Table of Contents

1st Workshop on Refactoring Tools (WRT'07)
Organization

Chair and Organizer: Danny Dig (University of Illinois at Urbana-Champaign)

Program Committee:
Jan Becicka (NetBeans Refactoring Engine, Sun Microsystems)
Danny Dig (University of Illinois at Urbana-Champaign)
William G. Griswold (University of California, San Diego)
Ralph Johnson (University of Illinois at Urbana-Champaign)
Markus Keller (Eclipse Refactoring Engine, IBM)
Oege de Moor (Oxford University Computing Laboratory)
Frank Tip (IBM T.J. Watson Research Center)

KABA: Automated Refactoring for Improved Cohesion

G. Snelting, M. Streckenbach
Universität Passau

1 Overview

Cohesion is one of the most important software engineering principles. Cohesion demands in particular that data and function operating on these data are defined together in one class definition. But not all existing Java class hierarchies define a cohesive software architecture.

KABA analyses a class hierarchy together with a set of client programs. It refactors the hierarchy such that data members (fields) and methods operating on the data are always grouped together, and that any class in the refactored hierarchy contains only fields and methods which are indeed needed by some client code using the class. Every class-typed variable is given a new type (namely a class from the refactored hierarchy), while the program statements remain essentially unchanged. In contrast to some other refactoring tools, KABA guarantees that program behaviour is unchanged after refactoring.

KABA is based on fine-grained (static or dynamic) program analysis, namely points-to analysis, type constraints, and mathematical concept lattices. Typically, an original class is split if different clients use different subsets of a class's functionality. In order to avoid too fine-grained refactorings, the initial refactoring is usually simplified by automatic or manual semantics-preserving transformations.

Thus KABA provides fine-grained insight into the true usage patterns of classes in a program, can serve to evaluate cohesion of existing systems, and will automatically generate semantics-preserving refactorings which improve cohesion. KABA is implemented in form of a refactoring browser.

2 Examples and Experiences

KABA reacts to usage patterns of a class by client code. In figure 1, client objects `A1`, `A2`, `B1`, `B2` access different subsets of class `A` and `B`:

- object `A1` accesses only `x` from class `A`, while `A2` also accesses `f()` and thus `y`; therefore class `A` is split into two subclasses

```
class A {
  int x, y, z;
  void f() {
    y = x;
  }
}

class B extends A {
  void f() {
    y++;
  }
  void g() {
    x++;
    f();
  }
  void h() {
    f();
    x--;
  }
}
```

```
class Client {
   public static void
   main(String[] args) {
     A a1 = new A();  // A1
     A a2 = new A();  // A2
     B b1 = new B();  // B1
     B b2 = new B();  // B2

     a1.x = 17;
     a2.x = 42;
     if (...) { a2 = b2; }
     a2.f();
     b1.g();
     b2.h();
   }
}
```

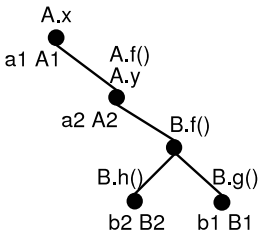

Figure 1: A small program and its KABA refactoring

- `B1` calls `g()` while `B2` calls `h()`, hence two new subclasses for `B` are introduced.

The new subclasses give fine-grained insight into what objects really do, and maximise cohesion as in the new hierarchy members are grouped together iff they are used together. A nice by-product is that objects are minimized and dead members are eliminated. But since the refactored hierarchy may be quite fine-grained, the engineer may remerge classes (eg the two top classes in figure 1). In any case, KABA guarantees preservation of semantics. For the complex technical details, see [1].

The resulting refactored hierarchy, as well as cross references between new and old classes, as displayed by the KABA refactoring browser, are

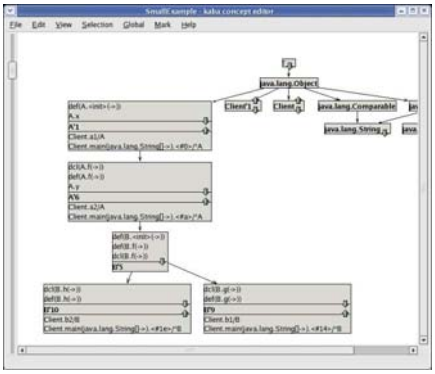

Figure 2: KABA screenshot for figure 1

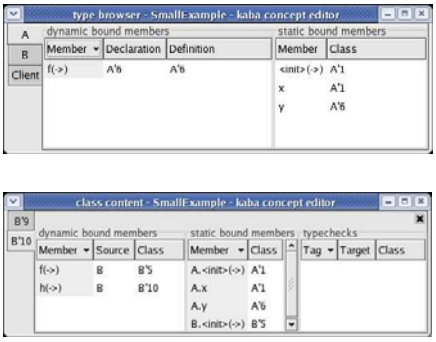

Figure 3: Browser for original types (top); Browser for class content (bottom).

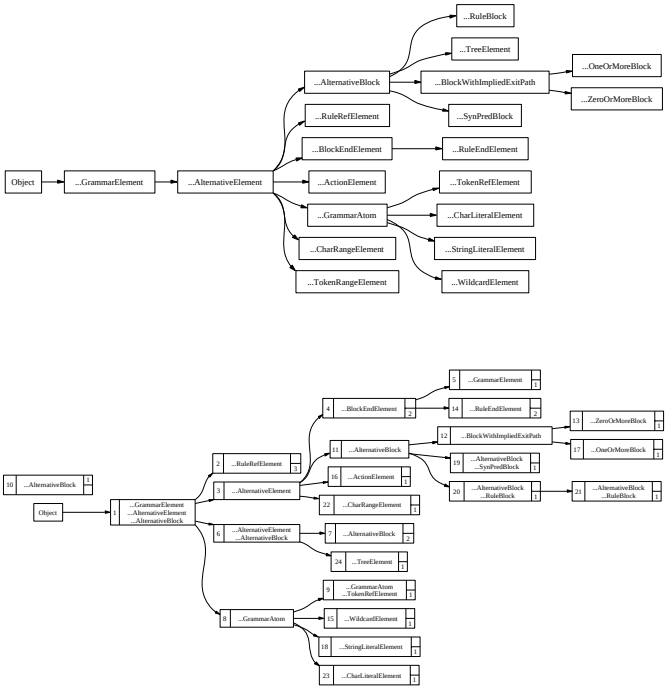

Figure 4: KABA refactoring (lower) for a part of `antlr` (upper)

shown in figures 2 and 3. The user may remerge splitted classes, which can reduce cohesion again but may improve other software quality factors. More details about the KABA refactoring browser and semantics-preserving class merging and elimination of multiple inheritance can be found in [3].

A more realistic case study was the `antlr` parser generator. KABA discovered problems with cohesion in the original design and generated an improved class hierarchy (figure 4). For example, KABA discovered that the original classes `GrammarElement` and `AlternativeElement` can be merged, while preserving semantics and improving cohesion. On the other hand, some original classes were split to 5, 6 or 9 new classes. Thus the hierarchy structure was changed quite a bit. But note that the number of classes remained about the same.

As a third example, we mention `javac`, where KABA reproduced the original class hierarchy almost exactly. This proved that the original `javac` design was good. More details about refactoring `antlr` and `javac` can be found in [2].

KABA is a research prototype. It can handle full Java (without reflection). KABA has been applied to several small and medium-scale example systems, and generated useful refactorings. For large systems, the refactorings may be quite fine-grained, thus we are exploring even more agressive simplifications of the original refactoring which still improve cohesion and guarantee behaviour preservation.

References. Technical details and more experience with KABA can be found in

1. G. Snelting, F. Tip: Reengineering Class Hierarchies Using Concept Analysis. ACM Transactions on Programming Languages and Systems (TOPLAS), May 2000, pp. 540-582.

2. M. Streckenbach, G. Snelting: Refactoring Class Hierarchies with KABA. Proc. OOPSLA 2004, Vancouver, British Columbia, Canada, October 2004, pp. 315-330.

3. Mirko Streckenbach: KABA - A System for Refactoring Java Programs. PhD thesis, Universität Passau, April 2005.

Automation of Refactoring and Refactoring Suggestions for TTCN-3 Test Suites
The TRex TTCN-3 Refactoring and Metrics Tool

Helmut Neukirchen and Benjamin Zeiss
Software Engineering for Distributed Systems Group,
Institute for Informatics, University of Göttingen,
Lotzestr. 16–18, 37083 Göttingen, Germany
{neukirchen|zeiss}@cs.uni-goettingen.de

Abstract *Refactoring is not only useful for source code of implementations, but as well for test specifications. The open source TRex tool automates the application of refactorings and the detection of refactoring opportunities for test suites that are specified using the standardised* Testing and Test Control Notation *(TTCN-3). Depending on the refactoring, the behaviour preserving transformations may include syntax tree transformations and direct modification of the source code; for suggesting refactorings, metrics are calculated and code smell patterns are detected.*

Introduction. The *Testing and Test Control Notation* (TTCN-3) [1] is a mature standard from the telecommunication and data communication domain that is widely used in industry and standardisation to specify and execute test suites. Just like any other software artifact, tests suffer from quality problems [2]. To remove such quality problems from TTCN-3 test suites, we use refactoring [3]. For suggesting refactorings, we use a combination of metrics and code smell detection.

In the following, we first present our approach for the quality assessment and improvement of TTCN-3 test suites. Subsequently, the TTCN-3 Refactoring and Metrics Tool TRex and its implementation are described. Finally, future work is discussed in the outlook.

Refactoring, Metrics, and Code Smell Detection for TTCN-3 Test Suites. Refactoring of test suites has so far only been studied in the context of JUnit [2]. Thus, we have developed a refactoring catalogue for TTCN-3 [4, 5] which includes 23 refactorings using language specific concepts of TTCN-3. Furthermore, we found 28 refactorings from Fowler's Java refactoring catalogue [3] to be applicable to TTCN-3.

For being able to automatically identify locations in source code where a refactoring is worthwhile, we investigated corresponding TTCN-3 metrics and TTCN-3 code smells. For example, a *Number of References* metric is used to identify definitions that are never referenced and can thus be removed or that are referenced only once and can thus be inlined using a corresponding refactoring. Even though we experienced that metrics are able to detect various issues, they are not sophisticated enough to detect more advanced problems. Therefore, we investigated pattern-analysis of source code and as a result, we have developed a catalogue of TTCN-3 code smells [6]. So far 38 TTCN-3 code smells have been identified.

TRex Implementation. To automate refactoring for TTCN-3 test specifications, we have implemented the open source TTCN-3 Refactoring and Metrics tool TRex [4]. The initial version has been developed in collaboration with Motorola Labs, UK [7]. TRex implements state-of-the-art editing capabilities, assessment and improvement techniques for TTCN-3 test suites based on the calculation of metrics, automated smell detection, and refactoring. TRex is based on the Eclipse Platform [8] and thus makes use of the infrastructure offered, e.g. the *Language Toolkit* (LTK) or the *Eclipse Test & Performance Tools Platform* (TPTP) static analysis framework. The analysis infrastructure including lexer, parser, symbol table, pretty printer, etc. for TTCN-3 have been implemented using *ANother Tool for Language Recognition* (ANTLR) [9].

The automated refactorings we currently provide concentrate mostly on the improvement of test data descriptions (TTCN-3 templates). The refactoring implementations can be applied in two different ways: either the test developer invokes the refactoring from the code location

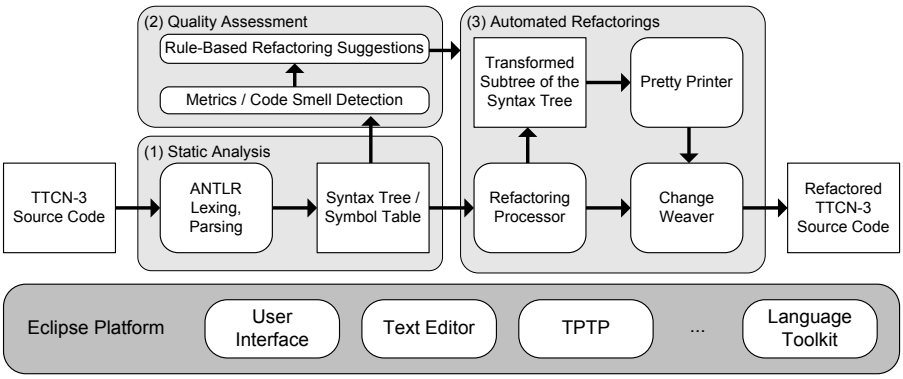

Figure 1: The TRex Toolchain

which should be subject to the refactoring or the refactoring is invoked directly by a quick-fix which is provided by the analysis results of the automated quality assessment.

For the assessment, a considerable number of size metrics (such as counting the number of references to a definition) and structural metrics (using control-flow graphs and call graphs) are calculated. Furthermore, a total number of 11 TTCN-3 code smell detection rules have been implemented that partially allow the use of quick-fixes to invoke automated refactorings.

The overall toolchain of TRex is depicted in Fig. 1. Based on the syntax tree and symbol table, the automated refactorings can either be applied directly or invoked through the refactoring suggestions obtained by means of metrics and code smell detection. The refactorings are applied directly to the source code using a programmatic text editor and may as well involve syntax tree transformations. The corresponding text representation from transformed subtrees is reobtained by a pretty printer component and weaved back into the surrounding TTCN-3 source code.

