9 Command

In this chapter, I want to show you a behavior pattern that detaches a command call from command execution. What is meant by this? Imagine the head of your company saying in a department head meeting, "I need the latest statistics!" Malice is guaranteed to strike in the form of your own department head raising his finger and saying, "I'll take care of it!" Predictably, he will also assign you to do it, saying, "You do it, you can do it!" What happened there? The boss requests an action, you execute it, and in between there is an entity (your department head) that passes on the task. In other words, command invocation and execution are detached from each other and only loosely coupled via a command object – and that is exactly the goal of the Command Pattern.

9.1 Encapsulating Commands in Classes

But first, let's leave the scene in the office and think about how to book a vacation trip. You go to the travel agency and state your travel wishes. The person in the travel agency types on his computer for a while and you can go on holiday with peace of mind. But in fact, it's not the travel agent who runs the trip, it's a tour operator who takes care of the flight and, at the destination, the hotel. Let's take a closer look at this, but in addition we will also deal with the Date and Time API added in Java 8, which replaces the old java.util.Date.

9.1.1 Version 1 – Basic Version

In the first step, which has nothing to do with the Command Pattern, I will program a travel agency and a tour operator. The travel agency defines the method `bookTrip()`. It is passed the destination, departure day and return day as arguments. The data is packed into

O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_9

an object of the class Trip and passed to the tour operator, which then carries out the trip in real terms, i.e., executes the trip command. You can find the following example in the sample project Travel_1.

```
public class Trip {
    final String destination;
    final String from;
    final String to;
    final DateTimeFormatter dtFormatter =
           DateTimeFormatter.ofPattern("MM/dd/yyyy");
    Trip(String destination, LocalDate from, LocalDate to) {
        this.target = target;
        this.from = from.format(dtFormatter);
        this.to = dtFormatter.format(to);
    }
    @Override
    public String toString() {
        return "Travel to " + destination + " from "
                                   + from + " to " + to;
    }
}
```

Note the use of the date functions: First, the constructor of the Trip class receives the start and end dates of the trip as values of type `LocalDate`. For this, the Trip class must use the `import java.time.LocalDate.`

This LocalDate belongs to the Date and Time API, which was implemented in Java 8 with the Java Specification Request JSR-310. The reason for replacing the `java.util.Date` used until then were its weaknesses: Lack of type safety, lack of extensibility, unclear responsibilities (Date with time specification built in but without time zone) and some others.

In contrast, the `java.time` package and its four subpackages `chrono`, `format`, `temporal`, and `zone` provide largely unified commands for various date and time types, consistent and clear command definitions, and thread-safe immutable objects.

Take another look at the sample code above:

The constructor is about converting a LocalDate into a string. This is done by means of a DateTimeFormatter, which is provided with the pattern "`MM.dd.yyyy`" for the representation or conversion, which is common for the US. This conversion is not mandatory, but without it the date would be output according to the ISO standard ISO-8601 in the format yyyy-MM-dd. Decide for yourself what suits you better.

By the way, a variable of type LocalDate has no time parts at all. It contains the year, month, and day, as well as all the necessary methods. The counterparts for processing time information will be discussed separately in a later chapter.

The actual conversion into a string is then possible in two ways, both of which I've used to introduce you to them

- Either you use the `format` method of the LocalDate and specify the DateTimeFormatter as parameter
- Or you use the `format` method of the DateTimeFormatter and pass it the LocalDate as parameter

In both cases, however, the appropriately formatted text comes out. We will see how to create a LocalDate in a moment when we create the test class.

The tour operators negotiate contracts with hotels, airlines and local bus companies and run the tour. Each tour operator gets its own company.

```
public class Tour Operator {
    private final String company;
    Tour operator(String company) {
        this.company = company;
    }
    void execute(Trip trip) {
        System.out.println(company + " operates the
            following trip: " + trip);
    }
    @Override
    public String toString() {
        return company;
    }
}
```

An instance of the travel agency is created by passing an object of type TourOperator to its constructor. The method `execute()` is passed the data of the trip as parameters, which are stored as strings.

```
public class TravelAgency {
    private final TourOperator operator;
    TravelAgency(TourOperator operator) {
        this.operator = operator;
    }
   void bookTrip(String destination, LocalDate from, LocalDate to) {
        var trip = new Trip(destination, from, to);
        operator.execute(trip);
    }
}
```

