Towards Automated Cross-Language Refactorings between Java and DSLs used by Java Frameworks *

Philip Mayer Andreas Schroeder

Programming & Software Engineering Group
Ludwig-Maximilians-Universität München, Germany
{mayer,schroeder}@pst.ifi.lmu.de

Abstract

Today, software applications are usually not written in just one programming language. In many cases, a general-purpose language such as Java is combined with multiple domain-specific languages (DSLs) for diverse purposes such as system configuration, UI description, or database querying. The artifacts defined in those different languages reference each other, often by name; in most cases these references are essential for the functionality of the overall system. This introduces problems if an artifact is refactored in any single language, since most current refactoring tools are not aware of language-external uses of the artifact. What is therefore needed is extended refactoring support across language boundaries. In this work, we explore the area of cross-language linking and refactoring, and present an approach and tool which we evaluate in a systematic fashion using automated renaming and unit testing on an open-source case study.

Categories and Subject Descriptors D [*2*]: 6

Keywords multi-language software applications, polyglot programming, cross-language, refactoring, automation, DSLs, Java

1. Introduction

Arguably, any non-trivial software application today is a multi-language software application (MLSA) [2], i.e. an application which is written using multiple formal (programming) languages. We shall confine ourselves in this work to the combination of one general-purpose language (Java) with multiple DSLs. Such a setup is usually achieved through the use of *frameworks* which come equipped with a DSL, and which parse and evaluate the DSL code at application runtime. In this paper, we use the industrial and open-source frameworks Spring, Hibernate, and Wicket, which each come with their own DSL(s).

Using multiple languages in one project follows the "right tool for the job" metaphor. Expected benefits are reduced amount of code, thus increased development speed, and (ideally) better readability and maintainability of domain-specific aspects of the application. However, there are downsides as well, the most obvious being the increased mental workload due to cross-language links that must be considered *in addition to* the semantics of each individual language during development, program understanding activities, and in particular, refactoring. In fact, even the existence of cross-language links may make developers wary of changes.

As we have argued previously [3], we feel that overall the benefits prevail, and thus that automating discovery, management, and refactoring of cross-language links is a worthwhile endeavor.

In the following, we first discuss a case study in Section 2. Afterwards, we describe our approach to MLSA development automation, which is done in several phases with the end result of refactoring possibilities (Section 3). We discuss how we evaluated our approach in Section 4 and summarize in Section 5.

2. Case Study

We use the open-source issue tracking system JTrac as our case study. JTrac is a web application written in Java that uses the frameworks Spring (a dependency injection framework), Hibernate (an object-relational mapper with both entity definition and querying languages), and Wicket (a HTML-based UI framework), which represent a fairly good spread of DSL use cases. JTrac uses six languages: Besides Java, these are the Spring DSL, the two languages HBM and HQL from Hibernate, and the HTML dialect of Wicket as well as the internal Wicket DSL in Java.

Listings 1 to 4 show examples of cross-language links (Java, HBM, and Wicket) taken directly from JTrac.

The combination of Java with each of the frameworks separately already creates three interesting set-ups for investigation. Having all three inside one application leads to additional linking and refactoring challenges, since changes may need to be propagated across more than one language border.

First, the HBM code in listing 1 defines a persistence mapping for the entity `Item` and its property `summary`. On the Java side, this mapping refers to the class `Item` and a Java getter and setter method in listing 2. If the property name in the mapping file is changed, the getter and setter need to be renamed as well.

Listing 4 shows a Wicket HTML page in which an input field is tagged with the Wicket ID `summary`. This identifier is first only linked to the corresponding form class, which is shown in listing 3.

However, as the `ItemForm` class uses a bound compound property model, the Wicket ID is not only used as a cross-reference between the HTML input element and the Java-defined text field, but additionally refers to a property of the `Item` object given in the constructor, i.e. we have two additional links from the Wicket ID definition in Listing 3 to the Java getter and setter methods of `Item`.

