick Sort also works on the divide and conquer principle. Next to us the unsorted array from above:

This array is again divided into two parts. You calculate a value, a pivot element, so that about half of the values are smaller and the other half are larger than this value. The smaller values are written to the left list, and the larger values are written to the right list. For demonstration purposes, I'll take the value 40 as the pivot value here, so you have two lists:

"Purely by chance", the right-hand list contains only two values – so there's no point in dividing it again; sort it and you're done. However, consider the list on the left. You could take 18 as the pivot value. Put all values less than 18 on the left, and the others on the right. So you have:

The right list contains "purely by chance" again only two elements, which you sort. You may then merge them with the already sorted values:

You divide the "left list" again according to a given pivot element and so on.

Both Merge Sort and Quick Sort work recursively. Quick Sort is fast if you manage to compute a pivot value at each step such that there's roughly an equal amount of numbers in each of the left and right lists. In my implementation of the `QuickSortStrategy` class, I simply pick the first element of the list. There is a lot of literature about sorting algorithms; so much that I don't want to give a preference here. My descriptions here are only very superficial, but also not really relevant to the topic of design patterns per se.

10.2.5 The Context

All sorting algorithms are of type `SortStrategy`. So you can rely on them to define the `sort()` method that triggers the sorting process. You declare a variable within the context that holds a reference to the desired strategy. An assignment then specifies the strategy to use. When you want to sort the array, you simply call the strategy's sort method. A snippet from the `Context` test class:

```
public static void testeLaufzeit() {
    var field size = 100000;
    var value range = 1000000;
    SortStrategy sortStrategy;
    var numbers_1 = createArray(fieldsize, range);
    var numbers_2 = numbers_1.clone();
    var numbers_3 = numbers_1.clone();
    System.out.println("Three arrays containing the " +
                "same unsorted numbers were created");
    sortStrategy = new SelectionSortStrategy();
```

```
        executeStrategy(sortStrategy, number_1);
        sortStrategy = new MergeSortStrategy();
        executeStrategy(sortStrategy, numbers_2);
        sortStrategy = new QuickSortStrategy();
        executeStrategy(sortStrategy, numbers_3);
    }

    private static void executeStrategy(SortStrategy s, int[] z) {
        Instant istart;
        Instant iend;
        Long idifference;
        System.out.println("Start " + s);
        istart = Instant.now();
        s.sort(z);
        iend = Instant.now();
        Duration elapsed = Duration.between(istart, iend);
        idifference = elapsed.toMillis();
        System.out.println("Duration " + s + ": " +
                           idifference + " milliseconds");
    }
```

In Sect. 9.1 we took a first look at the Date and Time API and looked at the LocalDate. Remember the note about it not containing any information about times? In the code above, you can see the counterpart to that: The `Instant` class. This class contains only time information and appropriate methods for calculations accordingly. You can see a simple application here: A look at the starting time of the sort. Then a look at the clock when everything has run, followed by the calculation of the duration (as a separate class `Duration` for the length of time intervals) and its output in milliseconds.

By the way, this benchmarking approach is very naive and only meaningful at first glance. It is enough for us here to make a rough comparison of the different algorithms. But the startup times of the Java Virtual Machine (and also under different environmental conditions) are not taken into account here. If you have serious performance measurements of Java programs in mind, you should look into the "Java Microbenchmarking Harness" (JMH) added in Java 12 with the JEP 230. However, a description of this small collection of tools goes too far here and misses the actual topic of the book.

10.2.6 Evaluation of the Approach and Possible Variations

What do you think of this approach? Surely one advantage jumps right out at you: The context is much clearer! But there are two more advantages:

The project can be extended to include any number of sorting algorithms. If you want to implement the Heap Sort, define a new class that implements the `SortStrategy` interface. The context can use your `HeapSortStrategy` immediately. For the Selection Sort, I suggested an enhancement option. You can also implement that as a variant of a SortStrategy and compare it directly against the Selection Sort I created.

In principle, the context does not even need to know how his problem is solved. But why only "in principle"? You offer a variety of algorithms that all solve the same problem. In order for a programmer to know when to choose which algorithm, you need to document very precisely under which conditions which algorithm is appropriate. In doing so, you will not be able to avoid going into implementation details in the documentation.

In the implementation above, you as the user have specified which strategy you want to use. However, it would also be conceivable that the program asks the strategies how well they can solve a certain task under certain conditions. An algorithm that sorts data in main memory very efficiently might fail if the data is stored in a file and cannot be fully loaded into main memory. So the context could ask the strategy classes, "How well do you solve the task under the condition that the data is stored on disk?" Each strategy class would return a score, say in the range zero to 100, and the context can then select the strategy that returns the highest score.

Another advantage of the Strategy Pattern is the reusability of the algorithms. Imagine you are programming an office suite. Within the spreadsheet, you need to be able to sort data. But you also want to sort data in word processing. After all, the algorithms are the same, so you can reuse them and, in the case of the spreadsheet, limit yourself to the functionality that is typical of the spreadsheet. For word processing, you program only the parts that are typical for word processing. You can reuse the sort algorithm.

In the example above, I have assumed that the data to be sorted is passed to the `sort()` method. This specification certainly makes sense in this constellation. However, think of a family of algorithms where one implementation takes a lot of parameters, but another implementation takes significantly fewer. Because of the common interface, the context would still have to pass all arguments – an avoidable overhead.

10.3 The Strategy Pattern in Practice

Where can the Strategy Pattern be found in the Java class library? In the area of GUI programming, you will find the Strategy Pattern in several places. For example, there are LayoutManagers or the "Look and Feel"; containers are provided with a certain strategy by default, which can, however, be exchanged by the user.