In the main method of the test class, a travel agency and a tour operator are created. Then three trips are booked. Let's take a closer look at this because of the Date and Time API. I have highlighted the relevant code passages in bold.

```
public class Testclass {
    public static void main(String[] args) {
        var operator = new TourOperator("ABC-Travel");
        var travelAgency = new TravelAgency(operator);
        LocalDate from, to;

        // book a trip
        of = LocalDate.of(2023, Month.NOVEMBER, 4);
        to = from.withDayOfMonth(15);
        travelAgency.bookTrip("Washington", from, to);

        // book another trip
        from = toDate("12/30/2023");
        to = from.with(nextOrSame(DayOfWeek.TUESDAY));
        travelAgency.bookTrip("Rome", from, to);

        // and book another trip
        from = toDate("02.10.2023");
        to = from.plusWeeks(2);
        travelAgency.bookTrip("Beijing", from, to);
    }
    private static LocalDate toDate(String date) {
        LocalDate tempDate;
        try {
            tempDate = LocalDate.parse(date,
            ofLocalizedDate(FormatStyle.MEDIUM));
        } catch (DateTimeParseException ex) {
            tempDate = LocalDate.now();
            ex.printStackTrace();
        }
        return tempDate;
    }
}
```

Note that you need to specify some imports to use the Date and Time API. For this example, these are.

```
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.Month;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import static
    java.time.temporal.TemporalAdjusters.nextOrSame;
```

Let's go over the bolded lines in the test class again in detail.

I create the start date of the first trip using `from = LocalDate.of(…)`. Note that no new is used in this. The `of` method creates the object for me (more precisely, a private method `create` called by `of` does that) and returns it. We will learn more about this "factory" approach in the chapters on the Abstract Factory and the Factory Method. As parameters for the call, I use year, month, and day each separately. For the month I use the standard enumeration `Month` with the English month names, in this case `Month.NOVEMBER`. The first trip therefore starts on 11/4/2023.

However, the customer wants to be back on the 15th of the same month, so I simply calculate an appropriate end date using `from.withDayOfMonth(15)`. In doing so, the Date and Time API now creates a new LocalDate with the day changed to the 15th.

For the second and third journey I use the self-written method `toDate`, which determines a corresponding LocalDate from a text. For this I use the `parse` method of the LocalDate and give it – as already explained in the class `Trip` – also a DateTimeFormatter. This can be used in both directions – for parsing as well as for formatting. Since the parser can throw an exception if it can't do anything with the text, there must be a corresponding try-catch block here. For the case of the exception, I return `null`. You should not be tempted here to set the current date, for example. This may cause major problems down the line and is hard to reproduce.

The duration of the second trip is a tricky thing. The client wants to be back on "the following Tuesday". Instead of flipping through the calendar myself now, I can elegantly solve this with the command `with(nextOrSame(DayOfWeek.TUESDAY))`: Adjust the start date to "next or same day of week Tuesday" and return it to me as the new date. For the days of the week, there is also an enumeration that simply saves me from having to ask which day of the week started the index (was Monday now the 0 or the 1?).

The third trip should last exactly two weeks. This can also be easily solved using `plusWeeks`. This command category also includes plusDays, plusMonths and plusYears.

In the Travel class and in the Test class you will find commented out alternative lines of code for the DateTimeFormatter. Just have a look at the comments in the Travel class on your own.

After this excursion into the possibilities of the Date and Time API, now back to the actual topic: What you should note in the analysis is that the travel agency calls methods of the tour operator directly. But that's about to change!

9.1.2 Other Suppliers Appearing

The travel business is booming – the travel agency has yet to open a branch. I have combined both travel agency and branch under the interface provider. The interface prescribes the method `bookTrip()`, which you already know from the last version. The sample project Travel_2 simulates this situation.

