JQuery-Based Refactorings as a Framework for Managing Java Annotations

CPSC 449 Honours Thesis
Alex Bradley
University of British Columbia
Supervisor: Dr. Kris De Volder

April 2008
Abstract

This thesis aims to provide a solution to the problem of managing large numbers of Java 5 annotations. We introduce the idea of JQuery-based refactorings, which are essentially Eclipse Language Toolkit refactorings that take the results of a JQuery query as input. These refactorings are integrated with the JQuery user interface through two types of TyRuBa configuration predicates, one for refactorings that apply to an entire working set and the other for context-sensitive refactorings that apply to a user-selected set of code elements. By implementing JQuery-based refactorings for a number of usage scenarios and evaluating their ease of implementation and configuration, we aim to demonstrate that these refactorings provide a powerful and customizable framework for annotation manipulation.
Contents

1 Introduction 3

2 JQuery-Based Refactorings 4
   2.1 Eclipse Refactoring Support .................................. 4
       2.1.1 LTK Refactoring Process .............................. 5
       2.1.2 ASTs and AST Rewritings in the Eclipse JDT ......... 6
   2.2 Java API for Query-Based LTK Refactorings ................. 6
   2.3 Integrating Refactorings into the JQuery UI .................. 7

3 Evaluation 9
   3.1 Usage Scenarios .............................................. 9
       3.1.1 Adding an annotation to a set of methods ............ 9
       3.1.2 Adding @Override wherever applicable .............. 11
       3.1.3 JUnit 4 test case annotations ....................... 12
       3.1.4 Feature annotations .................................... 14
   3.2 Analysis ...................................................... 16

4 Related Work 18
   4.1 Annotation generation ....................................... 18
   4.2 Languages for specifying refactorings ....................... 19

5 Conclusion 21
   5.1 Future Work .................................................. 21

A JQuery Predicates for Annotations 26

B Source Code Listings 27
   B.1 QueryBasedRefactoring ................................. 27
B.2 RefactoringTargetSet, RefactoringTarget ............. 28
B.3 RenameFeatureRefactoring ......................... 29
Chapter 1

Introduction

Java 5 annotations, introduced in [GJJB05], are increasingly becoming an important tool for developers. Widely used frameworks such as JUnit[ Gon06] and Enterprise JavaBeans[O’C07] promote the use of annotations to specify object properties and roles. However, manually adding and maintaining large numbers of annotations in a large existing codebase is time-consuming and error-prone. A framework for managing annotations on large sets of annotatable code elements in a consistent manner would allow developers to make greater use of annotations and gain greater access to the benefits they provide.

Our thesis is that JQuery-based refactorings provide a powerful and customizable framework for managing Java 5 annotations.

We will begin by defining and describing an implementation of our notion of “JQuery-based refactorings” (or “query-based refactorings”) in Chapter 2. In Chapter 3, we will validate our approach by applying it to a number of usage scenarios and evaluating the power and ease of implementation and configuration provided by our framework. In Chapter 4, we will briefly survey some of the related work in this area. Chapter 5 will sum up our work and provide some possible directions for future research.
Chapter 2

JQuery-Based Refactorings

At its most general, a JQuery-based refactoring is any refactoring that takes the results of a JQuery query as its input. (In this discussion, we assume that the reader is familiar with JQuery and its query language TyRuBa; [JV03], [Vol06], [Vol98] and [Vol03] can be consulted for more details.) In our implementation of JQuery-based refactorings, the refactorings are implemented in Java using the Eclipse Language Toolkit framework, and can be integrated into the JQuery user interface through configuration rules written in TyRuBa. We will first give a basic overview of how LTK refactorings work, then describe how we have built JQuery-based refactorings using LTK. We will then describe our TyRuBa configuration rules for refactorings.

