HotSpotter: a JavaML-based approach to discover Framework's HotSpots

Nuno Flores , Diana Soares, Helder Ferreira, Marco Rodrigues

Faculdade de Engenharia Universidade do Porto
R. Dr. Roberto Frias s/n,
4200-465 Porto, Portugal
{mei03009, mei03014, mei03001, mei03016}@fe.up.pt

Abstract. Object-oriented frameworks are designed with reusability as the main design concern. In order to develop concrete solutions by customizing existing frameworks, developers must be aware of its areas of flexibility, which are commonly called hot-spots. A common problem with framework reuse is the extension and detail of existing documentation, often resulting insufficient to communicate the level of knowledge the developer needs to be effective in reusing the framework. This paper addresses this problem by providing a tool that detects those hot-spots, called the HotSpotter, thus enabling the developer to rapidly identify where is possible to customize the framework in order to obtain the desired solution. The HotSpotter is a tool targeted for frameworks written in Java, and it follows a multi-phased process that starts from a JavaML base representation of the framework's source-code and evolves through a series of XSL transformations until reaching the desired results. This approach intends to uncover the hot-spots of frameworks or applications, by identifying templates and hooks and then grouping them using predefined heuristics. The results obtained with the XML-based version of the HotSpotter tool was compared and validated against those of a similar existing tool, using the JUnit framework as a case-study.

1 Introduction

Object-oriented application frameworks are intended to provide software for reuse [4]. Their generic design within a given domain and its reusable implementation spurs rapid development of software solutions. Despite this great potential for reuse, its practical feasibility can only be reached if the framework design becomes thoroughly understood and its extensibility parts are clearly recognizable. The complexity behind the design and implementation of the framework makes it difficult to grasp its flexibility, forcing a good documentation support vital. Most often, the existing documentation lacks in extent and detail and is insufficient to make the developer aware of its flexibility.

This paper addresses the need to such awareness by providing a tool that detects those areas of flexibility, called the hotspots, thus enabling the developer to rapidly identify where to act upon, configuring the framework to evolve to the desired solu-

tion. Due to its potentially large size and complexity, the ability to quickly understand and apply a framework is a critical issue [4] in software development.

The HotSpotter is a tool targeted for frameworks written in Java and which goal is to detect and show the framework's hot-spots. It follows a multi-phased process that starts from a JavaML base representation of the framework's source-code and evolves through a series of XSL transformations until reaching the desired results. This approach intends to uncover the hot-spots of frameworks or applications, by identifying templates and hooks and then grouping those using predefined heuristics.

The development of this approach was part of a dual-approach-based project using JavaML [6] and Eclipse [2] in parallel. The results obtained with the XML-based version of the HotSpotter tool were compared and validated against those of the similar Eclipse tool, using the JUnit [3] framework as a case-study.

Section 2 gives a brief explanation on the concepts of hotspots, template and hook methods and their relationships. In order to maintain expressiveness, a few considerations had to be made and specific domain definitions were adopted.

Section 3 describes the goals and the development methodology: a multi-step process aiming at detecting and recognizing the relevant reusability structures and the appropriate way of aggregating the results into two views for broader understanding.

In Section 4, all the process steps are explained in detail. Beginning at the JavaML source code representation, several XSL transformations are applied accordingly to achieve the desired outcome.

Section 5 illustrates some results after applying the process to the JUnit framework, whereas Section 6 presents the conclusions and the forthcoming work.

2 HotSpots, Template Methods and Hooks Relationships

As a concept, a hotspot has its foundations built upon a framework's Open-Closed Principle [15]. This attribute describes the framework's potential flexibility and proneness to reuse. The principle encompasses two definitions: the "closed" and the "open" parts. The latter represents the areas that are variant, configurable and redefinable, whereas the former describes those areas that remain immutable, yet group the variant parts together [14].