The implementations of metric calculation, code smell detection, and refactoring are tested using *Plug-in Development Environment* (PDE) JUnit tests, e.g. by comparing source code snippets before and after the refactoring.

Outlook. A remaining issue open for research is the validation of test refactorings: Unlike Java refactorings, for example, there are no unit tests available for tests and simply running a test suite against an implementation is not enough if the test behaviour consists of more than one path. We are thus investigating bisimulation to validate that the observable behaviour of a refactored test suite has not changed. In addition, we are extending TRex by implementing further TTCN-3 refactorings and more sophisticated code smell detection techniques.

References

[1] ETSI: ETSI Standard (ES) 201 873 V3.2.1: The Testing and Test Control Notation version 3; Parts 1-8. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2007)

[2] van Deursen, A., Moonen, L., van den Bergh, A., Kok, G.: Refactoring Test Code. In: XP2001. (2001)

[3] Fowler, M.: Refactoring – Improving the Design of Existing Code. Addison-Wesley, Boston (1999)

[4] TRex Team: TRex Website. `http://www.trex.informatik.uni-goettingen.de` (2007)

[5] Zeiss, B.: A Refactoring Tool for TTCN-3. Master's thesis, Institute for Informatics, University of Göttingen, Germany, ZFI-BM-2006-05 (2006)

[6] Bisanz, M.: Pattern-based Smell Detection in TTCN-3 Test Suites. Master's thesis, Institute for Informatics, University of Göttingen, Germany, ZFI-BM-2006-44 (2006)

[7] Baker, P., Evans, D., Grabowski, J., Neukirchen, H., Zeiss, B.: TRex – The Refactoring and Metrics Tool for TTCN-3 Test Specifications. In: TAIC PART 2006, IEEE Computer Society (2006) 90–94

[8] Eclipse Foundation: Eclipse. `http://www.eclipse.org` (2007)

[9] Parr, T.: ANTLR parser generator v2. `http://www.antlr2.org` (2007)

A visual interface for type-related refactorings

Philip Mayer
Institut für Informatik
Ludwig-Maximilians-Universität
D-80538 München
plmayer@acm.org

Andreas Meißner
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
meissner@acm.org

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org

ABSTRACT

In this paper, we present our approach to a visual refactoring tool, the Type Access Analyzer (TAA), which uses program analysis to detect code smells and for suggesting and performing refactorings related to typing. In particular, the TAA is intended to help the developers with consistently programming to interfaces.

1. INTRODUCTION

When looking at currently available type-related refactoring tools, a noticeable gap shows between simple refactorings like *Extract Interface* and more complex, "heavyweight" ones like *Use Supertype Where Possible* and *Infer Type* [2]: While the former do not provide any analysis-based help to the user, the latter perform complex program analyses, but due to their autonomous workings – without interacting further with the user except for preview functionality – it is not always clear when to apply them, what result to expect, and just how far the changes of the refactorings will reach. For example,

- *Extract Interface* keeps programmers in the dark about which methods to choose,
- *Use Supertype Where Possible* replaces all declaration elements found without a proper way of restricting it,
- *Infer Type* creates new types guaranteeing a type-correct program, but often lacking a conceptual justification.

As a remedy, we propose a new approach to refactoring. The contributions of this approach consist of:

- moving precondition checking and parameterization from refactorings to a dedicated program analysis component,
- presenting the analysis results visually in such a way that they suggest refactorings, and
- breaking down existing refactorings into simpler tools which perform predictable changes immediately visible and controllable by the visual refactoring view.

This approach is prototypically realized in our Type Access Analyser (TAA) tool for type-related refactorings.

2. THE TYPE ACCESS ANALYZER

A loosely coupled and extensible software design can be reached by consistently programming to interfaces [1], specifically to what we have called *context-specific interfaces* [3]. An interface is considered to be context-specific if it contains exactly – or, in a more relaxed interpretation, not much more than – the set of members of a type required in a certain area of code (which is comprised of variables and methods declared with the interface as their types and their transitive assignment closure).

Refactoring to the use of such interfaces requires an analysis of what is really needed in contexts by analyzing the code to find used or unused members. With this information, the code can be refactored in an informed way by:

- creating, adapting, or removing interfaces, and
- retrofitting existing variable types to the newly introduced, or adapted, interfaces.

The TAA follows the approach discussed in the introduction by analyzing the code using the type inference algorithm we have introduced in [2], presenting the results in a visual form, and providing access to and feedback from simplified versions of refactorings such as *Extract Interface* or *Infer Type*.

To give the programmer a comprehensive and concise view of the program that is tailored to the specific problems of interface-based programming, we have developed the supertype lattice view described in [4]. In this view, the supertype hierarchy of the type under consideration is enhanced by displaying a bounded lattice of the set of members of the type, each node being enriched with various kinds of information.

Figure 1 shows a screenshot of the TAA in action on a four-method class (due to space limitations, only a part is shown).

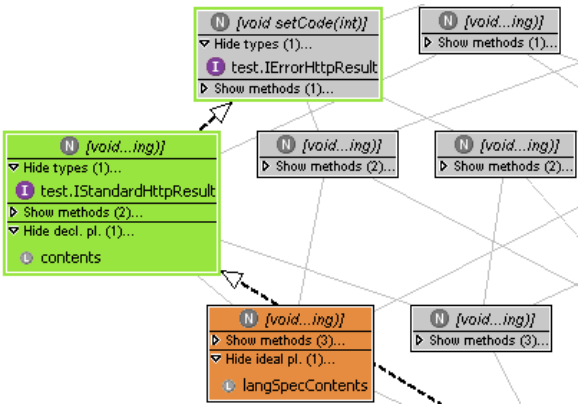

Figure 1: TAA Visual View

Four types of information are immediately visible from the graph:

- **Possible types** – each node is a possible supertype of the class (which is situated at the bottom; not shown).
- **Available types** are shown in the *types* section on a node. Subtyping relations between types are indicated by UML-style subtyping arrows.

- **Variables and methods**. Each variable or method typed with one of the type(s) under consideration is included in the graph. Assignments between these elements are shown with red arrows.
  - **Declared placement**. A variable or method is shown in the *declared placement* section of the node containing the declared type of the variable or method.
  - **Ideal placement**. If different from the declared placement, a variable or method is shown in the *ideal placement* section of the node which corresponds to the set of members (transitively) invoked on this variable or method.

The quality of the *variable* and *method declarations* (i.e. the matching between types and their usage contexts) is shown by the colours of the background of the node. A green colour represents the use of context specific types, while a red colour signals a mismatch between types and usage contexts.

Selecting variables or methods in the graph further enriches the display:

- A line is drawn connecting the ideal and declared placements of an element (if different).
- Additional lines are drawn connecting all elements which the current element is being assigned to (transitively).

This data may be used to detect smells in the code and take appropriate action. The following section will detail this.

3. VISUAL REFACTORINGS

By analyzing a type in the TAA view, the developer has complete overview of the usages of this type. The annotations on the supertype lattice suggest a number of ways of improving the typing situation; specifically, the arrangements of types and variables/methods visualize code smells which can be removed by applying refactorings.

The following table associates design problems in the code, the way these problems show up in the visual view (as smells), and the actions to be taken by the developer to deal with those problems. Later on, we will present refactorings for executing these actions.

Problem	Smell	Action
No interfaces available for a context	Nodes in the graph with ideal placement of variables/methods, but without interfaces	Extract interface and redeclare variables/ methods with new interface
Poorly designed interface	Ideal placement of variables/methods swarming around existing interfaces	Move existing interface up or down in hierarchy
Two interfaces for the same purpose	Two interfaces share the same/ neighbouring nodes, each with ideally placed variables/ methods	Merge interfaces
Superfluous interface	Interface present in a node without declared placements; no ideal placements in vicinity	Remove interface from hierarchy
Interface is not (yet) used	Interface is present in a node with ideally but no declared placements	Redeclare variables / methods with existing interface

Table 1: Code Smells

As can be seen from the table above, the TAA suggests a number of actions to be taken as a result of identified smells. These actions are implemented as refactorings. In line with our approach of putting the developer in charge, these refactorings may be selected as (semantically) appropriate by the user.

Contrary to existing refactorings, the ones invoked from the TAA in general do not require further dialog-based parameterization – all information required for a refactoring is already available in the graph and the way the user invokes the refactoring in a visual way. These visual start procedures are shown in Table 2:

Refactoring	Visual start procedure
Extract interface	Alt + Drag an interface to another, higher node in the graph
Move interface in hierarchy	Drag an interface to another node in the graph (up or down)
Remove interface from hierarchy	Select an interface, select delete
Merge interfaces	Select two interfaces, select merge
Redeclare declaration elements (transitively, i.e. following assignments)	Select a variable or method, select redeclare

Table 2: Refactorings

While the *Extract Interface* refactoring is already available as-is in many tools, the others have been either adapted or specifically written for the TAA.

4. SUMMARY

In this paper, we have described our approach to visual refactoring. The TAA tool aids the developer by providing valuable information about the typing situation in the code – in itself suitable for program understanding – and thereby suggests refactorings whose effect is more predictable and which can be executed directly from the visual view. The results of the refactorings are likewise directly shown in the graph.

In the future, we will investigate ways of further improving the user interface and add more refactorings to the TAA.

5. REFERENCES

[1] E Gamma, R Helm, R Johnson, J Vlissides *Design Patterns - Elements of Reusable Software* (Addison-Wesley, 1995).

[2] F Steimann "The Infer Type refactoring and its use for interface-based programming" *JOT* 6:2 (2007) 67–89.

[3] F Steimann, P Mayer "Patterns of interface-based programming" *JOT* 4:5 (2005) 75–94.

[4] F Steimann, P Mayer "Type Access Analysis: Towards informed interface design" in: *TOOLS* (2007) to appear

This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

ITCORE: A Type Inference Package for Refactoring Tools

Hannes Kegel
ej-technologies GmbH
Claude-Lorrain-Straße 7
D-81543 München
hannes.kegel@ej-technologies.com

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org

1. Introduction

ECLIPSE's current distribution comes with various type-related refactoring tools, some of which rely on a built-in framework for solving a set of type constraints derived from the declarations and expressions of a type-correct program. We have adapted this framework to embody the type inference underlying our INFER TYPE refactoring tool for JAVA [6], which extracts the maximally general type for a declaration element (variable or method) based on the use of that element in a program. The new implementation surpasses the original one by greater memory efficiency; it also fully embodies the language extensions of JAVA 5, most notably generics. It has been extensively tested by automatically applying it to all declaration elements of several large code bases.

Experimenting with INFER TYPE, it soon became clear that its type inference procedure can be used for several other refactorings and also for program analysis tools such as the TAA [5]. We therefore decided to develop the type inference part as a package in its own right. However, with several uses of this package still under development, its API has not yet settled; therefore, we will only provide an overview of its inner workings here, and of the refactorings that it enables.

2. Type inference with ITCORE

Type inference computes type annotations for program elements. It is most useful in languages in which type annotations for these elements are not mandatory ("untyped" languages). In typed languages such as JAVA, type inference can compute the difference between a declared and a derived (inferred) type.

In the context of refactoring statically type-checked, type-correct object-oriented programs, type inference can be used to change the declared types of program elements in one of two directions: to the more specific, or to the more general. The former is of interest for instance when `Object` or a raw type is used where all objects are known to be of a more concrete class or parameterized type; such type inference is, e.g., the basis for the refactoring tool INFER GENERIC TYPE ARGUMENTS described in [3]. The latter is useful for decoupling designs, for instance when turning a monolithic application into a framework; it is the basis of our INFER TYPE refactoring described in [6]. It is this latter variant of type inference that ITCORE performs and that we are looking at here.

The type inference algorithm of ITCORE computes for a given declaration element its maximally general type (interface or abstract class), i.e., that type that contains only the members accessed through the declaration element and the ones it gets assigned to. It does so by performing a static program analysis, (class hierarchy analysis, specifically), which is perfectly adequate in our setting, since the static type checking of JAVA would reject any additional precision offered by a dynamic analysis. The inferred type is "minimal" in the sense that if a declaration element is typed with its inferred type, no member can be removed from the type without introducing a typing error.

The constraints generated by ITCORE are similar to that of INFER GENERIC TYPE ARGUMENTS in that parameterized types are decomposed and separate constraint variables for all type arguments are created. For assignments, individual constraints are generated for the main types and all type arguments. If the receiver of a method invocation is a parameterized type, method and type arguments are also connected through constraints.

ITCORE differs from the engines of its predecessors in various respects:

1. The constraints generated by ITCORE cover the complete type system of JAVA 5. This lets ITCORE change type arguments including those of nested parametric types, as e.g. `Test` in `class MyComparator implements Comparator<Test>`. (Only type bounds are currently not considered for change.)
2. ITCORE attaches to each constraint variable (representing a declaration element) the required protocol (the *access set* [7]) of that element, which is the basis for the creation of new types during the solution of the constraint set.
3. The constraint solution procedure works constructively, by visiting the nodes of the constraint graph reachable from the start element(s) in depth-first order and collecting the (transitively determined) access sets in the nodes. After the traversal, the new type for the start element has been determined and is created. A second traversal then satisfies all type constraints, by introducing the new type (and possibly others [6]) where necessary.

