15 Abstract Factory (Abstract Factory)

Do you remember the singleton pattern from Chap. 3? You created an object without using the new operator. Instead, the client could use a static access method to get the instance of the class. The abstract factory delegates the creation of objects to a method, so that you, the user, can do without the actual creation with `new`.

Using two examples, let's take a closer look. We start with a beautiful garden.

15.1 Create Gardens

In our example there are different types of gardens; first we have monastery gardens where herbs are planted. But there are also ornamental gardens where roses grow. The respective enclosure is different, of course. The monastery garden is surrounded by a stone wall, while the ornamental garden is framed by a nicely cut hedge. The ground is also different in each case: in the monastery garden there are stone slabs, while in the ornamental garden you walk on fresh grass. The monastery garden and the ornamental garden are therefore two different *product families*. Each product family has the same characteristics (plants, fencing, soil), but each has a different design; the respective characteristics are called *products* in the abstract factory.

15.1.1 The First Attempt

Let's start with the example project Abstract_Garden, with the package Trial1, where we define a class Garden, where the garden is created and maintained. Like this:

O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_15

```
public class Garden {
    private enum garden type {
        MonasteryGarden, OrnamentalGarden
    };

    private GardenType garden =
                            GardenType.MonasteryGarden;

    public void layFloor() {
        switch (garden) {
            case MonasteryGarden -> {
                // lay old flagstones
            }
            default -> {
                // Sowing grass
            }
        }
    }

    public void plant() {
        switch (garden) {
            case MonasteryGarden -> {
                // Planting herbs
            }
            default -> {
                // Set roses
            }
        }
    }

    public void enclose() {
        // … as with the previous two methods.
    }
}
```

You can guess that this approach seems somehow wrong. But why are you bothered by it? Quite simply: You have three methods in which a switch query is needed. If you want to create another garden – for example a kitchen garden with tomatoes – you have to change a switch query in three methods. This is not very maintenance-friendly, but all the more error-prone.

The garden certainly also needs to be managed: The plants want to be watered and pruned; you also need to weed regularly. I will not print the methods for managing the garden here. However, you have a violation of the Single Responsibility Principle here:

you put the creation of the garden and its maintenance into one class. True, I had written that it is okay to violate it if you are aware of it and have a good reason. But that good reason is exactly what is missing here. Violating the SRP could be problematic, though, because both creating and managing the garden are very costly; these two tasks had better not be mixed. So the approach above implements a lot of code in a single class that, in case of doubt, is not needed at all. If you redefine a responsibility, both responsibilities need to be retested. Neither of the responsibilities can be reused. Here, the cohesion is decidedly weak and that is an indication of inappropriate design.

15.1.2 The Second Attempt – Inheritance

Perhaps the problem would be solved if you used inheritance? In the Trial2 package, you define an abstract class `AbstractGarden`, from which the subclasses `MonasteryGarden`, `KitchenGarden`, or `OrnamentalGarden` inherit. The garden is created in the constructor of the subclass.

```
public class MonasteryGarden extends AbstractGarden {
    MonasteryGarden() {
        super.layFloor(new OldStonePlate());
        super.plant(new Herbs());
        super.enclose(new StoneWall());
    }
}
```

This solution looks cleaner, but in fact it doesn't solve the dual responsibility problem: you still have an object that lavishly creates the garden and manages it. Moreover, this approach violates a principle that I consider much more essential than the SRP: prefer composition to inheritance. So let's find an alternative!