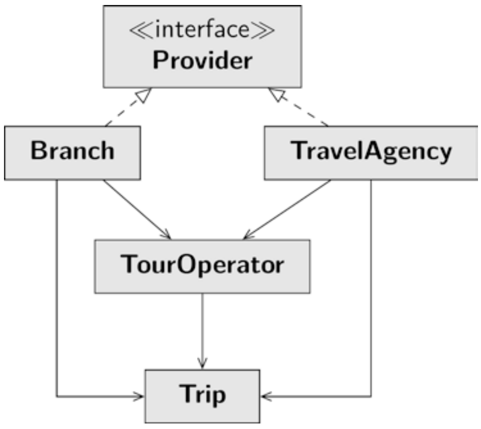

Fig. 9.1 Class diagram of the sample project Travel_2

```
public interface Provider {
    void bookTrip(String destination, LocalDate from,
                                               LocalDate to);
}
```

Otherwise, the code of the classes has not changed. How are the classes used? In the example code of the test class, you will find the following procedure: You create a tour operator. You pass the instance to the constructor of a provider. To book a trip, you call the provider's method `bookTrip()` – nothing has changed since the previous version. Figure 9.1 shows the class diagram of this project version.

However, I used a few more date manipulation options when compiling the travel dates to show you more possibilities of the Date and Time API.

If you analyze the project Travel_2, you notice that the code of the classes `Branch` and `TravelAgency` is almost identical. Redundant code is always an indication of an inappropriate program design. Above all, redundant code is a source of errors that should not be underestimated. It is therefore necessary to eliminate duplicate code.

How would you proceed if you have many equal factors in mathematics, for example, 5 * 3 + 5 * 2 + 5 * 9? You factor out the same factor in front of the bracket: 5 * (3 + 2 + 9). You do exactly the same with the redundant program code at the providers.

9.1.3 Encapsulating a Command

In the sample project Travel_3, you now pull the same program code in front of the parenthesis by introducing the class `TripCommand,` which lies between the suppliers and the tour operator. This class solely defines the command that is executed to book a trip. When it is created, an instance of a tour operator is passed to the constructor and its reference is stored in a data field. The `book()` method will be supplied with the same arguments that were passed to the `bookTrip()` method in the previous project.

```
public class TripCommand {
    private final TourOperator operator;
    public TripCommand(TourOperator operator) {
        this.operator = operator;
    }
    public void book(String destination, LocalDate from,
                                      LocalDate to) {
        Trip trip = new Trip(destination, from, to);
        organizer.conduct(travel);
    }
}
```

The providers – I'm only showing the travel agency here – have now become quite slim. An instance of the class `TripCommand` is passed to their constructor, whose reference is stored in a data field. The method `bookTrip()` now accesses the instance of the class `TripCommand` instead of a travel agent itself.

```
public class TravelAgency implements Provider {
    private final TripCommand tripCommand;
    TravelAgency(TripCommand tripCommand) {
        this.tripCommand = tripCommand;
    }
    @Override
    public void bookTrip(String destination,
                    LocalDate from, LocalDate to) {
        tripCommand.book(destination, from, to);
    }
}
```

The changes that have been added in this program version are shown in Fig. 9.2.

But what are the changes in the usage of the classes? In the main method of the test class, which you can find in the sample project Travel_3, the following procedure is provided: First, two objects of type `TourOperator` are created. These objects are passed to the constructors of two instances of the class `TripCommand`. Finally, the TripCommand instances are passed to the constructors of the providers as arguments. It is irrelevant to the provider which tour operator is working in the background. It only ever calls the `book()` method of its TripCommand instance.

What is gained by this approach? At first, it looks like this solution is quite cumbersome – after all, an additional class is needed. But in fact, several things happen: First, you have eliminated duplicate code as much as possible. This means you can easily add more travel providers. You can also pass the same command to different providers. In addition, the suppliers are now as decoupled from the travel providers as you are from the boss of your company with his statistics.

Swapping out a command into its own class is the core of the Command Pattern.

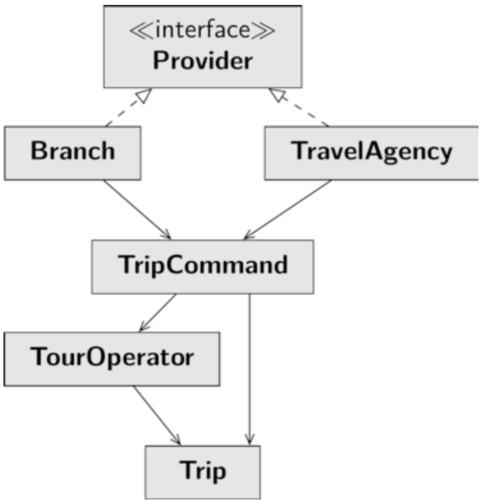

Fig. 9.2 Class diagram of the sample project Travel_3

The Command Pattern recognizes several terms, which I will present more formally to conclude the introduction. There is the *caller* or *invoker*; this is the head of the company requesting a statistic, but this is also a provider selling a trip. The invoker makes use of a *command*; that's an instance of the class `TripCommand`, for example; but a command object is also your department head, who passes on the boss's requests to you. And finally, there are those who execute the command, these are the *receivers* – for example, the operators that actually carry out the trips.