3. Refactoring tools building on ITCORE

3.1 Existing refactoring tools

A number of existing type-inference based refactoring tools can profit from being migrated to ITCORE:

GENERALIZE DECLARED TYPE The purpose of GENERALIZE DECLARED TYPE is to replace the declared type of a variable or method with a supertype, if that is possible. However, in its current form GENERALIZE DECLARED TYPE can only change the type of the selected declaration element; if this element gets assigned to others of the same type or some other subtype of the generalized type, the refactoring is impossible. This restricts its practical applicability considerably.

ITCORE can serve as the basis for the implementation of a new GENERALIZE DECLARED TYPE, since any existing (structural) subtype of a declaration element's inferred type that is also a (nominal) supertype of the declared type can be used in the declaration of this element and others of the same type in the assignment chain (subject to some very rare exceptions described in [6]). In addition, it would extend GENERALIZE DECLARED TYPE to cover generics, significantly increasing its applicability.

USE SUPERTYPE WHERE POSSIBLE In a similar vein, ITCORE can serve as the basis for USE SUPERTYPE WHERE POSSIBLE: by computing the inferred types for all declaration elements of a certain type, it can be checked for a selected supertype where this supertype could be used instead. Again, using ITCORE would make full coverage of generics available to this refactoring.

EXTRACT INTERFACE Current implementations of EXTRACT INTERFACE let the developer choose the members of the interface. By naming the declaration elements for which this interface is to be used, ITCORE can be used to preselect the members that are needed, allowing the developer to add (but not delete!) members if deemed appropriate. It is then guaranteed that the new interface can be used where it is intended to be.

INFER TYPE INFER TYPE is a relatively new refactoring described in some detail elsewhere [6]. It is currently the only one that has already been migrated to ITCORE; see `http://www.fernuni-hagen.de/ps/prjs/InferType3/`.

3.2 Refactorings awaiting tool support

Many useful refactorings still lack automation. Some of them can be implemented using ITCORE, as the following list suggests.

REPLACE INHERITANCE WITH FORWARDING Suppose a class inherits many members but needs only few of them, so that its interface is needlessly bloated; in this case, it may be better to define a new class with just the required interface, and have it hold an instance of its former superclass to which it forwards all service requests. The protocol of the new class can be computed using ITCORE, simply by inferring the types of all declaration elements typed with the class.

In case instances of the new class need to substitute for instances of its former superclass (which is often the case in frameworks), subtyping cannot be avoided. However, in these cases either an existing or an inferred interface of the new class and its former superclass can be used in the places where the substitution occurs (the plug points of the framework). Both can be derived using ITCORE. In fact, as has been pointed out in [7], even the question of whether such a substitution is possible (and corresponding subtyping, i.e., interface implementation, is therefore necessary) can be computed by means of ITCORE; all that needs to be done is search for assignment chains from (declaration elements of) the new class to (declaration elements of) its former superclass [7].

REPLACE INHERITANCE WITH DELEGATION A simple, but easily overlooked, twist to the previous refactoring is that a superclass may contain calls to its own, non-final methods. In this case, and if any of the called methods are overridden in the subclass to be refactored, replacing inheritance with forwarding changes program semantics, because the formerly overriding methods in the subclass no longer override (the `extends` is removed; the new class is no longer a subclass of its former superclass) so that dynamic binding of these methods does not apply. In these cases, forwarding must be replaced by delegation [9].

In class-based languages such as JAVA, delegation must be mimicked by subclassing. IDEA's REPLACE INHERITANCE WITH DELEGATION refactoring tool does exactly this [4]; combining it with REPLACE INHERITANCE WITH FORWARDING as described above would further automate this refactoring.

INJECT DEPENDENCY DEPENDENCY INJECTION is an increasingly popular design pattern that lets components to be assembled into applications remain independent of each other [1]. Components are independent (decoupled) only if they do not reference each other directly, neither through declarations nor through instance creation (constructor calls). One prerequisite to successful dependency injection is therefore that all such references are removed. As has been outlined above, this can easily be done by ITCORE, namely by inferring the common type of all references to the class depended upon. The rest, removing constructor calls (as for instance described in [8]) and instrumenting the assembly of components, is independent of ITCORE; it can be done with the standard means of AST manipulation. We are currently working on an implementation for SPRING and EJB3.

CREATE MOCK OBJECT Unit testing requires that each unit is tested independently of others. In practice, however, units depend on others that have not been sufficiently tested to be assumed correct, or whose behaviour cannot be controlled by a test case. In these cases, units depended upon are replaced by mock objects (really: mock classes). These objects exhibit the same provided interface as those they replace; their implementation, though, is different ("mocked").

Mocking complete classes with many methods may be more than is actually needed. To avoid this, ITCORE can compute from a test case (or test suite) the interface actually required from a mock class, which may be less than what is offered by the class it mocks. This computed interface can drive the creation of the mock class, or the necessary overriding of methods if the mock is derived from the original class. Also, it can serve as a common abstraction of the original and the mock class in the context of the test (suite). Cf. also [2].

4. Conclusion

There seems to be a whole class of refactorings that rely on the determination of the minimal protocol, or the maximally general type, of a program element. The reasons for this may vary: decreasing coupling and increasing variability as for GENERALIZE DECLARED TYPE, USE SUPERTYPE WHERE POSSIBLE, INJECT DEPENDENCY, or INFER TYPE; cleaning up the interface as for REPLACE INHERITANCE WITH FORWARDING/DELEGATION; or simply reducing the amount of necessary work, as in CREATE MOCK OBJECT. All have in common that they can be built on the type inference procedure offered by ITCORE.

References

[1] `http://www.martinfowler.com/articles/injection.html`

[2] S Freeman, T Mackinnon, N Pryce, J Walnes "Mock roles, not objects" in: *OOPSLA Companion* (2004) 236–246.

[3] RM Fuhrer, F Tip, A Kiezun, J Dolby, M Keller "Efficiently refactoring Java applications to use generic libraries" in: *ECOOP* (2005) 71–96.

[4] `http://www.jetbrains.com/idea/docs/help/refactoring/replaceinheritwithdelegat.html`

[5] P Mayer, A Meißner, F Steimann "A visual interface for type-related refactorings" submitted to: *1st Workshop on Refactoring Tools*.

[6] F Steimann "The Infer Type refactoring and its use for interface-based programming" *JOT* 6:2 (2007) 67–89.

[7] F Steimann, P Mayer "Type Access Analysis: Towards informed interface design" in: *TOOLS* (2007) to appear.

[8] `http://www.fernuni-hagen.de/ps/prjs/InferType/ReducingDependencyWithInferType.html`

[9] LA Stein "Delegation is inheritance" in: *OOPSLA* (1987) 138–146.

Flexible Transformation Language

Alexandre A. Santos[1], Luis Menezes[1], and Márcio Cornélio[1]

[1] Departamento de Sistemas Computacionais
Rua Benfica, 455, 50720-001, Brazil.
{aasj, lcsm, mlc}@dsc.upe.br

1. Introduction

The use of automatic refactoring [1] in large scale projects has increased in the last few years. Unfortunately the available tools provide a fixed set of program transformations. Thus, if a user needs a refactoring that is not supplied by a development environment, it is necessary to know deeply the environment architecture in order to implement it, which takes a lot of time and usually is not in accordance with project constraints.

This paper presents the Flexible Transformation System, which can be easily extended with language and refactoring descriptions.

2. The Flexible Transformation System

The transformation system is composed by a set of tools that process a refactoring description and produces a refactoring environment according to the description. The refactoring description makes use of the abstract and concrete syntax of the target language, and its static semantics description, which are written using traditional notations such as BNF and attribute grammars.

During the transformation execution, a validation is performed in order to capture errors like ill-term usage, invalid semantic production, etc. Any error found makes the system abort and return to the initial state with the purpose of avoiding inconsistencies.

The Flexible Transformation System is being developed using Java [2] as programming language and JavaCC [3] as the parser generator. However, the most relevant contribution of our system is a language for describing refactorings.

3. Transformation Language Operators

The transformation language contains operators designed to ease operations against the decorated abstract syntax tree such as tree searching, pattern matching and term rewriting that are useful in refactorings.

In order to demonstrate the applicability of those operators, a sample program wrote in a pseudo-language is used and analyzed focusing on what is going to be affected by the transformations; afterwards the transformation code is explained in detail.

3.1. The Sample Program

Our sample program, written in a pseudo-language, has an attribute `x` and a get method to return it, called `getName`. After the transformation, it is going to have the attribute name and all its usage renamed from `x` to `name` for a better legibility. This transformation is shown in Figure 1.

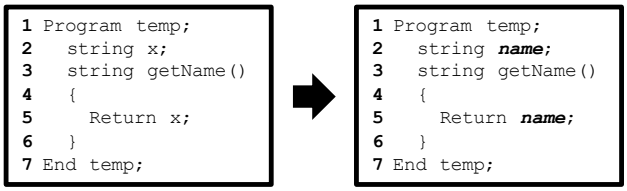

Figure 1 – A Sample Program Transformation.

3.2. The Transformation Code

In this Section, a transformation code written using the Flexible Transformation Language is described to demonstrate how easy are the usage of the operators of this language along with their purposes. Figure 2 presents the code that will be detailed later during this Section.

Basically, it looks for an identifier declaration node which has `x` as name and `string` as type. If found, it verifies if does not exist an identifier declaration node with any type called `name`. If not, it replaces each usage of `x` with `name` and also renames the node declaration from `x` to `name`.

```
01 start {
02   foreach(W in IdDecl("string", "x")) {
03     if (not exist(IdDecl(A, "name"))) {
04       foreach(R in IdUsage("x")) {
05         R=>IdUsage("name");
06       };
07       W=>IdDecl("string", "name");
09     };
10   };
11 }
```

Figure 2 – The Transformation Code.

The lines *01* and *11* delimit the program scope called the *program start point*. Each line inside this block will be executed. However, the start point is not mandatory and someone might implement just procedures to be imported and used by others program transformations. In this case, the program start point will not exist.

Before explaining the `foreach` statement, it is necessary to elucidate the node function as the `foreach` uses it. *Node function* is an expression in the language, responsible for looking in the AST to find a specific node or a set of nodes based on the arguments passed which are filter criteria. Besides searching, the node function could also be used in pattern matching expressions to verify if some variable matches the specified format. In the above example, `IdDecl` and `IdUsage` are node functions defined in the abstract syntax of the pseudo-language.

The lines *02,10* and *04,06* delimit the scope of the first and second `foreach` statement, respectively shown in Figure 2. The command `foreach` is iterative and uses the node function structure to bring back all tree nodes that matches a format defined. This command iterate over the result list making possible the execution of operations over each node. The node function used in the example was the `IdDecl` and `IdUsage` as said before. The first one returns all identifier declarations that have `x` as name and `string` as type and the second one returns all identifier usages that have `x` as name.

The `If-Else` command could be used to verify the existence of a specific node using the `exist` clause, or to verify if the current node matches a specific format using the `is` clause. Both are flexible validations available on the language. The lines *03* and *09* delimit its scope. In this case, it checks if it does not exist an identifier declaration with any type called "name".

Finally, the term rewriting operation, described by the symbol "=>", which is used in the lines *05* and *07*, responsible for transforming a node in a new one. In the first case, it renames the identifier usage from `x` to `name`; in the other case, it renames the identifier declaration from `x` to `name`, preserving the same type.

It is important to mention that the language has other features that were not shown here, such as procedures definitions and string operations since they are standard for the existent languages.

4. Future Plans

In the short term, we are going to have a flexible transformation system based on a generic transformation language, tested against different programming languages. This system will help both refactoring community and developers, in which anyone will be able to define his own transformations and automate the application of complex refactoring as the transformation language defined is easy to understand and to use.

5. References

1. W. Opdyke: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
2. Sun Microsystems. Retrieved 07-2007, from Java SE: http://java.sun.com/javase/.
3. CollabNet. Retrieved 07-2007, from JavaCC Home: https://javacc.dev.java.net/.

A Refactoring Discovering Tool based on Graph Transformation

Javier Pérez, Yania Crespo
Departamento de Informática
Universidad de Valladolid
{jperez,yania}@infor.uva.es

Abstract

One of the problems of documenting software evolution arises with the extensive use of refactorings. Finding and documenting refactorings is usually harder than other changes performed to an evolving system. We introduce a tool prototype, based on graph transformation, to discover refactoring sequences between two versions of a software system. When a refactoring sequence exists, our tool can also help reveal the functional equivalence between the two versions of the system, at least, as far as refactorings can assure behaviour preservation.

1 Introduction

Efforts to include refactorings as a regular technique in software development have led refactoring support to be commonly integrated into development environments (*i.e.* Eclipse Development Platform, IntelliJ® Idea, NetBeans, etc.). Finding refactorings, now they are extensively used, is one of the problems of software evolution [2, 3]. For version management tools, for example, refactorings are modifications more difficult to deal with than any other kind of changes.

Our tool prototype implements a method, based on graph transformation [5, 8], to discover refactoring sequences between two versions of a software system. In case a refactoring sequence exists, our tool can also help reveal the functional equivalence between the two versions of the system, at least, as far as refactorings can assure behaviour preservation.

2 Graph parsing approach

To search refactorings we use graph transformation as an underlying formalism to represent Object-Oriented software and refactorings themselves. Refactorings involve modification of the system structure, so we believe that graph transformation, which focuses on description and manipulation of structural information, is a quite appropriate formalism.