* Partially sponsored by the EU project ASCENS, 257414

WRT '13, October 27, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2604-9/13/10. . . $15.00.
http://dx.doi.org/10.1145/2541348.2541350

Listing 1: Hibernate mapping

```
<class name="Item" table="items">
  <property name="summary" column="summary"/>
</class>
```

Listing 2: Java domain class

```
public class Item {

  public String getSummary() {
    return summary;
  }

  public void setSummary(String summary) {
    this.summary = summary;
  }
}
```

Listing 3: Wicket form class

```
private class ItemForm extends Form {

  public ItemForm(String id, final Item item) {
    super(id);
    model = new BoundCompoundPropertyModel(item);
    setModel(model);
    add(new TextField("summary"));
  }
}
```

Listing 4: Wicket form HTML

```
<form wicket:id="form">
  <input wicket:id="summary"/>
</form>
```

To sum up, this example shows a total of five cross-language links across two different frameworks (Wicket and Hibernate) such that five artifacts are affected by a change in any one of them: As an example, whenever the `wicket:id` attribute value of the input field in Listing 4 is changed, all linked artifacts need to be altered accordingly: the string literal used as text field id in `ItemForm`, the getter and setter in the `Item` class, and transitively, the Hibernate mapping defined for the `Item` class.

3. The XLL Approach

Our approach, which we call "XLL" for cross-language linking, has the aim of creating awareness and an architecture for supporting MLSAs within IDEs — specifically, Eclipse — in a comprehensive approach. We feel that existing implementations (such as the tool support for Spring, Hibernate, and Wicket) are focused on isolated solutions (mostly, linking and refactoring between two languages at a time). There are several reasons why we feel that a framework approach to MLSA linking and refactoring, where additional languages "plug in" to a generic refactoring algorithm, is beneficial.

First, as our case study shows, more than two languages may be affected when changing artifacts in a single language — even when starting from a DSL, and even if the languages are not directly linked. This can be supported by a language-agnostic refactoring propagation algorithm.

Second, we believe that the work on language linking has more to offer than just refactorings. Before we get to refactorings, there are linking errors and warnings to consider, and we may even need to refuse refactoring in case artifacts relevant for cross-language linking cannot be fully resolved (detailed below).

Third, there is already refactoring support out there, especially for Java, which can be re-used in a MLSA refactoring solution. However, we also want to be able to start by refactoring DSLs and thus have bidirectional refactoring support (contrary to the, e.g. Eclipse refactoring participant approach).

We focus our work on Java and Java frameworks with DSLs since examples of these abound, and their interactions are non-trivial. In our three example frameworks, artifacts are linked between languages either by name (ID references) or position (due to a parent/child relationship, or by index in an (argument) list).

The solution approach we use is a three-step process in which the steps build upon one another.

- *Language Models and Discovery*. The first step consists of discovering models from source code, leading to artifacts which can then be linked based on their position and properties.
- *Linking*. The second step is a pairwise linking process which investigates two models, identifying the framework-specific induced links between artifacts.
- *Refactoring*. Finally, our refactoring algorithm builds on the links discovered in the previous step. Based on changes to artifact properties, changes are propagated across languages until closure is reached.

Each step has its own results and benefits besides allowing the next step to continue. We detail the three steps in the following three sections.

3.1 Language Models and Discovery

Our aim is automating the process of linking identifiers in different languages; i.e. our approach is not based on manual identification of identifiers (as, e.g., in [4]). Instead, we opt for a model-based approach, i.e. extracting the code of each individual programming language into a model corresponding to a meta-model of that language, and use those model artifacts for linking (another option would have been just one meta-model across all languages, as done, e.g., in [5]). Meta-models and discovery routines are provided by *language adapters* for each individual language.

Such a meta-model of a language has been called a semantic model [1], which is a higher-than-AST representation of a language. It describes the concepts of the language — such as, in Java, the fact that there are `TypeDeclaration`s which contain `MethodDeclaration`s — and which ideally has all in-language bindings resolved (such as linking a `MethodInvocation` to its `MethodDeclaration` in Java), but is still language-specific. The meta-model must also be source code aware, i.e. keep the source position of each element.