Instances of the `JPanel` class are needed to combine components of a GUI. By default, the FlowLayout is used for this. You can understand this, for example, by executing the following code once in the Java shell:

```
javax.swing.JPanel pnlLayoutTest =
                                new javax.swing.Jpanel();
java.awt.LayoutManager layout =
                            pnlLayoutTest.getLayout();
System.out.println(layout);
```

The console will then output `java.awt.FlowLayout[hgap = 5,vgap = 5,align = center]`. You can replace the strategy `FlowLayout` by passing a new strategy with `pnlLayoutTest.setLayout(layoutManager)`. The variable `layoutManager` must be of type `LayoutManager`.

The `LayoutManger` interface prescribes five methods. The methods `addLayout-Component()` and `removeLayoutComponent()` are needed if you want to address the components of the Jpanel with a string, otherwise the method bodies may remain empty. The `preferredLayoutSize()` method calculates and returns the optimal size of the Jpanel, while `minimumLayoutSize()` calculates and returns the minimum size of the Jpanel. The positioning of the components is done by the `layoutContainer()` method. In this method, horizontal and vertical position as well as width and height are assigned to each component with `setBounds()`.

In the source code for this book, you will find the example project LayoutStrategy, in which I implemented the LayoutManager – the Strategy – `TwoColumnLayout`. When you place components on the Jpanel, they are displayed in two columns. If you drag the GUI to the width, the components in the right column are enlarged accordingly. Figure 10.2 shows you what a GUI you create with TwoColumnLayout might look like.

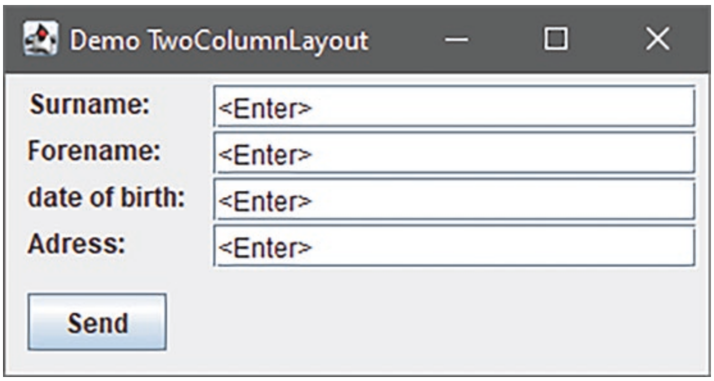

Fig. 10.2 Positioning components with the two-column layout

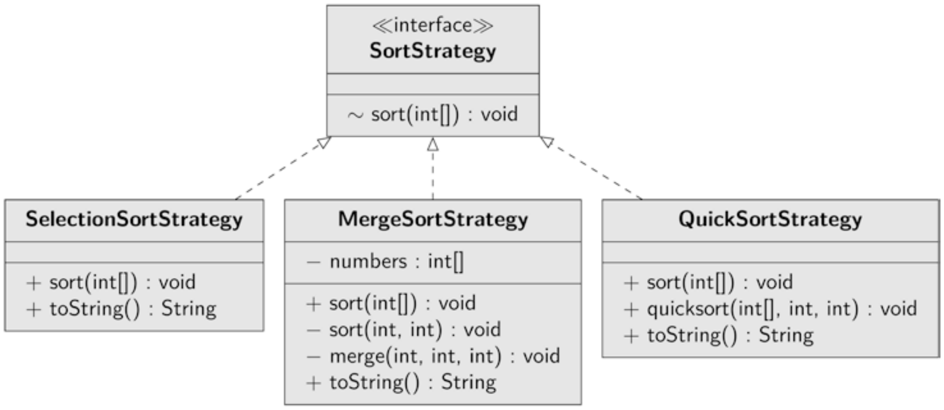

Fig. 10.3 UML diagram of the Strategy Pattern. (Example project Sort)

Please note that the implementation of the TwoColumnLayout is extremely simple – it is certainly far from perfect. I just want to demonstrate with this example how you can design and use your own strategy, your own LayoutManager.

10.4 Strategy – The UML Diagram

From the Sort example project, you can see the UML diagram in Fig. 10.3.

10.5 Distinction from Other Designs

You know the Command Pattern and the State Pattern, and in this chapter, you learned about the Strategy Pattern. All three patterns encapsulate behavior. In fact, the class diagrams of State and Strategy look pretty much the same. You can keep track of them by visualizing the different goals:

The Command Pattern is where you encapsulate commands. You need it to pass different action listeners to all buttons on your GUI. One command opens a file, another saves it. The Command Pattern does not describe the behavior of the calling object.

The Strategy Pattern encapsulates algorithms. Behavior of an object is implemented in different ways: There are many algorithms to compress files, to encrypt or to sort data. But you will only ever need one algorithm. So, out of the multitude of strategies, choose the one that solves your task most efficiently.

You use the state pattern to encapsulate state characteristics. The behavior of an object is based on its state. When an object has a certain state, it exhibits different behavior than when it is in a different state. Think of the open gate – the gate can be closed, but not locked. Only a closed gate can be locked.

10.6 Summary

Go through the chapter again in key words:

- The Strategy Pattern is used when you have multiple algorithms for a task.
- Each algorithm is defined in its own class.
- All Strategy classes implement the same interface.
- The context is programmed against the interface.
- At runtime, an algorithm is passed to the context.
- The context calls the solution strategy, the algorithm, without knowing which algorithm is behind it.
- The algorithm can be reused.

10.7 Description of Purpose

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

> Define a family of algorithms, encapsulate each one, and make them interchangeable. The strategy pattern allows the algorithm to vary independently of clients using it.