Scripting a Refactoring with Rascal and Eclipse

Mark Hills   Paul Klint   Jurgen J. Vinju

Centrum Wiskunde & Informatica, Amsterdam, The Netherlands   INRIA Lille Nord Europe, Lille, France

{Mark.Hills,Paul.Klint,Jurgen.Vinju}@cwi.nl

Abstract

To facilitate experimentation with creating new, complex refactorings, we want to reuse existing transformation and analysis code as orchestrated parts of a larger refactoring: i.e., to script refactorings. The language we use to perform this scripting must be able to deal with the diversity of languages, tools, analyses, and transformations that arise in practice. To illustrate one solution to this problem, in this paper we describe, in detail, a specific refactoring script for switching from the Visitor design pattern to the Interpreter design pattern. This script, written in the meta-programming language Rascal, and targeting an interpreter written in Java, extracts facts from the interpreter code using the Eclipse JDT, performs the needed analysis in Rascal, and then transforms the interpreter code using a combination of Rascal code and existing JDT refactorings. Using this script we illustrate how a new, real and complex refactoring can be scripted in a few hundred lines of code and within a short timeframe. We believe the key to successfully building such refactorings is the ability to pair existing tools, focused on specific languages, with general-purpose meta-programming languages.

Categories and Subject Descriptors D.2.7 [*Distribution, Maintenance, and Enhancement*]: Restructuring, reverse engineering, and re-engineering

General Terms refactoring tools; meta-programming

Keywords refactoring scripts; design patterns; program transformation

1. Introduction

Refactorings [11, 21, 22] are semantics-preserving program transformations, applied by hand or with the help of refactoring tools. Most IDEs now include such tools to automate common refactorings, such as rename method [11, pp. 273–274] or encapsulate field [11, pp. 206–207]. Unfortunately, creating new refactorings is still quite challenging, requiring in-depth knowledge of the language being refactored, tight integration with IDE features (wizards, undo capabilities, etc), and knowledge of many analysis and transformation algorithms and supporting data structures. One approach is to use language support provided by APIs targeted at a specific IDE, such as the Eclipse LTK [12]. However, this still requires writing the refactoring in a general-purpose language (in the case of the LTK, Java) not targeted at the domain. Another approach is to use a meta-programming language, such as Rascal [18, 19], which gives the benefits of using a domain-specific language, but requires rewriting a significant amount of analysis code to gain access to information needed in the refactoring code (e.g., which definitions a name refers to).

In an attempt to have the best of both worlds, in this paper we focus on providing tight integration of an IDE with existing analysis and refactoring capabilities — the Eclipse Java Development Tools, or JDT[1], which provides IDE support for developing Java programs and includes a number of tools (e.g., for refactoring Java code) and analyses (e.g., bindings of types to names, links between name uses and declarations) — with a meta-programming language for the analysis and transformation of source code — Rascal. Specifically, we focus on using Rascal as a prototyping environment for the creation, evaluation, and improvement of new refactorings and refactoring scripts[2]. As used here, a *refactoring script* is a high level meta-program that orchestrates and implements the mechanics of a refactoring by:

- calling existing parsing, name resolution and type resolution functionality,
- building an abstract model of the system under investigation,
- analyzing this abstract model to check preconditions and compute parameters for the transformation,
- performing source-to-source transformations, given previously computed parameters, potentially using existing refactorings.

[1] `http://www.eclipse.org/jdt`

[2] Note that this differs from a refactoring script in Eclipse, which allows manually applied refactorings to be replayed.

In this paper we report on our experience in applying Rascal and the Eclipse JDT in constructing a script-support refactoring[3] called Visitor-to-Interpreter (V2I for short). V2I takes source code organized according to the Visitor design pattern [14, page 331] and transforms it to source code organized according to the Interpreter design pattern [14, page 243]. V2I appeared in our earlier paper [15], where we studied the *effect* of this refactoring on the maintainability of source code. Here we focus instead on the *mechanics* of the refactoring, providing an overview of the entire refactoring and highlighting the interaction between Rascal and Eclipse. In fact, we have used V2I to refactor the Rascal interpreter, written in Java, which has been running the converted code since December 2010. This conversion has opened up additional opportunities for optimizing the interpreter, such as through caching information related to both optimizing lookups and to pattern matching.

In summary, we believe that the V2I case supports our claims that Rascal is a good language for scripting unforeseen refactorings and that the Rascal/Eclipse JDT integration provides an effective way to access facts about the syntax and static semantics of Java.

Roadmap Section 2 provides a short introduction to Rascal and to the JDT library, which provides access to the Eclipse JDT from within Rascal code. Section 3 then discusses the V2I transformation in detail, including the needed analyses, the Eclipse and Rascal portions of the transformation, and any manual steps that need to be taken. Following this, Section 4 discusses refactoring scripts in the context of refactoring tools and presents related work, including a comparison of different meta-programming tools that might be used instead of Rascal. Finally, Section 5 concludes.

2. Overview

Rascal [18, 19] is a DSL for source code analysis and manipulation. Below we provide an overview of the language by listing its main design ingredients. More details can be found at `http://www.rascal-mpl.org`.

Procedural control-flow Functions, procedures and structured control-flow primitives provide a well known and easy-to-understand framework for computation. This is extended with more advanced control flow mechanisms, such as lexically scoped backtracking, general traversal, and closures, to provide the power needed for meta-programming.