2.1 Eclipse Refactoring Support

Since version 3.1, Eclipse has provided a standard, language-independent framework for implementing source code refactorings: the Language Toolkit (LTK) API. In this section, we will give a brief overview of the components involved in a simple LTK refactoring process. We will then give a short description of AST rewriters, which provide a powerful way to calculate changes to Java source code. These Eclipse components provide the basic building blocks for our query-based refactorings.
2.1.1 LTK Refactoring Process

The logic for the refactoring—the actual computation of the changes to make and the associated validity checks—is provided by subclasses of `org.eclipse.ltk.core.refactoring.Refactoring`. The wizard-based user interface for the refactoring is managed by subclasses of `org.eclipse.ltk.ui.refactoring.RefactoringWizard`. When a refactoring is launched, these components interact in roughly the following lifecycle:

1. A `RefactoringWizard` is created and associated with a `Refactoring`.

2. The `checkInitialConditions()` method of the `Refactoring` runs. Its function is to perform basic sanity checks to determine whether it makes any sense to apply the refactoring in the given situation. If this method returns an error status, the refactoring process aborts and an error is displayed to the user.

3. The `RefactoringWizard` may obtain additional input from the user through one or more interactive wizard pages (provided by subclasses of `UserInputWizardPage`).

4. The `checkFinalConditions()` method of the `Refactoring` runs. Its function is to perform more thorough checks to determine whether any user input given is valid and whether the refactoring can be applied. If this method returns an error status, the `RefactoringWizard` will display an error, and the user will be permitted to return to previous wizard pages and correct their input.

5. The `createChange()` method of the `Refactoring` runs. This method computes the change to be performed based on the available data. Changes are represented by subclasses of `org.eclipse.ltk.core.refactoring.Change`. (In our refactorings, we will use `TextFileChanges`, which represent alterations to text files, and `CompositeChanges`, which are simply a way of combining multiple `Changes` into one “composite” change.)

6. The `RefactoringWizard` displays a preview page which uses the usual Eclipse file difference viewer to show the user all the changes that the refactoring would perform. (For examples of these preview pages, see Figures 3.3 and 3.5.) If the user approves the changes by clicking the “OK” button, LTK proceeds to perform the refactoring.
A more detailed description of this lifecycle can be found in [Fre06].

2.1.2 ASTs and AST Rewritings in the Eclipse JDT

As described in [KT06], the Eclipse Java Development Tools (JDT) suite provides tools for constructing an abstract syntax tree from Java source code. Modifications to abstract syntax trees (changes to the properties of ASTNodes) can be modelled with AST rewritings (ASTRewrite and associated classes.)

2.2 Java API for Query-Based LTK Refactorings

We have developed a new family of refactorings which take a set of results from JQuery queries as input and calculate the changes to perform based on those results (and possibly some extra user input.) We call such refactorings query-based refactorings.

Core functionality for these refactorings is provided by an abstract class QueryBasedRefactoring (a subclass of Refactoring; see Appendix B.1 for details.) This class aims to provide reasonable default implementations of checkInitialConditions, checkFinalConditions, and createChange. checkInitialConditions checks that the set of query results provided is not empty. createChange essentially iterates through the set of JQuery results and performs the following operations for each one:

1. Find the Java source file corresponding to the element $E$ in the first bound variable.

2. Parse that source file into an AST.

3. Traverse that AST to find the ASTNode $N$ corresponding to $E$ (matching is done based on $E$'s source location information.)

4. To compute the AST rewriting for $N$, call handleTargetNode($N$, $T$), where $T$ is a map giving the elements bound to each variable for the current

---

1This process has a quirk: annotation AST nodes can sometimes have the same “starting location” in the Eclipse AST as the things they annotate, which could cause a problem if the ASTVisitor performing the traversal reaches the wrong node first. We work around this problem by requiring that QueryBasedRefactorings have a method getMatchingNodesType that returns a Class corresponding to the type of AST node applicable for the refactoring.
result (encapsulated by the class RefactoringTarget; see Appendix B.2 for details.)

Developers creating new query-based refactorings need not worry about the createChange method or initial-conditions checks; they can simply focus on implementing handleTargetNode.

2.3 Integrating Refactorings into the JQuery UI

As described in [Vol06], JQuery’s context menus provide access to predefined queries according to certain configuration rules expressed in TyRuBa. Two types of predefined queries are provided: top-level queries (defined with the topQuery predicate), which are available in all contexts; and subqueries (defined with the menuItem predicate) which are only made available in a menu if the selected tree node(s) match a given query, and which may take the selected node(s) as input when executed.