This combination of open and closed parts embodies the framework's reusability regions, the so called hotspots. The notions of *template methods* and *hook methods* support adherence to the Open-Closed Principle as they materialize the concepts of "open" and "closed". A template ("closed") method defines an invariant part of the framework which links ("calls") the hook ("open") methods together. These hook methods are the re-definable elements to where the user should aim at, thus adapting the framework to provide a particular solution.

Cardino [1] states that "a reuser should not have to worry […], since hotspots should be sufficient to customize the framework to new needs". That is to say that the awareness and understanding of hotspots is crucial to exploit the extensibility aspects of a framework. Identifying these extensibility regions heightens the development speed and improves the quality of the process [4].

According to [14], the relationship between templates and hooks translates into what is called *meta-patterns*. These relationships can be observed through the analysis of several design patterns [13]. Knowing the composing patterns of a framework's design enables the assertion of its degree of reusability. Starting from a lower level of abstraction, one must first identify the templates and hooks contained in a framework and then identify its relationships. Albeit seven known meta-patterns exist, as far as hotspots are concerned, template methods and hook methods can be combined by either inheritance or composition [12]:

1. Through Inheritance, abstract hook methods, defined in the same class as the template method, are overridden in descendant classes (Fig. 1).
2. Through Composition, hook methods are defined in interface classes which are subsequently implemented by one or more concrete classes. The template class calls the hook methods defined in the interface, not knowing which concrete class is implementing its behaviour. (Fig. 2). Many GoF design patterns base their flexibility on this concept, such as Builder, Strategy or Bridge [13].

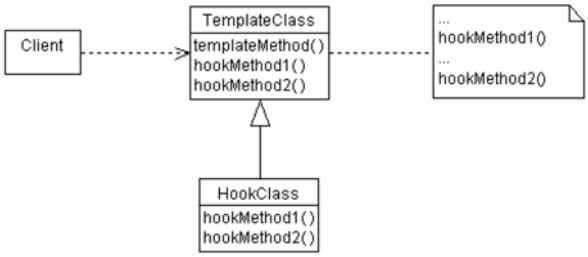

Fig. 1. Inheritance Template Method (IHS).

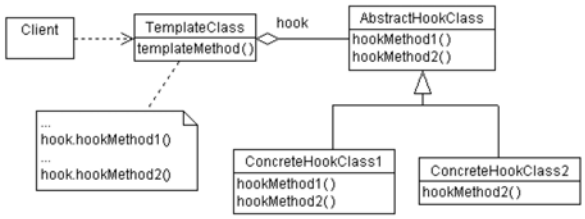

Fig. 2. Composition Template Method (CHS).

Upon template and hook detection and their relationship, one can group them into hotspots. This raises a concern: expressiveness. This issue is addressed by [12] which come up with a definition of hotspot, neither too fine-grained nor too course-grained:

[a hotspot is] a set of hook methods and their associated template methods, in which each hook method is invoked by exactly the same set of template methods.

From this definition of a hotspot (Fig. 3), Schauer [12] divides them into two sub-categories: IHS (Inheritance hotspots) and CHS (Composition hotspots). IHS stands for hotspots exclusively based on Inheritance Template Methods (Fig. 1), whereas CHS stands for hotspots exclusively based on Composition Template Methods (Fig. 2).

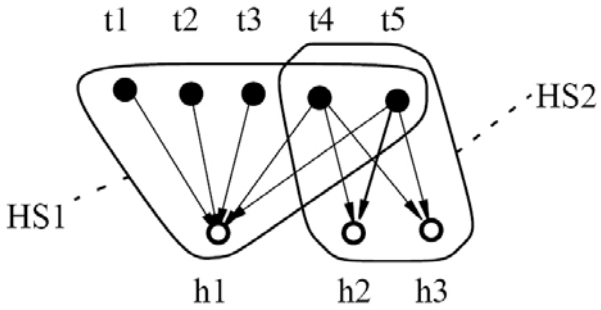

Fig. 3 Hot Spot definition, according to [12].

3 Goals and development methodology

The process of developing a tool for detecting and visualizing hotspots undertook three successive steps:

1. *Source Code Parsing*. The primary step would be to parse through the source code and identify the relevant elements, that is, candidate methods and their enclosing classes that might be classified as templates or hooks.