Immutable data Algebraic data types, a static type system with local inference, as well as built-in types such as lists, sets, relations and maps provide a reuse-friendly environment without the problems associated with references and destructive updates.

[3] The refactoring is not fully scripted, since it requires some human intervention at specific points in its execution.

```
data Entity = entity(list[Id] id);
data Id = package(str name)
| class(str name)
| class(str name, list[Entity] params)
| interface(str name)
| interface(str name, list[Entity] params)
| method(str name, list[Entity] params,
         Entity returnType)
| field(str name)
;
public Entity Object =
entity([package("java"), package("lang"),
class("Object")]);
```

Figure 1: Rascal JDT Entities.

Integrated syntax definition Rascal provides support for generalized scannerless context-free parsing and fully general disambiguation filters. This allows for parsing a wide range of legacy and embedded languages. Rascal also provides ambiguity detection to detect potential ambiguities in defined grammars [3].

Domain specific expression operators Rascal expressions target operations commonly needed in meta-programming, including relational operators and comprehensions, pattern matching combinators (regular expressions, algebraic signatures, set/lists, paths, etc.), string templates with an auto-indent feature, and concrete syntax fragments.

Java/JVM based Rascal is built with Java, which allows us to deploy it anywhere, and is usable inside Eclipse as a plugin. Java methods can also be invoked as Rascal functions, a feature used extensively in this paper to reuse existing Eclipse JDT functionality (Section 2.1).

Eclipse Interaction Rascal includes an interactive console available outside of Eclipse or as part of the Eclipse Rascal plugin. This is convenient for prototyping Rascal programs, including the refactoring steps shown in this paper. Rascal can also directly instantiate language specific Eclipse editors [8] for any language implemented in Rascal, enabling the addition of new annotators, code outliners, and menu items.

2.1 The Rascal JDT Library

When we decided to refactor the Rascal interpreter, we could have created our own fact extraction framework for Java. However, we believe that it makes sense to try and reuse the existing features provided by the Eclipse JDT for analysis and code manipulation. We previously created a small bridge between the JDT and Rascal, referred to as the JDT library[4].

The JDT library implements the omnipresent Extract-Analyze-SYnthesize (EASY) design pattern in software analysis. Given an Eclipse Java project, the JDT library extracts facts about the code and stores these facts in a number of typed sets and relations. These sets and relations generally

[4] Acknowledgment: Bas Basten wrote this library.

Extracted Fact	Description	Decls	Uses
types	classes, interfaces, enums	x	x
methods	methods	x	x
methodDecls	methods	x	
fields	fields	x	x
fieldDecls	fields	x	
variables	variables, method parameters	x	x
classes	classes	x	

Figure 2: Rascal JDT Interface: Extracted Entities.

Extracted Fact	Description
modifiers	modifiers on definitions (e.g., public, final)
implements	interface $\times$ implementer
extends	class or interface $\times$ extender
declaredMethods	class or interface $\times$ method declaration

Figure 3: Rascal JDT Interface: Extracted Relationships.

contain either information on *entities* (packages, classes, methods, etc) or relationships between entities. Entities are represented as a Rascal datatype made up of a list of identifiers representing different Java constructs. Figure 1 provides an example of some of the different identifiers, including those used to represent classes (with or without type parameters), interfaces, methods, and fields, and also shows an example entity for class `Object`.

Figure 2 lists the extracted entity information (mapping entities to locations where those entities are declared and/or used) that is used in the V2I transformation discussed in Section 3. The first column in this table shows the name of the extracted fact, as used in the Rascal code for the transformation, with the second column providing a brief description. Some of the facts include both declarations and uses of the entities, while others include just declarations. This is indicated by an `x` in the column for either **Decls**, **Uses**, or both. For instance, `methods` maps both uses (invocations) and declarations of methods to the location in the source code where the use or declaration appears, while `methodDecls` is a subset of this that contains just the declarations[5].

Similarly, Figure 3 shows the extracted relationship information used in V2I. Again, the first column shows the fact's name, while the second provides a description. For instance, `implements` maps interfaces to the classes that implement these interfaces, while `extends` maps interfaces and classes to the interfaces and classes (respectively) that extend them.

2.2 Rascal Code Examples

Figure 4 provides several example snippets of Rascal code from the V2I refactoring, occasionally with slight modifications to remove dependencies that are not shown.

Example 1 shows a function that, given a set of interfaces, returns all interfaces that directly or indirectly extend one of the interfaces in the set, as well as all classes that directly or indirectly implement one of these interfaces. The `solve` statement grows this set one level at a time until no new interfaces or classes are added.

Example 2 gets all methods named `visit*` in a specified class. First, all class $\times$ method pairs are enumerated where the class matches that passed to `getVisitorsInClass`; then, matching is used to extract the method name from the entity representing the method. Regular expression matching ensures this name starts with `visit`. Finally, the inverse of the `methodDecls` relation is used to get back the locations (there should be just one, but the result of subscripting a binary relation is a set). This information is then returned as name $\times$ location $\times$ class entity $\times$ method entity pairs.