We have taken a similar approach to integrating our refactorings into the JQuery context menus. We provide two ways for JQuery users to access refactorings: top-level refactorings, which execute a predefined query and apply a given refactoring to the results; and contextual refactorings, which are only made available in a menu if the selected node(s) match a given query, and which execute a subquery on the selected nodes, then apply a given refactoring to the results of that subquery.

Top-level refactorings are defined with the topRefactoring predicate, which has the following specification:

// topRefactoring([labelString], refactoringInfo)
topRefactoring :: [String], QueryRefactoring
MODES
   (F,F) IS NONDET
END

Contextual refactorings are defined with the refactoring predicate, which has the following specification:

// Refactorings based on properties of selected item (sort of like menuItems)
// refactoring(?this, [labelString], refactoringInfo) :- applicExp.
refactoring :: Object, [String], QueryRefactoring
In each of these predicates, the label string and ?this arguments have the same function as in the analogous menuItem and topQuery predicates. The last argument to each predicate gives the refactoring to apply, specified with the new TyRuBa type QueryRefactoring:

// query, java refactoring class, Tyрубa var-to-Java arg correspondences
TYPE QueryRefactoring<> AS <String,String,[String]>

A QueryRefactoring is a 3-tuple giving information about a single refactoring:

• the query to execute to find input for the refactoring (in which the currently selected nodes are bound to !this in the case of a contextual refactoring);

• the fully qualified name of the Java class which performs the refactoring; and

• the “chosen variables” for the above query, which will specify which results are passed to the refactoring and the order in which they are passed.

Examples of refactorings defined with these predicates can be found in the next chapter.
Chapter 3

Evaluation

In the initial planning phase of this project, we established a number of usage scenarios for an annotation generation tool, some of which were drawn from annotations and frameworks in widespread use. In this chapter, we describe those scenarios, and evaluate the power and customizability of our query-based refactorings by demonstrating how they can be used to perform the specified manipulations.

Descriptions of the annotation-related predicates used in the TyRuBa queries can be found in Appendix A.

3.1 Usage Scenarios

3.1.1 Adding an annotation to a set of methods

One of the most simple annotation management tasks is adding an annotation to a set of methods or fields. For instance, a developer might want to add the @Deprecated annotation to certain methods; to add the @Ignore annotation to certain JUnit test methods; or to add the Java Persistence API\cite{O’C07}’s @Id annotation to the getId methods of some entity bean classes.

Using our API for query-based refactorings, we can construct the following simple\footnote{Some details, such as addition of import statements and caching of resolved annotations, have been omitted.} Java class for adding a marker annotation to a code element. It expects query results based on a query with two chosen variables. The first variable gives the code element to which to add an annotation, and the second variable...
gives the type of the marker annotation to add to that element.

```java
public class AddAnnotationsRefactoring extends QueryBasedRefactoring {
    public AddAnnotationsRefactoring(RefactoringTargetSet targets) {
        super(targets);
    }

    // Name of the refactoring for the wizard UI
    @Override public String getName() { return "Add Annotations"; }

    protected void handleTargetNode(ASTNode foundNode, RefactoringTarget target)
        throws JavaModelException {
        // target.get(1) returns the second (zero-based "1th") argument to this refactoring,
        // i.e., the result corresponding to the second chosen variable for the associated
        // query.
        Object annotationType = target.get(1);
        if (foundNode != null && foundNode instanceof BodyDeclaration) {
            Annotation annot = getAnnotation(annotationType, foundNode.getAST());
            BodyDeclaration bodyDecl = (BodyDeclaration)foundNode;
            // currentRewrite is a superclass field that allows us to access the current
            // ASTRewrite.
            ListRewrite typeDeclRewrite = currentRewrite.getListRewrite(bodyDecl,
                bodyDecl.getModifiersProperty());
            typeDeclRewrite.insertFirst(annot, null /* TextEditGroup */);
        } else {
            throw new IllegalStateException("Element not found or not a BodyDeclaration");
        }
    }

    @Override
    protected Class getMatchingNodesType () { return BodyDeclaration.class; }

    private Annotation getAnnotation (Object obj, AST ast) {
        Annotation annot = ast.newMarkerAnnotation();
        // Note that we can access JQueryAPI and make further queries within a refactoring.
        Name simpleName = ast.newName(JQueryAPI.getStringProperty(obj, "name"));
        annot.setTypeName(simpleName);
        return annot;
    }
```

