Factory Method

16

In Chap. 15 you got to know the abstract factory. If you now work on the factory method pattern, you can transfer many structures and arguments from there. While both patterns are about creating objects, they differ in their goals; the abstract factory creates unified product families, while the factory method creates only a single product. They also differ in structure – the Abstract Factory is based on object composition, the Factory Method is based on inheritance, as you'll see in a moment.

16.1 A First Example

For the first, introductory example, meals are to be prepared. Since everything is kept very simple at the moment, there are only the classes `Doner` and `Pizza`. Both implement the interface `Meal`. Fittingly, there is a `Pizzeria` and a `Takeaway`. Both inherit from the abstract class `Restaurant`. The abstract class defines the `order()` method, which dictates that the guest should first be asked for their request, then the requested meal is prepared, and finally it is served. This method is called by the guest when he wants to order a meal. The method `prepareMeal()` is a factory method. Its job is to create a specific product. The actual execution is delegated to subclasses, i.e. `Takeaway` and `Pizzeria`. Note that while the `takeOrder()` method is also defined by the subclasses, this is irrelevant to the discussion of the pattern; I intended this coding to ask the guest for their pizza request at the pizzeria and their doner request at the takeaway. You can find the complete code in the sample project Meal.

O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_16

```
public abstract class Restaurant {
    protected abstract String takeOrder();

    protected abstract Meal prepareMeal();


    private void serveMeal(Meal meal) {
        System.out.println("Meal is here! It's " + meal);
    }

    public final Meal order() {
        var order = takeOrder();
        var meal = prepareMeal();
        serveMeal(meal);
        return meal;
    }
}
```

Takeaway and pizzeria create either a doner or a pizza – each varied according to customer preferences. All meals implement the same interface. The guest – in the example the test class – creates the instance of a subclass of the class Restaurant and calls the method `order()` on it.

```
public class Test {
    public static void main(String[] args) {
        Restaurant mammaMia = new Pizzeria();
        mammaMia.order();
        Restaurant istanbul = new Doenerbude();
        istanbul.order();
    }
}
```

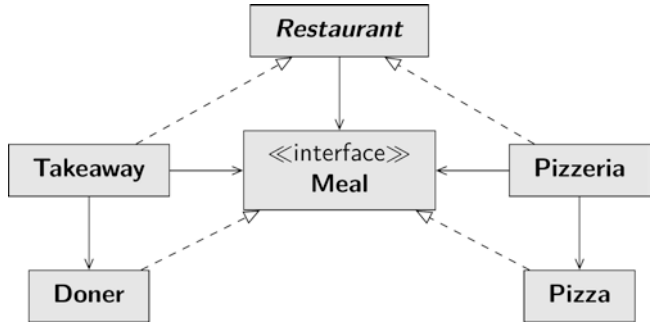

Fig. 16.1 Class diagram of the sample project Meal

The following text is output on the console:

```
Dinner's ready! There is Pizza Margharita
Dinner's ready! There is kebab with everything
```

If the guest wants to eat a pizza, he creates an instance of the class `Pizzeria`, otherwise an instance of the class `Takeaway`. At no point is there an `if` query. Just by selecting the "right" subclass, the desired product is created.

Figure 16.1 shows you the class diagram of the project. In the terminology of the pattern, the interface `Meal` is a product. The derived classes `Doner` and `Pizza` are concrete products. The interface `Restaurant` is a producer and the concrete localities (pizzeria and takeaway) are concrete producers.

16.2 Variations of the Example

The factory method can optionally be parameterized. The project is already prepared for this variation. The method `takeOrder()` returns a string with the order. The previous project version ignores the customer request, which is not very friendly. In the new version, the order is passed to the factory method as a parameter, which then creates the desired meal. The pizza example demonstrates this procedure. The following code from the MealVariation sample project shows the parameterized factory method of the pizzeria.

```
protected Meal prepare(String order) {
    if (order == null || order.isEmpty())
        return new Pizza();
    else
        return switch (order) {
            case "Calzone" -> new Calzone();
            case "Hawaii" -> new Hawaii();
            default -> { System.out.
                println("We don't offer this pizza!");
                yield null;
        }
}
```

Note that I added a switch expression to the return statement in the else branch, and then I don't have to write another `return` in every single case, but give the result expression of every case distinction to the return statement. In the above example you can also see that firstly you can have multiple statements executed in a case distinction (here the

default case), and secondly the keyword `yield` which is used instead of a return. A `return` is intended exclusively for terminating a method. The termination of a command sequence within a switch expression is done with `yield` and returns the subsequent value as the value of the expression.

