The Addison-Wesley Signature Series

A Martin Fowler Signature Book

Refactoring to Patterns

Joshua Kerievsky

software development 15th annual productivity award

Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts

List of Refactorings

factory method declaration in the superclass into a concrete factory method that performs the default ("majority case") instantiation behavior.

✔ Compile and test.

Example

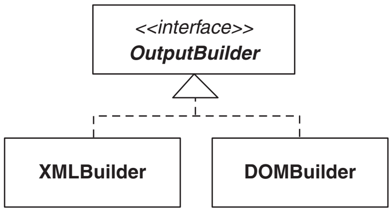
In one of my projects, I had used test-driven development to produce an XML-Builder—a Builder [DP] that allowed clients to easily produce XML. Then I found that I needed to create a DOMBuilder, a class that would behave like the XML-Builder, only it would internally produce XML by creating a Document Object Model (DOM) and give clients access to that DOM.

To produce the DOMBuilder, I used the same tests I'd already written to produce the XMLBuilder. I needed to make only one modification to each test: instantiation of a DOMBuilder instead of an XMLBuilder:

```
public class DOMBuilderTest extends TestCase...
  private OutputBuilder builder;

  public void testAddAboveRoot() {
   String invalidResult =
   "<orders>" +
     "<order>" +
     "</order>" +
   "</orders>" +
   "<customer>" +
   "</customer>";
   builder = new DOMBuilder("orders");  // used to be new XMLBuilder("orders")
   builder.addBelow("order");
   try {
     builder.addAbove("customer");
     fail("expecting java.lang.RuntimeException");
   } catch (RuntimeException ignored) {}
  }
```

A key design goal for DOMBuilder was to make it and XMLBuilder share the same type: OutputBuilder, as shown in the following diagram.



After writing the DOMBuilder, I had nine test methods that were nearly identical on the XMLBuilderTest and DOMBuilderTest. In addition, DOMBuilderTest had its own unique tests, which tested access to and contents of a DOM. I wasn't happy with all the test-code duplication, because if I made a change to an XMLBuilder-Test, I needed to make the same change to the corresponding DOMBuilderTest. I knew it was time to refactor to the Factory Method. Here's how I went about doing that work.

1. The similar method I first identify is the test method, testAddAboveRoot(). I extract its instantiation logic into an instantiation method like so:

```
public class DOMBuilderTest extends TestCase...
  private OutputBuilder createBuilder(String rootName) {
    return new DOMBuilder(rootName);
  }

  public void testAddAboveRoot() {
    String invalidResult =
    "<orders>" +
      "<order>" +
      "</order>" +
    "</orders>" +
    "<customer>" +
    "</customer>";
    builder = createBuilder("orders");
    builder.addBelow("order");
    try {
      builder.addAbove("customer");
      fail("expecting java.lang.RuntimeException");
    } catch (RuntimeException ignored) {}
  }
```

   Notice that the return type for the new createBuilder(…) method is an OutputBuilder. I use that return type because the sibling subclass, XMLBuilderTest, will need to define its own createBuilder(…) method (in step 2) and I want the instantiation method's signature to be the same for both classes.

   I compile and run my tests to ensure that everything's still working.

2. Now I repeat step 1 for all other sibling subclasses, which in this case is just XMLBuilderTest:

```
public class XMLBuilderTest extends TestCase...
  private OutputBuilder createBuilder(String rootName) {
    return new XMLBuilder(rootName);
  }

  public void testAddAboveRoot() {
    String invalidResult =
      "<orders>" +
        "<order>" +
        "</order>" +
      "</orders>" +
      "<customer>" +
      "</customer>";
      builder = createBuilder("orders");
      builder.addBelow("order");
      try {
        builder.addAbove("customer");
        fail("expecting java.lang.RuntimeException");
      } catch (RuntimeException ignored) {}
    }
```

I compile and test to make sure the tests still work.