10
With the Java refactoring implementation in place, the following contextual refactoring predicate allows a developer to select some methods, then apply any of the potential marker annotations in JQuery’s working set to those methods:

\[
\text{refactoring(?this, ["Add annotation to method(s)", ?N], QueryRefactoring<\{method(!this),annotationDeclaration(??T),name(??T,?N)\}, "ca.ubc.jquery.annotations.AddAnnotationsRefactoring", ["!this", "?T"]> :- method(?this),annotationDeclaration(?T),name(?T,?N),}
\]

// Marker annotations cannot have any required attributes (i.e. declared attributes without default values.)

\[
\text{NOT(EXISTS ?Attrib : child(?T,?Attrib),NOT(hasDefault(?Attrib))).}
\]

The context menu displayed to the user for this refactoring is shown in Figure 3.1.

![Image](image.png)

Figure 3.1: Adding the @Deprecated annotation to selected nodes

### 3.1.2 Adding @Override wherever applicable

The tool should be able to add and remove the @Override, @Deprecated and @SuppressWarnings annotations at a given set of locations in the code. The
following top-level refactoring uses a query to find all the methods in the current working set that could be given the @Override annotation but do not currently have it, and uses the AddAnnotationsRefactoring to add @Override to all those methods.

topRefactoring(["Add Override annotation where applicable"],
QueryRefactoring<"method(?X), overrides(?X, ?),
   NOT(EXISTS ?A : hasAnnotation(?X,?A), name(?A, Override)),
   annotationDeclaration(?T), name(?T, Override)"]
,"ca.ubc.jquery.annotations.AddAnnotationsRefactoring", ["?X", "?T"]).

(Line breaks have been added to the query string for readability.) The context menu generated for this configuration entry is shown in Figure 3.2. Figure 3.3 shows the RefactoringWizard preview page when this refactoring is applied to a version of the JHotDraw source code.

![Figure 3.2: Top-level refactoring: adding @Override where necessary](image)

3.1.3 JUnit 4 test case annotations

As described in [Gon06], JUnit 3 and JUnit 4 have different conventions for specifying test cases. One of the differences between the two versions is that
JUnit 3 detects test cases based on method signatures (a JUnit 3 test case method has a name beginning in `Test` and takes no parameters), while JUnit 4 detects test cases based on annotations (a JUnit 4 test case has the annotation `@Test`). If we were converting JUnit 3 tests to the JUnit 4 style, we could use the following top-level refactoring to add the `@Test` annotation to all old JUnit 3 test case methods:

```java
topRefactoring(["Upgrade JUnit 3 test cases to JUnit 4"],
    // JUnit 3 test case classes must extend TestCase
    QueryRefactoring<"extends(?C,?T), qname(?T,junit.framework.TestCase),
        method(?C,?M), re_name(?M,/^test/), params(?M,[]),
        annotationDeclaration(?A), qname(?A,org.junit.Test)",
    "ca.ubc.jquery.annotations.AddAnnotationsRefactoring",
    ["?M", "?A"]).
```

(Line breaks have been added to the query string for readability.)
3.1.4 Feature annotations

Rick Chern and Kris De Volder have developed a method for using Java annotations to delineate modular structure in Java projects.[CV08] The “modules” (hereafter called features) thus constructed are heterogeneous and potentially overlapping collections of source code artifacts (e.g. classes, methods, method call relationships, inheritance relationships, etc.) The annotations used include @Feature, which specifies that a code element belongs to one or more features, and @Export, which specifies that a code element is “exported to” (used by) one or more features. We will collectively call these annotations feature annotations. Here is a hypothetical example in which both annotations are applied to a method:

```java
@Feature(names = {"./featureSAVE", ".//featureWRITE")
@Export(to = {"./featureDUPLICATE", "./featureCOPY", ".//featureCUT")
public void writeInt(int i) { /* . . . */ }
```

It can easily be seen that in a large codebase organized into features with these annotations, there could be hundreds or thousands of annotations to manage. Our query-based refactorings can be of use in managing feature annotations in various ways. A simple case is that of renaming a feature, which requires finding all the relevant annotations and substituting the new feature name for the old feature name in their attributes. The Java code for this operation is slightly more complicated than that of the preceding example; it can be found in Appendix B.3.