Example 3 shows how the set of non-public fields in a visit method (described more in Section 3) is calculated. Relation `frel` contains all uses of fields in one of the source files with locations stored in `classPaths`, filtered to include only fields declared in the Rascal interpreter code (based on package name). Relation `frel2` then filters this by the location of the fields in the source files (the comparison of offset `o` with information from `overlapsAux`, which has locations of the visit methods), leaving only those fields used inside visit methods. Relation `frel3` further filters this, leaving only those fields not declared `public`. Finally, `npFields` is defined as just the set of field entities, with one entry for each field used in a visit method and declared as non-public.

The last example, Example 4, shows an example of generating a string using Rascal's string comprehension and indenting capabilities. `methodCode` is the code for one of the new methods being generated by the V2I refactoring. String comprehensions are used to insert the values of various expressions into the correct positions in the string: `paramsForSig` for type parameters, `readable(mr)` for the method type (`readable` pretty-prints the entity representing the type), `readable(instance)` for the parameter type, and `methodBody` for the generated method code. The `'` maintains alignment on the left in the generated code.

3. The V2I Transformation

Figure 5 provides an overview of the V2I transformation. At a high level, the transformation needs to identify the visitor methods to convert into interpreter methods; perform this conversion by applying a number of simple transformations to the code; ensure that fields and methods used in this transformed code will be visible to the interpreter code (e.g., that the fields have public getters and setters); write these converted methods into new interpreter classes; and clean up by removing the visitor methods. This requires support both from the Eclipse JDT, which provides information on the Java source being transformed, and which provides several refactorings used in the script; and from Rascal, which performs the source transformation, generates the new classes,

[5] This distinction is mainly for backwards compatibility, since `methodDecls` was added later, but we did not want to remove declarations from `methods` and potentially break existing code.

```
// Example 1: Find all interfaces and classes that implement one of the
// interfaces in the set of interfaces given as a parameter.
public set[Entity] findImplementers(Resource r, set[Entity] interfaces) {
  implementers = { *getInterfaceImplementers(r, i) | i <- interfaces };
  solve (implementers) {
    implementers += { *getClassExtenders(r, i) | i <- implementers };
  }
  return implementers;
}

// Example 2: Given a class, get all methods in this class named visit*
// along with their source code locations.
public rel[str mname, loc mloc, Entity owner, Entity method] getVisitorsInClass(Resource r, Entity class) {
  im = invert(r@methodDecls);
  return { <mns,l,cn,mn> | tm:<class,mn> <- r@declaredMethods, entity([_*,method(mns,_,_),_*]) := mn,
                           /visit.*/ := mns, l <- im[mn] };
}

// Example 3: This builds the set of non-public fields to refactor over several steps. frel1 is all
// fields used in the same source file as the methods being refactored; frel2 then limits this to
// just those within the methods being refactored, based on where the field is used in the file; frel3
// then restricts this further to only those that are not public. npFields is then just these fields.
frel = { < l.path, l.offset, e > |  <loc l,e> <- rascal@fields, l.path in classPaths,
                                    entity([package("org"),package("rascalmpl"),_*]) := e };
frel2 = { <e,fp,o,i,dm,l> | fi:<fp,o,e> <- frel,
                            <fp,bn,en,l,i,dm> <- overlapsAux, bn <= o, o <= en };
frel3 = { fri | fri <- frel2, \public() notin (rascal@modifiers)[fri[0]] };
npFields = frel3<0>;

// Example 4: Generate the evaluate method code as a string.
methodCode = "@Override
             'public<paramsForSig> <readable(mr)> __evaluate(<readable(instance)> __eval) {
             '  <methodBody>
             '}";
```

Figure 4: Rascal Code Examples from V2I.

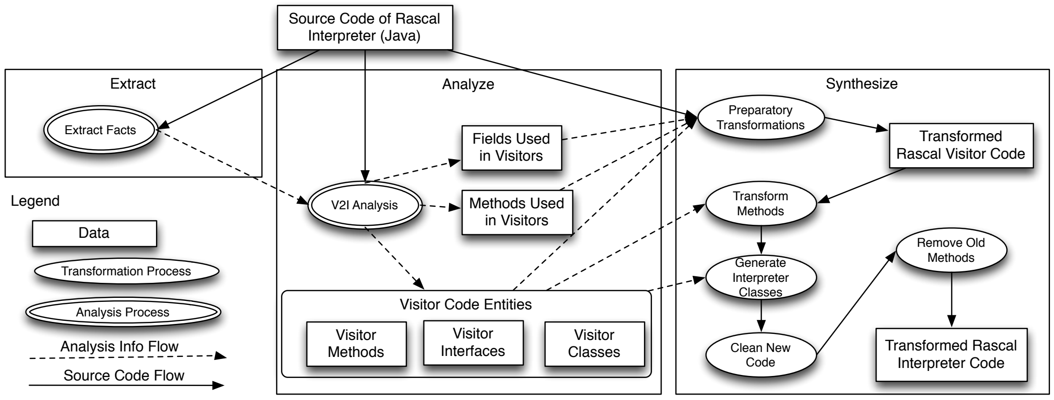

Figure 5: V2I Transformation Overview.

tracks various analysis facts in sets and relations, and, in general, drives the entire process, coded as a Rascal program.

The first step shown in Figure 5 for V2I is to extract information about the source program, in this case the Rascal interpreter itself (written in Java). This information is fed into an analyzer, which determines which code implements the Visitor pattern (interfaces, classes, and visit methods) as well as which dependencies of this code (specifically, fields and methods) may need to be modified. Extraction and analysis are both covered in Section 3.1.