9.2 Command in the Class Library

In this chapter I would like to give you two examples that show the use of the Command Pattern in the Java class library.

9.2.1 Example 1: Concurrency

If you want to implement concurrency in a program, create an object of type `Runnable`. The `Runnable` interface requires you to override the `run()` method. This method will contain the code that you want to execute concurrently. You can implement the execution of the command in two ways: Either the `Runnable` object executes the code itself, or it delegates the execution to another object. In our example, the travel agency object has delegated the command to execute a trip to the tour operator. However, it is not mandatory to have the recipient execute the command. It would also be conceivable to have the entire business logic, or at least a large part of it, executed by the command.

This would look like this, for example:

```
Runnable runnableDemo = new Runnable() {
    @Override
    public void run() {
        // ... concurrent code
    }
};
```

You then create an instance of the Thread class and pass an instance of that class to the constructor:

```
Thread threadDemo = new Thread(runnableDemo);
```

And finally, you call the `start()` method on the Thread instance, which results in the code of a Runnable type class being executed. As an Invoker, the Thread class is as loosely coupled to the Receiver as travel providers are to tour operators.

```
threadDemo.start();
```

While it is obvious, please note that the `Runnable` does not define any behavior of the `Thread` class. The interface `Runnable` and the prescribed method `run()` are only created so that an object can execute a command of a different object, of whose definition it has no knowledge. If I may formulate the principle casually: The socket supplies power at a defined interface – whether a lamp, a computer or a washing machine is operated with it, it is quite indifferent. None of the devices mentioned defines any behavior of the socket.

9.2.2 Example 2: Event Handling

You are programming a GUI. On the GUI there should be a button that can be activated by the user. You pass an `ActionListener` to the button that contains the code to be executed when the button is activated. An object of type `ActionListener` must override the `actionPerformed()` method. When the button is activated, this method is called. To demonstrate the procedure, first create the button:

```
JButton btnExample = new Jbutton(“Non-Sense“);
```

Then follows an anonymous class for the code to be executed in the action:

```
ActionListener actionListener = new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // ... something
    }
};
```

And finally, pass the Command object, the ActionListener, to this button:

```
btnExample.addActionListener(actionListener);
```

The whole thing also works much shorter and clearer with the lambda expressions added in Java 8, but does the same thing:

```
JButton btnExample = new JButton("Non-Sense");
btnExample.addActionListener((ActionEvent e) -> {
    // … something
}
```

Did you notice that this code is structurally quite similar to concurrency example from before? The logic is indeed the same: you need an object of a certain type to whose previously defined interface (`run()` in one case, `actionPerformed()` in the other) another object can send messages. Once the invoker has called the defined method of the command object, the work is done for it. The programmer has defined what should happen when the Invoker has become active. But the Invoker has no knowledge of how it happens – and that's exactly the knowledge it doesn't need to have, and shouldn't have.

9.3 Reusing Command Objects

The classes `JButton` and `JMenuItem` have the same superclass `AbstractButton`. So, you can reuse the ActionListener from the previous section and add it to the MenuItem in the main menu of your interface at the same time:

```
JMenuItem mtmExample = new JMenuItem("Non-Sense");
mtmExample.addActionListener(actionListener);
```

In the next paragraph, you take it a step further.

9.3.1 The "Action" Interface

The `Action` interface enhances the `ActionListener` interface. You can pass both `Action` and `ActionListener` to a caller as a command. The binding between the caller and the command object is much tighter for an object of type `Action.` Among other things, the `Action` interface provides the `setEnabled()` method, which determines whether the Action object is enabled or disabled. You can also use an action object to specify the text and icon of the caller.