With a parameterized factory method, you can easily introduce new products. Without parameters, you would have to rely on creating a new concrete producer for each new concrete product. However, since the production effort for the different pizzas is similar and calzone even inherits from pizza, it makes sense to parameterize the factory method.

16.3 Practical Application of the Pattern

You have already seen the pattern in practical use.

16.3.1 Recourse to the Iterator Pattern

In Chap. 11, you created iterators. To use a collection in an extended for-each loop, it must implement the `Iterable` interface. The interface prescribes the `iterator()` method, which returns an object of type `Iterator`. The Iterable is implemented by the `ArrayList` class, for example, but the two collections you created in the Iterator Pattern chapter also have this interface implemented. All of these classes must be able to return a product, an object of type `Iterator`; the Iterator prescribes three methods that you can use to iterate element-by-element through the collection. In most cases, it might be convenient to define the iterator as an anonymous class, that is, `return new Iterator{...}`. The diagram in Fig. 16.2 shows this interaction.

You define an interface, the `Iterable` interface, which prescribes a method for creating objects: `iterator()`. This method is the factory method. An object of the type of the interface `Iterable` must return an object of the type `Iterator`, which works together with its own specification. In the diagram, I have helpfully named this object `MyIterator`. The example project Iterator shows this (rough) structure.

> If the `iterator()` method is a factory method because it returns an object, then you could say that any method that returns an object is a factory method, which is not uncommon in object-oriented programming. A factory method is characterized by the fact that subclasses decide on the exact specification of the object. The client does not know by which class the object it receives was created.

The factory method pattern is useful when you want to write general-purpose code. A for-each loop, which I mentioned in the last example, has no idea from which class the

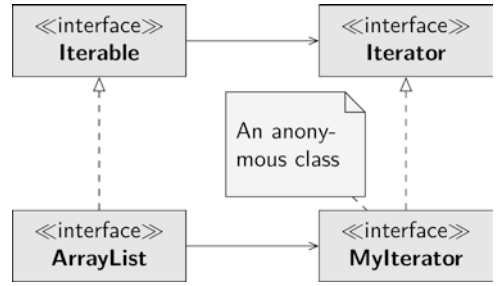

Fig. 16.2 Class diagram Factory Method Pattern

object of type `Iterable` was instantiated – it doesn't even need this knowledge. Similarly, the for-each loop has no interest in which concrete product it is working with. Since the binding of the concrete iterator object to the for-each loop is very loose, you can define any collection classes and use them in the for-each loop. The concrete creator is responsible for creating a matching concrete product.

If you create a new object with the new operator, you must always call a constructor of the class. When you define a factory method, on the other hand, you can choose any identifier you want. For example, consider the procedure of the `Color` class. Here, there are numerous static methods whose only task is to create a Color object from certain parameters. There is, for example, the method `getHSBColor()`, which describes its purpose much more succinctly.

> If you compare the Factory Method pattern up to this point with the Abstract Factory, you'll notice that the Abstract Factory is much more complex. In fact, many projects start with the Factory Method and end up with the Abstract Factory.

16.3.2 At the Abstract Factory

In the Abstract Factory chapter, when you programmed the Haunted House, you referred back to the Factory Method several times. There was the creator, the class `ComponentFactory`. In this class, there are the methods `createRoom()`, `createDoor()`, and `createWall()`, which all create doors, rooms, and walls that have no special feature, that is, neither a ghost nor an enchantment. This class is the interface and there is nothing wrong with defining the interface to bring default behavior. Therefore, other classes, called concrete creators, were derived from this class; one class creates doors with spells. The other class creates doors with spells and rooms with ghosts. The derived classes override certain create methods of the superclass. These are exactly the "factory methods". It looked similar with the products. Products were the classes `door`, `room` and `wall`. From these, other concrete products were derived: Doors with spells and Rooms with ghosts.