Once the extraction and analysis are complete, a number of discrete transformation steps are applied. The preparatory transformations make small semantics preserving changes to the code, allowing it to be relocated into other classes (Section 3.2). The visit methods are then transformed into interpret methods, which are placed into newly generated interpreter classes. (Section 3.3). Finally, using Rascal code and a few manual steps, the code is cleaned up and the old visit methods are removed. The end result is a Rascal interpreter, equivalent to the original, but now structured according to the Interpreter pattern (Section 3.4).

Figures 6 and 7 show, at a high level, how the code looks before, and should look after, the transformation. In the initial system of Figure 6, multiple classes are defined that implement visit methods to perform different types of evaluation. The first method, defined in `Evaluator`, takes an

```
class Evaluator implements IASTVisitor {
  public A visitN(T x) {
    visitBody1;
  }
}
class PatternEvaluator implements IASTVisitor {
  public B visitN(T x) {
    visitBody2;
  }
}
```
Figure 6: Before Transformation.

```
class T extends ast.T {
  public A interpret(Evaluator e) {
    transformedBody1;
  }
  public B interpret(PatternEvaluator pe) {
    transformedBody2;
  }
}
```
Figure 7: After Transformation.

AST node of type `T`, performs some computation, and returns a result of type `A`. The second method, defined in `PatternEvaluator`, is an implementation of the same visit method, with the same name and parameter, but instead returns a result of type `B`. After the transformation, the code should be structured as shown in Figure 7. The `visitN` method from `Evaluator` is now an `interpret` method accepting a parameter of type `Evaluator`, with a transformed method body and the same return type `A`, while the `visitN` method from `PatternEvaluator` is now an `interpret` method accepting a parameter of type `PatternEvaluator`, with a transformed method body and the same return type `B`. Both methods are in the same subclass of AST node class `T`.

Table 1 provides the lines of code for (in order) the Rascal portion of the V2I code; the Java portion of the V2I code; the overall size of the Rascal interpreter, before the refactoring; the size of the seven implementation classes whose methods were transformed; and the size of the generated interpreter classes. Overall development time for the refactoring was approximately 2 person-weeks.

3.1 Fact Extraction and Fact Analysis

The V2I analysis is used to identify the locations of the code that must be transformed to convert from the Visitor to the Interpreter pattern. To find this information, the analysis uses `extract` functions provided by the JDT library. These functions are designed to extract facts, including those shown above in Figures 2 and 3, from either specific Java classes or entire Eclipse projects. This information is built by traversing the internal DOM representation of Java source files provided by the JDT, with specific DOM nodes yielding facts such as the locations of method invocations, the interfaces implemented by a class, and the declarations of fields.

Two facts of immediate interest are the `extends` and `implements` facts, which model the underlying relations of the same name in Java. To find the visit methods, the analysis first identifies all the visit classes. This is done in two steps. First, all interfaces which extend a base visitor interface[6] (for Rascal, this is `IASTVisitor`) are identified using a fixpoint computation[7] over the `extends` relation. Second, using these interfaces as a seed, all classes which implement one of these interfaces, or which extend a class that does, are identified using a second fixpoint computation. For Rascal, this finds 14 classes; this is manually pared down to 7 classes, with the other 7 being small, focused classes that only visit a small number of node types in the AST.

Item	Lines of Code
V2I: Rascal Code	489
V2I: Java Code	500
Rascal Interpreter	102997
Source Visitor Code	5892
Generated Interpreter Code	9926

Table 1: Lines of Code for Rascal and V2I.

In the 7 visitor classes, the analysis next must identify the methods to transform. The analysis uses a heuristic to identifying the visitor methods in `IASTVisitor`: visitor methods are those that have names starting with `visit`, a heuristic that works in this case because `IASTVisitor` is generated from the Rascal grammar. If a non-uniform naming scheme were used instead, some information about the Visitor pattern could be used to identify candidate methods (e.g., visit methods should accept one parameter, which should be an AST node), but human intervention would ultimately be required. Using this heuristic, along with the `methodDecls` fact, 536 methods in `IASTVisitor` are identified as visit methods. These are identified by using pattern matching over the entity representation of the method names: a matching method entity begins with the entity for `IASTVisitor` and ends with a method entity id whose name (checked with regular expression matching) begins with "visit". The implementations of these methods are then identified in each of the 7 classes, again using pattern matching over the entities representing methods in these classes, leading to a total of 910 methods to refactor. Of these, 536 methods are in a class that provides a default behavior for each visit method (returning `null`), while the rest are spread through the other 6 classes.

With these methods identified, the analysis next finds any dependencies the methods have on local fields and methods which are non-public. This is needed because the code in the visit methods is being moved to new classes which do not inherit from the current visitors and are not in the same packages. More restrictive access levels would thus prevent access to these fields and methods from within the transformed code. At the time V2I was created the Rascal JDT library did not provide direct access to the ASTs of Java methods (the library now provides these, based on the underlying AST representa-

[6] The analysis assumes that all visitors of interest extend, directly or indirectly, a specific interface

[7] In Rascal, fixpoint computation is provided as a control flow construct, the `solve` statement.

```
public Result<IValue> visitStatementAssert(Assert x) {
  Result<IValue> r = x.getExpression().accept(this);
  if (!r.getType().equals(tf.boolType()))
    throw new UnexpectedTypeError(tf.boolType(), r.getType(), x);
  if (r.getValue().isEqual(vf.bool(false)))
    throw RuntimeExceptionFactory.assertionFailed(x, getStackTrace());
  return r;
}
```

