18 Builder

From the set of generation patterns, you know the singleton pattern, the two factories, and the prototype pattern. The Builder Pattern is the last pattern you use to create objects without using the new operator. With the builder pattern, you have an object that is complex or complicated to construct. This object cannot be created in one pass, but goes through an elaborate construction process.

18.1 An Object Creates Other Objects

To get started, let's start with a simple example. If you want to book a trip online, you can specify a variety of search parameters. The most important parameters are probably the time period and the number of days you want to travel. It is also interesting to know how many people you will be traveling with and how many children (under 14) you will be taking with you. You will certainly also want to specify how many stars your accommodation should have. And important are the rating of the accommodation and the recommendation rate. Maybe you want to be on the safe side and only want to see hotels that have already been rated by at least x guests. You save all these search parameters in the data fields of a travel object.

18.1.1 Telescoping Constructor Pattern

Take a look at the Telescoping_Constructor_Pattern sample project. You pass the search parameters to the constructor.

© The Author(s), under exclusive license to Springer Fachmedien Wiesbaden GmbH, part of Springer Nature 2023

O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_18

```
public class Trip {
public final LocalDate startDate;
public final LocalDate endDate;
public final int duration;
public final int numberTravellers;
public final int numberKids;
public final int minimumStars;
public final int minimumRecommendations;
public final int rating;
public final int minimumNumberRatings;

Trip(LocalDate startDate, LocalDate endDate,
        int duration, int numberTravellers,
        int numberKids, int minimumStars,
        int minimumRecommmendations,
        int rating, int minimumNumberRatings)
    {
        this.startDate = startDate;
        this.endDate = endDate;
        this.duration = duration;
        this.numberTravellers = numberTravellers;
        this.numberKids = numberKids;
        this.minimumStars = minimumStars;
    this.minimumRecommendations = minimumRecommendations;
        this.rating = rating;
        this.minimumNumberRatings = minimumNumberRatings;
    }
}
```

Calling this constructor is sheer horror! The main method of the `ApplStart` class demonstrates this:

```
public static void main(String[] args) {
    var trip = new Trip(LocalDate.now(), LocalDate.now()
                , 7, 2, 3, 3, 80, 5, 30);
    System.out.println(travel);
}
```

The problem is not only that you have to pass a lot of parameters – they are of the same type. You will never find out what each parameter stands for without documentation. Errors will creep in very reliably with such a call. One solution might be to overload the constructor. If the client doesn't require a rating or even a minimum number of ratings, it should be allowed to omit them. The new constructor calls the one just presented with this() and passes default values:

```
Trip(LocalDate startDate, LocalDate enddate,
        int duration, int numberTravellers,
        int numberKids, int minimumStars,
        int minimumRecommendations) {
    this(startDate, enddate, duration, numberTravellers,
            numberKids, minimumStars,
            minimumRecommendations, 0, 0);
}
```

The client now creates a travel object with a slightly shorter parameter list:

```
public static void main(String[] args) {
    var trip = new Trip(LocalDate.now(), LocalDate.now(),
                    7, 2, 3, 3, 80);
    System.out.println(travel);
}
```

Is this call more appealing or clear? No! This solution, called the Telescoping Constructor Pattern, has made its way into the list of antipatterns. It's never a good idea to have to pass too many parameters to a method – even more so when they are of the same type. Let's look at another solution.

18.1.2 JavaBeans Pattern

In Java introductions you will find the advice that data fields must be encapsulated and may only be read and modified via getters and setters – and the version in the sample project JavaBeans_Pattern is oriented to this. Information that cannot be dispensed with during construction, i.e. time frame, duration, and number of travelers, is stored in final data fields that are initialized by the constructor. The remaining information is preset with default values and can optionally be overwritten with setters.

```
public class Travel {
    public final LocalDate startDate;
    public final LocalDate endDate;
    public final int duration;
    public final int numberTravellers;
    private int numberKids = 0;
    private int minimumStars = 0;
    private int minimumRecommendations = 0;
    private int rating = 0;
    private int minimumNumberRatings = 0;
```

```
    Trip(LocalDate startDate, LocalDate endDate,
                    int duration, int numberTravellers) {
        this.startDate = startDate;
        this.endDate = endDate;
        this.duration = duration;
        this.numberTravellers = numberTravellers;
    }

    public void setNumberKids(int numberKids) {
        this.numberKids = numberKids;
    }

    // … abridged
}
```

