Can Undergraduates Script Their Own Refactorings?

Jongwook Kim
Dept. of Computer Science
University of Texas at Austin
jongwook@cs.utexas.edu

Don Batory
Dept. of Computer Science
University of Texas at Austin
batory@cs.utexas.edu

Danny Dig
School of EECS
Oregon State University
digd@eecs.oregonstate.edu

Abstract

We present a status report on a project to build a refactoring engine whose primary goal is to allow undergraduate students to write classical and neo-classical refactorings (pull-up, class partitioning) and design patterns (visitor, framework) as parameterized refactoring scripts in Java. We explain the first step of our work that creates a reflection-like interface to expose the structure of an Eclipse JDT application as Java objects; methods of these objects are refactorings. Doing so hides the complexity of JDT refactoring code and tools, so that refactoring scripts can be written as compact Java methods. We present preliminary performance results of scripting JDT refactorings and sketch the next steps of our work.

Categories and Subject Descriptors D.2.7 [*SoftwareEngineering*]: Distribution, Maintenance, and Enhancement

General Terms prototype, refactoring, case study

Keywords refactoring scripts

1. Introduction

I (Batory) teach an undergraduate class in software design, where the central theme is to understand program construction from an automated perspective. Fig. 1 illustrates the basic idea. A programming task is to produce program P_4. One starts with an initial program P_0, which could be the empty program or an existing program, and progressively makes changes $\tau_1, \tau_2, \tau_3, \tau_4$ to transform (map) P_0 to P_4, henceforth written $P_0 \Rightarrow P_4$. Today each τ is manually hacked. In the world of automated design, each τ is a programmed transformation, making the process of $P_0 \Rightarrow P_4$ automatic. To accomplish this change in perspective and development, tools must be available to create transformations, add them to transformation libraries, and be able to invoke them on user programs.

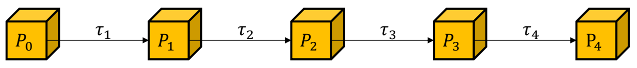

Figure 1. Program Derivation.

A key part of this course is material on *object-oriented (OO)* refactorings and OO design patterns. Most core refactorings (rename, move, delegate, partition-class, push-down, *etc.*) are conceptually easy to understand. Contemporary texts on design patterns provide informal step-by-step descriptions on how each pattern can be introduced into a program [9, 11, 13]. A few, such as the visitor pattern, can be created automatically just by identifying a "seed" method. (All methods in a class hierarchy that have the same return type, name, and argument signature of the seed can be automatically moved into a newly created visitor class). Others, such as framework, require more information and more activity from users. (Identify the classes to be moved into the framework, which methods of these classes are to be partitioned – via the template method – into a framework method and hook method, *etc.*) Given this extra preparation and information, the class scaffolding that separates a framework from its plug-in can be erected automatically.

One of the best ways to teach students about refactorings and design patterns is not only to *use* them, but also to have them *write* programs that sequence refactoring steps to mechanize program changes. Doing so introduces them to *metaprogramming*; objects are programs and methods are refactorings (or more generally, transformations that do not necessarily preserve behavior). In effect, students are writing the τ transformations to mechanize the $P_0 \Rightarrow P_4$ mappings of Fig. 1.

The obvious question is: what language should be used to script refactorings? There are many proposals with distinguished merit [1–5, 12, 15, 17, 19, 20], but all fall short. There is no time for undergraduates to fully learn yet another language (functional or otherwise) to manipulate programs. Or for them to learn to use program transformation systems/utilities, such as the *Eclipse Language Toolkit (LTK)* framework [10]. Although these systems are monuments of engineering prowess, their learning curve is measured in increments of weeks or months. Undergraduates don't have the time, and perhaps not the background, to learn how to use such tools.

Since classroom instruction uses Java, the obvious answer to the above question is to use Java as a metaprogramming language. Our thought was to use *Eclipse Java Development Tools (JDT)* [7] and provide a programming interface/façade that presents an Eclipse project, its packages, classes, methods, fields, *etc.* as Java objects. Their methods are JDT refactorings. Scripting existing refactorings would be fundamentally no different than having undergraduates import an existing Java package and use it to write their programs (in this case, refactoring metaprograms). At least, this is our conjecture.

This paper describes our experiences to date using the Eclipse JDT Refactoring Engine and to chart our next steps to simplify the construction of refactoring engines so that undergraduates can script their own refactorings. Henceforth, we refer to "Eclipse" as "Eclipse JDT" and "Eclipse Refactoring" as "JDT Refactoring" in the rest of this paper.