In [4, 7] the graph transformation approach is shown to be valid for refactoring formalisation. In these works, programs and refactorings are represented with graphs, in a language independent way, using a kind of abstract syntax trees with an adequate expressiveness level for the problem. This representation format is claimed to be language independent and very simple, with the purpose of making it easy to use and as flexible as possible. Therefore, as suggested in [7], it was necessary to extend the graph format to represent 'real' programs containing specific elements and constructions of a particular language. We have developed an extension to represent Java programs which we have named 'Java program graphs'.

Once programs have been represented with graphs, and refactoring operations have been described as graph transformation rules, we apply graph parsing algorithms to find the refactoring sequence between two different versions of a software system. We address the problem of finding a transformation sequence from one version of the system to another as a state space search problem. With this approach we identify: **the original/old system** as a start state, **refactoring operations** as state changing operations (edges), **the refactored/new system** as the goal state, **the problem of whether a refactoring sequence exists** as a reachability problem, and **a refactoring sequence** as the path from the start state to the goal state.

We propose a basic search algorithm to look for refactoring sequences. In order to allow some kind of guided search, we base our solution in the use of refactoring preconditions and postconditions. So our approach needs refactoring definitions which include preconditions and postconditions.

The main idea of our algorithm is to iteratively modify the start state applying refactoring graph transformation rules. The set of selectable refactorings at each

iteration is composed just of refactorings whose preconditions are held in the current state and whose postconditions are held in the goal state. When no more refactorings are selectable, the algorithm backtracks to the last transformation applied. The algorithm ends up in success when the current state graph is isomorphic to the goal state graph. The refactoring sequence is the path found to the goal state. The algorithm ends up in fail when no more refactorings can be executed, and the current and goal states are not isomorphic.

3 Our tool so far

Our refactoring discovering tool consists mainly of a **refactoring searching graph grammar** and a **plugin for the Eclipse Development Platform** (see Fig. 1), which has a strong refactoring support.

We have developed a sample implementation of some searching rules to test the validity of our approach. Up to date, the refactoring searching graph grammar searchs for *pullUpMethod*, *renameMethod* and *useSuperType*, supported by the Eclipse refactoring engine, *removeMethod* and *removeClass*. Using graph representation for source code enables to adjust the detail level by adding or removing elements from the graph model. The set of searchable refactorings can be easily extended by adding more searching rules to the grammar.

We use the AGG graph transformation tool [1] as the back-end of our prototype implementation. AGG is a rule-based tool that supports an algebraic approach to graph transformation, and allows rapid prototyping for developing graph transformation systems. We have chosen AGG mainly because it supports graph parsing. Graph parsing can be used to perform depth first search with backtracking, and our algorithm can be partially implemented that way.

Our initial Eclipse plugin [6] obtains the Java program graph representation from the source code of the two versions of a system, launches the graph transformation parser and shows a part of the output dumped by the parser. From this raw information, we are able to identify the refactoring sequence found, but this is only valid for the purpose of testing our approach. There is a clear need to improve the front-end to show up the search results in a more convenient way.

4 Results and future work

We have developed a tool prototype based on graph transformation to find whether a refactoring sequence exists between two versions of a software system or not. Our implementation is a proof of concept that offers very promising results.

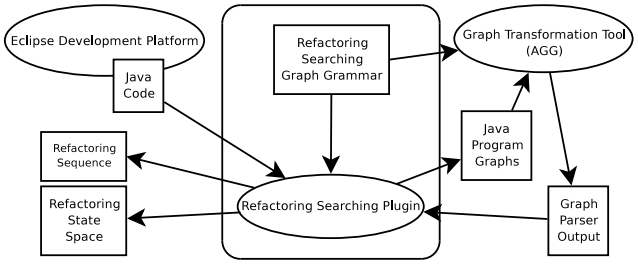

Figure 1. Outline of our tool

Our innmediate objectives are to improve the visualisation of results, to implement refactoring searching rules to support more refactoring operations and to measure the scalability of our technique over industrial-size systems. This will include improving rule descriptions to take benefit of new features being added in the newest versions of the AGG tool or even testing other graph transformation tools for the back-end.

References

[1] Agg home page, graph grammar group, Technische Universität Berlin. `http://tfs.cs.tu-berlin.de/agg`.

[2] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA*, pages 166–177, 2000.

[3] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automatic detection of refactorings in evolving components. In *ECOOP 2006 - Object-Oriented Programming; 20th European Conference, Nantes, France, July 2006, Proceedings*, pages 404–428, 2006.

[4] N. V. Eetvelde and D. Janssens. Refactorings as graph transformations. Technical report, Universiteit Antwerpen, 2005.

[5] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume II: Applications, Languages and Tools*, volume 2. World Scientific, 1999.

[6] B. Martín Arranz. Conversor de Java a grafos AGG para Eclipse. Master's thesis, Escuela Ténica Superior de Ingeniería Informática, Universidad de Valladolid, September 2006.

[7] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, July/August 2005.

[8] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume I: Foundations*, volume 1. World Scientific, 1997.

Refactoring with Contracts

Yishai A. Feldman
IBM Haifa Research Lab
yishai@il.ibm.com

Maayan Goldstein
IBM Haifa Research Lab
maayang@il.ibm.com

Shmuel Tyszberowicz
The Academic College
of Tel-Aviv Yaffo
tyshbe@tau.ac.il

1 Introduction

Design by contract [9] is a practical methodology for developing object-oriented programs together with their specifications. It offers immediate benefits in terms of early error detection as well as long-term process improvement. The contract consists of class invariants and method pre- and postconditions. Method preconditions must hold on entry to the method; it is the responsibility of the caller to make sure they hold when the call is made. Method postconditions must hold on exit from the method, provided that the preconditions held on entry. Class invariants must hold for every object of the class on entry to and exit from every non-private method. The contract is part of the public information of the class, for use by clients.

The contract has several methodological implications [9]:

- It clearly specifies the assumptions underlying the use of existing classes.
- It places constraints on inheritance, according to the behavioral subtyping principle [8]. A subclass must honor the contracts made by its parent classes. Hence, subclasses can only weaken preconditions, and can only strengthen postconditions and invariants. (Violations of this principle are bugs even when contracts are not explicitly specified; they are more likely to go unnoticed in that case.)
- The assertions can be used to prove the correctness of the program at various levels of rigor.

Design by contract is an integral part of the Eiffel programming language [9]. There are a number of tools that instrument Java programs with the contract (e.g., [4, 2, 1]).

Design by contract is synergistic with many agile practices, and can be used to replace unit tests to a significant degree [3]. When refactoring the code, any unit test for affected code must be modified accordingly. If the assertions in the unit tests are replaced by contract checks, tests need only exercise the code, and can therefore be written at a higher level than a single method or even class. However, the contracts still need to be refactored with the code. This is easier than refactoring unit tests, since what programmers have in mind when refactoring code is the intended functionality of the modified code, which is directly expressed by the relevant contracts. Furthermore, it is possible to automate contract refactoring to a large extent [3, 7].

Automating contract refactoring, in conjunction with other tools (such as code instrumentation, verification, and contract discovery), also has the potential to increase acceptance and widespread use of the design-by-contract methodology.

2 Refactoring Contracts

There are several levels of automation required for contract refactoring. Typically, the contract for Java programs is expressed using Javadoc tags such as `@inv`, `@pre`, and `@post`. Contract assertions are just boolean-valued expressions, and therefore need to be treated as code rather than comments in refactorings such as Rename Method or Inline Method. This is relatively easy to automate.

Some refactorings, such as Introduce Null Object and Self-Encapsulate Field, introduce new assertions in fairly obvious ways. Others, such as Push Down Method and Replace Constructor with Factory Method, move or transform contracts in simple ways. Refactorings such as Extract Superclass (in its full generality, when applied to more than one class), require the computation of a contract for the new superclass based on the contracts of the existing classes, in compliance with the behavioral subtyping principle. For example, the class invariant must be weaker than the class invariants of each of the subclasses; it can therefore be computed as their disjunction. In general, a theorem prover is necessary to simplify the computed contract, and various heuristics may be employed in order to obtain the "best"

contract (which is not always the strongest [7]).

Some refactorings, most notably Extract Method, require completely new contracts for arbitrary pieces of code. This is very difficult to do in general, although we have made some steps in this direction [5].

Because of the behavioral subtyping principle, the applicability of some refactorings, such as Move Method, may depend on the contract. If the contract of the method violates behavioral subtyping in its new position, either the contract must be modified appropriately first, or the refactoring is invalid. Here, too, a theorem prover is necessary to discover such cases.

According to Feldman's analysis [3], of the 68 refactorings mentioned in Chapters 6–11 of Fowler's book [6], 32% do not impact contracts except for treating assertions as code. The applicability of some 13% of Fowler's refactorings are constrained by the contracts. In 59% of the refactorings (including 4% that overlap with the previous category), new or modified contracts need to be computed.

Feldman estimated that 71% of Fowler's refactorings can be automated with relative ease, and about 12% require theorem proving for checking constraints. Some 17% require the computation of completely new contracts, which is an open problem.

3 Crepe

In order to investigate the techniques necessary for contract refactoring, we have implemented Crepe (Contract REfactoring Plugin for Eclipse) [7]. Crepe implements a small number of refactorings, including the full form of Extract Superclass. It demonstrates the treatment of assertions as code, the computation of new or modified contracts in the simpler cases, and the use of theorem prover for checking constraints and simplification. We estimate that 83% of Fowler's refactorings can be implemented using the same techniques.

References

[1] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass—Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[2] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Eighth Int'l Workshop on Formal Methods for Industrial Critical Systems (FMICS '03)*, pages 73–89, June 2003.

[3] Y. A. Feldman. Extreme design by contract. In *Fourth Int'l Conf. Extreme Programming and Agile Processes in Software Engineering (XP 2003)*, pages 261–270, Genova, Italy, 2003.

[4] Y. A. Feldman, O. Barzilay, and S. Tyszberowicz. Jose: Aspects for design by contract. In *Proc. Fourth IEEE Int'l Conf. Software Engineering and Formal Methods*, pages 80–89, September 2006.

[5] Y. A. Feldman and L. Gendler. DISCERN: Towards the automatic discovery of software contracts. In *Proc. Fourth IEEE Int'l Conf. Software Engineering and Formal Methods*, pages 90–99, September 2006.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[7] M. Goldstein, Y. A. Feldman, and S. Tyszberowicz. Refactoring with contracts. In *Proc. Agile 2006 Int'l Conf.*, pages 53–64, July 2006.

[8] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Programming Languages and Systems*, 16(6):1811–1841, 1994.

[9] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

Synchronizing Refactored UML Class Diagrams and OCL Constraints

Slaviša Marković and Thomas Baar
École Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
Email: {slavisa.markovic, thomas.baar}@epfl.ch

***Abstract*—UML class diagrams are usually annotated with OCL expressions that constrain their possible instantiation. In our work we have investigated how OCL annotations can be automatically updated each time the underlying diagram is refactored. All our refactoring rules are formally specified using a QVT-based graphical formalism and have been implemented in our tool ROCLET.**

I. Refactoring Class Diagrams

In this section we give a motivation for performing UML/OCL refactorings and show on an example, how OCL constraints have to be treated when the underlying UML class diagram changes. Note that our approach does *not* aim to improve the structure of OCL expressions in order to get rid of OCL smells (see [1]). We are just concerned about smells in UML class diagrams, how to eliminate these smells by class diagram refactorings, and how to keep the annotated OCL constraints in sync with these changes.

Figure 1 shows the application of refactoring *MoveAttribute* on a class diagram annotated with one OCL invariant. The refactoring moves attribute `telephone` from class `Person` to class `Info`. In order to preserve the syntactical correctness of annotated constraints, it is necessary to rewrite all navigation expressions of form *exp*`.telephone` by *exp*`.info.telephone`.

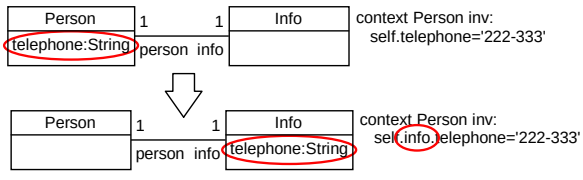

Fig. 1. *MoveAttribute* Example

The only preconditions for this refactoring are (1) that the attribute is moved over an association with multiplicities 1-1 and (2) that in the destination class (`Info`), or in any of its descendants and ancestors, there is no attribute with the same name as the name of the moved attribute (`telephone`).

There are other refactoring rules, which do not influence annotated OCL constraints but whose applicability depends on the absence of OCL expressions of a certain type. One example is the rule *PushDownAttribute*; Fig. 2 shows an application where attribute `color` is pushed down from class `Vehicle` to class `Car`. This refactoring is only possible if for all occurring expressions *exp*`.color` the type of subexpression *exp* conforms to destination class `Car`.

TABLE I
Overview of UML/OCL refactoring rules

Refactoring rules	Influence on OCL	Precondition
RenameClass	No*	UML
RenameAttribute	No*	UML
RenameOperation	No*	UML
RenameAssociationEnd	No*	UML
PullUpAttribute	No	UML
PullUpOperation	No	UML/OCL
PullUpAssociationEnd	No	UML/OCL
PushDownAttribute	No	UML/OCL
PushDownOperation	No	UML/OCL
PushDownAssociationEnd	No	UML/OCL
ExtractClass	No	UML
ExtractSuperclass	No	UML
MoveAttribute	Yes	UML
MoveOperation	Yes	UML
MoveAssociationEnd	Yes	UML

*–*Rename* refactorings influence textual notation of OCL constraints but not their metamodel representation

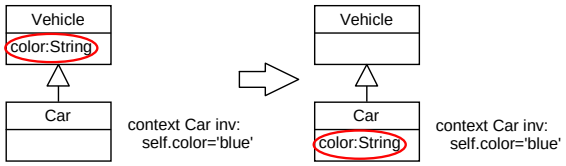

Fig. 2. *PushDownAttribute* Example

In [2], we have investigated and formalized a catalog of class diagram refactorings together with necessary changes of OCL constraints. Table I gives an overview of refactorings that can be applied on a class diagram, together with information whether the refactoring influences OCL constraints, and which part of the UML/OCL model is checked by the refactoring's application condition.