The client can now create an instance of the Trip class much more clearly.

```
public static void main(String[] args) {
    var trip = new Trip(LocalDate.now(),
                    LocalDate.now().plusDays(14), 14, 2);
    trip.setMinimumStars(3);
    trip.setMinimumRecommendations(80);
    System.out.println(trip);
}
```

The code is indeed much more speaking now. However, the other data fields are not final and cannot be. Subsequent modification of the values is therefore allowed, even if it may not be desired. The approach is not quite the same as the JavaBeans specification, but merely adopts the essential principle. The JavaBeans specification requires that you always define a default constructor. Incidentally, you will also occasionally come across opinions in the literature that relegate the JavaBeans specification to the realm of antipatterns – for the reasons just mentioned.

18.1.3 Builder Pattern

Since the toolbox of OOP does not allow a satisfactory solution, it is time for a pattern. The Builder Pattern assumes an object whose task is limited to constructing another object. In the version in the Builder_Pattern sample project, you now realize the travel class by storing all information in final data fields. As with Singleton, the constructor is private, so that an object can only be created within the class. The constructor expects an object of type `Builder` as parameter.

```
public class Trip {
    // … abridged
    private final LocalDate startDate;
    private final LocalDate endDate;
    public final String start;
    public final String end;
    public final int duration;
    public final int numberTraveller;
    public final int numberKids;
    public final int minimumStars;
    public final int minimumRecommendations;
    public final int rating;
    public final int minimumNumberRatings;

    private Trip(Builder builder) {
        this.startDate = builder.startDate;
        this.endDate = builder.endDate;
        var formatter =
            DateTimeFormatter.ofPattern("MM/dd/YYYY");
        start = formatter.format(startDate);
        end = formatter.format(endDate);
        this.duration = builder.duration;
        this.numberTravellers = builder.numberTravellers;
        this.numberKids = builder.numberKids;
        this.minimumStars = builder.minimumStars;
        this.minimumRecommendations =
                        builder.minimumRecommendations;
        this.rating = builder.rating;
        this.minimumNumberRatings =
                            builder.minimumNumberRatings;
    }
}
```

The `Builder` type is defined as the static inner class of the trip. The data fields of the trip are also declared here. The important data fields are initialized in the constructor, the others get default values. For each data field, there is a setter that, contrary to language convention, only repeats the name of the field. Inside the setter, the corresponding data field is assigned a value; finally, the setter returns the builder instance.

```
public static class Builder {
    private LocalDate startDate;
    private LocalDate endDate;
    private int duration;
    private int numberTravellers;
    private int numberKids = 0;
```

```
        private int minimumStars = 0;
        private int minimumRecommendations = 0;
        private int rating = 0;
        private int minimumNumberRatings = 0;

        public Builder(LocalDate startDate,
                       LocalDate endDate, int duration,
                       int numberTravellers) {
            this.startDate = startDate;
            // … abridged
        }

        public Builder numberKids(int numberKids) {
            this.numberKids = numberKids;
            return this;
        }

        public Builder minimumStars(int minimumStars) {
            this.minimumStars = minimumStars;
            return this;
        }

        // … abridged
    }
```

The `build()` method then finally creates the journey object and passes the builder object to its constructor for this purpose.

```
    public journey build() {
        return new Trip(this);
    }
```

In the first step, the client creates a builder, supplies it with the relevant data – if required – and only then has a trip object returned.

```
    public static void main(String[] args) {
        var builder = new Trip.Builder(LocalDate.now(),
                                LocalDate.now(), 15, 2);
        var trip = builder
                .minimumStars(3)
                .rating(5)
                .numberKids(0)
                .build();
        System.out.println(trip);
    }
```

This first realization of the Builder Pattern should give you a sense that it can be useful to have objects whose functionality is limited to creating other objects. Please don't be surprised that for simplicity reasons I have given `LocalDate.now()` for the startDate as well as for the endDate in the first examples; in practice it is of course impracticable to have 15 days of vacation in the time "from today to today".