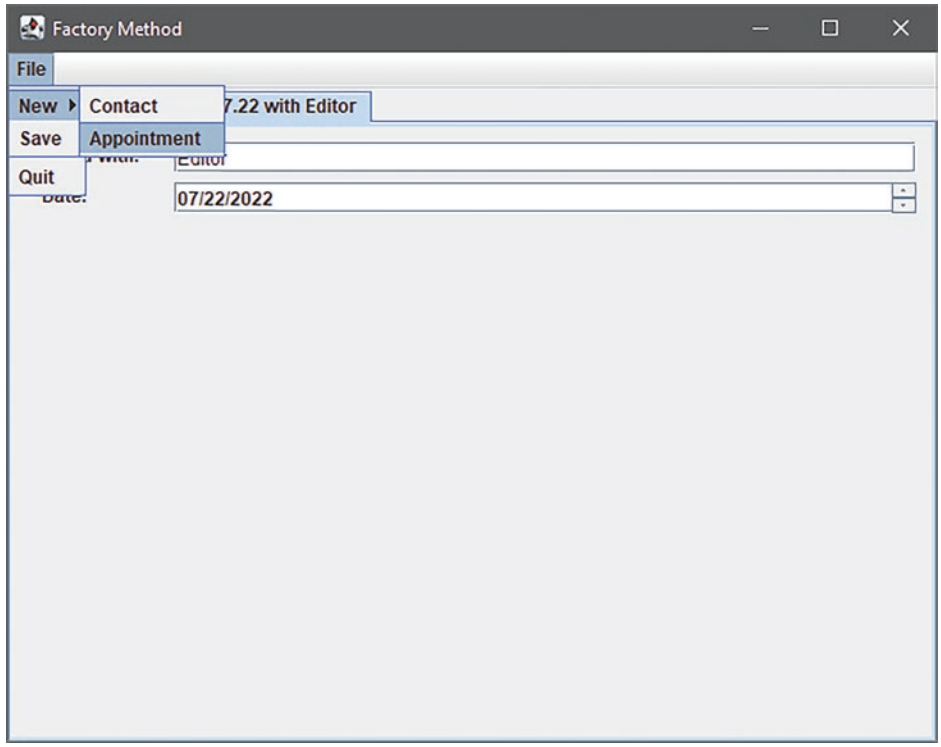

Fig. 16.3 Screenshot of the finished sample project AddressBook

16.4 A Larger Example – A Framework

The Factory Method finds practical application in the creation of frameworks.

16.4.1 And Another Calendar

You create a framework that manages appointments and contacts. In the screenshot in Fig. 16.3, you can see that I have created several contacts. I also have an appointment with my editor on July 22, 2022 – I must not forget it!

The frame of the program provides a blank interface and the menu. You can create entries (appointments and contacts). If you have created a new entry, it will be displayed on a new tab. The program should remain flexible to be able to display new entries such as e-mails or entire address books in a further expansion stage. You can guess that it's all about the frame and the entries being separated from each other and having only loose interfaces.

Let's take a closer look at the connection between the frame and the entry. Entries are certainly responsible for their own data. A contact consists of first and last name and a date

of birth. An appointment consists of the name of the other participant and the date of the appointment. The question now is which entity composes the controls for each data. If you delegate this task to the frame, this assumes that the frame knows all the dates of all conceivable entries. You are restricting yourself in that an existing entry may not get any additional data, because otherwise the frame would have to be modified. Also, implementing new entries would be extremely costly because the new controls would have to be coded in the frame. However, the frame has already been extensively tested and delivered to the customer – you will never want to touch it again under any circumstances. So the responsibility for designing the editor must be delegated to the class that best knows the controls you need: the entry itself.

You can find the following code in the AddressBook sample project.

16.4.2 The Interfaces for Entries and Their Editors

The entries implement the interface `Entry`. It prescribes two methods; one requests a textual description from the entry, the other retrieves an object of type `EntryPanel`. The returned `EntryPanel` can be optimized for either an appointment or a contact – this is like the iterator returned by any list with its own specifics.

```
public interface Entry {
    EntryPanel getEntryPanel();
    String getDescription();
}
```

The editor of an entry is of the type of the interface `EntryPanel`. This interface again prescribes two methods. One method asks the editor to save the entry. The other method returns a `JPanel` on which the controls are arranged.

```
public interface EntryPanel {
    void save();
    JPanel getEditor();
}
```

The following section shows how these interfaces are implemented.