II. Model Transformations

UML class diagrams and their OCL constraints can be seen as models (i.e. instances of corresponding metamodels). The refactoring of UML/OCL models is a special type of model transformation and can, thus, be specified by the OMG standard QVT (Query/View/Transformation).

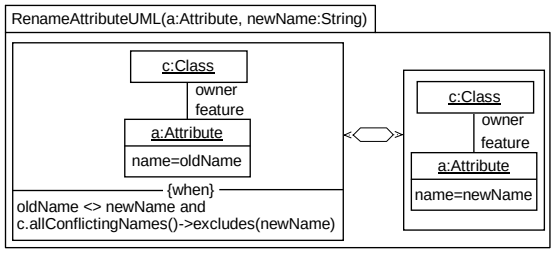

Fig. 3. QVT Formalization of *RenameAttribute* Refactoring

The formalization of the refactoring (*RenameAttribute*) is shown as a QVT rule in Fig. 3. QVT rules consist of basically two patterns (LHS, RHS). When applying the rule, occurrences of LHS are searched for in the non-refactored model that we want to change. If an occurrence is found, it is substituted with the corresponding instantiation of RHS. Additional constraints specified in the "when" clause specify formally the application conditions for the refactoring rule (ignoring them could result in syntactically invalid target models). For more information on the formalization of refactorings, we refer the interested reader to [2]. The refactoring rule *RenameAttribute* does not have an influence on attached OCL constraints. More complicated rules that have an influence (e.g. *MoveAttribute*), are formalized by two QVT rules; one describing the changes in the class diagram and a second for updating the OCL (see [2] for details).

III. Lessons Learned

A model refactoring is usually defined as a model transformation for which source and target model are instances of the same metamodel. During our work on implementing QVT-specified refactoring rules we have noticed that it is sometimes useful to relax this definition and to allow source and target model to have different metamodels.

A. Syntax Preservation

Refactoring rules should be syntax-preserving; i.e. syntactically correct source models should always be mapped to syntactically correct target models. However, syntax preservation is sometimes technically difficult to achieve, especially, if the metamodel contains hundreds of well-formedness rules.

Syntax preservation becomes easier to handle when refactoring is seen as a two-step process: (1) the source model is transformed to an intermediate model, which is an instance of a different metamodel; (2) from the intermediate model the final target model is recovered by a second transformation. In case of UML/OCL refactorings, the intermediate metamodel could represent OCL constraints as text and the refactoring rules just have to "produce" text in order to represent synchronized OCL constraints. The second recovery step would then parse the produced text as OCL constraints and create an instance of the original UML/OCL metamodel. Another possibility for an intermediate metamodel could be to use the original UML/OCL metamodel, but without any of its derived model elements. In this case, the only task of the recovery step would be to complete the intermediate model to an instance of the original UML/OCL metamodel by adding the (so far missing) derived model elements.

B. Behavior Preservation

In case of UML class diagram refactorings, the definition of behavior preservation in traditional program refactoring as "same inputs lead to the same output" is not applicable because class diagrams represent only the static structure of a system.

Our criterion for behavior preservation is based on the evaluation of OCL constraints in a system snapshot. In [3], we propose to call UML/OCL refactorings *behavior preserving* if the evaluation of a non-refactored OCL constraint on a valid instance of a non-refactored UML class diagram yields always the same result as the evaluation of the refactored OCL constraint on the corresponding instance of the refactored UML class diagram.

Contrary to some authors, like [4], we allow object diagrams also to be refactored. We believe that our definition of semantic correctness gives more freedom in performing refactorings and allows wider spectrum of refactoring rules to be applied on a UML class diagram.

IV. Conclusions

In this paper we have presented our approach of refactoring UML class diagrams annotated with OCL constraints. All refactorings that can be applied on class diagrams are specified as model transformation rules and implemented in our RoclET tool [5].

Moreover, an overview of lessons learned during the process of formalization and implementation is given. We think that the technique to handle refactorings as a 2-step process can help to simplify the refactoring of many other software artifacts as well.

References

[1] Alexandre Correa and Cláudia Werner. Applying refactoring techniques to UML/OCL. In *UML 2004*, volume 3273 of *LNCS*, pages 173–187. Springer, 2004.

[2] Slaviša Marković and Thomas Baar. Refactoring OCL annotated UML class diagrams. *Software and Systems Modeling (SoSym)*, 2007. In press. Online available under DOI 10.1007/s10270-007-0056-x.

[3] Thomas Baar and Slaviša Marković. A graphical approach to prove the semantic preservation of UML/OCL refactoring rules. In *PSI 2006*, volume 4378 of *LNCS*, pages 70–83. Springer, 2007.

[4] Rohit Gheyi, Tiago Massoni, and Paulo Borba. A static semantics for alloy and its impact in refactorings. *Elsevier's Electronic Notes in Theoretical Computer Science (To appear)*, 2006.

[5] RoclET homepage. http: //www.roclet.org, 2007.

Code Analyses for Refactoring by Source Code Patterns and Logical Queries

Daniel Speicher, Malte Appeltauer, Günter Kniesel
Dept. of Computer Science III, University of Bonn - Germany
{dsp, appeltauer, gk}@cs.uni-bonn.de - roots.iai.uni-bonn.de

***Abstract*—Preconditions of refactorings often comprise complex analyses that require a solid formal basis. The bigger the gap between the level of abstraction of the formalism and the actual implementation is, the harder the coding and maintenance of the analysis will be. In this paper we describe a subset of GenTL, a *gen*eric analysis and *t*ransformation *l*anguage. It balances the need for expressiveness, high abstractness and ease of use by combining the full power of a logic language with easily accessible definitions of source code patterns. We demonstrate the advantages of GenTL by implementing the analyses developed by Tip, Kiezun and Bäumer for generalizing type constraints [11]. Our implementation needs just a few lines of code that can be directly traced back to the formal rules.**

I. Introduction

Preconditions for refactorings require thorough analyses based on a solid formal foundation. In order to reduce the implementation effort, risk of errors, and cost of evolution it is desirable to have an implementation language at a similar abstraction level as the formal foundation. Part of this problem has been addressed by various approaches to logic meta-programming [4], [13]. They have demonstrated that the expressive power of logic meta-programming enables the implementation of powerful program analyses for refactoring (e.g. [12]).

Unfortunately, logic meta-programming requires programmers to know the meta-level representation of the analysed language and to think and express their analyses in terms of this representation. For instance, JTransformer, our own logic meta-programming tool for Java [8], [5] represents methods by a predicate with seven parameters. The full Java 1.4 AST is represented by more than 40 predicates. Mastering these predicates and the precise meaning of each of their parameters can be error-prone and forces programmers to think at the abstraction level of the meta-representation.

In this paper we offer the expressive power of logic meta-programming but raise the abstraction level by providing means of expressing constraints on the structure of source code elements without having to learn a new API. This is achieved by a predicate that selects source elements based on source code patterns containing meta-variables as place-holders. Thus the concept of meta-variables is all that programmers have to learn in addition to mastering the analysed language. The corresponding concepts of GenTL are introduced in Section II. In Section III we introduce the problem of type generalizing refactorings and the corresponding formal basis elaborated by Tip, Kiezun and Bäumer in [11]. In Section IV we show how GenTL can easily express the analyses of [11].

II. GenTL

GenTL is a generic program analysis and transformation language based on logic and source code patterns. For lack of space we describe here only its analysis features. We start by the introduction of meta-variables and code patterns, then we introduce the selection of program elements based on code patterns and finally we show how arbitrary predicates can be easily built on this simple infrastructure.

A. Code Patterns

A *code pattern* is a snippet of base language code that may contain meta-variables. A *meta-variable* (MV) is a placeholder for any base language element that is not a syntactic delimiter or a keyword. Thus meta-variables are simply variables that can range over syntactic elements of the analysed language. In addition to meta-variables that have a one-to-one correspondence to individual base language elements, *list meta-variables* can match an arbitrary sequence of elements, e. g. arbitrary many call arguments or statements within a block. Syntactically, meta-variables are denoted by identifiers starting with a question mark, e. g. ?val. List meta-variables start with two question marks, e. g. ??args. Here are two examples:

```
(a)  ?call()        (b)  ?called_on.?call(??args)
```

The pattern (a) above specifies method calls without arguments. If evaluated on the program shown in Figure 1 it matches the expressions x.a(), m() and y.b(). For each match of the pattern, the meta-variable ?call is bound to the corresponding identifier (a, m and b). Pattern (b) only matches x.a() and y.b() because it requires the calls to have an explicit receiver. Each match yields a tuple of values (a *substitution*) for the MV tuple (?called_on, ?call, ?args). In our example the substitutions are (x,a,[]) and (y,b,[]), where [] denotes an empty argument list.

B. Element Selection

The predicate is (written in infix notation) enables selection of program elements based on their structure expressed using code patterns:

```
<metavariable> is [[<codepattern>]]
```

```
class A          { void a(){} }
class B exends A{ void b(){} }
class C {
  B m() {
    B x = new B();   // [new B()] <= [x]
    x.a();           // [x] <= A
    return x;        // [x] <= [C.m()]
  }
  void n() {
    B y = m();       // [C.m()] <= [y]
    y.b();           // [y] <= B
  }
}
```

Figure 1. Method invocations, assignments, parameter passing and returns impose constraints on the types [E] of contained expressions E.

The predicate unifies the meta-variable on the left hand side with a program element matched by the code pattern on the right hand side. If the pattern matches multiple elements, each is unified with the corresponding metavariable upon backtracking.

C. Element Context

For many uses, it is not sufficient to consider only a syntactic element itself but also its *static context*. For example, the *declaring type* contains important information about a method or a field declaration. Also the statically resolved binding between a method call and its called method (or a variable access and the declared variable) is necessary for many analyses. This information is available via *context attributes*, which can be attached to meta-variables by double colons. Figure 2 describes the attributes used in this paper.

D. Self-Defined Predicates

The is predicate provides an intuitive way to specify the assumed *structure* of program elements. Context attributes let us concisely express a few *often used relations* between elements. However, for complex analyses, these features need to be complemented by a mechanism for expressing *arbitrary relations* between program elements. Therefore, GenTL lets programmers define their own predicates based on the concepts introduced so far.

Predicates are defined by rules consisting of a left hand side and a right-hand-side separated by ':-'. Multiple rules for the same predicate (that is, with the same left-hand-side) express disjunction. The right-hand-side (the body) of a rule can contain conjunctions, disjunctions and negations. Predicates can be defined recursively, providing Turing-complete expressiveness.

For example, the term 'declaration element' used in [11] denotes the declaration of the static type of methods, parameters, fields and local variables. The predicate decl_element implements this rule, associating each element with its declared type as follows:

?mv::decl	The statically resolved declaration of the element bound to ?mv. Calls reference the called method; variable accesses the declaration of the accessed field, local variable or parameter; type expressions reference a class or interface.
?mv::type	The statically resolved Java type of the expression bound to ?mv.
?mv::encl	The enclosing method or class of a statement or expression bound to ?mv.

Figure 2. Context attributes used in this paper

```
decl_element(?method,?type) :-
 ?method is [[??modif ?type ?name(??par){??stmt}]].
decl_element(?parameter,?type) :-
 ?parameter is [[?type ?name]].
decl_element(?field_or_var,?type) :-
 ?field_or_var is [[?type ?name;]].
decl_element(?elem,?type) :-
 ?elem is [[?type ?name = ?value;]]
```

Each rule describes one possible variant of a declaration element. Each element's structure is specified by a pattern. For instance, the first rule states that the declared type of a method declaration is its return type. The ?method argument of the element predicate called within the rule represents the method declaration. The second argument contains a code pattern describing the structure of method declarations. The pattern contains several meta-variables: ??modif, matching an arbitrary number of modifiers, ?type for the return type, ?name for the method name, ??par for possible parameters and ??stmt for the statements of the method body.

The second clause selects parameter declarations (they are not terminated by a semicolon). The third clause selects field and local variable declarations without an initializer. The fourth one captures initializers. The syntax of code patterns in GenTL generalized the one described in [10].

III. Type Generalization Refactorings and Type Constraints

In this section we introduce by example the challenge of type generalization analysis and the solution approach based on type constraints.

Let us consider the method m in Figure 1. It defines the local variable x to be of type B although only the method a, defined in the more general type A is actually invoked on x. Therefore one might hope to be able to generalize the type of x to A. This would eliminate an unnecessary dependency on a too concrete type. The utility of such dependency reduction becomes obvious if we consider m as a substitute for a whole subsystem that should be decoupled from the subsystem containing the type B.

Unfortunately, the intended generalization is not possible in our example. Method n indirectly enforces the use of B in m: As b is called on y, y has to be of type B. Because the result of m is assigned to y, the return type of m must be B,too.