Creating such meta-models requires effort; for our approach, we require one such model for each language (here: Java, Spring, Hibernate/HBM, Hibernate/HQL, Wicket/HTML and Wicket/API). For Java, such models already exist (e.g., MoDisco); we have created EMF meta-models for the other languages ourselves. The creation of a meta-model is not trivial, but should be no great effort for framework developers. There are non-cross-language benefits to such meta-models too: they can be used for other tasks such as metrics calculation or visualization.

Once a meta-model has been created, corresponding models need to be extracted from source code ("model discovery"). Effort for model discovery is very different depending on the language format. For example, Spring (in its base form) uses an XML dialect for describing beans and their properties. Such code is easy to read and extract. However, things are different when we discuss frameworks such as Hibernate and Wicket. Here, some of the relevant code of the DSL is fragmented and distributed in strings within Java code, and furthermore embedded in API calls. The identifiers present may be called *dynamic identifiers* since they are created within the control and data flow of Java.

It is, in general, not possible to find and/or resolve all identifiers in such a context. However, since Hibernate and Wicket are certainly much-used real-life frameworks, we cannot just ignore such identifiers either. Our approach to handling dynamic identifiers is similar to that of Tatlock et al. [6] in that we perform a static analysis of MoDisco Java models in order to retrieve possible values of argument expressions in specific framework method calls.

The capabilities of our implementation for finding dynamic identifiers correspond to an interprocedural flow-sensitive analysis over the domain of method and object environments, i.e. local variable and object field bindings to either object environments, strings, or custom entities. In terms of dataflow analysis, our static analysis approach uses standard transfer functions for application code (corresponding to the semantics of Java), and custom transfer functions for all library method calls (identity function for irrelevant calls and transfer functions performing environment updates and custom object manipulation for relevant framework calls).

From well-defined starting points such as all constructors inside `WicketPage` subclasses in a Wicket project, we perform a single pass over a block of statements following method invocations. Loops are handled by performing a single pass of the loop body, and recursive methods by skipping recursive method calls, such that our approach will miss dynamic links fabricated through looping or recursive program code. In the context of cross-language linking, however, this is no severe limitation, as every link must point to a corresponding static name in another language, and thus fabrication of names is not encouraged in any of the frameworks.

In both Hibernate and Wicket, the representation of the object trees created is an over-approximation of the parent-child relationships actual code executions can create. Note also that in case identifiers are passed in from the outside, we may know that an identifier was used, but not which one — this information is highly relevant for creating refactoring warnings (see below).

Although our approach to dynamic identifier resolution is similar, our work differs from Tatlock et al. in other parts. First, their work focuses on Java and JPA queries, where it goes beyond what we offer (in particular, type checking of parameters). However, the type checking concepts such as query completeness and output correctness are specific to queries, as is the summarizing as regular expressions. The approach thus cannot be trivially transferred to other DSLs. Our approach differs in that it encompasses three very different frameworks and DSLs. We aim at a comprehensive, integrative approach with support for plugging in DSLs, bi-directional refactoring, and multi-language linking and refactoring support.

3.2 Linking

Having extracted the source code from each individual language into an easily accessible model, we can now proceed to linking identifiers from different languages. In [3] we have shown that describing algorithms for linking identifiers between languages is not trivial due to the freedom frameworks give to developers. Separating this step into its own dedicated phase, different from both model discovery before and refactoring afterwards, has proven to be very beneficial for us since it clarifies and precisely defines the linking problem as such.

In our approach, a *linker* creates links between two languages at a time, for example Java and Hibernate HBM (see Listings 1 and 2). The task of the linker is thus to investigate the input models based on the rules given by the framework looking for artifacts which reference one another. In the example, this would mean linking the HBM class artifact with the name `Item` in the first listing to the class `Item` in the second listing (considering fully qualified names, etc.). Other examples include linking constructor arguments by index to method parameters, Wicket HTML identifiers to Java strings (considering parent/child relationships), and so on.

All of the languages we have looked at have a strong hierarchical structure. In Spring, beans contain properties and constructor arguments. In Hibernate, classes have IDs, properties, collections; in Wicket, pages have components which may be nested. This structure is important for linking as well, since nested artifacts may only be linked in the context of their relevant parents.