Remember what was said about the Template Method Pattern? In that context, I introduced you to the `AbstractListModel` class as a template-like implementation of the `ListModel.` The `AbstractListModel` class overrides all methods for which a default behavior can be reasonably implemented. The methods that are context-dependent, for example to describe the database, are delegated to subclasses. Exactly the same thing takes place here. The interface Action specifies the algorithm and the class `AbstractAction` partially implements it. The logic of how to change the state is already implemented; if you change the enabled state of the Action object, the state of the Invoker is also changed. This behavior should make sense in most cases; however, you can override it. In any case, the actionPerformed`()` method defies standard behavior, and you must override it.

9.3.2 Use of the "Action" Interface

I would like to demonstrate the reusability of commands with the following example. Take a look at the Action sample project. In it, two buttons and two menu items are created. The same action object – a command object – is passed to each button and menu item, i.e. to two different callers.

First, you create a button and a menu item:

```
var mnDisable = new JMenuItem();
var btnDisable = new JButton();
```

These are added to a GUI. A Command object, an object of type `Action`, is also created. The constructor of the class takes a string containing the display text of the caller. The only method that needs to be overridden is the `actionPerformed()` method. When triggered, it disables the object. The default implementation of the `AbstractAction` class is to disable the invokers as well.

```
AbstractAction actDisable = new AbstractAction("Disable") {
    @Override
    public void actionPerformed(ActionEvent evt) {
        this.setEnabled(false);
    }
};
```

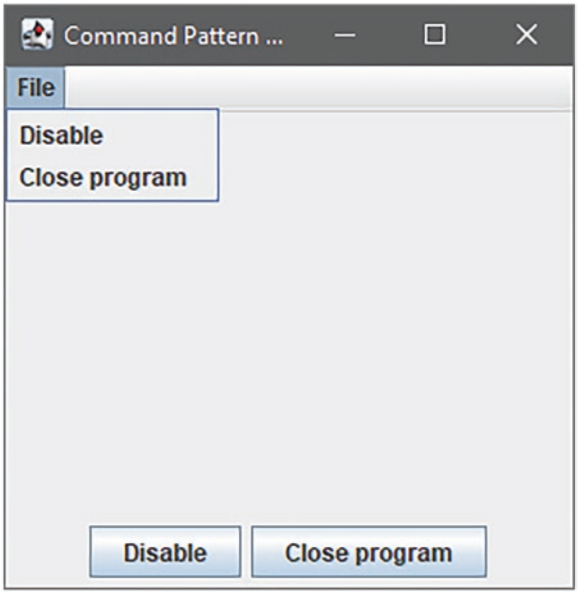

Fig. 9.3 Same labels and same actions

Both button and menu item are supplied with the same command object.

```
btnDisable.setAction(actDisable);
mnDisable.setAction(actDisable);
```

The text you passed to the constructor is used as the display text by both components. If you click either the button or the menu item, both button and menu item are disabled. Figure 9.3 shows you what the GUI looks like.

9.4 Undo and Redo of Commands

In this section I describe one last aspect of the Command Pattern. Commands can be undone and redone. You will find two examples of this. The first example is fairly simple. The second is a bit more extensive; I will only present the source code in broad strokes – I want to give you something to tinker with for long winter evenings with this example.

9.4.1 A Simple Example

The example project RadioCommand shows how a radio (the older ones among us may remember) can be operated with the Command Pattern. Besides the frequency adjustment, which I'll leave out here, there are four very simple commands: turn on, turn off, turn up and turn down. All commands implement the interface Command, which specifies two methods. The `execute()` method executes the command, the `undo()` method returns a command that must be executed to undo its own execute method.

```
public interface Command {
    void execute();
    Command undo();
}
```

I will explain the command for switching on the radio here as a representative of all other commands. An object of the type radio is passed to the constructor of the command. The command is executed on this object. So if the `execute()` method is called, the command turns the radio on. If the `undo()` method is called, the radio is turned off, so the turn on command returns a turn off command.

```
public class OnCommand implements Command {
    private final Radio radio;
    public OnCommand(Radio radio) {
        this.radio = radio;
    }
    @Override
    public void execute() {
        System.out.println("The radio will turn on.");
        radio.turnOn();
    }
    @Override
    public Command undo() {
        return new OffCommand(radio);
    }
}
```