Finally, x also has to be of type B because it is returned as the result of m.

The approach described in [11] enables us to deduce this relation formally from type constraints implied by method invocations and assignments (including the implicit assignments represented by return statements and parameter passing). The comments in Figure 1, for example, show the constraints for the statements on their left-hand-side. For example the initialization of y with the result of a call to m implies that the return value of m has to be a subtype of y's type ($[C.m()] \leq [y]$). Combination of the inequalities shown in Figure 1 yields the inequality $[x] \leq [C.m()] \leq [y] \leq B$, thus formally proving the necessity of x being of type B.

Summarizing, our example illustrates that

1) the invocation of the method a on x (resp. b on y) implies that the receiver type must be at least A (resp. B);
2) assignments and return types propagate these restrictions to the types of further expressions;
3) based on these constraints we can derive a chain of inequalities proving x must be typed with B, hence cannot be generalized.

In the following section we present the related formal constraints from [11] and our implementation in GenTL[1].

IV. Analysis for Type Generalization

We first implement predicates that capture the type constraints required for our example. Then we show how these predicates can be used to implement the test for non-generalizability.

A. Type Constraints

Method calls. The type of an expression that calls method M must be a subtype of "the most general type containing a declaration of M", denoted $Encl(RootDef(M))$[2]. This is expressed in [11] by the following type constraint:

$$(Call) \quad call\ E.m() \text{ to a virtual method } M \Rightarrow [E] \leq Encl(RootDef(M))$$

We can map the rule (Call) directly to the following rule of the predicate constrained_by_type(?elem,?type), which states that the type of ?elem is at most ?type:

```
constrained_by_type (?elem,?type) :-
  ?call is [[?E.?m(??args)]],
  ?elem = ?E::decl,
  ?M_decl = ?call::decl,
  root_definition(?M_decl, ?rootMethod),
  ?type = ?rootMethod::encl.
```

The first line of the right-hand-side specifies the structure of method calls which the rule is applicable to. The second says that the call constrains the type of the declaration of the message receiver. Unification of two variables is denoted with the infix operator '='. The fourth line determines the root definition of the called method, using the predicate root_definition, which implements the function $RootDef(M)$. The last line says that the type of the message receiver is constrained by the declared type of the root definition. .

Assignment. The type of the right hand side of an assignment must be a subtype of the one of the left hand side:

$$(Assign)\ E1 = E2 \Rightarrow [E2] \leq [E1]$$

This is implemented as a rule for the predicate constrained_by(?e2,?e1). It represents the restriction of the type of the element ?e2 by the declaration of the element ?e1:

```
constrained_by(?E2_decl, ?E1_decl) :-
  ?assign is [[?E1 = ?E2]],
  ?E1_decl = ?E1::decl,
  ?E2_decl = ?E2::decl.
```

Return. The type of a an expression returned by a method must be a subtype of the method's declared type. This is expressed formally as:

$$(Ret)\ return\ E \text{ in method } M \Rightarrow [E] \leq [M]$$

In GentTL, this reads:

```
constrained_by (?E_decl, ?M_decl) :-
  ?return_stmt is [[return ?E;]],
  ?M_decl = ?return_stmt::encl,
  ?E_decl = ?E::decl;
```

B. Test for Generalizeability

The non-generalizability of declarations in a given program P is checked on the basis of the inferred type constraints. The set of non-generalizable elements, $Bad(P,C,T)$, contains all elements of P whose declared type C cannot be replaced with the more general type T. This is the case if the type constraints imply that an element must be typed with a type that is *not* a supertype of T (second line below) or that is a subtype of a non generalizable element (fourth line below):

$$\begin{aligned}(Gen) \quad & Bad(P,C,T) = \\ & \{E \mid E \in All(P,C) \wedge [E] \leq C' \in TC_{fixed}(P) \\ & \quad \wedge \neg T \leq C'\} \cup \\ & \{E \mid E \in All(P,C) \wedge [E] \leq [E'] \in TC_{fixed}(P) \\ & \quad \wedge E' \in Bad(P,C,T)\}\end{aligned}$$

In the above definition, $E \in All(P,C)$ means that E declares an element of type C in program P. $TC_{fixed}(P)$ is the set of type constraints derived for P. The second line corresponds to the test implemented by the predicate constrained_by_type. The fourth line corresponds to the test implemented by constrained_by.

The rule (Gen) is implemented by the predicate not_generalizable(?elem,?generalizedType). It succeeds if

[1] Due to space limitations we omit some details (definition of *root_definition* and handling of multiple subtypes) in the formalism and in our implementation.

[2] We slightly adapted the original notation $Decl(RootDef(M))$ of [11] in order to avoid confusion with the 'decl' context attribute of GenTL.

the declaration of ?elem is not generalizable to the type ?type. Each line on the right hand side of the two implementing rules corresponds to a line of the formal rule (Gen). The two rules express the disjunction in (Gen):

```
not_generalizable(?elem,?generalizedType) :-
  constrained_by_type(?elem,?type),
  !subtype(?generalizedType, ?type).

not_generalizable(?elem,?generalizedType) :-
  constrained_by(?elem,?upper),
  not_generalizable(?upper, ?generalizedType).
```

V. Implementation & Application

The implementation of GenTL is still work in progress. Pattern predicates are successfully implemented in LogicAJ2 [10], a fine-grained aspect-oriented language that is a predecessor of GenTL. GenTL is translated to the logic meta-programming representation supported by the JTransformer system [5], [8]. This mapping is described in [1].

By now, we provide an implementation of type generalization analysis in JTransformer. This analysis has been integrated into our Cultivate plugin for Eclipse [3]. It is run automatically, whenever a source file is saved. Statements that can be generalized are marked and the usual Eclipse 'warning' tooltip indicates the most general types to which they could be generalized. This is illustrated in Figure 3, which shows a slight variation of our example. Here, it is possible to generalize the type of the variable x, the method m and the variable temp to A because b() is not invoked on temp but on the wrapper object y. All lines affected by the possible generalization are highlighted and the ones where generalizations are possible get an additional warning marker.

VI. Conclusion

In this paper we have presented GenTL, an extension of a logic language by a predicate supporting program element selection based on source code patterns containing meta-variables. We have demonstrated that this concept fosters a direct mapping of formal program analysis specifications to their logic based implementation. The formal foundations for GenTL are laid by the theory of logic-based conditional transformation [6], [7], [9]. The implementation of pattern predicates is based on JTransformer [5]. Efficiency and scalability of this system in conjunction with the compilation of logic programs supported by the CTC [2] is demonstrated in [8]. For instance, the identification of all instances of the observer pattern in the Eclipse platform implementation (≈ 11.500 classes) needs less than 8 seconds. Therefore, we think that the design of GenTL opens the door for a desirable mix of high run-time performance and extremely short development time enabled by the high abstraction level supported.

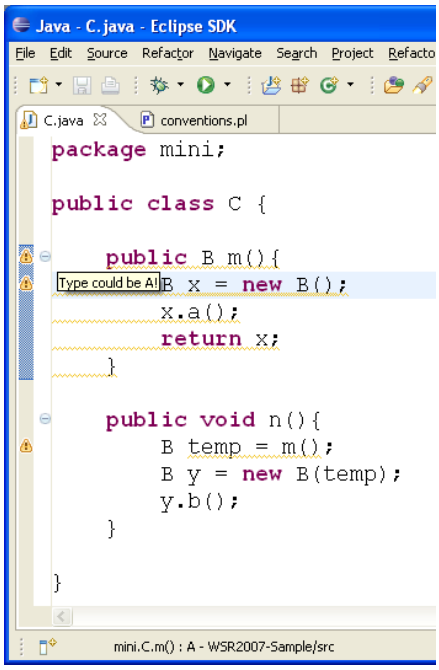

Figure 3. Tool tips indicating possible type generalizations detected by automated logic-based analysis.

References

[1] Malte Appeltauer and Günter Kniesel. Towards concrete syntax patterns for logic-based transformation rules. In *Eighth International Workshop on Rule-Based Programming*, Paris, France, July 2007.

[2] CTC homepage. http://roots.iai.uni-bonn.de/research/ctc/, 2006.

[3] Cultivate homepage. http://roots.iai.uni-bonn.de/research/cultivate/.

[4] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[5] JTransformer homepage http://roots.iai.uni-bonn.de/research/jtransformer/.

[6] Günter Kniesel. A Logic Foundation for Conditional Program Transformations. Technical report IAI-TR-2006-01, ISSN 0944-8535, CS Dept. III, University of Bonn, Germany, January 2006.

[7] Günter Kniesel and Uwe Bardey. An analysis of the correctness and completeness of aspect weaving. In *Proceedings of Working Conference on Reverse Engineering 2006 (WCRE 2006)*, pages 324–333. IEEE, October 2006.

[8] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *Linking Aspect Technology and Evolution*, March 12 2007.

[9] Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming (Special issue on Program Transformation)*, 52(1-3):9–51, August 2004. http://dx.doi.org/10.1016/j.scico.2004.03.002.

[10] Tobias Rho, Günter Kniesel, and Malte Appeltauer. Fine-grained Generic Aspects, Workshop on Foundations of Aspect-Oriented Languages (FOAL'06), AOSD 2006. Mar 2006.

[11] Frank Tip, Adam Kieżun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, pages 13–26, Anaheim, CA, USA, November 6–8, 2003.

[12] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *7th European Conference on Software Maintenance and Reengineering, Proceedings*, pages 91–100. IEEE Computer Society, 2003.

[13] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, 2001.

Reuse based refactoring tools

Raúl Marticorena, Carlos López
Area of Languages and Computer Systems
University of Burgos (Spain)
{rmartico, clopezno}@ubu.es

Yania Crespo, Francisco Javier Pérez
Deparment of Computer Science
University of Valladolid (Spain)
{yania, jperez}@infor.uva.es

Abstract

Current refactoring tools work on a particular language. Each time it is intended to provide refactoring support for new languages, the same refactoring operations are defined and implemented again from scratch. This approach ignores reuse opportunities in this matter. It is possible to define a way of collecting code information suited for several languages (a family of languages) and define refactoring operations over that representation. On the other hand, it is also possible to define and implement each refactoring operation by composing previously defined and developed elements. In this paper we show the current implementation of a reuse oriented refactoring engine and its specialization for a particular language.

Key Words: *refactoring, reuse, composition, language-independence*

1 Introduction

One of the open trends in refactoring [2] is the construction of language-independent refactoring tools. Language independence, or at least certain language independence, allows to reuse previous efforts in defining and implementing refactoring when support for a new language must be provided. It is also aimed at obtaining a rational solution to provide refactoring operations for development environments, specially for those which support several languages.

We present a refactoring tool, using a framework based on a Minimal Object-Oriented Notation, named MOON. The use of this minimal notation allows to abstract the main concepts over a set of object-oriented languages. Language particularities must be provided by framework specialization and extension.

A refactoring engine, based on the MOON core and extensions, is responsible of checking and executing the refactoring elements on the code. Finally, the refactored code is generated. In order to provide more powerful reuse capabilities, refactoring operations are defined by composition. A refactoring is composed of preconditions, actions and postconditions (following [3], [4]). On the one hand, conditions allow to check applicability from the point of view of behavior preservation. On the other hand, actions transform the code, changing its current state through add, remove and rename operations. Pre and postconditions are functions and predicates that query the model and actions are model transformers. Each pre, postcontion or action is stored in a repository to be reused when defining new refactoring operations. We have built an extension of the MOON framework core to deal with Java code information, in order to manage all the information of the source code.

1.1 Refactoring Engine

The refactoring engine runs the refactoring definitions and obtains a new object model with the new state. A framework definition has been used to allow a simple scheme of reuse, as can be seen in Fig. 1.

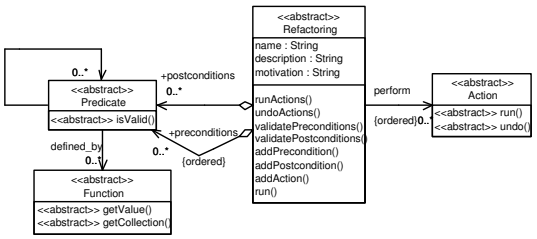

Figure 1. Refactoring Engine Framework

Using the Template Method Pattern Design [1], each refactoring has to be defined with stages, using the repository content.

1.2 Refactoring Repository

Refactoring elements are implemented as classes (see Fig. 2). These classes query or transform the current model instance. Although the model extension contains the information of real code (i.e. Java), most of the classes work with the MOON metamodel abstractions. This proposal allows to reuse the same query or action, when the related concepts are the same in several languages. For example, the precondition `ExistParameterWithName` or the action `MoveAttributeAction`, stored in the repository, are reusable for several languages.

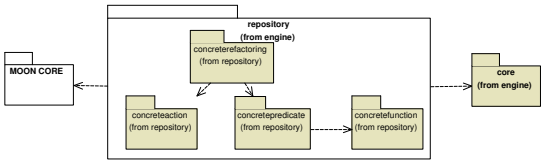

Figure 2. Repository Architecture Overview

2 Current State

The current version of the tool implements eleven refactoring operations (Fig. 3):
- add, rename and remove parameter.
- rename classes and methods.
- move attributes and methods.
- four refactoring operations, we have defined, on generic classes.