15.1.3 The Third Approach – The Abstract Factory

In the Trial3 package you have different gardens (*product families*) each with different *products*, i.e. different types of plants, soil types and enclosures. The products vary and are now encapsulated. I mentioned earlier that it makes sense to program against interfaces. This gives you the greatest possible flexibility. How can the products be abstracted? Both herbs and roses are plants; both the stone wall and the hedge are enclosures; both the flagstones and the lawn are soil. So design the interfaces, the different abstract products, and the implementations, the different concrete products. As an example, let me show the plants.

```
public interface Plant {
    // the abstract product "plant
}
```

There are at least two implementations of plants:

```
public class Rose implements Plant {
    // the concrete product Rose
}
public class Herbs implements Plant {
    // the concrete product herb
}
```

To create the different gardens, define a factory. For a monastery garden, implement a `MonasteryGardenFactory` and for the ornamental garden, implement an `OrnamentalGardenFactory`. You expect both factories to provide the same methods and create the same products. To ensure this, you define an interface (abstract class or interface), the abstract factory, from which the concrete factories inherit. And in it are the methods that must be implemented in each of the concrete factories.

```
public interface AbstractGardenFactory {
    Plant plant();
    Floor layFloor();
    Enclosure enclose();
}
```

The MonasteryGardenFactory returns the elements typical of a monastery garden.

```
public class MonasteryGardenFactory
                    implements AbstractGardenFactory {
    @Override
    public Plant plant() {
        return new Herb();
    }

    @Override
    public Floor layFloor() {
        return new Flagstone();
    }

    @Override
    public Enclosure enclose() {
        return new StoneWall();
    }
}
```

And how can the client who wants to have a garden deal with this? It creates an instance of the desired garden factory and gets the individual elements from it. In the following example, we create the elements for a monastery garden.

```
public class Gardenin {
    Gardening() {
        AbstractGardenFactory factory =
                            new MonasteryGardenFactory();
        var floor = factory.layFloor();
        var enclosure = factory.enclose();
        var plant = factory.plant();
}
```

Note that I explicitly specify the superclass `AbstractGardenFactory` for the declaration of the variable `fabrik.` Using `var.` for this declaration would create a factory of type `MonasteryGardenFactory,` which I explicitly do not want here.

Let's look at what we gain from this.

15.1.4 Advantages of the Abstract Factory

For a better overview, the class diagram Fig. 15.1 only shows the project in abbreviated form – the floor classes end interface are left out here.

The abstract garden factory knows the abstract declaration of the individual products, i.e., the interfaces floor, enclosure and plant. The client creates a variable of the abstract factory type that references an instance of a concrete factory: `AbstractGardenFactory factory = new OrnamentalGardenFactory`(). The concrete factory knows its specific products: In the example, the ornamental garden knows only the lawn, rose, and hedge – it has no reference to the flagstones, herb, or stone wall. When the client requests floor, plants, and enclosure, the factory is able to return its special products. Since the client

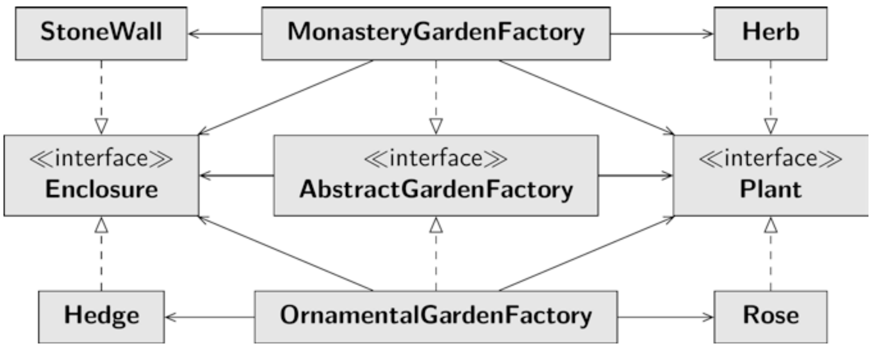

Fig. 15.1 Abbreviated class diagram of the garden project

relies on abstractions, i.e. the interfaces, it has no idea what specific products it will receive – and thanks to polymorphism, it doesn't have to worry about the runtime type of the object.

The client is free to instantiate a different factory and thus be given completely different products. A single line of code changes the nature of the garden:

```
AbstractGardenFactory factory =
                            new MonasteryGardenFactory();
```

The client is independent of changes to existing factories. If lilies are to be planted in the ornamental garden instead of roses, this change can be made in the `OrnamentalGardenFactory`; the client will not even notice this change, it does not have to be tested again.