The radio must now provide the appropriate methods so that the commands can be executed.

```
public class Radio {
    private int volume = 0;
    public void turnOn() {
        volume = 1;
        System.out.println(">Radio: I'm on now.");
    }
    public void turnOff() {
        volume = 0;
        System.out.println(">Radio: I'm off now.");
    }
    public void decreaseVolume() {
        if (volume >= 1) {
            volume--;
            System.out.println("›Radio: I'm playing" +
                                        "softer: " + volume);
    }
}
    public void increaseVolume() {
        volume++;
        System.out.println("›Radio: I'm playing louder: "
                                                + volume);
    }
}
```

The radio, in Command Pattern terminology, is the receiver that executes the commands. Invoker is a class Logbook to which the context sends the command call and the undo. The Invoker logs all command invocations. This allows the context to undo the last command in each case.

```
public class Logbook {
    private final List<Command> history =
                                    new ArrayList<>();
    public void execute(Command command) {
        history.add(command);
        command.execute();
    }
    public void undo() {
        int size = history.size();
        if (size > 0) {
            Command command = history.remove(size - 1);
            Command undoCommand = command.undo();
            System.out.println("\tundo: " + undoCommand);
            undoCommand.execute();
        }
    }
}
```

The test class creates an object of the type Radio. In addition, the commands are created and parameterized with the radio. The commands are then passed to the logbook and executed.

```
Radio radio = new Radio();
var onCommand = new onCommand(radio);
var offCommand = new offCommand(radio);
var softerCommand = new SofterCommand(radio);
var louderCommand = new LouderCommand(radio);
var logbook = new Logbook();
logbook.execute(onCommand);
// … abridged
log.undo();
```

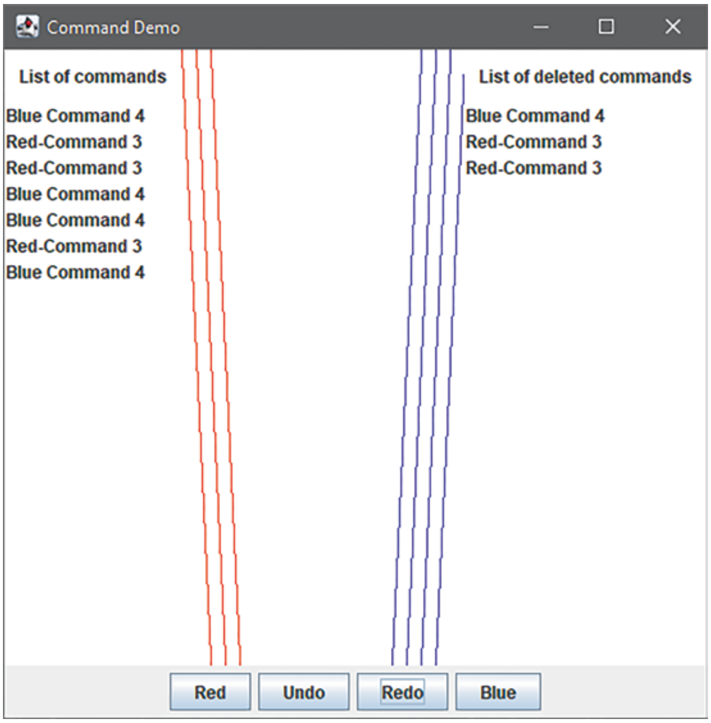

Fig. 9.4 GUI of the "Swing" project

Let's tackle a bigger swing example in the next section!

9.4.2 A More Extensive Example

The Swing sample project demonstrates the undo and redo functionality. On the GUI of the application you will find four buttons. One button is labeled Red and one is labeled Blue. When you click one of these buttons, red or blue lines are painted on the canvas. The last command executed is appended to the end of the list on the left side. If you click the undo button, the last executed command is undone – deleted, that is. However, it is not actually deleted, but written to the list of commands to be restored on the right. If you click redo, the command is restored, which is equivalent to calling it again. If you select a command in the list on the left and click undo, the selected command is deleted. Likewise, if you highlight a command from the right-hand list and click redo, that exact command is restored. In Fig. 9.4, you can see what the GUI will look like. I have drawn a few lines and deleted them again.

The idea for this project goes back to James W. Cooper (Cooper, James William (2000): Java design patterns. A tutorial. Reading, Mass.: Addison-Wesley. ISBN 0201485397.).