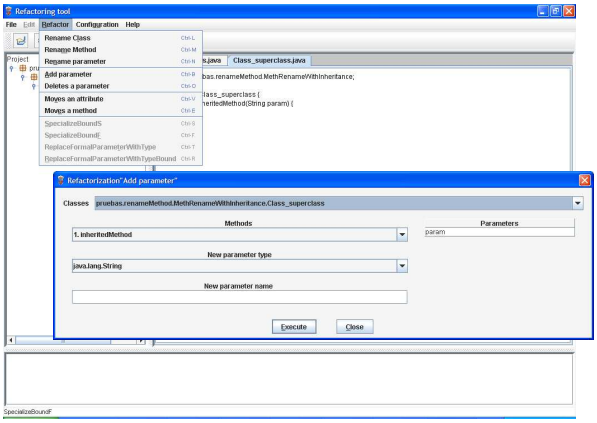

Figure 3. Refactoring Tool

Each one of these refactorings are implemented as concrete classes (extending the `Refactoring` abstract class as can be seen in Fig. 1). The refactorings are built from instances of pre and postconditions classes and action classes, using the corresponding `add` methods of the template (Fig. 1).

The `repository` contains the implementation of these elements, allowing the programmer to compose the refactoring. If the element is not available, the programmer should add the new code needed to the repository. Hence, last refactorings to be added are implemented with a minor effort, because the complete set of their elements is already available in the repository in order to be reuse.

3 Future Works

We have presented a refactoring tool which intents to provide some advantages: certain language independence, which allows to reuse the same refactoring implementation (or a very similar one) for different languages and refactoring construction by composition, which allows to implement new refactorings from pieces already available from previously introduced refactoring operations. The current version of the tool allows to run the refactorings over a simple set of toy codes. The Java parser is being completed to support commercial code, and a C# parser (with its own framework extension) is under development to validate the solution.

We are also currently working on a declarative definition of refactorings using XML. This makes easy to compose refactorings from the repository elements using a graphical interface. Since specialization could be necessary, the declarative definition could be also specialized for different languages with a high degree of reuse.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[2] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[3] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.

[4] Donald Bradley Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1999.

SORMASA: A tool for suggesting model refactoring actions by metrics-led genetic algorithm

Thierry Bodhuin Gerardo Canfora Luigi Troiano

RCOST – University of Sannio
Viale Traiano – 82100 Benevento, Italy
{bodhuin,canfora,troiano}@unisannio.it

ABSTRACT

In this paper we introduce SORMASA, SOftware Refactoring using software Metrics And Search Algorithms, a refactoring decision support tool based on optimization techniques, in particular Genetic Algorithms.

1. INTRODUCTION

During the development of object oriented software, dependencies (e.g. method call and use of attributes) between classes emerge that were not identified or specified explicitly at design time. This leads to a more complex software system than one desired having the same functionalities, hence, design modifications could be done in order to create a more manageable software in terms of maintenance and code reuse. Hereby, design decisions should be revised in the spirit of improving coupling and cohesion [4]. In order to reach this result, refactoring actions are generally performed aiming to improve the quality of the software architecture [1]. The ultimate goal is to increase cohesion of code and reducing coupling, but still keeping the initial idea of the solution architecture and semantics. Agile programming emphasized the role of refactoring up to support a process of *continuous refactoring*. This resulted into a pressing demand for tools able to automate refactoring tasks, or to support refactoring decisions [3].

Refactoring is often done by applying a series of transformations on an existing and often incomplete software system. When such transformations are applied, the software system remains fully compliant to the original requirements, differing only the implementation. While a single transformation (e.g. moving a method from a class to another) may not improve too much the software system, a series of such transformations may produce significant effect. Obviously, not all refactoring primitives (more than 80 primitives are listed at www.refactoring.com) are suitable for keeping low the software complexity. Among them, we have used two primitives, namely the `MoveField` and `MoveMethod`, that concerns respectively a field and method movement between two classes, hence affecting the coupling and cohesion of such classes. However, even considering few simple primitives, the effect of combining several transformations at the same time can lead to hard decisions due to the high number of possible combinations.

Refactoring can be viewed as an optimization problem, where each solution represents a set of refactoring actions, that if applied lead the system to an architecture entailing a different cohesion and coupling. A search based approach provides an interesting and viable solution to this problem, as it is able to automatically consider a high number of refactoring alternatives, then suggesting those that, if undertaken, can lead to a more cohesive and less coupled architecture. In this paper we describe SORMASA, SOftware Refactoring using software Metrics And Search Algorithms, developed at our research center with the goal of supporting decision making in refactoring a software architecture.

The current version of SORMASA makes some simplifying assumptions, that are (i) only field and method movements between classes are considered, (ii) transformations are assumed in isolation, ignoring the effect they can produce on each other, and (iii) all original classifiers are kept, with no new class or interface introduced or removed. This leads to transformations that are independent on the order in which they are applied and without modifications of the overall architecture. In the context of suggesting redesign actions, as addressed by SORMASA, these assumptions are not very limitative. Indeed, the goal is to suggest actions at model level concerning the best way of allocating class properties (i.e. methods and attributes), leaving the final decision to the user. Preliminary experimentation shows encouraging results.

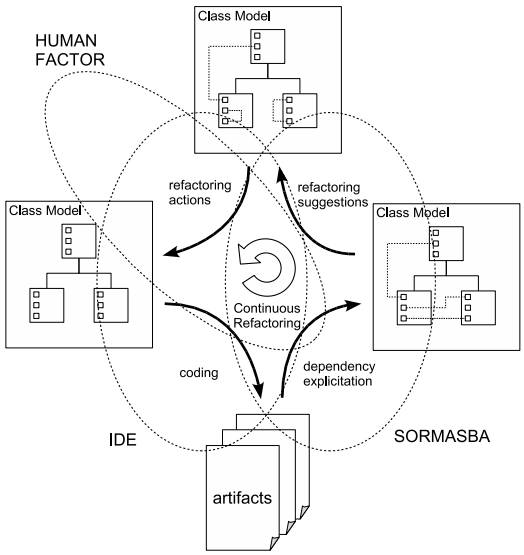

Figure 1: The role of SORMASA in a process of continuous refactoring.

Figure 1 depicts the role of SORMASA in the process of continuous refactoring. The initial class model is coded in software artifacts (i.e. `.java` source files, `.class` bytecode), generally using a Java IDE such as Eclipse or NetBeans. SORMASA analyzes them (in particular bytecode) in order to identify and make explicit structural dependencies. The optimization process of SORMASA is aimed to identify refactoring opportunities that could improve the model quality in terms of high cohesion and low coupling. Opportunities are presented to the user that can decide which refactoring actions to undertake. The modifications will lead to a new revised software model from which the refactoring process may start again.

2. SEARCH ALGORITHM

SORMASA's architecture is designed to work with different quality measurements (e.g. fitness function based on cohesion, coupling, complexity, etc.) and search algorithms. The current release supports Cohesion, Coupling [4], and Distance (that is the number of changes applied to the initial model) metrics for quality measurements and Genetic Algorithms (GA) as search algorithm. A set of refactoring primitives that are under consideration are coded like in Figure 2.

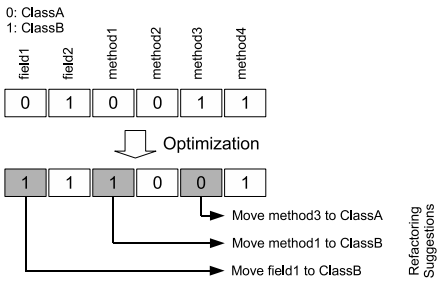

Figure 2: The chromosome structure and usage.

Refactoring suggestions can be obtained by comparing the solution to the initial model, then identifying the relocation of properties. SORMASA uses a Simple GA as described in [2], and that be outlined as follows:

1. An initial population of model candidates is randomly chosen.
2. The fitness of each candidate is evaluated.
3. (a) Select individuals for mating, according to their fitness
   (b) Perform crossover of selected individuals
   (c) Perform mutation
   (d) Replace individuals on population with offsprings
4. Until the maximum number of generations is reached repeat from 4.

Key aspects in the algorithm are (i) the fitness function, (ii) the genetic operators and (iii) the replacement policy. SORMASA allows to specify each of these aspects.

The effectiveness (fitness) of refactoring actions fit can be obtained using a function based on structural metrics such as cohesion ch and coupling cp, as they respectively represent the relatedness of class functionalities and the degree of dependency of a class on other classes. Moreover, we also consider the distance d from the initial model as we prefer solutions that do not disrupt the original architecture. These variables are combined by weighted product, as

$$fit = ch^{w_{ch}} \cdot (1 - cp)^{w_{cp}} \cdot (1 - d)^{w_d} \quad (1)$$

All metrics are within the unary interval $[0, 1]$. This is the function we used in our experimentation, but other fitness functions may be specified, also including additional metrics.

The available genetic operators are selection (Tournament, Roulette Wheel), crossover (One-point, Two-points) and mutation (Simple) [2]. The replacement policies supported by SORMASA are (i) replacement of worst individual, that is slower but facilitating the algorithm convergence, and (ii) random replacement of individuals, that is faster [2].

3. AN EXAMPLE OF APPLICATION

Considering the example depicted in Figure 3, we can notice a set of dependencies between `ClassA` and `ClassB`, resulting into structure coupling. Moreover the structure is not cohesive, as methods can be partitioned according to the usage of class attributes. Moving `method3` to `ClassA` and `method1` to `ClassB` provides a structure that is minimally coupled and maximally cohesive.

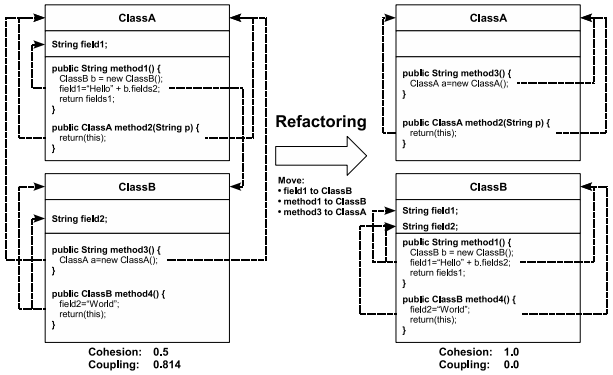

Figure 3: An example of refactoring aimed at minimizing coupling and at maximizing cohesion.

We notice that, these dependencies can emerge and become explicit mostly during the coding phase, as they depend on the actual use of class members. Hereby the need for refactoring. Obviously, the decision in undertaking refactoring actions depends also on the semantics of code, and this is a task left to the user. SORMASA only provides support in exploiting refactoring opportunities that can lead to code that is more cohesive and less coupled. SORMASA is able to deal with situation more complex than the one depicted in Figure 3, including inheritance of properties along with class generalization/specialization, and interface implementation.

4. CONCLUSIONS AND FUTURE WORK

SORMASA is a tool for supporting refactoring decisions aiming at optimizing the quality of software system (e.g. maximizing the cohesion and minimizing the coupling). This is obtained by implementing a search-based approach that is able to identify refactoring opportunities and to propose them to the user. SORMASA is at an early stage of development, and we plan to expand the feature set in order to:

1. Include more optimization techniques. In particular genetic programming looks a promising approach for searching a structured set of refactoring primitives that optimizes a quality function (i.e. fitness function). This would make possible to consider refactoring procedures, instead of simple primitives, such as the move of properties among classes.
2. Integrate SORMASA with Eclipse and NetBeans IDE. This can lead to have online refactoring suggestions during the coding of software solutions, thus enabling a continuous refactoring process.
3. Consider explicit refactoring and semantic constraints, able to better preserve software requirements.

5. REFERENCES

[1] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1st ed edition, 1999. ISBN 0-201-48567-2.

[2] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Professional, January 1989.

[3] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[4] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured Design. *IBM Systems Journal*, 13(2):115–139, 1974.

Model-driven software refactoring

Position paper for the 1st ECOOP Workshop on Refactoring Tools

Tom Mens, *Université de Mons-Hainaut, Belgium*

Gabriele Taentzer, *Philipps-Universität Marburg, Germany*

Abstract. In the current state of the practice on software development, there are two very important lines of research for which tool support is becoming widely available. The first one is *program refactoring*, the second one is *model-driven software engineering*. To this date, however, the links and potential synergies between these two lines of research have not been sufficiently explored. Therefore, we claim that more research on model-driven software refactoring is needed, and we explore the obstacles that need to be overcome to make this happen.

Introduction

In the emerging domain of model-driven software engineering (MDE), we are witnessing an increasing momentum towards the use of models for developing software systems. By raising the level of abstraction, the uniform use of models promises to cope with the intrinsic complexity of software systems, and thus opens up new possibilities for creating, analyzing, manipulating and formally reasoning about these systems. To reap all the benefits of MDE, it is essential to develop languages, formalisms, techniques and tools that support *model transformation*. They will enable a wide range of different automated activities such as translation of models, generating code from models, model refinement, model synthesis or model extraction, and *model refactoring*. The latter activity can be considered as the model-level equivalent of program refactoring (Fowler, 1999), a well-known technique to improve the quality of software. A detailed survey of this very active research domain can be found in (Mens and Tourwé, 2004). In this position paper, we discuss how the ideas of model transformation and program refactoring may be combined, and we explore the research challenges associated to this combination.