Chern and De Volder have created predicates called feature and exportMarked which make it straightforward to find the elements contained in and exported to a feature. They have also created a “feature browser” top-level query which allows the developer to view all the features in the code (see Figure 3.4.) We can use a contextual refactoring such as the following to perform this task:

```java
refactoring(?this, ["Rename feature"],
QueryRefactoring<"(feature(!this,E),hasAnnotation(E,?Annot),name(?Annot,Feature));
 (exportMarked(?E,!this),hasAnnotation(?E,?Annot),name(?Annot,Export))",
"ca.ubc.jquery.annotations.RenameFeatureRefactoring",
["?Annot", "!this"]>) :- feature(?this).
```

(Line breaks have been added to the query string for readability.) Figure 3.5 shows this refactoring in action.
Figure 3.4: JQuery feature-browsing query and context menu

Figure 3.5: Renaming a feature
In a similar way, we could design refactorings that add or remove code elements to or from features, or delete a feature.

**Declarative completeness**

Chern and De Volder’s features have an associated notion of *declarative completeness*. A feature-annotated codebase is declaratively complete if for any element $A$ in a given feature $F$ that accesses some other element $B$ in the codebase, either

- $B$ belongs to $F$, or
- $B$ is exported to $F$.

Suppose that a predicate `missingElement(?F,?B)` exists that find all the elements “missing” from a given feature—i.e., elements $B$ that do not satisfy the criteria above. For any such $B$, the developer has two possible ways of resolving the problem: add $B$ to $F$, or export $B$ to $F$. Therefore, we cannot construct an overall top-level refactoring that makes a codebase declaratively complete in one deterministic process. However, one can imagine a natural way of providing the developer with a choice at each element with twin contextual refactorings like the following:

```prolog
refactoring(?this, ["Declarative completeness", {Add to feature ?F}],
            QueryRefactoring<{missingElement(?F,!this)},
            "ca.ubc.jquery.annotations.AddToFeatureRefactoring", // not implemented
            ["!this", "?F"]>) :- missingElement(?F,?this).
refactoring(?this, ["Declarative completeness", {Export to feature ?F}],
            QueryRefactoring<{missingElement(?F,!this)},
            "ca.ubc.jquery.annotations.ExportToFeatureRefactoring", // not implemented
            ["!this", "?F"]>) :- missingElement(?F,?this).
```

With these contextual refactorings, the developer can resolve declarative completeness problems by running a top-level `missingElement` query, then selecting groups of results and adding or exporting them as he/she pleases.

### 3.2 Analysis

We fully implemented the refactorings necessary to realize the usage scenarios of sections 3.1.1–3.1.3 and the feature renaming scenario of section 3.1.4. We
did not implement the Java refactorings necessary for declarative completeness refactoring since we lacked the time to do so. They would not be much more complicated, in principle, than RenameFeatureRefactoring, but would have more cases to consider (e.g., feature annotation not yet present vs. feature annotation present with some features already specified.)

Even in these relatively simple examples, it is apparent that our framework has significant power for implementing refactorings. The Java refactorings have full access to a compilation unit’s AST and can, in principle, rewrite any part of it. The power of using queries to determine the contexts in which certain menu items will appear, and to dynamically generate the labels of the menu items, has already been shown in the case of context-sensitive subquery menu items.[Vol06] Using subqueries to perform further processing on selected nodes before passing them on to a Java refactoring class provides a further level of power, as shown in the feature-renaming example.

We now turn to the question of ease of implementation and configuration. Coding the Java refactoring classes was the most time-consuming part of implementing query-based refactorings. Even though our API abstracts away many of the details into QueryBasedRefactoring, it may still be necessary to dig down several layers into the AST to find the necessary information, as RenameFeatureRefactoring demonstrates. However, once a Java refactoring is written, writing Tyruba configuration facts to integrate it into the JQuery menu is easy (a matter of minutes), and as sections 3.1.1–3.1.3 demonstrate, the same Java refactoring class can be reused for several fairly different applications.
Chapter 4

Related Work

We will examine previous work in two major areas related to the topic of our
thesis: tools for generation of Java annotations (including comment annotations)
and languages for specifying refactorings.