Somewhat unattractive is still the new Trip.builder in the main method. You can still create a builder in the trip class with a static method:

```
public class Trip {
    // … abridged
    public static Trip.Builder builder(
            LocalDate startDate, LocalDate endDate,
            int duration, int numberTravellers) {
        return new Trip.Builder(startDate,
                endDate, duration, numberTravellers);
    }
}
```

If you then want to do without a separate builder variable in the main method, the following code (which you will also find in the example) will also work:

```
public static void main(String[] args) {
    var trip = Trip.builder(LocalDate.now(),
                    LocalDate.now().plusDays(15), 15, 2)
            .minimumStars(3)
            .rating(5)
            .numberKids(0)
            .build();
    System.out.println(trip);
}
```

18.2 A More Complex Design Process

The purpose of the Builder Pattern is to outsource the construction process to a separate class. This always makes sense if the construction of an object is complex or should be replaceable. In the following example, I would like to take up the budget book from the last chapter. To do this, open the sample project BudgetBuilder. In practice, you probably don't want to store the various items and categories in the source code, as I showed you with the composite pattern in Chap. 12. You may want to read the data from an XML file and display it in a JTree. The Builder Pattern is ideally suited for this, since a node in the JTree cannot be created in one pass; only when it has no further subnodes can the object creation be completed.

The XML shortened file in the example looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Budgetbook
[
    <!ELEMENT Budgetbook (Item*, Month+)>
    <!ELEMENT Month (Category*)>
    <!ELEMENT Category (Item| Category)*>
    <!ELEMENT Item EMPTY>

    <!ATTLIST Budgetbook description    CDATA #REQUIRED>
    <!ATTLIST Month description         CDATA #REQUIRED>
    <!ATTLIST Category description      CDATA #REQUIRED>
    <!ATTLIST Item  description         CDATA #REQUIRED
                        amount          CDATA #REQUIRED
                        required (yes|no)     "yes" >
]>
<Budgetbook description = "Budgetbook 2022">
    <Item description="Life insurance"   amount = "-1600" required
= "yes"/>
   <Month description = "January">
       <Category description = "Income">
           <Item description = "Main job" amount = "2000" required
= "yes"/>
           <Item description = "Lectures" amount = "5000" required
= "yes"/>
       </Category>
       <Category description = "Expenses">
          <Category description = "Rent">
              <Item description = "Apartment" amount = "-700"/>
              <Item description = "Garage" amount = "-150" required
= "no"/>
          </Category>
          <Category description = "Insurances">
              <Item description = "Car" amount = "-34.50"/>
          </Category>
       </Category>
   </Month>

   … abridged


</Budgetbook>
```

Your task will now be to turn the XML document into a TreeModel and display it in a JTree.

18.2.1 Converting XML Files into a TreeModel

I first took the last project version of the chapter about the composite pattern. In this, I created a new package `builder`, where you can find the abstract class `Builder`. Builder inherits from `org.xml.sax.helpers.DefaultHandler` and forces its subclasses to define the `startElement()` and `endElement()` method. When an element of the XML document is opened or closed, one of these methods is called. If the XML data is not valid, the `warning()`, `error()`, or `fatalError()` methods emit appropriate error messages. The client first lets an instance of the `Builder` type create an object and then fetches the finished product from this instance; each subclass must therefore also offer the `getProduct()` method.