Figure 8: An Example Visit Method, Before Refactorings are Applied.

```
public org.rascalmpl.interpreter.result.Result<org.eclipse.imp.pdb.facts.IValue>
  visitStatementAssert(org.rascalmpl.ast.Statement.Assert x) {
    org.rascalmpl.interpreter.result.Result<org.eclipse.imp.pdb.facts.IValue> r =
      x.getExpression().accept(this);
    if (!r.getType().equals(org.rascalmpl.interpreter.Evaluator.__getTf().boolType()))
      throw new
        org.rascalmpl.interpreter.staticErrors.UnexpectedTypeError(
          org.rascalmpl.interpreter.Evaluator.__getTf().boolType(), r.getType(), x);
    if(r.getValue().isEqual(this.__getVf().bool(false)))
      throw org.rascalmpl.interpreter.utils.RuntimeExceptionFactory.assertionFailed(x,
        this.getStackTrace());
    return r;
}
```

Figure 9: The Method from Figure 8, After Type and Field Name Qualification.

tion provided by the JDT). Because of this, the analysis uses an alternative means of identifying these fields and methods: source locations. First, the range of each of the 910 visit methods identified in the prior step is computed using information in the `methodDecls` fact (which includes locations). Next, field and method uses which occur at a location inside these ranges are identified using the location information in the `fields` and `methods` facts, with the location check encoded as a condition on the match. Using the declarations of the used fields and methods, available in `fieldDecls` and `methodDecls`, respectively, the fields and methods are then filtered to remove any that are declared to be `public`, available in the `modifiers` fact. For Rascal, this yields 38 fields and (by coincidence) 38 methods.

3.2 Preparatory Eclipse-Based Refactorings

After the analysis is complete, a number of refactorings are applied by the refactoring script to prepare the code for the main transformation. These refactorings are available using functions in the Rascal `JDTRefactoring` library.

First, the code cleanup engine is applied to each of the selected visitor classes, qualifying field and method accesses with `this` (for non-static member accesses) or the name of the declaring class (for static member accesses). This allows the transformer to syntactically distinguish local variables, instance variables, and statics. Second, the Eclipse Encapsulate Field refactoring is applied to each of the non-public fields identified in the analysis. This adds getters and, for updatable fields, setters for each field, ensuring they are still accessible once the visit code is moved into the new interpreter classes. Third, for similar reasons the Eclipse Change Method Signature refactoring is applied to each of the non-public methods identified in the analysis to make these methods public.

Finally, a custom refactoring is applied to each of the selected visitor classes (technically, to the files containing the classes) to fully qualify all type names. This transformation makes it possible to move the code without also moving the imports, which prevents possible import “collisions”. For instance, given two visit methods m_1 and m_2 in different files which will be copied into the same interpreter class, if m_1 uses imported class c from package p_1 and m_2 uses imported class c from p_2 (with p_1 and p_2 different), we would need to fully qualify at least some of the uses of c. To avoid making this decision on a case by case basis, V2I just qualifies all the names, using a later refactoring (after the code is moved) to remove as many of the qualifiers as possible.

Figure 8 shows an example of a visit method before any of these four transformations is applied. Figure 9 shows the same code after these four transformation steps. The fourth transformation step shows the most obvious changes, since all the type names are now fully qualified; these added type qualifications are shown in this color. Changes from the first step can be seen in the additions of `this` to the call to `__getVf` and the class name to the call to `__getTf` (which is declared as `static`), and are shown in this color. Changes from the second step, field encapsulation, can be seen in the use of these same `get` methods, neither of which existed before the transformation, but were added to ensure that non-public fields `vf` and `tf` remain available to the code after it is moved, and are shown in this color . Changes from the third step are not visible here, since they do not change the code inside the visit methods, only the actual declarations of any non-public methods used in the visit code.

3.3 Generating and Moving Code using Rascal

Once the preparatory refactorings are complete, V2I applies a number of transformation steps, written in Rascal, to convert the visitor methods into interpreter methods and create the new interpreter classes. This includes the steps Transform Methods and Generate Interpreter Classes from Figure 5. These steps make heavy use of both regular expression matching and string operations.

In Transform Methods, V2I first reads in the source of each visit method; each of these is then transformed, using a number of purely textual transformations, into an equivalent interpret method. For ease of discussion, below *V* refers to the class containing the visit method, *T* refers to the class of the AST node visited by the visit method, and *I* refers to the new class (a child of *T*) containing the created interpret method. First, uses of `this`, representing an instance of *V*, are replaced with `__eval`, a formal parameter of type *V* in the new method. Second, calls to `accept` are replaced by calls to `__evaluate`, the default name for all interpret methods. Third, uses of the formal parameter of type *T* are replaced by `this`, valid since the new methods are created in a subclass of *T*. Fourth, calls to `super` are replaced by calls to `__evaluate`, with uses of `__eval` in these calls cast to the type of the parent of *V* to ensure the correct method is called via overloading. The transformed method body is then inserted into a new public interpret method named `__evaluate`, returning the same type as the visit method and taking one formal parameter of type *V* named `__eval`. Type parameters declared on *V* are added directly to the new interpret method (not to class *I*). Because these are just textual transformations, all comments are maintained in the method source.