2. *Template/Hook detection heuristics*. Upon gathering the filtered source code elements, its structure and relationships would have to be confronted against the definitions of templates and hooks. Namely, methods that call other methods defined in descendant classes (IHS approach) or in interface classes (CHS approach) would be scrutinized.

3. *Aggregate into hotspots and generate results*. Coupling templates and hooks would provide candidate hotspots. In order to conform to the pre-defined hotspot definition, an aggregation step would have to take place to broaden the scope of the detected areas of flexibility. Without loss of information, the collapsing of contiguous candidate hotspots into a single element would define wider regions of reuse. Consequentially, granularity would decrease and a better vision of the overall flexibility would be easily acquired.

While detecting templates and hooks, two views emerged from the progressing development and occurring results. Amidst the process, it made sense to come up with two kinds of aggregation heuristics for candidate hotspots:

1. "Vista1" (View1). *Each template method calls exactly the same hook methods, grouping by the higher number of hook methods as possible*. That is, templates and hooks are joined together into a container-content structure. A template "contains" (calls) a certain number of hooks. Aggregation of the final hotspots is then made by

intersecting the template methods through their hook set. The new hotspot is generated by maximizing the number of hooks methods intersected.

2. “Vista2” (View2). *Each hook method is called by exactly the same template methods*. In this case, templates and hooks are joined together into a container-content structure, having the hook as the container. The hook “contains” the templates which call it. Aggregation of the final hotspots is made by intersecting the hooks through their templates set. The new hotspot is generated by grouping only those hooks that have exactly the same templates as callers.

In the next section, a further insight of the JavaML approach will enlighten the purpose behind these views.

4 JavaML Approach

The lack of a canonical structured representation of the java source code, and its convenient plain-text representation to the programmers induced the need for a universal format, able to directly represent the program structure and its contents. This transformation would supply other software tools with an easy platform for source code analysis and manipulation. XML [5] presented itself as a good solution and led to the creation of JavaML [6] [7]. The work presented in this paper followed a JavaML approach, as it easily allows the manipulation of Java source code, providing a simple and fast way of getting results.

4.1 System’s Overview

As mentioned before (see abstract) the JUnit framework was used as a case-study for HotSpotter. The first step in the JavaML approach consisted in the transformation of the JUnit java source code into JavaML files.

_____Extract of junit-all.java.xsl (abstract method declaration)_____

```
  <method name="testEnded" id="Ljunit/runner/BaseTestRunner;testEnded(Ljava/lang/String;)V"
idkind="method">
    <modifiers>
      <modifier name="public"/>
      <modifier name="abstract"/>
    </modifiers>
    <type name="void" primitive="true"/>
    <formal-arguments>
      <formal-argument name="testName" id="Ljunit/runner/BaseTestRunner;arg391" idkind="formal">
        <type name="String" idref="Ljava/lang/String;" idkind="type"/>
      </formal-argument>
    </formal-arguments>
  </method>
```

_____ Extract of junit-all.java.xsl (hook call by template method) _____

```
  <method name="endTest" id="Ljunit/runner/BaseTestRunner;endTest(Ljunit/framework/Test;)V"
idkind="method">
    <modifiers>
      <modifier name="public"/>
      <modifier name="synchronized"/>
    </modifiers>
    <type name="void" primitive="true"/>
    <formal-arguments>
      <formal-argument name="test" id="Ljunit/runner/BaseTestRunner;arg338" idkind="formal">
        <type name="Test" idref="Ljunit/framework/Test;" idkind="type"/>
      </formal-argument>
    </formal-arguments>
```

```
    <block>
      <send message="testEnded"
idref="Ljunit/runner/BaseTestRunner;testEnded(Ljava/lang/String;)V" idkind="method">
        <arguments>
          <send message="toString" idref="Ljava/lang/Object;toString()Ljava/lang/String;"
idkind="method">
            <target>
              <formal-ref name="test" idref="Ljunit/runner/BaseTestRunner;arg338"
idkind="formal"/>
            </target>
          <arguments/>
        </send>
      </arguments>
    </send>
    </block>
  </method>
```