```
public abstract class Builder extends DefaultHandler {
    public abstract Object getProduct();

    @Override
    public abstract void startElement(String uri,
                        String localName, String name,
                        Attributes attributes);

    @Override
    public abstract void endElement(String uri,
                        String localName, String name);

    @Override
    public void warning(SAXParseException exception) {
        System.err.println("-> warning: " +
                                exception.getMessage());
    }

    @Override
    public void fatalError(SAXParseException exception) {
        System.err.println("-> FATAL ERROR: " +
                                exception.getMessage());
    }

    @Override
    public void error(SAXParseException exception) {
        System.err.println("-> error: " +
                                exception.getMessage());
```

```
        }
    }
```

The first concrete builder you will create is the TreeModelBuilder. It has two data fields: the root node and a list where all nodes are temporarily stored. The `startElement()` and `endElement()` methods are responsible for the actual construction process. When an element is opened, the `startElement()` method first checks if the element is called "Item". If so, the attributes are retrieved and a new leaf is created; otherwise, a new composite is created. If the root is `null`, that is, only for the very first element, the currently created element is passed to the `root` data field. If the `root` data field is not `null`, that is, starting with the second element, the new element is appended to the last composite stored in the list as a child node. The node is then stored in the list.

```
public class TreeModelBuilder extends Builder {
    private Composite root;
    private final LinkedList<Node> stack =
                                    new LinkedList<>();

    @Override
    public void startElement(String uri,
                String localName, String name,
                Attributes attributes) {
        Node node;
        if (name.equalsIgnoreCase("Item")) {
            var tempDescription =
                    attributes.getValue("description");
            var tempAmount = Double.
            parseDouble(attributes.getValue("amount"));
            var tempRequired =
                        attributes.getValue("required").
                        equalsIgnoreCase("yes");
            node = new Leaf(tempDescription, tempAmount,
                                        tempRequired);
        } else {
            var tempDescription =
                    attributes.getValue("description");
            node = new Composite(tempDescription);
        }
        if (root == null)
            root = (Composite) node;
        else {
            var tempNode = (Composite) stack.peekLast();
            tempNode.add(node);
        }
```

```
            stack.add(node);
        }

        // … abridged
    }
```

The method `endElement()` is called when the element is closed. It can be limited to removing the last element stored in the list there. The method `getProduct()` creates a MyTreeModel with the data field `root` and returns it.

```
    @Override
    public void endElement(String uri, String localName,
                                              String name) {
        stack.pollLast();
    }

    @Override
    public TreeModel getProduct() {
        return new MyTreeModel(root);
    }
```

The client first creates a builder instance, in this project a TreeModelBuilder. It passes this builder to the `build()` method, which first reads the XML file, converts it into a string and parses it with the builder as handler.

```
Budget() {
    var builder = new TreeModelBuilder();
    build(builder);
    var treeModel = (TreeModel) builder.getProduct();
    var frmMain = new JFrame("Builder Pattern Demo");
  frmMain.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    var trvBudgetBook = new JTree(treeModel);
    trvBudgetBook.setCellRenderer(
                                new MyTreeCellRenderer());
    trvBudgetBook.setEditable(true);
    trvBudgetBook.setCellEditor(new MyTreeCellEditor());
    var scrTree = new JScrollPane(trvBudgetBook);
    frmMain.add(scrTree);
    frmMain.setSize(500, 500);
    frmMain.setVisible(true);
}
public void build(Builder builder) {
```

```
        var filePath = Paths.get(file);
        try {
            var content = Files.readString(filePath);
            var factory = SAXParserFactory.newInstance();
            factory.setValidating(true);
            var saxParser = factory.newSAXParser();
            saxParser.parse(new InputSource(
                  new StringReader(content)),builder);
        } catch (IOException | ParserConfigurationException |
                                            SAXException ex) {
            ex.printStackTrace();
        }
    }
```

Background Information

If you critically review the example, you will notice one disadvantage of the pattern: The builder is strongly tied to the object being created. Since it queries whether an element is called "position", this builder is bound to the given DTD. Likewise, the builder must have precise knowledge of the construction of leaf and composite classes.

On the other hand, you can see the big advantage: You can pass any XML file that conforms to the DTD to the Builder – you will always be able to generate a suitable TreeModel from it.

Your program is now much more modularized. As a result, the routine that reads the XML document can be replaced by a routine that obtains the XML document from another source, such as the network. This change has no effect on the Builder.

In the next section, you will define a new builder that will bring the data into HTML format.

18.2.2 Displaying XML Files as HTML

The following example extends the project with a new builder. It should be possible to generate and display the XML file as an HTML document. Follow the necessary steps in the sample project BudgetBuilder_Ext. In the first step, you create a new builder, the `HTMLBuilder`, which inherits from the `Builder` class. Nothing changes in the `startElement()` and `endElement()` methods. The `getProduct()` method generates an HTML string from the information in the root and returns it.