After Transform Methods, new interpret classes are generated to mirror the existing AST class hierarchy, built according to the Composite pattern. Each syntactic category is represented by an abstract class, e.g., `Statement` or `Expression`, with each production then represented as a nested concrete class, e.g. `Statement.IfThenElse` or `Expression.Addition`. Each of the new classes inherits from the existing AST class of the same name, allowing substitution, and each of the non-abstract classes includes a constructor with the same signature as the parent class, making a super call to execute any of the logic already defined therein[8]. The interpreter methods built above are then each inserted into the appropriate interpreter class. For instance, an interpret method based on a visit method which took a parameter of type `Expression.Addition` is placed in the new `Expression.Addition` class. A destination package is defined for all interpreter classes; this is used when generating the interpreter source file and to identify where the newly-created classes are saved in the project.

Other Approaches: There are other potential approaches to the transformations discussed above. One potential would be to use a combination of the Move Method and Push Down Method refactorings, first to move each visit method out of the visitor class into the AST class, and then to push it down into the correct interpreter class. Unfortunately this does not take care of all the transformations of the method bodies that are needed – we would still need to transform `accept` calls to `__evaluate` calls which, given the number of methods being transformed, needs to be scripted to be feasible. More seriously, this technique does not always work. First, there are several scenarios where this will create broken code, including cases where two methods with the same signature are generated and cases where (if type names are not fully qualified) type names will be captured in the move. Second, sometimes move cannot actually move a method, for instance, if the method uses non-local type parameters, or if the method uses the `super` keyword. Third, in some cases we want to only copy the code (e.g., with a default visitor extended by the real visitor classes), not move it.

3.4 Manual Changes, Cleanup, and Noise Reduction

As part of the Clean New Code step (Figure 5) several changes need to be made by hand, since only the code inside the visitors has been directly modified. First, a custom `ASTFactory` is created which returns instances of the new interpreter classes, with the `ASTFactoryFactory` modified to return an instance of this new factory. Next, calls to `accept` outside of the original `visit` methods which take instances of the transformed visitors are changed to `__evaluate` calls. The base `__evaluate` methods are also added to `AbstractAST`, the parent of all the AST nodes. Several other minor changes are also made, such as making several enumerators public. In theory, all these changes could have been made programmatically using Rascal. However, given the time it takes to make them by hand – around half an hour, using features of Eclipse and queries in Rascal – V2I instead focused on automating the more complex, time-consuming part of the transformation.

Several additional steps are then performed as part of the V2I script. First, another custom refactoring is invoked which, for each of the implementers and for each of the new interpreter classes, moves the qualifiers on the type

[8] The doubled AST hierarchy implements the "Generation Gap" design pattern `http://www.research.ibm.com/designpatterns/pubs/gg.html`.

```
public Result<IValue> interpret(Evaluator __eval) {
  Result<IValue> r = this.getExpression().interpret(__eval);
  if (!r.getType().equals(__eval.__getTf().boolType()))
    throw new UnexpectedTypeError(__eval.__getTf().boolType(), r.getType(), this);
  if (r.getValue().isEqual(__eval.__getVf().bool(false))) {
    throw RuntimeExceptionFactory.assertionFailed(this, __eval.getStackTrace());
  return r;
}
```

Figure 10: The Refactored Version of Figure 8.

names into imports. This step uses several rules to ensure that collisions do not occur. For instance, given two classes with the same unqualified name, only the longest qualified name is imported, with uses changed to just the unqualified name; also, in cases where the name would clash with a local name, it is left as is. This essentially backs out the changes made when the types were fully qualified (at least where possible). Second, source formatting is invoked over all modified or newly created code to ensure it has consistent formatting. These steps both ensure the code is much more readable, and thus easier to maintain going forward. An example of a fully transformed method, after all steps have been applied, is shown in Figure 10. This is the Interpreter version of the Visitor method shown in Figures 8 and 9.

Finally, to ensure the old visit methods are no longer invoked, the Remove Old Methods step (Figure 5) removes all the methods which have now been translated from visitor to interpreter variants (*except* those in `NullASTVisitor`, which are still used by some of the visitors that have not been converted). This step also helps ensure the correctness of the transformation – any code still trying to invoke these old methods will fail, including the suite of JUnit tests used for regression testing.

4. Discussion

In this section we raise four issues. First, how does Rascal compare with other meta-programming systems that seem equally applicable? Second, how do refactoring scripts relate to generic refactoring frameworks? Third, what is the trade-off between reusing an existing front-end (with the integration this entails) versus making one from scratch that is integrated from the start? Fourth, is the V2I refactoring a representative example of a refactoring script?

Meta-programming systems. Rascal is an integrated, language–independent analysis and transformation framework[9], providing concrete syntax trees, abstract syntax trees and relations as general analytical representations [9]. There are many meta-programming systems which can fill a role similar to that shown for Rascal above, including TXL [9], ASF+SDF [29], CodeSurfer [1], CrocoPat [6], DMS [5], Grok [16], Stratego [7], TOM [2], JastAdd [10] and Kiama [26][10].