Linking may also fail. That is, the linker code must include mechanisms for identifying if and where an artifact has no corresponding element on the other side of the language gulf. This may then yield warnings or errors which can be presented to the user — a benefit of linking, even without considering refactorings. For resolved (successful) links, the benefit is enabling visual annotation and allowing navigation right in the IDE.

A linker is only ever responsible for linking two languages at a time. However, one language may be linked in multiple directions with multiple linkers, thus we can later propagate changes through the linked artifacts without requiring linkers between all languages.

Again, writing such linkers means effort. Of course, individual framework code already contains algorithms for these links, but they may or may not be readily accessible or usable in this context. A deep knowledge of the framework involved is certainly a requirement for writing a linker.

3.3 Refactoring

One goal of the XLL framework identified above was creating a generic framework for refactoring across language boundaries. Based on the links identified in the previous section, there is actually not a lot more required for adding this functionality. As pointed out already, we restrict ourselves to *rename refactorings*: We only consider links based on names (identical or transformed).

To enable refactorings, we need to be aware of the artifact properties whose change may lead to cross-language refactoring; this information is implicit in the algorithm required for linking in the previous step. An example would be the information that `WicketElement.name` is linked to `MethodDeclaration.propertyName` (where `propertyName` is the bean name of the property, i.e. without the get- or set- prefix).

Given this information, we know which changes to propagate through each link. Thus, we can design a generic refactoring algorithm in the XLL framework which, given a single original property change from the user, can traverse all links from artifact property to artifact property until a transitive closure of changes has been reached. This leads to a number of artifact property changes in a number of languages which then need to be transferred — by language-specific refactoring routines — to actual text edits.

The latter task (along with forwarding a refactoring invoked by the user in the first place) falls again to the language adapters. Each language adapter is handed the artifacts and properties which need to change after the XLL algorithm has been run, and will invoke language-local refactorings. For example, if the `name` of a Java `ClassDeclaration` was changed, the Java language adapter will invoke the Rename Type refactoring which may lead to many text edits, a compilation unit rename, etc.

An example of a refactoring propagation has been given in Section 2, where renaming a Wicket identifier leads to renaming Java methods, which in turn leads to renaming properties in Hibernate.

A refactoring may not always be possible: There may be conditions in which we need to warn the user about possible side-effects, or prevent him from executing a refactoring. As an example, the JTrac project contains a HQL query in which the name of a property is injected and is thus unknown to the model discoverer (`"SELECT from Item i where i." + getName() + "..."`). In this case, model discovery must store information about the non-resolvability of *some* property of `Item`. In refactoring, we must then flag *all* properties of class `Item` as potentially problematic,

and thus warn the user about possible problems (as, e.g., the Java rename refactoring does in case of parsing errors).

Again, implementing refactoring support in each language requires effort, which however is in this case less than in the previous two steps. Identifying which artifact properties are linked is a by-product of creating the linker; the only thing left is plugging into and invoking pre-existing language-specific refactorings. This obviously only holds if the refactorings exist in the first place, which is not the case in most DSLs, where they have yet to be created. However, since many DSLs are (in themselves) quite simple, this is less of an effort than in languages such as Java.

4. Implementation and Evaluation

Our implementation of the XLL approach[1] covers all three of the above steps for the three frameworks discussed. All subsequent numbers are taken from the JTrac case study.

A key concern for us was how to validate if the model discovery, linking process, and the refactorings we carry out are actually correct. In the case of discovery and linking, we had to manually inspect each artifact to make sure none are missing, and all are either reported as errors or linked correctly. With regard to refactoring, we have been able to automate validation with the same approach as in "normal" refactorings: By using unit tests. For each language pairing, tests were created which covered *both* languages under test, succeeding if cross-language links were correct, failing if not.

We used the following experimental setup which was run on all identified links in all languages within JTrac.