```
@Override
public String getProduct() {
    html.append("<html><body><h1 align=\"center\">");
    html.append(root.getDescription());
    html.append("</h1>");
    html.append("<b>" + "Annual items:</b><br/>");
    for (var i = 0; i < root.getNumberOfChildNodes();
                                                  i++) {
        var tempNode = root.getIndex(i);
        if (tempNode.getClass() == Leaf.class) {
            html.append(" ");
            var item = (Leaf) tempNode;
            formatLeaf(item);
            html.append("<br/>");
        } else {
            html.append("<p>");
            appendElements(tempNode, 0);
            html.append("</p>");
        }
    }
    html.append("</body></html>");
    return html.toString();
}
```

In this method, the `appendElements()` method is called, which recursively traverses all nodes and expands the HTML string.

```
private void appendElements(Node node, int tab) {
    html.append("<br/>");
    for (var i = 0; i < tab; i++)
        html.append(" ");
    if (node.getClass() == Leaf.class)
        formatSheet((sheet) node);
    else {
        if (tab == 0)
        html.append("<b>");
        html.append(node);
        if (tab == 0)
            html.append("</b>");
    }
    for (var j = 0; j < node.getNumberofChildNodes();
                                                  j++) {
        var childNode = node.getIndex(j);
        appendElements(childNode, tab + 1);
    }
}
```

Both methods call the `formatLeaf()` method, which checks an output item to see if it was required. If not, the display text is colored red.

```
private void formatLeaf(Leaf item) {
    if (!item.expenseIsRequired())
        html.append("<font color=\"#FF0000\">");
    double amount = item.getValue();
    html.append(item
            .getDescription())
            .append(": ")
            .append(NumberFormat
                    .getCurrencyInstance()
                    .format(amount));
    if (!item.expenseIsRequired())
        html.append("</font>");
}
```

Background Information

To create a string in multiple steps, you could concatenate strings: String `x + = stringY;`. Since strings are immutable, each concatenation creates a new string object with the value x assigned to it. The cost is correspondingly high. Alternatively, access a StringBuilder or a thread-safe StringBuffer and append new strings to the previously stored text. Only when the construction is complete do you return the finished string with `StringBuilder.toString();`. Here you can see the builder pattern in practical use again.

The client creates a builder and gets the HTML string from there. This is displayed in a JEditorPane:

```
Budget() {
    var builder = new HTMLBuilder();
    build(builder);
    var html = builder.getProduct();

    var frmMain = new JFrame("Builder Pattern Demo");
frmMain.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    System.out.println(html);
    var editorPane = new JEditorPane();
    editorPane.setContentType("text/html");
    editorPane.setText(html);
```

