Toward Refactoring in a Polyglot World

Extending Automated Refactoring Support Across Java and XML

Nicholas Chen, Ralph Johnson
University of Illinois at Urbana-Champaign
{nchen,johnson}@cs.uiuc.edu

Abstract

Many automated refactoring tools only work with one programming language. However, most complex programs rely on polyglot programming — entailing the use of different programming languages, libraries and frameworks. While finding a *general* solution for extending refactoring across multiple languages is *hard*, it is *simple* and *possible* to support automated refactorings for some common cases that programmers already encounter in their programs today. Supporting those common cases is essential in encouraging programmers to make changes that would increase the understandability and maintainability of their programs.

This paper is a step toward encouraging tool developers to think about supporting multi-language refactorings. We present our tool for supporting multi-language refactorings from studying the interactions between Java and XML configuration files. Our tool extends *Rename Refactoring* support across three different popular Java frameworks: Struts, Hibernate and Spring.

1. Introduction

Refactorings are behavior-preserving program transformations that make a program more understandable and maintainable [4]. While complex refactorings require programmers to perform them by hand, many less-complex refactorings can be *automated* using some tool.

The first tool to support automated refactorings was the Refactoring Browser for Smalltalk [8]. In Smalltalk, *everything* is written in Smalltalk itself. Essentially, the Refactoring Browser needs to work only in a single language.

Contrast this to current programs written in languages such as Java. Such programs rely on various libraries and frameworks. Each library or framework might employ its own *domain-specific language*. While the majority of the program is in Java, non-trivial components that are not written in Java exist as well. Currently, executing an automated refactoring such as *Rename Refactoring* on a Java class would preserve the behavior in the Java components but would break the non-Java components, requiring the programmer to manually perform the remaining changes by hand — a task that is both tedious and error-prone.

Ideally, an automated refactoring tool should also extend refactoring support to those non-Java components. Though it is well-known that refactoring a Java component could require changes in other non-Java components, little has been done to study how to support such multi-language refactorings. Many programs today already rely on polyglot programming. Without proper support for automated refactoring across multiple-languages, maintaining such programs would be onerous.

Presently, most frameworks use XML as their domain-specific language for configuration. Like other programming languages, valid XML requires proper syntax (as specified by the XML standards) and semantics (as specified by the schema of the XML document). Moreover, XML is the most common domain-specific language for Java frameworks.

The similarities of XML with other languages and its ubiquity make it a suitable *model language* for studying the feasibility of multi-language refactoring support. Its similarities with other languages imply that the technique we develop would be applicable for other languages. And its ubiquity implies that our tool can be used for many different systems already out there.

In this paper, we present our tool for extending refactoring support from Java to XML configuration files under the Eclipse IDE [2]. In Section 2, we present an example of how our tool works for refactoring a Java project that uses the Struts framework. And in Section 3, we describe the general principle behind our tool. The current version of our tool extends *Rename Refactoring* across Java and XML configuration files and we are adding support for refactorings that require moving or deleting Java components.

WRT'08, October 19, 2008, Nashville, Tennessee, USA.

2. Example — Supporting Struts Refactoring

We have implemented our tool for three popular Java frameworks: Struts, Hibernate and Spring. In fact, our general technique was distilled from those *three examples* [9] as we implemented and refined our implementations of them. In this paper, we focus on the Struts framework and detail the steps that were taken to extend refactoring support for Struts in the Eclipse IDE.

Struts [1] is one of the most popular open-source Java frameworks for creating Java web applications. It follows the Model-View-Controller architecture and decouples each component to promote maintainability. It relies on XML configuration files to "wire" those different components together.

Struts is a mature and robust framework with many different features. In this section, we focus on how we extended refactoring support to its Model component. Below is a walkthrough of our implementation for this component:

Trigger The programmer decides to rename the Java class `SearchForm` to `MySearchForm`. He invokes the *Rename Refactoring* operation from Eclipse. This operation renames the Java class and triggers our Struts refactoring participant to perform necessary changes to the XML configuration files.

Filter relevant XML files Once triggered, our participant goes through and locates *all* the XML files in the current working directory. The participant goes through all the files and checks the DOCTYPE declaration to see if any file matches the declaration for the Struts framework. Listing 1 shows an example of a valid declaration for Struts.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE struts-config PUBLIC
3 "-//DTD Struts Configuration 1.3//EN"
4 "http://apache.org/struts-config_1_3.dtd">
```

Listing 1. Struts DOCTYPE (actual declaration has been truncated to fit)

Analyze relation in XML files The participant analyzes the list of files from the previous step to check if any file references the newly refactored file i.e. SearchForm. To do so, it must first resolve the fully-qualified name of the class to `edu.uiuc.nchen.SearchForm`. It then searches each XML file for such a reference. In this case, this search can be expressed in the form of an XPath[10] query: `//form-bean[@type="edu.uiuc.nchen.SearchForm"]`. Executing this query matches the XML node on lines 4-5 in Listing 2.