The V2I refactoring makes intensive use of relations and computations over relations. Relations are commonly used in meta-programming systems designed for analysis in the domains of reverse engineering and re-engineering, like Grok and CrocoPat. Similarly, these systems are applied to extract abstract models of source code and perform queries on these abstract models. Tree-only meta-programming systems, such as Stratego and TXL, are able to encode relations, but do not support them natively. By comparison, Kiama and JastAdd support *reference attributes*, allowing these systems to construct graph-like structures "superimposed" over abstract syntax trees. Rascal, like Grok, has native (immutable) relations.

Some work on refactoring using the JastAdd Extensible Java Compiler has focused on specifying correct versions of a number of popular refactorings [23–25]. The important distinctions between JastAdd and Rascal are: (a) as mentioned above, relations are first class data in Rascal, while they are represented by reference attributes in JastAdd; and (b) Rascal control flow is specified operationally with control flow constructs and functions, rather than declaratively using attributes in JastAdd. We designed Rascal to make what a program does and when it does it transparent, as opposed to using high level declarations executed by an engine. Which is preferable depends on the audience, but note that the same level of conciseness is attained.

Overall, we believe that a number of the cited meta-programming systems, including JastAdd, Kiama, and DMS, could be used in a way similar to Rascal for developing such refactorings. We believe that support for relations and for manipulating programs, either as strings or as syntax trees, are both critical features. As shown in V2I, the ability to integrate with a host IDE (here, Eclipse), gaining access to facts already computed about source programs, provides significant opportunities for reuse. There is a trade-off between reuse of existing tools and libraries versus native implementations, however, which is discussed further below.

Refactoring frameworks Of course, several IDEs provide built-in refactoring tools. For Java, this includes Eclipse [4,

[9] Terms from WRT CfP `http://refactoring.info/WRT12/`.

[10] There exist many more such systems that are left unmentioned here.

13], Netbeans[11], and JetBrains' IntelliJ IDEA[12]. The Eclipse LTK, mentioned above, provides language-independent support for creating new refactorings, encapsulating reusable scheduling and user interaction (IDE integration) functionality, but not the needed semantics support [28]. To enable scripting of refactorings on the level supported by Rascal these frameworks should be completed with high level libraries for AST querying and database/graph querying.

Specialized refactoring frameworks or DSLs, such as Refacola [27], JunGL [30], and Wrangler [20], also exist. As opposed to Rascal, these systems focus explicitly on refactoring, with features aimed directly at this domain. For example, Refacola provides the basic notions of constraints and constraint variables to support all kinds of constraint-based refactorings. The JunGL language, aimed specifically at scripting refactorings, has many similarities to Rascal: both languages support higher-order functions, references, and pattern matching; make use of predicates and comprehensions; and support relational algebra natively in the language. Additionally, JunGL supports features not found in Rascal, such as the lazy addition of edges in graph representations of programs and the use of streams. Wrangler, a code inspection and refactoring framework for Erlang, provides support for creating refactorings and for generating commands that can (in an eager or lazy fashion) apply refactorings at multiple positions in an Erlang AST. Wrangler also provides a DSL for scripting refactorings, which allows larger refactorings to be built by composing a number of finer-grained refactorings.

Reuse versus Reimplementation of front-ends. One common feature of meta-programming systems is an integral parser generator, allowing parse trees to be integrated with other language features. Rascal includes a parser generator for general context-free grammars; similarly, JastAdd, Kiama, Stratego, DMS, and TXL all provide parser generators for efficiently processing input source code. However, there are two main reasons why one may not want to create a grammar from scratch. First, the engineering of a context-free grammar represents a significant intellectual effort [17]. Secondly, the hardest part of front-ends for languages like Java is in the name resolution and the type resolution, with all of the quirks and exceptions. Regardless of the formalism used to specify or implement these features, the heavy-lifting is in understanding all these fine details and adequately implementing them. For example, both the JastAdd-based Java compiler [10] and Java-front for Stratego re-implement this functionality. While both provide good integration with the underlying platform, they both represent man-years of work.

Our view was that most of this "heavy lifting" in V2I was already provided by the designers of the Eclipse JDT and their user-base, and we believe we have reached a similar level of integration. The actual integration of a meta-programming system with an existing front-end is not a completely trivial undertaking. The JDT library for Rascal is in that sense a small contribution in itself.

Since the creation of V2I, ASTs for Java have been added to the JDT library, but full access to the underlying parse trees is still missing. V2I instead employs strings and string templates. A high-fidelity parse tree representation [31, 32] can provide the ability to easily rewrite source while retaining much of the original source code comments and indentation. For scripting refactorings, we can assume that such high fidelity is an important, but nevertheless secondary concern.

Threats to validity. We must note that the V2I refactoring may not be representative of all refactoring scripts. In particular, since it manipulates entire method bodies there is no necessity for fine-grained analysis of where source code comments should go after the transformation. Where we do manipulate the internals of method bodies, we have reused Eclipse's refactoring tools that do maintain source code comments. We do know that type-based refactorings, such as "Infer Generic Type Arguments", can concisely be prototyped in Rascal [18] as well.

A second concern is that the V2I refactoring, as presented, is quite Java-specific: Rascal is implemented in Java, uses facts extracted from a Java-based development environment, and refactors Java code. However, we believe that Java is not an essential element of the techniques presented here. Rascal can be used to build refactorings for any other language given the proper tools, which could be written directly in Rascal (e.g., a grammar for parsing programs, analyses to determine needed analysis facts such as types and name bindings) or based on interaction with a programmable development environment, as was done here with the JDT. As mentioned above, it should also be possible to develop V2I-like refactorings in other meta-programming languages or in languages focused on refactoring, but adopting these techniques to general purpose languages such as Java would be quite cumbersome.