WRT '13, October 27, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2604-9/13/10. . . $15.00.
http://dx.doi.org/10.1145/2541348.2541351

2. Reflective Refactoring

Let $\mathcal{P}$ denote a Java project in Eclipse. Our approach is to leverage the idea of reflection – we define a class $RClass$ whose instances are the class declarations in $\mathcal{P}$; we define classes $RMethod$ and $RField$ whose instances are the method and field declarations of $\mathcal{P}$, and so on. When $\mathcal{P}$ is compiled by Eclipse, a set of tables (one for $RClass$, $RMethod$, $RField$, *etc.*) is created, where each row corresponds to a class, method, and field instance of $\mathcal{P}$. The fields of $RClass$, $RMethod$, and $RField$ – henceforth called *Reflective Refactoring* ($\mathcal{RR}$) classes – define the association and inheritance relationships among table rows (A is a superclass of B, foo is a method of A, *etc.*). The methods of $\mathcal{RR}$ classes expose primitive Eclipse refactorings or composite refactorings.

2.1 A Simple Example

To get a flavor of the scripts that we envision students may write, here is a simple example. A GUI observer is to be created to display the current values of selected non-final static fields. A user can select which static fields to observe, run an $\mathcal{RR}$ refactoring that modifies the user's program. The program is run, debugged using the observer, and further developed. Eventually, another $\mathcal{RR}$ refactoring removes the observer from the program. The code in Fig. 2 shows the basic $\mathcal{RR}$ calls. The steps to introduce the observer are:

1. Identify the project and package (here: project $PTable$ and package $tables$),
2. Introduce a singleton $Observer$ class with its appropriate fields and methods into the package,
3. Identify static fields to observe (here: field $Node.ctr$ and $Node.age$),
4. Invoke an $observe$ $\mathcal{RR}$ refactoring to create a $setter$ method for each field, where the $setter$ method announces to the singleton $Observer$ that its field value has been updated.

```
RPackage pkg = RProject.getPackage("PTable", "tables");

// add Observer class to the "tables" package
RClass obs = pkg.makeObserver("Observer");

// observe static fields Node.ctr and Node.age
RClass cls = pkg.getClass("tables.Node");
RField fld = cls.getField("ctr");
obs.observe(fld);     // observe field

fld = cls.getField("age");
obs.observe(fld);     // observe field
```

Figure 2. An Observer Script.

This example relies on previously written $\mathcal{RR}$ methods that directly invoke Eclipse refactorings: $makeObserver(N)$ adds an observer class (with appropriately defined fields and methods) with name N to the specified package and $observe(F)$ invokes the **encapsulate field** refactoring on F and alerts the observer whenever the $setter$ is called. Fig. 3 shows the result of the $observe$ refactoring when the age field is the target of observation.

Fig. 4 shows the $\mathcal{RR}$ script that undoes the changes made by Fig. 2. We envision that $\mathcal{RR}$ scripts will eventually be listed with the predefined, hard-coded refactorings in the Eclipse refactoring menu and invoked via the Eclipse GUI. The idea here is that programmers can either import $\mathcal{RR}$ refactorings (perhaps from an on-line repository) or write their own.

```
// before refactoring
public class Node {
   public static int age;
   ...;
}
```

```
// after refactoring
public class Node {
   private static int age;
   public static void set(int x) {
      age = x;
      Observer.singleton.observe("age", age+"");
   }
   public static int get() {
      return age;
   }
   ...;
}
```

Figure 3. Transformation of $observe$ Refactoring.

```
RPackage pkg = RProject.getPackage("PTable", "tables");
RClass obs = pkg.getClass("tables.Observer");

// remove field subjects
RClass cls = pkg.getClass("tables.Node");
RField fld = cls.getField("ctr");
fld.removeSubject(obs, "public");

fld = cls.getField("age");
fld.removeSubject(obs, "public");

// delete Observer class
pkg.delete(obs);
```

Figure 4. A Script to Remove an Observer.

2.2 More Complex Examples and Current Status

We implemented an $\mathcal{RR}$ refactoring called $makeVisitor$ that automatically creates a visitor, given a method "seed" [14]. That is, all methods in a class hierarchy that have the same return type, name, and argument signature as the seeds are moved into a newly created visitor. We also implemented the inverse of this refactoring. Given a visitor that was created by $makeVisitor$, $undoVisitor$ returns the program to its original state.