1. All tests were run on the unchanged original code of the case study. The tests were expected to pass.
2. Only the artifact on the left-hand side of the link was renamed. Then, the tests were run again and were expected to fail. The change was then undone.
3. Only the artifact on the right-hand side of the link was renamed. The tests were run and were again expected to fail. The change was undone.
4. Finally, the multi-language refactoring algorithm was run on the link. The resulting changes were then executed in all languages, i.e. all relevant artifacts renamed. The tests were run again and were expected to pass. The change was again undone.

The result from the tests are shown in table 1. The column *L* shows the total amount of links found in the system; all of which were refactored. As mentioned above, we can warn the user if a refactoring might fail due to missing identifiers; thus, the next two columns show the number of refactorings run despite warnings. Of those, some succeeded (i.e., the tests passed) due to the overapproximating nature of the warnings (W/O), and some actually lead to test failures (W/F). There were no test failures for links for which no warning was given.

The last two columns show the mean value of the number of artifact changes (ØA) and resulting text changes (ØT) in each link.

As can be seen, the number of successfully executed refactorings is rather high (around 90%). In 45% of cases, more than two languages were involved, with a maximum of five (since the Spring artifacts were disjunct from all others). As said above, in no case did the tests fail without warning — however, there were a number of refactorings (around 27%) for which warnings were given which were in fact unnecessary. In each case, the reason are unknown (that is not fully resolvable) identifiers, which is a limitation of the model discovery routines.

[1] `www.xllsrc.net`

Table 1: Refactoring Results

Source	Target	L	W/O	W/F	ØA	ØT
Spring and Java						
Bean	Class/Method	34	0	0	2.1	5.5
Property	Method/Field	70	0	0	2.2	2.4
Hibernate/HBM and Java						
Entity	Class	13	0	0	7.2	77.6
Property	Method/Field	258	78	98	8.7	11.3
Hibernate/HQL and Hibernate/HBM						
Ent. Ref.	Entity	68	0	0	24.4	127
Prop. Ref.	Property	370	295	8	81.5	48.8
Wicket-API and Wicket-HTML						
Class	Page	46	0	0	2.0	6.7
Widget	Element	489	10	0	5.8	3.3
Wicket-API and Java						
Element	Method	199	35	17	11.4	16.2

We conclude that for frameworks making use of static identifiers (Spring, Hibernate/HBM), the approach chosen leads to full refactoring coverage. For frameworks with dynamic identifiers (Hibernate/HQL, Wicket/HTML, Wicket/API), we run into the usual analysis problems but are, at least for the case study and the chosen frameworks, able to identify unknown identifiers and warn the user.

5. Summary

This paper has shown an investigation into cross-language linking and refactoring between Java and DSLs used by Java frameworks; in particular, the frameworks Spring, Hibernate, and Wicket. We believe that support for developers in this area leads to better code understanding, less errors, and better maintainability of software.

We feel that multi-language linking and refactoring has not yet received the attention it deserves. The number of programming languages is already rather high, and with the advent of language workbenches will rise even more. Furthermore, there will always be legacy languages to be integrated.

As a long-term objective, we plan to investigate guidelines for how to best deal with cross-language links, which includes how to write frameworks which let developers reap the benefits of DSLs while still being amenable to analysis and refactoring.

References

[1] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[2] P. Linos, W. Lucas, S. Myers, and E. Maier. A metrics tool for multi-language software. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*, SEA '07, pages 324–329, Anaheim, CA, USA, 2007. ACTA Press.

[3] P. Mayer and A. Schroeder. Patterns of cross-language linking in java frameworks. In *Proceedings of the 21st IEEE International Conference on Program Comprehension*, ICPC'13, pages 1–10, 2013.

[4] R.-H. Pfeiffer and A. Wasowski. Cross-language support mechanisms significantly aid software development. In R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *MoDELS*, volume 7590 of *Lecture Notes in Computer Science*, pages 168–184. Springer, 2012.

[5] D. Strein, R. Lincke, J. Lundberg, and W. Löwe. An extensible meta-model for program analysis. *IEEE Trans. Softw. Eng.*, 33(9):592–607, 2007.

[6] Z. Tatlock, C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Deep typechecking and refactoring. *SIGPLAN Not.*, 43(10):37–52, 2008.