```
        var scrTree = new JScrollPane(editorPane);
        frmMain.add(scrTree);
        frmMain.setSize(500, 500);
        frmMain.setVisible(true);
    }
```

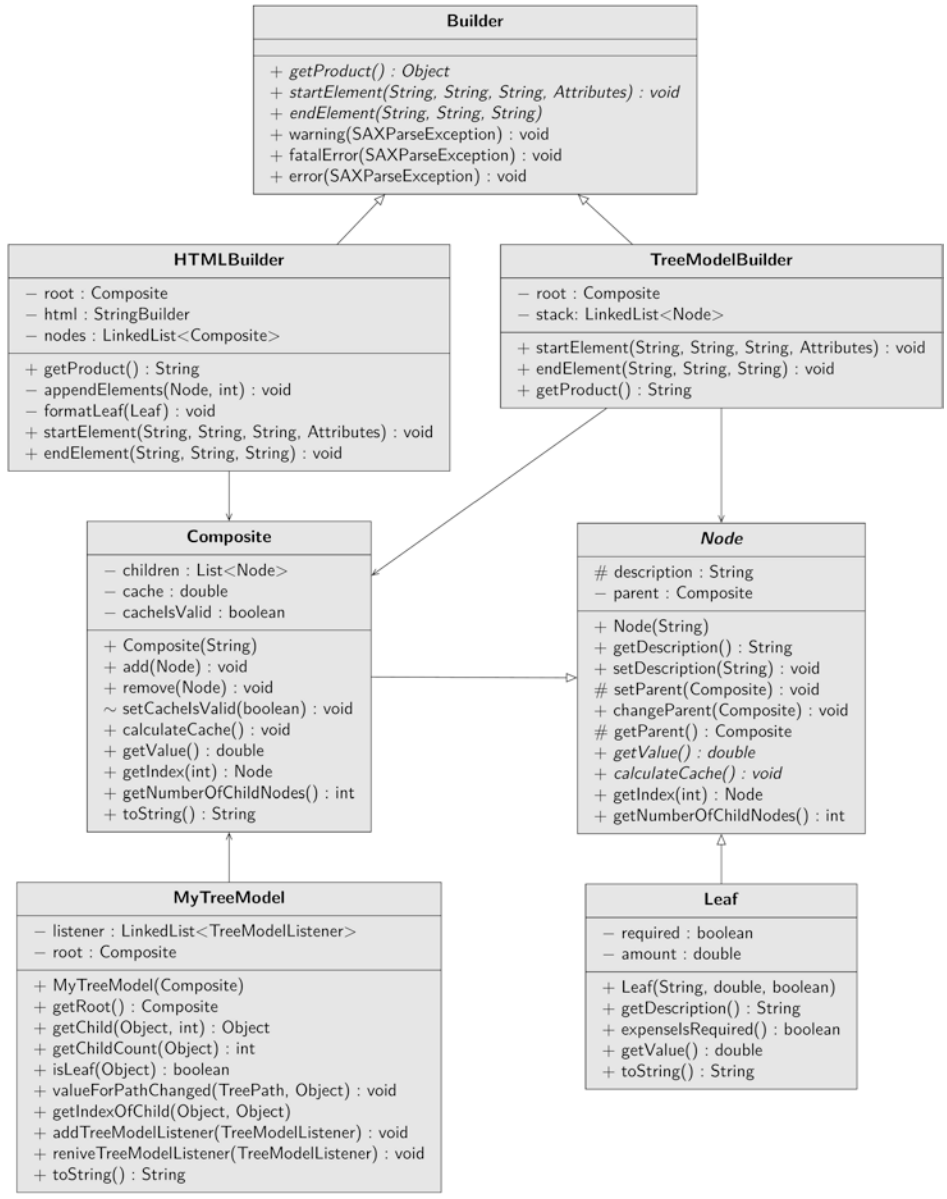

Fig. 18.1 UML diagram of the Builder Pattern (sample project HaushaltsBuilder_Ext)

By the way, you can also "stagger" builders if, for example, you need a very specific sequence in the creation process of the finished object. However, you must then name the corresponding builders individually. For example, for an SQL statement, this could be a selectBuilder, a fromBuilder, and a whereBuilder, each of which can only return the subsequent builder in its last build step. This then leads to chained builders, so-called "NestedBuilders" and also works, for example, for the construction of composite objects.

18.3 Builder – The UML Diagram

You can find the UML diagram from the sample project HaushaltsBuilder_Ext in Fig. 18.1.

18.4 Summary

Go through the chapter again in key words:

- A Builder is an object whose task is to build other objects-.
- The design process is isolated in the Builder.
- Usually the construction process of the objects to be built is complex or complicated; often several steps are required.
- Builders can also be chained together and thus made dependent on each other.
- Since the data is independent of the object creation, many similar objects can be created.
- Typically, builders are used to build a composite.

18.5 Description of Purpose

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

> Separate the construction of a complex object from its representation, so that the same construction process can produce different representations.