The transformation was performed with the IBM Jikes Java Compiler [8] complemented with the JavaML Patch for Jikes [9]. Next, the XML files generated were packed into one single JavaML file with the root element `<javaml>`. Afterwards, several XSL transformations [10] for both IHS and CHS methods and both "Vista1" and "Vista2" views were applied, producing several XML outputs. Each of these last XML outputs were then transformed by a PHP [11] script that grouped the hooks and templates, producing a single final HTML presentation file containing the Hotspots found in the JUnit framework. Fig. 4 shows the overall structure of this process.

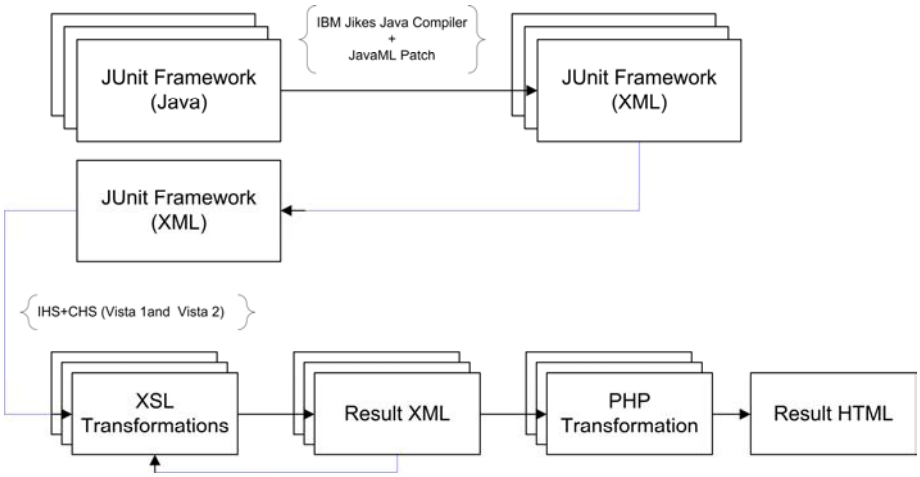

Fig. 4. System Overview of JavaML approach.

4.2 XSL Transformations

Several XSL files were developed with the purpose of extracting relevant information from the JavaML files. Each method (IHS and CHS) used the appropriate XSL.

For the IHS method the stylesheets do the following:

1. *Capture of Hooks and Templates*: templates are captured by searching the JavaML file for the `<method>` tag with the *abstract* attribute. Hooks are captured by searching for the `<send>` tag with a *message* attribute that refers the called method.

ihsa-vista1-getHooksandTemplates.xsl

```
<xsl:template match="javaml">
  <Java>
    <xsl:apply-templates select="//method/modifiers/modifier[@name='abstract']"/>
  </Java>
</xsl:template>

<xsl:template match="modifier">
  <xsl:param name="hookID" select="../../@id"/>
  <hook>
    <xsl:attribute name="name"><xsl:value-of select="../../@name"/></xsl:attribute>
    <xsl:attribute name="id"><xsl:value-of select="../../@id"/></xsl:attribute>
    <xsl:if test="ancestor::class/@name">
      <xsl:attribute name="class"><xsl:value-of select="ancestor::class/@name"/>
      </xsl:attribute>
    </xsl:if>
    <xsl:if test="ancestor::interface/@name">
      <xsl:attribute name="class"><xsl:value-of select="ancestor::interface/@name"/>
      </xsl:attribute>
    </xsl:if>
    <xsl:apply-templates select="//send[@idref=$hookID]"/>
  </hook>
</xsl:template>

<xsl:template match="send">
  <template>
    <xsl:attribute name="name"><xsl:value-of select="ancestor::method/@name"/></xsl:attribute>
    <xsl:attribute name="id"><xsl:value-of select="ancestor::method/@id"/></xsl:attribute>
    <xsl:attribute name="class"><xsl:value-of select="ancestor::class/@name"/></xsl:attribute>
  </template>
</xsl:template>
```

2. *Invert the position of Hooks and Templates*: this is only required for "Vista1" view and it concerns grouping hooks inside a template, and not the contrary.