A third concern is that we have not compared the time taken to build the refactoring script with the time it would take to perform the refactoring by hand using Eclipse. Given the number of methods to transform, with the opportunities for error that this could introduce, plus the limitations of the existing Eclipse Java refactorings pointed out in Section 3.3, we believe that the script saved time. Admittedly, though, we have not attempted to quantify this.

5. Conclusions

This paper detailed a single refactoring, "Visitor to Interpreter", implemented as a script-supported refactoring in Rascal. We emphasized the points of integration with Eclipse, including the use of JDT facts and existing JDT refactorings, while also detailing analysis and transformation steps that occur completely in Rascal. The resulting script is less than 1KLOC. We discussed meta-programming and refactoring

[11] `http://www.netbeans.org`

[12] `http://www.jetbrains.com`

frameworks, giving our idea of the key success factors for *scripting refactorings*. These are: (a) having strings, trees, and relations as first-class citizens with high-level operations and (b) tight integration with a reusable front-end.

Acknowledgments: We would like to thank the anonymous reviewers, whose helpful feedback has assisted us in improving the quality of this paper.

References

[1] P. Anderson and M. Zarins. The CodeSurfer Software Understanding Platform. In *Proceedings of IWPC'05*, pages 147–148. IEEE, 2005. ISBN 0-7695-2254-8.

[2] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking Rewriting on Java. In *Proceedings of RTA'07*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007.

[3] B. Basten and T. van der Storm. AMBIDEXTER: Practical Ambiguity Detection. In *Proceedings of SCAM'10*, pages 101–102. IEEE, 2010.

[4] D. Bäumer, E. Gamma, and A. Kiezun. Integrating Refactoring Support into a Java Development Tool. In *OOPSLA'01 Companion*. ACM Press, October 2001.

[5] I. Baxter, P. Pidgeon, and M. Mehlich. DMS®: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of ICSE'04*, pages 625–634. IEEE, 2004.

[6] D. Beyer. Relational programming with CrocoPat. In *Proceedings of ICSE'06*, pages 807–810. ACM Press, 2006.

[7] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008.

[8] P. Charles, R. M. Fuhrer, S. M. Sutton Jr., E. Duesterwald, and J. J. Vinju. Accelerating the Creation of Customized, Language-Specific IDEs in Eclipse. In *Proceedings of OOPSLA'09*, pages 191–206. ACM Press, 2009.

[9] J. R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 61(3):190–210, 2006.

[10] T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. In *Proceedings of OOPSLA'07*, pages 1–18. ACM Press, 2007.

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

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

[13] R. M. Fuhrer, M. Keller, and A. Kiezun. Advanced Refactoring in the Eclipse JDT: Past, Present, and Future. In *Proceedings of WRT*, pages 30–31, 2007.

[14] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[15] M. Hills, P. Klint, T. van der Storm, and J. J. Vinju. A Case of Visitor versus Interpreter Pattern. In *Proceedings of TOOLS 2011*, volume 6705 of *LNCS*, pages 228–243. Springer, 2011.

[16] R. C. Holt. Grokking Software Architecture. In *Proceedings of WCRE'08*, pages 5–14. IEEE, 2008.

[17] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for Grammarware. *ACM TOSEM*, 14(3):331–380, 2005.

[18] P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of SCAM'09*, pages 168–177. IEEE, 2009.

[19] P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In *Post-Proceedings of GTTSE'09*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.

[20] H. Li and S. J. Thompson. A Domain-Specific Language for Scripting Refactorings in Erlang. In *Proceedings of FASE'12*, volume 7212 of *LNCS*, pages 501–515. Springer, 2012.

[21] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[22] W. F. Opdyke and R. E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[23] M. Schäfer and O. de Moor. Specifying and Implementing Refactorings. In *Proceedings of OOPSLA'10*, pages 286–301. ACM Press, 2010.

[24] M. Schäfer, T. Ekman, and O. de Moor. Sound and Extensible Renaming for Java. In *Proceedings of OOPSLA'08*. ACM Press, 2008.

[25] M. Schäfer, M. Verbaere, T. Ekman, and O. de Moor. Stepping Stones over the Refactoring Rubicon. In *Proceedings of ECOOP'09*, volume 5653 of *LNCS*. Springer, 2009.

[26] A. M. Sloane. Lightweight Language Processing in Kiama. In *Post-Proceedings of GTTSE'09*, volume 6491 of *LNCS*, pages 408–425. Springer, 2011.

[27] F. Steimann, C. Kollee, and J. von Pilgrim. A Refactoring Constraint Language and Its Application to Eiffel. In *Proceedings of ECOOP'11*, volume 6813 of *LNCS*, pages 255–280. Springer, 2011.

[28] J. van den Bos. Refactoring (in) Eclipse. Master's thesis, Universiteit van Amsterdam, Aug. 2008.

[29] M. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *Proceedings of CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.

[30] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: A Scripting Language for Refactoring. In *Proceedings of ICSE'06*, pages 172–181. ACM, 2006.

[31] J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. PhD thesis, nov 2005.

[32] D. Waddington and B. Yao. High-fidelity C/C++ code transformation. *Science of Computer Programming*, 68(2):64 – 78, 2007.