With this solution, you ensure that the garden is consistent. You purchase the individual products from a specific factory. The factory uniformly supplies only products for either a monastery garden or an ornamental garden. The gardens are now consistent; it is ensured that no roses grow in the monastery garden and no herbs are planted in the ornamental garden.

15.1.5 Defining a New Garden

You have a cohesive family of products and you can add new families very easily. What would have to happen if you wanted to create a new garden, for example an allotment or garden plot? First, you need a new concrete factory that implements the `AbstractGardenFactory` interface. Take a look at the example project GardenPlot:

```
public class GardenPlotFactory
                    implements AbstractGardenFactory {
    @Override
    public Plant plant() {
        return new Tomato();
    }

    @Override
    public layFloor() {
        return new ConcreteApron();
    }

    @Override
    public Enclosure enclose() {
        return new ChainLinkFence();
    }
}
```

You now still create the concrete products, the classes `Tomato`, `ConcreteApron`, and `ChainLinkFence`, which must correspond to the interfaces of the abstract products, `Floor`, `Enclosure`, and `Plant`.

> Why is the pattern called "Abstract Factory"? Because both the factories and the products rely on abstraction.

15.2 Discussion of the Pattern and Practice

You will use the abstract factory whenever you need a set of individual products that all correspond to a certain type. The products form the product family. In the previous example you got to know the product families of the gardens. However, the following example would also have been conceivable: You are programming an address management system. When you code the postal code, you plan five digits for Germany. But the structure of a postal code will be different in America or in Russia. Likewise, the length of the telephone number varies from country to country. Maybe there is even a different structure or a certain pattern for it? So, when you instantiate a contact, you need a consistent set of products for each country: a country-specific postal code and a country-specific phone number. The `AmericaFactory` will provide these products, just like the `RussiaFactory`, but customized for that country. The contact generated by the factory, if the factory is programmed correctly, is consistent. Furthermore, the client can be written in such a general way that its code is valid for all conceivable product families.

Where do you find the abstract factory in practice? Think of the different look and feels that Java provides. You can set a specific look and feel and change it at runtime. Each component is drawn uniformly with the chosen look and feel. Third party developers can develop their own look and feel. And since all created objects come from the same family, they can interact with each other if necessary. One pitfall must be addressed, however: Once you've determined the products, stick to them. Imagine that you also want to create a pond in your garden. The `AbstractGardenFactory` interface must dictate a corresponding factory method `getPond`(). Then, all concrete factories must define a method that returns a pond object. In our small example, this is certainly not a problem. However, if you have created a large framework for which your clients have already created their own garden factories, then this change can become very high-maintenance and therefore very expensive.

15.3 Chasing Ghosts

Imagine you get the order to create a game in which you have to hunt ghosts and open magic doors in an old house. Remember the old text adventures that were popular in the early days of home computing and still have fans today? We're doing something similar.

15.4 Abstract Factory – The UML Diagram

Figure 15.2 shows the UML diagram from the example project House_4.

15.5 Summary

Go through the chapter again in key words:

- With the abstract factory, you create a product family that consists of various individual products.
- The products together form a unit, e.g. a uniform look and feel.
- During implementation, an interface for a factory is created: the abstract factory.
- The client programs against abstractions of products, that is, against abstract products.
- The abstractions can be abstract classes or interfaces.
- A concrete factory produces the concrete products that belong to a particular family.
- The bond between the products and the client is loosened.
- As a result:
- An entire product family can be replaced without affecting the client code,
- A specific product can be changed without affecting the client code,
- New product families can be added,
- Products can be reused, e.g. in other product families,
- The products can communicate with each other because they know their mutual interfaces.
- One disadvantage is that it is difficult to expand a product family with additional members.

15.6 Description of Purpose

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

> Provide an interface for creating families of related or interdependent objects without naming their concrete classes.