```
1  ...
2  <struts-config>
3    <form-beans>
4      <form-bean name="searchForm"
5          type="edu.uiuc.nchen.SearchForm"/>
6    </form-beans>
7    <action-mappings>
8      <action path="/search"
9          type="edu.uiuc.nchen.SearchAction"
10         name="searchForm"
11         scope="request" validate="true"
12         input="/search.jsp">
13     </action>
14   </action-mappings>
15         ...
16 </struts-config>
```

Listing 2. Partial listing of struts XML configuration file **before** refactoring

Create refactorings The participant now creates a change that modifies the term `edu.uiuc.nchen.SearchForm` to `edu.uiuc.nchen.MySearchForm` on line 5 of Listing 2.

Trigger additional refactorings However, our participant is not done yet. On line 4 of Listing 2, the programmer has given a reference name — `searchForm` — to the class. This reference name is no longer up-to-date. While leaving it as such is valid and will continue to work, it is better to refactor its name to improve understandability. Our participant presents the programmer with a user interface for renaming `searchForm`. Assuming that the programmer renames the reference to `mySearchForm`, our participant now has to create new internal and external refactorings:

- *Internal refactoring*
  Line 10 of Listing 2 (the current XML configuration file) holds a reference to the old `searchForm` name. Therefore, our participant needs to create a change that modifies the term `searchForm` to `mySearchForm` on that line.
- *External refactoring*
  In addition, the old name might be referenced in other files as well. In our example, the old `searchForm` name is referenced from one of the JSP files (we use the same technique of Section 3 to locate such references[1]). This is shown on line 8 of Listing 3. Our participant needs to create a change that modifies the term `searchForm` to `mySearchForm` on that line too.

```
1  <%@ taglib
2    uri="http://apache.org/tags-logic"
3    prefix="logic"%>
4
5  <html>
6  <body>
7
8  <logic:present name="searchForm"
9    property="queries">
10 ...
11 </logic:present>
12
13 </body>
```

[1] JSP files are very similar to XML files. In fact, we convert the JSP file to XML [3] so that we can use the same XML tools to perform the analysis.

```
</html>
```

Listing 3. Partial listing of JSP file **before** refactoring

Perform refactorings Finally our participant presents the changes that it will perform to the programmer. Upon accepting those changes, the XML configuration file will be refactored to its final version as shown in Listing 4. In addition, the JSP file will also be refactored to its final version as shown in Listing 5.

```
...
<struts-config>
  <form-beans>
    <form-bean name="mySearchForm"
        type="edu.uiuc.nchen.MySearchForm"/>
  </form-beans>
  <action-mappings>
    <action path="/search"
        type="edu.uiuc.nchen.SearchAction"
        name="mySearchForm"
        scope="request" validate="true"
        input="/search.jsp">
    </action>
  </action-mappings>
        ...
</struts-config>
```

Listing 4. Partial listing of struts XML configuration file **after** refactoring