4.1 Annotation generation

A number of tools have been developed for analyzing properties of source code
to generate JML[LLP\textsuperscript{+}00] annotations as an aid to code verification with tools
such as ESC/Java[FLL\textsuperscript{+}02]. (JML annotations are not “annotations” in the
Java 5 sense; they are stored in method comments and encode properties such
as preconditions, postconditions and invariants which are supposed to hold true
for the annotated methods.) These tools include:

- Houdini[FL01], which analyzes code to generate JML annotations describ-
ing code properties such as preconditions, postconditions and invariants;

- that of Pavlova et al.[PBB\textsuperscript{+}04] which performs JML annotation generation
  based on a user-created security policy as an aid to verifying that code
  complies with those policies;

- CANAPA[CFJJ06], which propagates JML non-null annotations (e.g.,
based on existing user-provided annotations stating that certain variables
should not be null, it tries to place non-null annotations on things that
are used to set the non-null variables); and
• JAG[GG06], which generates JML annotations for verifying temporal properties of Java classes.

These existing tools are limited to using specific predefined annotations (those of JML), and the particular application domain of testing and verification. In contrast, our tool can manipulate any Java 5 annotation and is not limited to any particular application domain. Furthermore, our tool allows a high degree of user interaction in its annotation manipulations, whereas these tools generally proceed according to predefined algorithms with little or no user interaction.

Since version 3.3, the Eclipse JDT provides limited support for annotation generation in its Clean Up refactoring[LT07], which gives the options of adding the @Override annotation where appropriate, and adding @Deprecated annotations based on the presence of @deprecated Javadoc annotation in method comments. This refactoring is limited to predefined specific cases, whereas our refactorings process arbitrary annotations based on any criteria that can be expressed as a JQuery query.

4.2 Languages for specifying refactorings

Tourwé and Mens[TM03] and Muñoz[Bra03] developed a system for using SOUL [Wuy01] logic metaprogramming queries to search for “bad smells” (a term coined by Kent Beck [Fow99] to describe code patterns that suggest refactoring is needed) and suggest refactorings. They integrated their refactoring suggestion system with the Smalltalk Refactoring Browser[RBJ97]. There are obvious similarities between our system and theirs: both use logic metaprogramming queries to determine applicable refactorings for source code in an object-oriented language, and both aim to integrate with a sophisticated existing refactoring UI. Their system of predicates for expressing refactoring recommendations based on the properties of a selected code element is in theory very similar to our system of contextual refactoring predicates. However, on the implementation level, their graphical tool (unlike ours) only appears to support searching for refactoring recommendations for classes, and not other types of code elements such as methods or fields. Another difference between their work and ours is that they focus on applying existing Refactoring Browser refactorings, not on providing a framework for creating new refactorings.

More generally, there are various systems that use some sort of language for specifying program transformations. Stratego[Vis01] is a rich language for
expressing complicated program rewritings; in comparison to our system and
the one above, it is a way of expressing a particular transformation, not a
way of suggesting possible transformations. Unlike our refactorings, it is “not
intended for interactive program transformation” (Ibid., §5, emphasis mine).

Sun’s APT[APT04] tool (integrated into the javac compiler in Java SE 6), and
tools built on top of it such as Spoon[Paw05], generate or transform program
code based on the content of Java 5 annotations in the code; like Stratego, their
proposed uses appear to be for specific non-interactive transformations.
Chapter 5

Conclusion

In this thesis, we have presented JQuery-based refactorings and demonstrated that they provide a powerful and customizable way to manipulate Java annotations. We have created a Java framework based on Eclipse LTK for developing query-based refactorings, and we have implemented some basic refactorings with this framework. We have created TyRuBa configuration rules that permit the integration of these refactorings with the JQuery user interface in a highly customizably way. We have demonstrated our framework’s power by successfully applying it to situations such as adding certain annotations to selected code elements; adding the `@Override` annotation wherever it applies in a codebase; upgrading JUnit tests; and managing features in Chern and De Volder’s feature annotation system. We have found that the Java parts of JQuery-based refactorings are reasonably straightforward, though not painless, to implement, while the TyRuBa configurations for the refactorings are very easy to create.