For the CHS method:

1. *Capture of all abstract methods, all methods declared in interfaces and all variables used inside those methods*: the methods are captured in a similar way as in IHS; methods in interfaces are captured by searching the method declaration inside an `<interface>` tag and variables are captured by searching the tags `<field>`, `<local-variable-decl>` and `<formal-argument>` inside the captured methods.

chsa-vista1-getHooksandTemplates.xsl

```
<!-- Variables -->
<xsl:template match="field">
  <field>
    <xsl:attribute name="id"><xsl:value-of select="@id"/></xsl:attribute>
    <xsl:attribute name="class"><xsl:value-of select="type/@name"/></xsl:attribute>
  </field>
</xsl:template>
<xsl:template match="local-variable-decl">
  <field>
    <xsl:attribute name="id"><xsl:value-of select="local-variable/@id"/></xsl:attribute>
    <xsl:attribute name="class"><xsl:value-of select="type/@name"/></xsl:attribute>
  </field>
</xsl:template>
<xsl:template match="formal-argument">
  <field>
    <xsl:attribute name="id"><xsl:value-of select="@id"/></xsl:attribute>
    <xsl:attribute name="class"><xsl:value-of select="type/@name"/></xsl:attribute>
  </field>
</xsl:template>
<!-- Abstract methods -->
<xsl:template match="modifier">
  <xsl:param name="MethodId" select="../../@id"/>
  <xsl:for-each select="//send[@idref=$MethodId]">
    <hook>
      <xsl:attribute name="name"><xsl:value-of select="@message"/></xsl:attribute>
      <xsl:attribute name="id"><xsl:value-of select="@idref"/></xsl:attribute>
      <xsl:if test="descendant::field-ref/@idref">
        <xsl:attribute name="classid"><xsl:value-of select="descendant::field-ref/@idref"/>
        </xsl:attribute>
        <xsl:attribute name="classtype">varclass</xsl:attribute>
      </xsl:if>
      <xsl:if test="descendant::var-ref/@idref">
        <xsl:attribute name="classid"><xsl:value-of select="descendant::var-ref/@idref"/>
        </xsl:attribute>
```

```
        <xsl:attribute name="classtype">varlocal</xsl:attribute>
      </xsl:if>
      <xsl:if test="descendant::formal-ref/@idref">
        <xsl:attribute name="classid"><xsl:value-of select="descendant::formal-ref/@idref"/>
        </xsl:attribute>
        <xsl:attribute name="classtype">varlocal</xsl:attribute>
      </xsl:if>
      <template>
        <xsl:attribute name="name"><xsl:value-of select="ancestor::method/@name"/>
        </xsl:attribute>
        <xsl:attribute name="id"><xsl:value-of select="ancestor::method/@id"/></xsl:attribute>
        <xsl:attribute name="class"><xsl:value-of select="ancestor::class/@id"/>
        </xsl:attribute>
      </template>
    </hook>
  </xsl:for-each>
</xsl:template>
<!-- Interface methods -->
<xsl:template match="method">
  <xsl:param name="MethodId" select="@id"/>
  <xsl:for-each select="//send[@idref=$MethodId]">
    <hook>
      <xsl:attribute name="name"><xsl:value-of select="@message"/></xsl:attribute>
      <xsl:attribute name="id"><xsl:value-of select="@idref"/></xsl:attribute>
      <xsl:if test="descendant::field-ref/@idref">
        <xsl:attribute name="classid"><xsl:value-of select="descendant::field-ref/@idref"/>
        </xsl:attribute>
        <xsl:attribute name="classtype">varclass</xsl:attribute>
      </xsl:if>
      <xsl:if test="descendant::var-ref/@idref">
        <xsl:attribute name="classid"><xsl:value-of select="descendant::var-ref/@idref"/>
        </xsl:attribute>
        <xsl:attribute name="classtype">varlocal</xsl:attribute>
      </xsl:if>
      <xsl:if test="descendant::formal-ref/@idref">
        <xsl:attribute name="classid"><xsl:value-of select="descendant::formal-ref/@idref"/>
        </xsl:attribute>
        <xsl:attribute name="classtype">varlocal</xsl:attribute>
      </xsl:if>
      <template>
        <xsl:if test="ancestor::method/@name">
          <xsl:attribute name="name"><xsl:value-of select="ancestor::method/@name"/>
          </xsl:attribute>
          <xsl:attribute name="id"><xsl:value-of select="ancestor::method/@id"/>
          </xsl:attribute>
          <xsl:attribute name="class"><xsl:value-of select="ancestor::class/@id"/>
          </xsl:attribute>
        </xsl:if>
        <xsl:if test="not(ancestor::method/@name)">
          <xsl:attribute name="name"><xsl:value-of select="ancestor::constructor/@name"/>
          </xsl:attribute>
          <xsl:attribute name="id"><xsl:value-of select="ancestor::constructor/@id"/>
          </xsl:attribute>
          <xsl:attribute name="class"><xsl:value-of select="ancestor::class/@id"/>
          </xsl:attribute>
        </xsl:if>
      </template>
    </hook>
  </xsl:for-each>
</xsl:template>
```