16.4.3 The Class "Contact" as an Example of an Entry

Next, I would like to introduce the Contact class as an example of a possible entry. The class stores the first name, last name, and date of birth in its data fields. In addition, the class creates an object of type `EntryPanel`. This object is an instance of an inner class, which will be described below.

```
public class Contact implements Entry {
    private String firstname = "<first name>";
    private String lastname = "<last name>";
    private LocalDate birthday = LocalDate.now();
    private final EntryPanel entryPanel = new MyEditor();

    @Override
    public EntryPanel getEntryPanel() {
        return entryPanel;
    }

    private class MyEditor implements EntryPanel {
        // … abridged
    }
}
```

In the constructor of the inner class, the editor panel is assembled. Here I use the TwoColumnLayout that you developed in the chapter about the Strategy Pattern. Further, I've given the input fields a FocusListener that highlights all the text when you click in the field. And finally, the JSpinner, where the date of birth is entered, gets its own date editor and date model. Please analyze the code of the project. Of course you could also use a DatePicker instead of the simple spinner, this is not available natively in the JDK, but in many free libraries, of which you can choose one. For the demo here, however, the JSpinner serves its purpose. You can also overwrite the date manually instead of tediously "spinning" back and forth day by day.

The input fields are saved as data fields. When the user gives the command to save the data, the editor transfers the values from the input fields and from the JSpinner to the data fields of the outer class. If you extend the project further, you could, for example, provide the functionality that the class gets serialized. The getEditor() method returns the panel built in the constructor.

```
private class MyEditor implements EntryPanel {
    private final JPanel pnlEntry = new JPanel();
    private final JTextField edtfirstname =
                        new JTextField("<First Name>");
    private final JTextField edtlastname =
                        new JTextField("<Last Name>");
    private final JSpinner spnBirthday = new JSpinner();

    MyEditor() {
        // … abridged
    }

    @Override
    public void save() {
```

```
            firstName = edtFirstName.getText();
            lastName = edtLastName.getText();
            birthday = (Date) spnBirthday.getValue();
        }

        @Override
        public JPanel getEditor() {
            return pnlEntry;
        }
    }
```

How does the client handle these classes?

16.4.4 The FactoryMethod Class as a Client

In this class there is a list where all entries are stored. Each new entry gets its own tab in a JTabbedPane on the GUI. When the user clicks Save on the menu, an object of type `Action` compares each entry in the list with each entry with the components of the JTabbedPane. From the currently selected component, the editor is determined and the `save()` method is called on it. The constructor assembles the controls and displays the GUI on the screen.

```
public class FactoryMethod {
    // … abridged
    private final JMenuItem mnSave = new JMenuItem();

    private final Action saveAction =
                            new AbstractAction("Save") {
        @Override
        public void actionPerformed(ActionEvent e) {
            listEntries.forEach((tempEntry) -> {
                Object selectedPanel =
                        tbbMain.getSelectedComponent();
                if (tempEntry.getEntryPanel().getEditor()
                                    == selectedPanel) {
                    tempEntry.getEntryPanel().save();
                    var index =
                             tbbMain.getSelectedIndex();
                    tbbMain.setTitleAt(index,
                                tempEntry.toString());
                }
            });
        }
    };
}
```

Most of the tricky bits in this class are in the Swing programming area. What is important for the factory method pattern is that the coupling of the client to the entries is loose, which allows you to develop more entries very easily.

You can see the power of the factory method in this example: the client does not need to know from which class the object of type `EntryPanel` was instantiated. It is enough that it calls the factory method `getEntryPanel()` on the object of type `Entry`. Since interfaces are classes with only abstract methods, you can say that the entry (contact or inheritance) has delegated the responsibility for creating the EntryPanel to a subclass.