5.1 Future Work

So far, we have only used our JQuery-based refactorings for annotation management. However, the generality of our framework would permit it to be used to develop more general refactorings and link them to JQuery. If we pursue this research direction, the similar existing work of Tourwé and Mens[TM03] and Muñoz[Bra03] might prove useful as a starting point. It would be interesting to see if we can “wrap” existing Eclipse LTK refactorings for use with JQuery, in a way similar to [Bra03]’s “wrapping” of Smalltalk Refactoring Browser refac-
It would not be hard to implement more extensive support for feature annotation management as described in Section 3.1.4. A full set of feature management refactorings would complement the ongoing work of Chern and De Volder and make it easier to experiment with their feature annotations.

Lloyd Markle has been working on a number of ways of improving the flexibility and power of the JQuery user interface.[MV08] In particular, he has started to add support for dragging and dropping JQuery nodes onto other nodes. This concept could provide new ways for the user to request refactorings; for instance, dragging an annotation to some methods could be interpreted as adding the annotation to the methods, or dragging some methods to a feature could be interpreted as adding the methods to the feature.
Bibliography


Appendix A

JQuery Predicates for Annotations

Here is a list of core fact predicates associated with annotations in JQuery and their meanings. Some existed prior to our work and some were added by the author. Precise definitions of the terms *annotation*, *annotation type declaration*, *annotation type element* and *element-value pair* may be found in [GJJB05] §9.6–9.7.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>annotationDeclaration(?T)</td>
<td>?T is an annotation type declaration</td>
</tr>
<tr>
<td>child(?T,?E)</td>
<td>Annotation type declaration ?T has declared attribute (“type element”) ?E</td>
</tr>
<tr>
<td>hasDefault(?E)</td>
<td>Annotation attribute (“type element”) ?E has a declared default value</td>
</tr>
<tr>
<td>annotType(?A,?T)</td>
<td>Annotation ?A is of type ?T</td>
</tr>
<tr>
<td>annotation(?A)</td>
<td>?A is an annotation</td>
</tr>
<tr>
<td>hasAnnotation(?E,?A)</td>
<td>?A is an annotation on program element ?E</td>
</tr>
<tr>
<td>attribute(?A,?name,?value)</td>
<td>Annotation ?A has attribute ?name with value ?value</td>
</tr>
</tbody>
</table>
Appendix B

Source Code Listings

B.1 QueryBasedRefactoring

The following listing shows the fields and methods visible to subclasses.

```java
/**
 * Abstract base class for Eclipse LTK refactorings based on the results of a JQuery query.
 * @author awjb
 */
public abstract class QueryBasedRefactoring extends Refactoring {
    // List of refactoring targets
    protected RefactoringTargetSet targets;
    // Current rewrite object
    protected ASTRewrite currentRewrite;
    public QueryBasedRefactoring (RefactoringTargetSet targets) {
        super();
        this.targets = targets;
    }
    @Override public RefactoringStatus checkFinalConditions(IProgressMonitor pm);
    @Override public RefactoringStatus checkInitialConditions(IProgressMonitor pm);
    @Override public Change createChange(IProgressMonitor pm);
    abstract protected Class getMatchingNodesType ();
    abstract protected void handleTargetNode (ASTNode foundNode, RefactoringTarget target) throws JavaModelException;
}
```
B.2 RefactoringTargetSet, RefactoringTarget