```
<%@ taglib
   uri="http://apache.org/tags-logic"
   prefix="logic"%>

<html>
<body>

<logic:present name="mySearchForm"
   property="queries">
...
</logic:present>

</body>
</html>
```

Listing 5. Partial listing of JSP file **after** refactoring

This example illustrates how our tool works in the context of the Struts framework. While the general idea of our technique is applicable for the different frameworks that we have investigated, each framework has its own subtle features that require special consideration to ensure that the program not only works after the refactoring but also maintains its readability and maintainability by conforming to standard conventions. In the case of Struts, it entails refactoring any reference in both the current XML configuration file and any referencing JSP files as well.

Our example show that even a simple rename refactoring on a Java class requires changes in multiple files. Without such a tool, the programmer would have to refactor the XML configuration file and its related JSP files by hand — a process that a tedious and error-prone.

Moreover, if the programmer makes a mistake while manually refactoring the XML file, the error is only detectable during runtime when the framework loads. So an error could be in the system for an extended time between its *insertion* and *detection*.

3. Technique

The main idea for multi-language refactorings relies on *detection* and *propagation*. There has to be a way to detect when an initial refactoring has been initiated by the user in one component and subsequently a way to propagate a chain of refactorings in other components.

Our implementation for extending refactorings across XML files employs the *Observer* design pattern [5]. The *subject* is the refactoring operation on Java files. And the *observers* are refactoring operations to be performed on the XML files. Once a refactoring on our subject is triggered, its observers are notified. Active *observers* would then proceed to refactor the related XML files.

The **one-to-many** dependency between a *subject* and its *observers* is a important requirement for our technique. It is possible — and very likely — that a Java program uses multiple frameworks. Each framework would require the use of its own XML configuration file. Thus, a refactoring operation on a Java file could trigger changes in zero or more XML files. So it is important that multiple observers can be registered for each subject.

Currently, our implementation is *uni-directional*. It only observes the refactoring of Java files and proceeds to refactor the XML files as necessary; it does not observe refactorings on XML files and thus cannot proceed to refactor the Java files as necessary. This limitation stems from the absence of support for refactoring operations on XML files in Eclipse. Thus, it is, currently, not possible to observe when a refactoring has occurred in a XML file and take the appropriate actions to extend that refactoring to Java files. *Bi-directional* support can be added once a full-fledged XML editing environment has been implemented in Eclipse.

Here are the general steps that each observer performs upon receiving a notification from its subject.

1. Filter relevant XML files — The observer locates all the XML files in the current working directory. It then goes through this list of files and filters those that it can handle. It determines the files that it can handle by parsing the DOCTYPE or XML schema declaration in the header of each of the XML files. All *valid* XML files must have DOCTYPE or XML schema declarations so this is a reasonable method for checking each file.
   This step is important in promoting *composability*. A program might use multiple frameworks and it is important that each observer only handles a file that it is configured for. By having this check, we enable multiple observers to take part in the refactoring without interfering with one another.

2. Analyze relation in XML files — The observer now goes through each of the XML files from the filtered list and checks to see if the file needs to be refactored. The analysis is done by comparing the information inside the XML file with the Java element that has just been refactored. For instance, if a Java class has been renamed, then all references to it from the XML file has to be refactored as well. For simple analysis, it is sufficient to rely on XPath queries to locate the relevant XML elements that might be affected. XML files that are not affected by the refactored Java element are ignored.
3. Create refactorings — After gathering the affected XML files, the observer proceeds to create the refactoring changes that need to be performed on the XML files. No change is actually performed at this stage.
4. Trigger additional refactorings — It is possible that additional refactorings might need to be performed both *internally* and *externally* to keep our program consistent with the new changes. Internal refactorings affect the current XML file and involve refactoring some other XML node that might be indirectly involved with the current refactoring. External refactorings affect other files that are indirectly involved with the refactoring. These additional refactorings are usually optional (since the behavior of the program is still preserved even without performing them) though they enhance the understandability of the underlying program. Refer to Section 2 for examples.
5. Perform refactorings — Before actually performing the refactorings, the observer presents a preview of the changes to the programmer. The programmer may decide to continue or abort the current refactoring operation.

Our implementation of this technique for extending refactoring was done in Eclipse by leveraging its support for *Refactoring Participants* [7] which were introduced in Eclipse version 3.2. A refactoring participant plays the role of the *observer* in our technique. In our current implementation, we create *Rename Participants* that take part in a *Rename Refactoring* on an existing Java class. Although refactoring participants have been available since Eclipse version 3.2, to the best of our knowledge, our implementation is the first to use them to perform multi-language refactorings.

This same technique can be employed for different IDEs either by starting from scratch or by using their existing support for user-contributed refactorings.

We have successfully applied this technique to two other frameworks — Hibernate and Spring — but because of space constraints we are only describing our implementation for the Struts framework.

4. Related Work

IntelliJ IDEA, a commercial IDE, first introduced the ability to support refactoring across Java and the different frameworks in version 7 of its product. However, because it is a close-source commercial product, we do not know what technique was used to support refactoring across Java and XML configuration files.

5. Conclusion

Programs today no longer exist in a monolingual world. Instead, they are crafted under a polyglot environment, utilizing many different languages. Naturally, our tools should also extend to this new paradigm for programming. Just as automated refactoring tools have lessen the burden of refactoring for programmers using a single language, new tools should be created to do the same for programmers using multiple languages.

We have presented a general technique that allows the extension of automated refactoring support across XML configuration files. While we have implemented the technique for XML configuration files, we believe that this technique is simple and applicable for other domain-specific languages. By providing support for the common cases of refactoring, we have made the task of refactoring much easier for programmers and, therefore, they are more likely to perform those refactorings since they are confident that their programs will continue to work after such operations.

References

[1] Apache Foundation Struts Framework. http://struts.apache.org

[2] Eclipse Foundation. http://eclipse.org

[3] Creating a JSP Document http://java.sun.com/javaee/5/docs/tutorial/doc/bnajq.html

[4] Martin Fowler, Kent Beck, John Brandt, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[5] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 1995.

[6] IntelliJ IDEA. http://www.jetbrains.com/idea/

[7] The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs http://www.eclipse.org/articles/Article-LTK/ltk.html

[8] Don Roberts, John Brandt and Ralph Johnson. A Refactoring Tool for Smalltalk. *Theor. Pract. Object Syst.*. 3(4):253263, 1997

[9] Don Roberts and Ralph Johnson. Evolving frameworks: A Pattern Language for Developing Object-Oriented Frameworks. In *Pattern Languages of Program Design 3*. Addison Wesley, 1997.

[10] XML Path Language (XPath). http://www.w3.org/TR/xpath