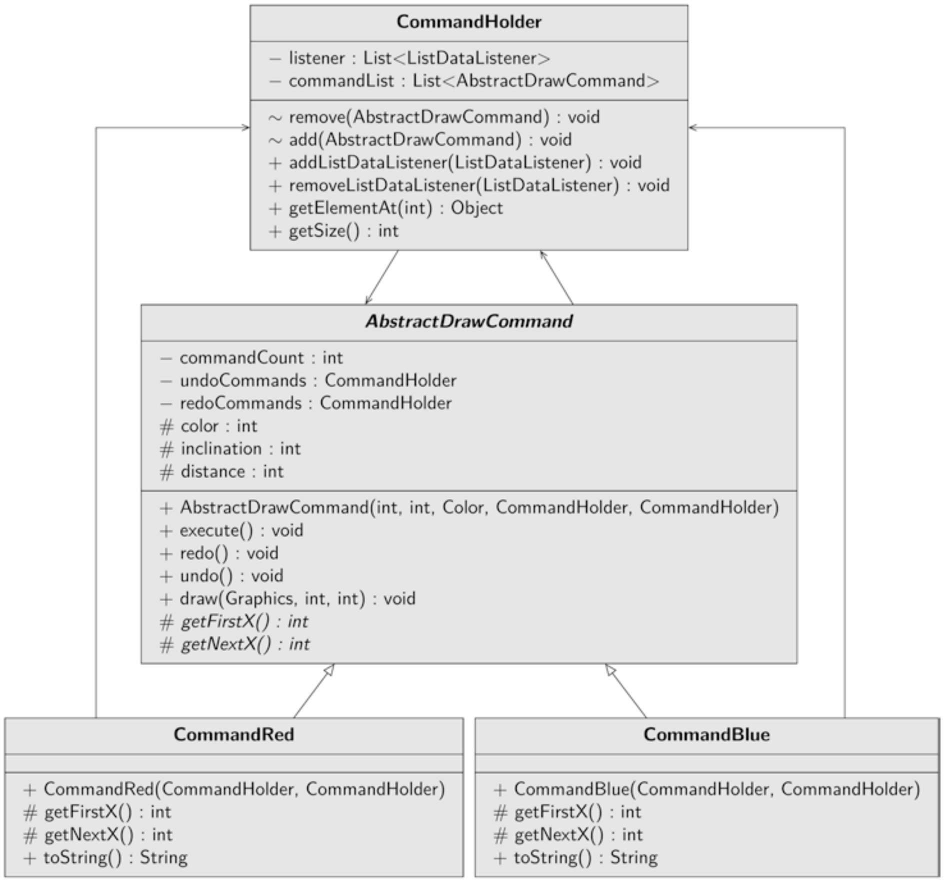
CommandHolder
− listener : List<ListDataListener>
− commandList : List<AbstractDrawCommand>
∼ remove(AbstractDrawCommand) : void
∼ add(AbstractDrawCommand) : void
+ addListDataListener(ListDataListener) : void
+ removeListDataListener(ListDataListener) : void
+ getElementAt(int) : Object
+ getSize() : int
AbstractDrawCommand
− commandCount : int
− undoCommands : CommandHolder
− redoCommands : CommandHolder
# color : int
# inclination : int
# distance : int
+ AbstractDrawCommand(int, int, Color, CommandHolder, CommandHolder)
+ execute() : void
+ redo() : void
+ undo() : void
+ draw(Graphics, int, int) : void
# getFirstX() : int
# getNextX() : int
CommandRed
+ CommandRed(CommandHolder, CommandHolder)
# getFirstX() : int
# getNextX() : int
+ toString() : String
CommandBlue
+ CommandBlue(CommandHolder, CommandHolder)
# getFirstX() : int
# getNextX() : int
+ toString() : String

Fig. 9.5 UML diagram of the command pattern (example project Swing)

9.4.3 Discussion of the Source Code

I would like to leave the source text essentially to you for your own research. I will therefore only present the essential key points to you. The class diagram of the project can then be found in Fig. 9.5.

9.4.3.1 The Classes Involved

There are two commands, the `CommandBlue` and the `CommandRed`. The Commands have partially the same code, which is outsourced to an abstract superclass `AbstractDrawCommand`. And finally, you will find the `CommandHolder` class, where the commands are stored in the order they are processed. The invoker is the `GUI`.