These encapsulate the data in a JQuery result set for easy use by our code.

```java
public class RefactoringTargetSet implements Iterable {
    private final String[] chosenVars;

    // Note: we use a set instead of a simple list because we can get duplicate
    // results out of the ResultSet when there are unchosen variables in the query
    // (e.g. if query contains ?A, ?B, ?C and only ?A and ?B are chosen variables,
    // then we may get redundant results with the same ?A and ?B but different ?C.)
    private Set results = new HashSet();

    public RefactoringTargetSet (String[] chosenVars) {
        this.chosenVars = chosenVars;
    }

    public RefactoringTargetSet (JQueryResultSet rs, String[] chosenVars)
        throws JQueryException {
        this(chosenVars);
        addResults(rs);
    }

    @Override
    public Iterator iterator() {
        return Collections.unmodifiableSet(results).iterator();
    }

    public boolean isEmpty () {
        return results.isEmpty();
    }

    public void addResults (JQueryResultSet rs) throws JQueryException {
        while (rs.hasNext()) {
            results.add(new RefactoringTarget(rs.next()));
        }
    }

    public class RefactoringTarget {
        private HashMap map = new HashMap();

        RefactoringTarget (JQueryResult result) throws JQueryException {
            for (String var : chosenVars) {
                map.put(var, result.get(var));
            }
        }

        public Object get (int index) {
            return map.get(chosenVars[index]);
        }
    }
}
```
### B.3 RenameFeatureRefactoring

Setting the `from` and `to` fields before generating changes is handled by a custom wizard page (source code not included.)

```java
public class RenameFeatureRefactoring extends QueryBasedRefactoring {
    private String from = null;
    private String to = null;

    public RenameFeatureRefactoring(RefactoringTargetSet targets) {
        super(targets);
        if (!targets.isEmpty()) {
            // Get the "from" feature name out of the target list.
            RefactoringTarget target = (RefactoringTarget)targets.iterator().next();
            from = (String)target.get(1);

            if (from != null && to != null && !from.isEmpty() && !to.isEmpty()) {
                try {
                    JQuery query = JQueryAPI.createQuery("feature(!featName) ");
                    query.bind("!featName", to);
                    JQueryResultSet rs = query.execute();
                    if (rs.hasNext()) {
                        rs.close();
                        return RefactoringStatus.createErrorStatus("Feature '" + to + "' already exists");
                    }
                    rs.close();
                } catch (JQueryException e) {
                    return RefactoringStatus.createErrorStatus("Error occurred while performing validation check");
                }
            }
        }
    }
}
```

@Override
public RefactoringStatus checkFinalConditions(IProgressMonitor pm) throws CoreException, OperationCanceledException {
    if (pm != null) {
        pm.beginTask("", 1); //$NON-NLS-1$
        pm.worked(1);
        pm.done();
    }
    if (from != null &

"Both name of feature to refactor and new name for feature must be provided";
}
}

@Override
public String getName() {
    return "Rename Feature";
}

@Override
protected void handleTargetNode(ASTNode foundNode, RefactoringTarget target)
    throws JavaModelException {
    if (foundNode != null && foundNode instanceof NormalAnnotation) {
        NormalAnnotation annot = (NormalAnnotation)foundNode;
        if (annot.getTypeName().getFullyQualifiedName().endsWith("Feature") ||
            annot.getTypeName().getFullyQualifiedName().endsWith("Export")) {
            for (Object obj : annot.values()) {
                MemberValuePair pair = (MemberValuePair)obj;
                if (pair.getName().getIdentifier().equals("names") ||
                    pair.getName().getIdentifier().equals("to")) {
                    if (pair.getValue() instanceof ArrayInitializer) {
                        ArrayInitializer array = (ArrayInitializer)pair.getValue();
                        for (Object o : array.expressions()) {
                            if (o instanceof StringLiteral) {
                                StringLiteral strLit = (StringLiteral)o;
                                if (strLit.getLiteralValue().equals(from)) {
                                    // Found it
                                    ListRewrite namesRewrite =
                                        currentRewrite.getListRewrite(array,
                                            array.EXPRESSIONS_PROPERTY);
                                    StringLiteral replacement = strLit.getAST().newStringLiteral();
                                    replacement.setLiteralValue(to);
                                    namesRewrite.replace(strLit, replacement, null);
                                }
                            }
                        }
                    } else {
                        throw new IllegalStateException("Element not found or not a feature annotation");
                    }
                }
            }
        } else {
            throw new IllegalStateException("Element not found or not a feature annotation");
        }
    }
}

@Override
protected Class getMatchingNodesType() {
    return NormalAnnotation.class;
}

/* getters and setters for from and to omitted */