3. I'm now about to modify the superclass of my tests. But that superclass is TestCase, which is part of the JUnit framework. I don't want to modify that superclass, so I apply *Extract Superclass* [F] to produce AbstractBuilderTest, a new superclass for my test classes:

```
public class AbstractBuilderTest extends TestCase {
}

public class XMLBuilderTest extends AbstractBuilderTest...

public class DOMBuilderTest extends AbstractBuilderTest...
```

4. I can now apply *Form Template Method (205)*. Because the similar method is now identical in XMLBuilderTest and DOMBuilderTest, the *Form Template Method* mechanics I must follow instruct me to use *Pull Up Method* [F] on testAdd-AboveRoot(). Those mechanics first lead me to apply *Pull Up Field* [F] on the builder field:

```
public class AbstractBuilderTest extends TestCase {
  protected OutputBuilder builder;
}

public class XMLBuilderTest extends AbstractBuilderTest...
  ~~private OutputBuilder builder;~~

public class DOMBuilderTest extends AbstractBuilderTest...
  ~~private OutputBuilder builder;~~
```

Continuing with the *Pull Up Method* [F] mechanics for testAddAboveRoot(), I now find that I must declare an abstract method on the superclass for any method that is called by testAddAboveRoot() and present in the XMLBuilderTest and DOMBuilderTest. The method, createBuilder(…), is such a method, so I pull up an abstract method declaration of it:

```
public abstract class AbstractBuilderTest extends TestCase {
  protected OutputBuilder builder;

  protected abstract OutputBuilder createBuilder(String rootName);
}
```

I can now proceed with pulling up testAddAboveRoot() to AbstractBuilderTest:

```
public abstract class AbstractBuilderTest extends TestCase...
  public void testAddAboveRoot() {
    String invalidResult =
    "<orders>" +
      "<order>" +
      "</order>" +
    "</orders>" +
    "<customer>" +
    "</customer>";
    builder = createBuilder("orders");
    builder.addBelow("order");
    try {
      builder.addAbove("customer");
      fail("expecting java.lang.RuntimeException");
    } catch (RuntimeException ignored) {}
  }
```

That step removed testAddAboveRoot() from XMLBuilderTest and DOMBuilderTest. The createBuilder(…) method, which is now declared in AbstractBuilderTest and implemented in XMLBuilderTest and DOMBuilderTest, now implements the Factory Method [DP] pattern.

As always, I compile and test my tests to make sure that they still work.

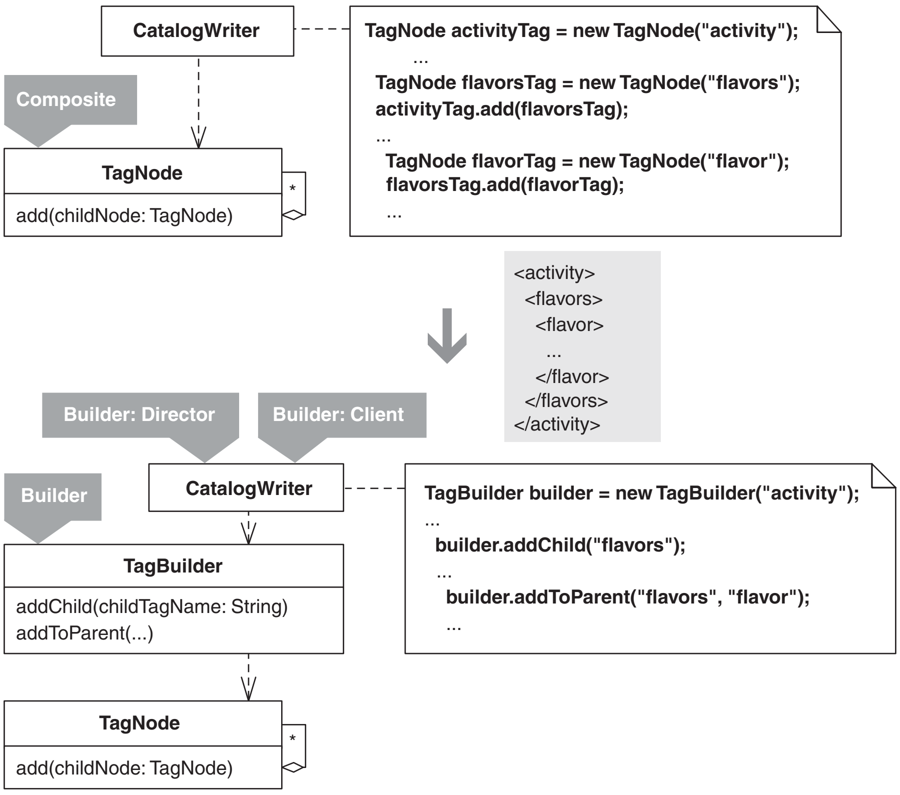
5. Since there are additional similar methods between XMLBuilderTest and DOMBuilderTest, I repeat steps 1–4 for each similar method.