We encountered plenty of challenges. Most were specific to Eclipse, which we discuss later in Section 3. One that is not limited to Eclipse is the lack of a common and precise definition of some refactorings. Fowler's definition of **move method** is a case in point [8]: it is extraordinarily vague. NetBeans (7.3), IntelliJ IDEA (12.1.4), and Oracle's JDeveloper (11g Release 2) refuse to move polymorphic methods. Eclipse 4.2.2, in contrast, *can* move polymorphic methods, providing that it does not invoke $super$ (complying with other constraints) and that one leaves behind a delegate. As many of the methods that we wanted to use as seeds for visitors reference $super$, we had to generalize **move instance method**. This required a simple refactoring that lifts $super$ calls into its own method $callSuper$, leaving behind the original method with a $callSuper$ instead of a $super$ reference. The modified original method no longer references $super$, and Eclipse can move it.

We executed $makeVisitor$ on programs where over 270 methods were moved into a single visitor in approximately 540 seconds (9 minutes). Automated support for visitor creation is essential; we are confident that visitors with many fewer methods could not be created manually. We are now preparing user-studies to confirm this hypothesis.

Interestingly, the individual length of our $makeVisitor$ and $undoVisitor$ scripts are a few lines longer than the Java scripts given in Fig. 2 and Fig. 4. As the Visitor pattern is among the most complex patterns, we are confident that most $\mathcal{RR}$ methods will not be very long.

Whether $\mathcal{RR}$ methods are easy for undergraduates to write remains to be seen. Eclipse refactorings, and refactorings in general, have many side-effects. For example, a single invocation of **change method signature** can alter all methods related by run-time polymorphism.[1] A fairly deep knowledge about the semantics of Eclipse refactorings is needed to write correct $\mathcal{RR}$ scripts. We are now preparing user studies with undergraduate classes to confirm this hypothesis (and to confirm results of prior work [18]). More details on our work are presented in [14].

3. Experiences with Eclipse

It is well-known that program transformation systems are difficult to use, and are typically used only by their creators. They are intimidating and are not for casual users. To us, the JDT Refactoring Engine is no different.

Eclipse refactorings use the LTK framework to provide language-neutral refactoring APIs [10]. LTK consists of a refactoring core, UI components, and incorporates the JDT's UI and language-specific support. We wanted to separate UI components from the refactoring core but failed at this task due to highly tangled source code. We also looked at JUnit tests where refactorings were invoked programmatically. Here again, we were unable to decipher the arguments to these calls – they too were highly tangled in the testing framework for us to understand how we could adopt JUnit codes to script refactorings.

Eventually we recognized two possibilities: One utilized basic LTK APIs only to invoke refactorings. The other used existing UI (refactoring dialogs) to trigger refactorings. Using UIs, we could automate exactly the same procedures that occur when Eclipse replays refactoring scripts, and it allowed us to measure the actual time spent on overall refactoring process *and* accurately estimate the overhead of JDT refactorings due to UI operations. We chose the second option to both understand refactoring inputs and to invoke refactorings using Eclipse's XML-scripts.

In general, we were unable to find useful (or up-to-date) documentation on the JDT Refactoring Engine or an XML schema for the scripts we had to write. It was unpleasant to reverse engineer, via experiments, the information that we needed. It is this kind of overhead that we feel disqualifies Eclipse for our project.

Here is another potential disqualification: Our experiments showed that Eclipse refactorings are a bit too slow to be interactive on large programs. Creating a visitor with 26 methods takes about 70 seconds. For 276 methods, $makeVisitor$ takes over 9 minutes to complete and about 7 minutes to undo. While this is not terrible, it is not the performance that would encourage programmers to dynamically create views of their applications.

Yet another disqualification is the number of Eclipse refactoring bugs that we encountered. In building the $makeVisitor$ transformation alone, we documented 6 bugs, 5 new and one reported five years ago. Some bugs we could fix ourselves within our $\mathcal{RR}$ methods; others required us to manually repair refactored programs. Obviously, there is a limit to how much manual reparation can be tolerated before the utility of $\mathcal{RR}$ automation becomes questionable. That is, if too many errors are introduced into a refactored program, it is hard to argue the benefit of applying $\mathcal{RR}$ transformations in the first place. We chose Eclipse because we thought it provided a solid foundation on which to build our work; to our dismay, it behaves more like quicksand. We recognize that writing a refactoring engine is not easy. Perhaps it is time to rethink how such engines can be built in a more reliable way. This is part of our future work.

[1] **Change method signature** alters methods beyond a single class hierarchy when interfaces containing a method seed are involved.

Bug ID	Eclipse Bugzilla #	Description
B1	217753	When a method with reference to $static\ import$ is moved, the reference type is qualified incorrectly.
B2	385550	When a method with reference to inherited fields is moved, the field access is not updated.
B3	385989	When a method with reference to $import$ type is moved, the reference type is not qualified as the $import$ type.
B4	404471	When a method with $@Override$ annotation is moved, the annotation is also moved with the method.
B5	404477	When a method is moved, wrong detection of duplicate methods occurs.
B6	411529	When a method with reference to $protected$ methods is moved to other package, $protected$ is not modified to $public$.