2. *Capture the Classes where methods are declared*: this is done by searching for the *idref* attribute of the called method tag which relates to the variables declaration *id* attribute.

3. *Invert the position of Hooks and Templates*: this is only required for "Vista1" view and its purpose is similar to the explained before in the IHS method.

4.3 Hotspot Detection

After applying all the XSL transformations, the XML output result file is processed by a PHP script that groups hooks or templates (depending whether its "Vista1" or "Vista2" views) and detects possible hotspots, outputting them into a HTML file.

In both views, the flow and set of activities are the same:

1. Read the file;
2. Parse the XML structure;
3. Find and group the hotspots;
4. Show the results.

Reading the file and showing the results is trivial. A brief description of activities 2 and 3 follows:

Vista1. The XML structure is parsed into a hash container. Whenever a new template is found, a new entry is inserted into the hash, with the template *id* being the key and an empty array as its value. The template "hook method" *id*s will then be added into this array.

At the end, a hash is available with all template *id*s as keys and the corresponding hook *id*s as its values.

Next, an attempt is made to find and group the hotspots. For each template method *id* in the hash, its hook methods are intersected with the remaining ones from the other templates in the hash. If the result of this intersection is not empty, it will be the key of a new hash with its value being an array with the corresponding templates. If that hash key already exists, the template *id*s are added to the already existing array of templates.

At the end, a new hash is available containing the candidate hotspots. Its keys are a set of hooks, and its values the templates who call them.

Vista2. In this view, the document is parsed with the same method as in "Vista1", with one slight difference: since the XML structure was swapped to a "hook to template" hierarchy, the resulting hash will contain all hook *id*s as keys and its values being the template methods which call them.

Finally, to find and group the hotspots, the template *id*s stored in the first hash are used as the key to a new hash of candidate hotspots. Its corresponding value is an array with the hook methods which are called by exactly those templates.

5 Some Results

Figure 5 illustrates the results of applying the "Vista2" view to the JUnit framework. In comparison with the similar tool developed for Eclipse, both "Vista1" and "Vista2" produced perfect matches, despite a few glitches in "Vista1", but of no relevancy to the overall goal. This comparison served to prove that the proposed solution produced accurate results and that the pre-defined heuristics for hotspot detection were indeed valid.