9.4.3.2 Task of the GUI

The GUI class creates the components: a JPanel as a canvas on which the lines are painted, two JList instances that display the executed and the deleted commands. And finally, on the GUI you will find four buttons, instances of the JButton class. The CommandHolder implements the ListModel interface, so it can be used as a model for the lists. The GUI class creates two CommandHolder instances, one for the undo list and one for the redo list, and passes both to the constructors of the CommandRed and CommandBlue commands.

```
private final CommandHolder undoCommands =
                                   new CommandHolder();
private final CommandHolder redoCommands =
                                   new CommandHolder();
private AbstractDrawCommand cmdRed =
            new CommandRed(undoCommands, redoCommands);
private AbstractDrawCommand cmdBlue =
            new CommandBlue(undoCommands, redoCommands);
```

When the btnRed button is activated, the execute() method of the CommandRed class instance is called. This draws a red line on the canvas – the canvas object – and causes it to redraw itself.

```
btnRed.addActionListener((ActionEvent evt) -> {
    cmdRed.execute();
    canvas.repaint();
});
```

The listener for the btnBlue button looks likewise. Before I go into the listeners for undo and redo, I would like to take a closer look at the command classes.

9.4.3.3 Operation of the Command Classes

The CommandRed and CommandBlue classes define two different commands: One command draws a red line that slopes toward one direction, and the other command draws a blue line that slopes toward the other direction. In doing so, each command must perform two tasks: It must first set the parameters for the lines, and it must also be able to draw its lines on the canvas as many times as it is called. The parameters of the line, i.e. color, distance and slope, are stored in the subclasses. The superclass holds in a non-static data field the information about how many times the command has been called.

If the execute method is now called, the counter is first incremented. Then a reference to the command is passed to the CommandHolder, which stores all commands in the order they are called. When the lines are to be drawn, the canvas passes a reference to its Graphics instance, the pen, to the command and causes it to draw as many lines as it has been called.

9.4.3.4 Undo and Redo

To the right and left of the canvas you see two lists in which the commands are stored one after the other. You can select any command object and call the undo command on it. The logic of undo is defined in the command itself. The command decrements its counter, clears itself from the undo list, and writes itself to the redo list – admittedly, unlike the tasks of real projects, this logic is very simple. When you initiate a redo, the command is deleted from the redo list and then executed again, i.e. the execute method is called.

9.4.4 Undo and Redo Considered in Principle

In both examples, you have perfect undo and redo functionality. The undo reverses the execution and the redo corresponds exactly to the original execute command. In reality, you will also encounter situations where an exact reversal is not possible at all. For example, if you bought chocolate in a supermarket and then ate it, you would certainly not be able to return it. However, it is also conceivable to encounter situations in which an order can only be partially reversed. Perhaps you bought a book. The retailer takes the book back, but does not refund your money, but gives you a voucher for it. This only partially restores the condition as it existed before. You are rid of the book, but you do not have your money back.

9.5 Command – The UML Diagram

The UML diagram for the command pattern comes from the Swing sample project. You can find it in Fig. 9.5.

9.6 Summary

Before I summarize the key points of the pattern, I want to draw your attention to one thing: its similarity to the Chain of Responsibility pattern. Both patterns decouple command invocation and command execution. In the Chain of Responsibility, the caller sends its command to a chain of possible recipients; however, it has no way of knowing whether or how the command will be processed. In the Command Pattern, there is a clearly defined command executor, the Receiver. It is loosely coupled with the invoker.

Go through the chapter again in key words:

- Commands are outsourced to their own classes: mnemonic: “factor out” the command in front of the brackets.
- The command pattern decouples invoker and receiver.
- A command class can execute the call itself or delegate it to an executing unit.

- Each command class can be parameterized with a different executing unit: for example, different tour operators.
- The Invoker sends messages to the Command object only; it does not need to know the Receiver.
- Invoker and command object exchange their messages via defined interfaces.
- A command object can be replaced at runtime.
- A command can be undone.
- An undone command can be restored.
- The Command Pattern allows you to keep a history.

9.7 Description of Purpose

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

> Encapsulate a command as an object. This allows you to parameterize clients with different requests, queue objects, keep a log, and undo operations.