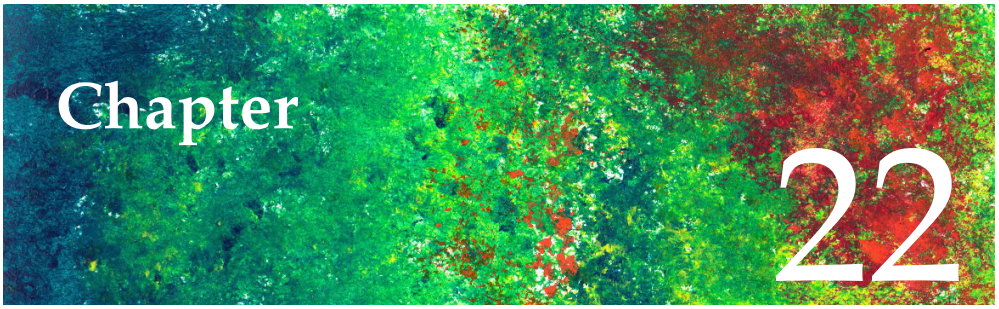


Builder

22.1 The Problem

Consider a word processor program that allows us to write documents with a standard set of typographical options: sections and subsections, emphasize, quotes, lists, etc. Our program maintains an internal data structure that stores both the text as well as the typography, the markup, of the text. When it comes to saving the data structure I would like to be able to save in different formats so it is possible to export my document to other word processing programs or save it as a web page.

For example, I would like it to be able to save in XML, like:

```
<document name=''The Pattern Book''>
  <section name=''Builder''>
    <paragraph>
      This is my section on builder.
    </paragraph>
    <subsection name= ``Analysis''>
      <paragraph>
        Here is my analysis of builder.
      </paragraph>
    </subsection>
  </section>
</document>
```

And the same document could be output in HTML in which the section would instead be encoded as:

```
<H1>Builder</H1>
```

while the section in ASCII would perhaps look like:

```
Builder
=======
```

The problem is that if I write software to build each representation then much of it would be the same code in all variants, namely the code that iterates through the data structure and determine the type of typographical node, like "document", "section", etc. Only the parts that build the actual output representation would differ: HTML, XML, ASCII, etc. Thus I once again face a multiple maintenance problem if I program each formatter from scratch.

22.2 A Solution

Thus it is a classic variability problem and when I apply the three principles and the ③-①-② process the analysis goes like this:

③ All outputs consist of the same set of "parts" (section, subsection, paragraphs, etc.) but how the parts are built varies. That is, concrete construction of the individual node is variable.

① I encapsulate the "construction of parts" in a **builder** interface. A builder interface must have methods to build each unique part: in our case methods like buildSection, buildQuote, etc. Instances realizing this interface must be able to construct concrete parts to be used in the data structure.

② I write the data structure iterator algorithm once, the **director**, and let it request a delegate builder to make the concrete parts as it encounters them.

Figure 22.1 illustrates how the word processor can save in, say, HTML format. The word processor first creates a HTML builder object (denoted b in the diagram), and next ask its associated director object to start constructing. The director contains the common iteration over the data structure containing the document, and request the builder to build the specific parts. Once the construction process is finished, the word processor requests the built output from the concrete builder.

☞ Study the source code provided in folder *chapter/builder* on the web site.

22.3 The Builder Pattern

This is the BUILDER design pattern (design pattern box 22.1, page 29). Its intent is

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

The two main roles are the **Director** that contains the construct() method responsible for iterating the data structure, and the **Builder** that contains the methods to build each particular part of the product. **ConcreteBuilder**s implement the particular representation of the **Product**, that is the output of the construction phase. Note that it is the responsibility of the concrete builders to create the product and ultimately give the client access to it when it is finished: only concrete builders have getResult methods. Finally the **Client** is responsible for creating the concrete builder, ask the director to construct and finally to retrieve the product from the builder.