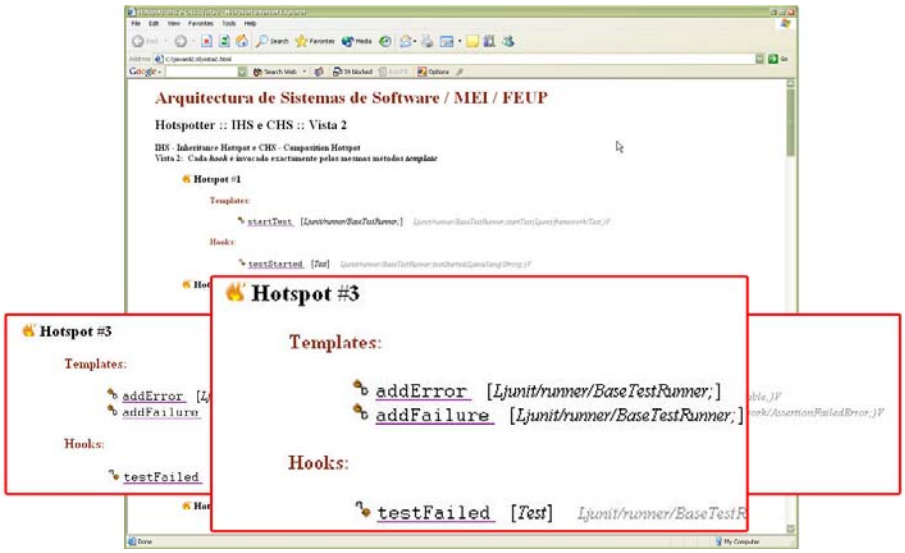

Fig. 5. Vista2 HTML generated results.

6 Conclusions and Future Work

This paper presented a JavaML-based approach to detect hotspots using successive XSL transformations over a multi-phased evaluation process. Using the JUnit framework as a case study, the results were confronted against a similar tool for Eclipse, proving the efficiency of the approach and validating its results. Despite the possibility of performing all the transformation steps using XSL, a final step was made using PHP. This decision had to do with project development schedule restrictions only, and not technological issues regarding XSL. At the time of development, PHP served as a quicker developing platform in order to uphold the project's deadline. At the writing of this article, the PHP step was being converted into XSL, with satisfying results. As future work, this approach could be extended to include the following features:

- Convert the PHP step to XSL (currently underway).
- Detect all seven relationships between templates and hooks (meta-patterns).
- Extend the JavaML notation to include "flexibility-oriented" tags, thus enriching the code representation.
- Infer what kind of design patterns can be found on the code with a degree of certainty.

References

1. Guido Cardino *et al. "The Evaluation of Framework Reusability".* ACM SIGAPP. Volume 5, Issue 2. September 1997
2. Eclipse website - *http://www.eclipse.org* .
3. JUnit website - *http://www.junit.org*.
4. Garry Froelich et al. *"Hooking into Object-Oriented Application Frameworks".* 19th International Conference on Software Engineering. 1997.
5. XML – eXtensible Markup Language (website). *http://www.w3.org/XML/*.
6. Greg Badros. "*JavaML: A Markup Language for Java Source Code*". 9th International World Wide Web Conference. 2000.
7. Ademar Aguiar et al. *"JavaML 2.0: Enriching the Markup Language for Java Source Code"*. XATA. 2004.
8. IBM Jikes Java Compiler - *http://www-124.ibm.com/developerworks/opensource/jikes/.*
9. JavaML Patch for Jikes website- *http://www.cs.washington.edu/homes/gjb/JavaML/.*
10. XSL website- *http://www.w3.org/Style/XSL/*.
11. PHP website - *http://www.php.net/*.
12. Reinhard Schauer. *"Hotspot Recovery in Object-Oriented Software with Inheritance and Composition Template Methods"*. ICSM. 1999.
13. Erich Gamma et al. *"Design Patterns: Elements of Reusable Object-Oriented Software"*. Addison-Wesley. 1995.
14. Wolfgang Pree. *"Design patterns for object-oriented software development"*. Addison-Wesley. 1995.
15. Martin, R. *"The Open Closed Principle"*, C++ Report, 1996