Table 1. Eclipse Bug Reports.

The JDT Refactoring Engine is typical of the state-of-the-art refactoring tools. It is not a system that can be easily given to typical undergraduates to pick up, use, and modify.

4. Conclusions and Future Work

We have spent the better part of two years on this project. We committed ourselves to use Eclipse, at least initially, to better understand the problems of contemporary refactoring engines. We also became familiar with Semantic Designs *DMS (Design Maintenance System)* [2], a very impressive, industry-hardened tool for large-scale program transformations. We found it had different problems of equal or greater magnitude (e.g., DMS has its own proprietary programming language called $Parlanse$).

We are now convinced that further development of $\mathcal{RR}$ within Eclipse, using the JDT and LTK, is not the way to go. We were inspired by Simonyi's *Intentional Programming (IP)* [16] and have built an IP-like prototype to write $\mathcal{RR}$ scripts in Java, but to call our refactoring implementations, not those of Eclipse. We hope to have a full-scale prototype of this engine by Spring 2014. Again, our goal is to create a refactoring engine that undergraduates can understand and use. We are not aiming our effort at people with hard-core interests writing arbitrary program transformations; the LTK framework and DMS engine are for them.

After all this, we still have to add analyses that evaluate preconditions for refactoring transformations. We know (and have been reminded repeatedly) that these analyses are the most difficult to write. We do not disagree, but the main problem is that we want to encode constraints into a more abstract, understandable and reusable form for our students (as opposed to, say, coding them in Java as is currently done in the JDT Refactoring Engine). Because $\mathcal{RR}$ takes a database viewpoint, where all $\mathcal{RR}$ data of a program is stored in tables (and $\mathcal{RR}$ methods act on rows of these tables), a main-memory Prolog or Datalog engine could be used to express and evaluate preconditions expressed in terms of database constraints, without inventing yet another DSL to do so. While this still may not give us the speed that we seek, there are plenty of ways to tune $\mathcal{RR}$ implementations. One is to ignore the access privileges of members until the program has been refactored, and then recompute the privileges of members to support a type-correct structure. Again, this is future work.

And finally, the broader picture of our research is that it is part of the *Change-Oriented Programming Environment (COPE)* project, whose goal is to raise transformations to first-class entities in software development in general, and the Eclipse IDE in particular [6]. Our ultimate goal is to allow people to create views of programs, to block out irrelevant features, and to reorganize code (using refactorings) to present a simpler representation of a program. Being able to create visitors on-the-fly to inspect a set of related methods, or to eliminate frameworks and plug-ins by compacting their classes and eliminating generality, and just as easily undo visitors and framework compactions, is a capability that COPE (and ultimately Eclipse) should have.

Acknowledgments. We gratefully acknowledge support for this work by NSF grants CCF-1212683 and CCF-1213091.

References

[1] E. Balland and et al. Tom: piggybacking rewriting on java. In *RTA*, 2007.

[2] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *Proc. 26th Int. Conf. on Software Engineering*, 2004.

[3] M. Boshernitsan and S. L. Graham. iXj: interactive source-to-source transformations for java. In *OOPSLA Companion*, 2004.

[4] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.*, June 2008.

[5] J. R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, Aug. 2006.

[6] D. Dig and et al. Science and Tools for Software Evolution. NSF Grant CCF 1212683, 2012.

[7] EclipseJDT. Eclipse Java development tools (JDT). `http://eclipse.org/jdt/`, 2013.

[8] M. Fowler. Move Method. `http://www.refactoring.com/catalog/moveMethod.html`, 2013.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[10] L. Frenzel. The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. `http://www.eclipse.org/articles/Article-LTK/ltk.html/`, 2013.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman, 1995.

[12] M. Hills, P. Klint, and J. J. Vinju. Scripting a refactoring with Rascal and Eclipse. In *Workshop on Refactoring Tools*, WRT, 2012.

[13] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.

[14] J. Kim, D. Batory, and D. Dig. Scripting Parameteric Refactorings in Java to Implement Design Patterns. submitted to ICSE, 2014.

[15] H. Li and S. Thompson. A domain-specific language for scripting refactorings in erlang. In *FASE*, 2012.

[16] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *OOPSLA ONWARD*, Oct. 2006.

[17] F. Steimann, C. Kollee, and J. von Pilgrim. A refactoring constraint language and its application to eiffel. In *ECOOP*, 2011.

[18] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *ICSE*, 2012.

[19] van den Brand and et al. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *Compiler Construction*, 2001.

[20] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *ICSE*, 2006.