16.5 Difference to Abstract Factory

Finally, let's look at the differences with Abstract Factory. At first glance, the difference seems relatively clear: The abstract factory creates a family of products; think of a unified look and feel or a specific garden. The factory method creates a single product. However, since nowhere does it say that a family must necessarily consist of two or more products, the difference should not be based on this alone.

The more important difference, in my opinion, is that the Abstract Factory is object-based, while the Factory Method is class-based. Object-based patterns rely on the interaction of objects selected by the client. Class-based patterns involve inheritance; the superclass calls the "right" method.

Background Information

I told you in Sect. 1.2 that patterns are divided into three categories: Creation Patterns, Behavior Patterns, and Structural Patterns. However, the GoF has made two further distinctions within these three categories: by scope.

Class-based patterns are the template method and the interpreter. In the case of the adapter, there are two types – one object-based and one class-based. All other patterns are object-based. So the GoF follows its own postulate that composition is preferable to inheritance. An overview of the pattern categories can be found in Table 1.1.

16.6 Factory Method – The UML Diagram

From the example project Address Book you can find the UML diagram in Fig. 16.4.

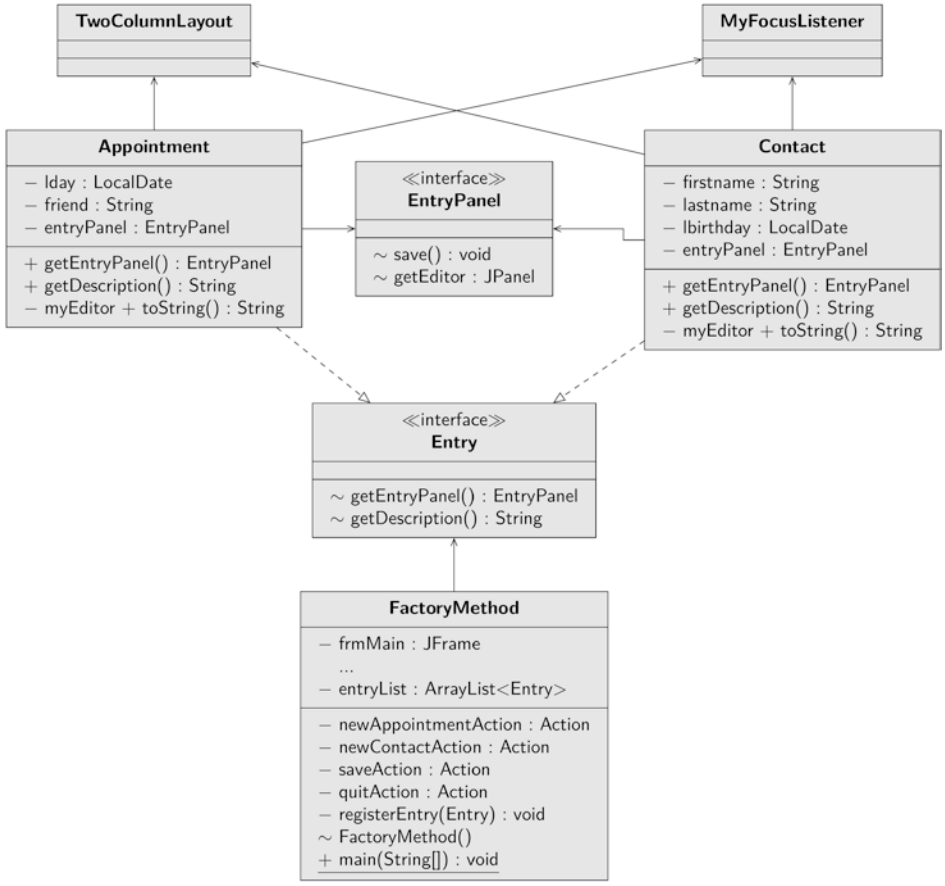

Fig. 16.4 UML diagram of the Factory Method Pattern (example project Address Book)

16.7 Summary

Go through the chapter again in key words:

- The Factory Method supports SRP: object creation and object usage are separated.
- The object creation is outsourced to a separate class.
- The creator delegates object creation to a subclass.
- To do this, it calls the factory method that gives it its name: e.g. `createCar()`.
- The subclass must define this method in its own unique way.
- The subclass decides which concrete product/object is created.