6. At this point I consider creating a default implementation of createBuilder(…) in AbstractBuilderTest. I would only do this if it would help reduce duplication in the multiple subclass implementations of createBuilder(…). In this case, I don’t have such a need because XMLBuilderTest and DOMBuilderTest each instantiate their own kind of OutputBuilder. So that brings me to the end of the refactoring.

Encapsulate Composite with Builder

Building a Composite is repetitive, complicated, or error-prone.

Simplify the build by letting a Builder handle the details.



```
TagNode activityTag = new TagNode("activity");
    ...
 TagNode flavorsTag = new TagNode("flavors");
 activityTag.add(flavorsTag);
 ...
  TagNode flavorTag = new TagNode("flavor");
  flavorsTag.add(flavorTag);
  ...
```

```
TagBuilder builder = new TagBuilder("activity");
...
 builder.addChild("flavors");
 ...
  builder.addToParent("flavors", "flavor");
  ...
```

Motivation

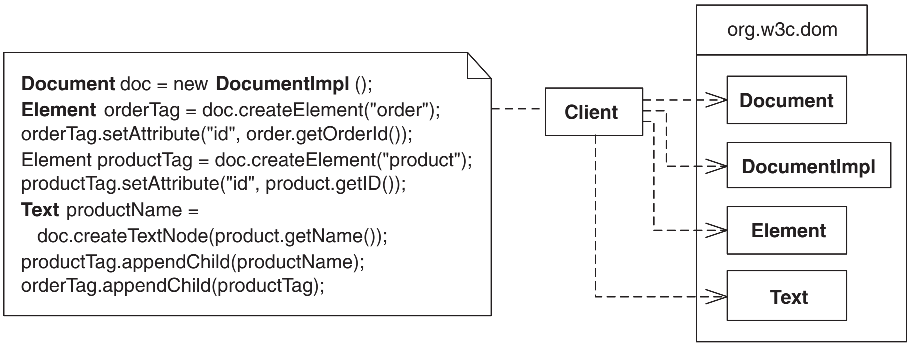
A Builder [DP] performs burdensome or complicated construction steps on behalf of a client. A common motivation for refactoring to a Builder is to simplify client code that creates complex objects. When difficult or tedious parts of creation are implemented by a Builder, a client can direct the Builder's creation work without having to know how that work is accomplished.

Builders often encapsulate Composites [DP] because the construction of Composites can frequently be repetitive, complicated, or error-prone. For example, to add a child node to a parent node, a client must do the following:

- Instantiate a new node
- Initialize the new node
- Correctly add the new node to the right parent node

This process is error-prone because you can either forget to add a new node to a parent or you can add the new node to the wrong parent. The process is repetitive because it requires performing the same batch of construction steps over and over again. It's well worth refactoring to any Builder that can reduce errors or minimize and simplify creation steps.

Another motivation for encapsulating a Composite with a Builder is to decouple client code from Composite code. For example, in the client code shown in the following diagram, notice how the creation of the DOM Composite, `orderTag`, is tightly coupled to the DOM's `Document`, `DocumentImpl`, `Element`, and `Text` interfaces and classes.



```
Document doc = new DocumentImpl ();
Element orderTag = doc.createElement("order");
orderTag.setAttribute("id", order.getOrderId());
Element productTag = doc.createElement("product");
productTag.setAttribute("id", product.getID());
Text productName =
  doc.createTextNode(product.getName());
productTag.appendChild(productName);
orderTag.appendChild(productTag);
```

Such tight coupling makes it difficult to change Composite implementations. On one project, we needed to upgrade our system to use a newer version of the DOM, which happened to have several differences from the DOM 1.0 version we had been using. This painful upgrade involved changing many lines of Composite-construction code that were spread across the system. As part of the upgrade, we encapsulated the new DOM code within a `DOMBuilder`, as shown in the diagram on the following page.