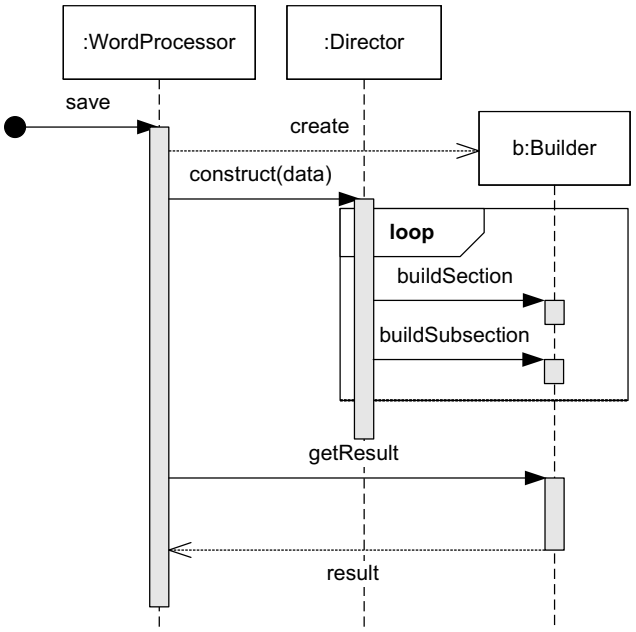


Figure 22.1: Saving a document using a builder pattern.

Exercise 22.1: Argue why it is not possible to put the getResult method in the Builder interface itself.

Separating the construction process and the concrete part building opens up for using builders for other purposes than pure construction. For instance you can easily do statistics with builders, like in the document case above where a CountingBuilder may count the number of sections, subsections, and paragraphs. Another benefit of the separation is a well-known consequence of the ② *favor object composition* principle, namely that each of the two roles becomes more focused and thus more cohesive leading to better maintainability and overview of the code.

BUILDER is a creational pattern, ie. its purpose is to create objects. ABSTRACT FACTORY is also a creational pattern. Compared to abstract factory, builder provides much more fine-grained control over the individual elements of the product. Thus, builders are typically used for creating complex data structures while abstract factory is typically for simple objects.

A liability of BUILDER is the relative complexity of setting up the construction process involving several different objects.

22.4 Selected Solutions

Discussion of Exercise 22.1:

The getResult method cannot be defined in the Builder interface because the concrete data structure to store the **Product** is not known. One concrete builder may want to

build a string representation, another a binary tree, and a third a bit image. This is also the reason that there must be a **Client** role whose responsibility it is to know the concrete type of builder whereas the **Director** does not.

22.5 Review Questions

Describe the BUILDER pattern. What problem does it solve? What is its structure and what is the protocol? What roles and responsibilities are defined? What are the benefits and liabilities?

22.6 Further Exercises

Exercise 22.2. Source code directory:
`chapter/builder`

Extend the builder example.

1. Make an XML builder that builds a XML representation of the document, e.g. uses tags like

   ```
   <HEADER>The Builder Pattern</HEADER>
   ```

   etc.

2. Make a builder that counts the total number of words in the document.

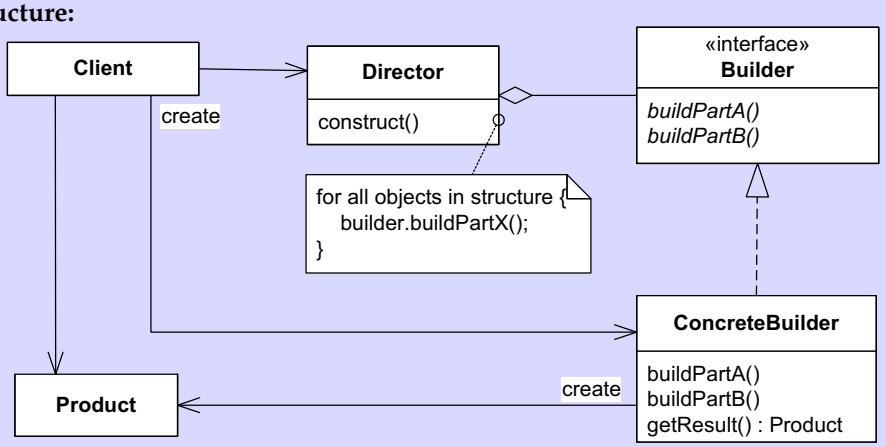
[22.1] Design Pattern: Builder

Intent Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Problem You have a single defined construction process but the output format varies.

Solution Delegate the construction of each part in the process to a builder object; define a builder object for each output format.

Structure:



Roles **Director** defines a building process but constructing the particular parts is delegated to a **Builder**. A set of **ConcreteBuilder**s is responsible to building concrete **Product**s.

Cost - Benefit It is *easy to define new products* as you can simply define a new builder. The *code for construction and representation is isolated*, and thus multiple directors can use builders and vice versa. Compared to other creational patterns (like ABSTRACT FACTORY) products are not produced in "one shot" but stepwise meaning you have *finer control over the construction process.*