One of the straightforward ways to address refactoring in a model-driven context is by raising refactorings to the level of models, thereby introducing the notion of *model refactoring*, which is a specific kind of model transformation that allows us to improve the structure of the model while preserving its quality characteristics. Dealing with model refactorings, however, is far from trivial. Consider the scenario depicted on the left of Figure 1. It clearly illustrates the potentially high impact a simple refactoring may have on the software system. We assume that a model is built up from many different views (e.g., class diagrams, state diagrams, use case diagrams, interaction diagrams, activity diagrams). We also assume that the model is used to generate code, while certain fragments of the code still need to be implemented manually. Whenever we restructure a single model view (step 1 in Figure 1), it is likely that we need to synchronise all related views, in order to avoid them becoming inconsistent (step 2 in Figure 1). Next, since the model has been changed, part of the code will

need to be regenerated (step 3 in Figure 1). Finally, the manually written code that depends on this generated code will need to be adapted as well (step 4 in Figure 1). This need for synchronisation between different model views and between the model and the code, respectively, upon model refactoring has not been addressed in detail in research literature. If a model is being refactored, how should the corresponding source code be modified accordingly? Vice versa, if source code is being refactored, how will the models be affected?

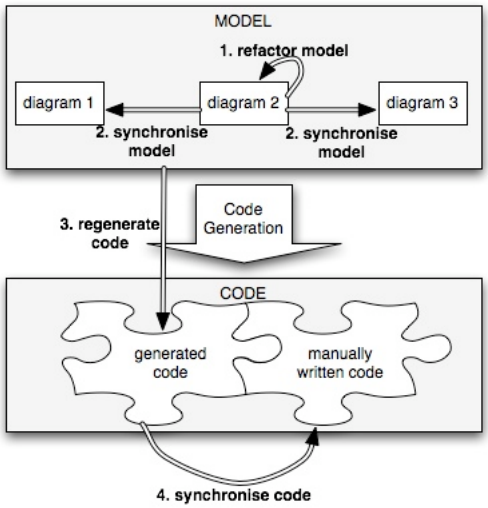

Figure 1: Scenarios for model-driven software refactoring.

From a technical point of view, the Eclipse Modeling Framework (EMF) defines an up-to-date standard for specifying models. In this context, model refactorings can be considered as model transformations within one and the same language. This kind of rule-based model transformation is performed "in place", i.e., the current model is directly changed and not copied. An EMF model transformation framework supporting these properties has been presented in (Biermann et.al. 2006). It consists of a visual editor, an interpreter and a compiler to Java.

We are currently carrying out a case study to get more insight in model-driven software refactoring. From a theoretical point of view, we rely on the theory of graph transformation (Mens, 2006, Ehrig et.al 2006) to reason about refactoring in a formal way. For example, one can reason about properties such as termination, composition, parallel dependencies, and sequential dependencies. From a technical point of view, we rely on AndroMDA, a state-of-the-art generator for web applications from UML models. AndroMDA drives code generation heavily by stereotypes and tagged values. Therefore, refactoring methods need to be adapted to AndroMDA models and extended with more domain-specific information. Furthermore, also entirely new "decidated" refactorings for AndroMDA models need to be discussed. Due to comprehensive code generation, model refactoring can affect the actual behaviour/functionality of the application, even those are considered as standard ones.

Challenges

Based on our experiences, we hope to shed more light on the following challenges:

- How can we formally define model quality, and how can we assess the (positive or negative) effect of refactoring on this quality?

- How can we deal with model synchronisation in an incremental way, when part of the model (or its corresponding source code) has been refactored?

- How can we ensure that a model refactoring *preserves the behaviour*? This requires a formal definition of (different notions of) "behaviour" in general, and for models in particular. A formalism could be used to verify which behavioural aspects are preserved by which model refactoring.

A more pragmatic approach would be to resort to *model testing* techniques: before and after each refactoring step, tests are executed to ensure that the behaviour remains unaltered. Even more challenging is to test or verify the model transformations directly.

- How can we provide refactorings for domain-specific modelling languages in a generic way? Given the large number of very diverse domain-specific languages, it is not feasible, nor desirable, to develop dedicated tools for all of them from scratch. A generic model transformation engine could be the basis to specify and maybe also analyse refactorings for domain-specific models. Model transformation engines based on different kinds of models have been developed by e.g. Zhang *et al.* (2004) and Biermann et.al. (2006).

- A final challenge is that all of the above should be implemented in model-driven development environments in an as efficient and scalable way as possible, otherwise it will never be adopted by practitioners (Egyed 2006).

References

Biermann, E., Ehrig, K., Köhler, C., Taentzer, G., & Weiss, E. (2006). Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. Proceedings of International Conference on Model Driven Engineering Languages and Systems, O. Nierstrasz (Ed.), Lecture Notes in Computer Science, 4199, 425-439, Springer

Egyed, A. (2006), Instant consistency checking for the UML. In: Proc. International Conference on Software Engineering (pp. 31-390), ACM

Ehrig, H., Ehrig, K., Prange, U. & Taentzer, G. (2006), Fundamental Approach to Graph Transformation, EATCS Monographs, Springer

Fowler, M. (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley.

Mens, T., & Tourwé, T. (2004). A Survey of Software Refactoring. IEEE Transactions on Software Engineering, 30(2), 126-162.

Mens, T. (2006). On the use of graph transformations for model refactoring. In Generative and Transformational Techniques in Software Engineering, Lecture Notes in Computer Science, 4143, 219-257, Springer.

Mens, T., Taentzer, G., & Runge, O. (2007). Analyzing Refactoring Dependencies Using Graph Transformation. Journal on Software and Systems Modeling 2007. Springer. To appear.

Zhang, J., Lin, Y., & Gray, J. (2005). Generic and Domain-Specific Model Refactoring using a Model Transformation Engine, In Model-driven Software *Development - Research and Practice in Software Engineering*, Springer.

The "Extract Refactoring" Refactoring

Romain Robbes and Michele Lanza
Faculty of Informatics, University of Lugano - Switzerland

Abstract

There is a gap between refactoring tools and general-purpose program transformation tools that has yet to be filled. Refactoring tools are easy to use and well-established, but provide only a limited number of options. On the other hand, program transformation tools are powerful but are viable only for large transformation tasks. We propose an approach in which a developer specifies transformations to a program by example, using an IDE plugin recording the programmer's actions as changes. These changes could be generalized to specify a more abstract transformation, without the need of a dedicated syntax. Defining refactorings and transformations from concrete cases would enable more frequent uses of medium scale transformations.

1 Introduction

Refactoring [1], [2] has become a well-established program restructuring technique. Indeed, several major Integrated Development Environments feature a refactoring engine which automates the most common refactoring operations [3]. However, the refactorings supported by a refactoring engine are often limited in number and extent: only a fixed number of transformations are implemented. If a more complex change to a program is needed, it must either be done manually, or with the help of a generic program transformation tool.

Such program transformation tools [4], [5] are very powerful and allow large scale transformations to be performed with a much lower cost than if done manually. However, these tools still have a rather high barrier to entry, making them only suitable for large-scale transformations: They require the user to learn a transformation syntax and to have a high capacity in abstracting and reasoning at the meta level in order to define the transformation. [5] describes how a tool named DMS was used to migrate an application from one component style to another. They mention that such an approach is not wortwhile for small applications.

2 Restructuring a Program by Example

To fill the gap between refactorings and program transformations we propose an approach based on change recording and generalization. In such an approach a programmer provides concrete instances of a transformation manually and generalizes them to fully specify a transformation. This approach relies on a framework which records a programmer's actions in an IDE and model them as program transformations or *changes*. Each of these changes takes as input an abstract syntax tree (AST) of the system being monitored and returns a modified AST of the program.

Transformations recorded this way operate on specific entities of the AST (*e.g.* add method *setConcreteBar* to class *AbstractBar*). To define a generic transformation, a programmer takes this concrete transformation and progressively abstracts it until his goal is reached (*e.g.* add method *setConcrete***X** to class *abstract***X**).

These transformations would then be instantiated: The programmer would fix set the variables of the transformation (telling which class is **X**), before executing it. He would then evaluate the results and modify the transformation before retrying, should the result be incorrect.

3 Example

A programmer, Bob, discovers that class **Bar** from the system he is working on has too many responsibilities. **Bar** should be split in two classes: each instance of **Bar** should hold an instance of class **Baz**. The behavior encoded in **Baz** could thus vary if a subclass of **Baz** is given. This change is not trivial: Several methods in **Bar** need to move in **Baz**, and be replaced by delegation stubs. In addition, some accesses to instance variables of **Bar** need to be replaced by accessors.

To implement this change, Bob performs it first concretely, by delegating method **foo** from **Bar** to **Baz** and changing a direct access to variable **bag** to an accessor. Bob then examines his actions in his change history to generalize his change. He ends up with a generic change affecting two classes **A** and **B**, a variable **v** and a set of methods **SM**. The

transformation moves the implementation of the methods in **SM** from **A** to **B**, defines delegation stubs in **A** (forwarding the call to instance variable **v**, an instance of **Baz**), and replaces accesses of variables belonging to **Baz** by accessors.

Bob then applies the change to the method **foo** he first modified to verify that the results are the same. He then applies it to all necessary methods in class **Bar**, and can store the transformation should he need to move behavior across classes in the future.

4 Related Work

Apart from program transformation tools, our work is close to the field of programming by example [6], [7]. Programming by example consists in recording user actions and generalize them in a program. Our approach is based on the same principle, but is restricted to defining transformations.

Boshernitsan and Graham defined a visual language aimed at easing program transformations [8]. The transformation task is simplified, but programmers still have to specify the transformation: They can not provide a concrete instance of it.

5 Conclusion

We proposed an approach in which programmers can specify program transformations by giving concrete examples of them. Transformations should be expressed more easily and hence used more often than with current approaches.

The ideas described in this paper are partially implemented. Recording developer actions and converting them to change operations is provided by our prototype, Spyware [9]. Spyware has been previously used for software evolution analysis. Change generalization and application need to be implemented, and a suitable user interface should be built.

References

[1] Opdyke, W.F.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois (1992)

[2] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)

[3] Roberts, D., Brant, J., Johnson, R.E., Opdyke, B.: An automated refactoring tool. In: Proceedings of ICAST '96, Chicago, IL. (1996)

[4] Roberts, D., Brant, J.: Tools for making impossible changes - experiences with a tool for transforming large smalltalk programs. IEE Proceedings - Software **152** (2004) 49–56

[5] Akers, R.L., Baxter, I.D., Mehlich, M., Ellis, B.J., Luecke, K.R.: Reengineering c++ component models via automatic program transformation. In: WCRE, IEEE Computer Society (2005) 13–22

[6] Halbert, D.C.: Programming by Example. Ph.D. thesis, Dept. of EE and CS, University of California, Berkeley CA (1984) Also OSD-T8402, XEROX Office Systems Division.

[7] Lieberman, H.: Your Wish Is My Command — Programming by Example. Morgan Kaufmann (2001)

[8] Boshernitsan, M., Graham, S.L.: Interactive transformation of java programs in eclipse. In: ICSE. (2006) 791–794

[9] Robbes, R., Lanza, M.: A change-based approach to software evolution. In: ENTCS volume 166, issue 1. (2007) 93–109

[10] Rubin, R.: Language constructs for pogramming by example. 3rd ACM-SIGOIS Conf on Office Information Systems, also SIGOIS Bulletin **7** (1986) 92–103

Advanced Refactoring in the Eclipse JDT: Past, Present, and Future

Robert M. Fuhrer
IBM Research
rfuhrer@watson.ibm.com

Markus Keller
IBM Zurich
markus_keller@ch.ibm.com

Adam Kieżun
MIT CSAIL
akiezun@csail.mit.edu

Abstract: In this position paper, we present the history, the present and our view of the future of refactoring support in Eclipse.

Eclipse was among the first IDEs to help bring refactoring to the mainstream developer. Eclipse version 1.0 included several highly useful Java refactorings, which are nowadays staple tools in most Java developers' toolbox. These included Rename, Move, and Extract Method. Eclipse 2.0 added a lot of statement-level refactorings such as Extract and Inline Local Variable, Change Method Signature, and Encapsulate Field. Some refactorings, such as Rename, offer great leverage because of the potential scale of the changes they perform automatically. Others, like Extract Method, are more local in scope, but relieve the developer from performing the analysis required to ensure that program behavior is unaffected. In both cases, the developer benefits from reduction of a complex and numerous changes to a single operation. This helps to maintain his focus on the big picture. Moreover, the ability to roll back the changes with a single gesture enables exploration of structural possibilities more easily, and without fear of irreparable damage to the code base.

Eclipse 2.1 included several type-oriented refactorings such as Extract Interface and Generalize Type, whose benefits derive from addressing the problems of both scale and analytic complexity. These used a common analysis framework [1] based on theoretical work from Palsberg et al. [2] for expressing the system of constraints that ensure the type-correctness of the resulting program. Such frameworks are important because they speed the development of entire families of refactorings, and help ensure their correctness (e.g., Steimann et al. proposed Infer Interface refactoring based on the type-constraint framework [8]). Our belief is that the incorporation of reusable and extensible frameworks for the various classical static analyses (type, pointer, data flow) into Eclipse will be critical to the expansion of our suite of refactorings.

Eclipse 3.0 saw the beginnings of the capability to deal with refactoring over multiple artifact types, which in part dealt with a problem well-known to Eclipse plug-in developers: the Rename refactoring had been previously oblivious to references located in plug-in meta-data, so that renaming an extension implementation class would break the reference, leaving the extension class unreachable by the extension point framework. Since extension points are the sole mechanism for providing functionality in Eclipse, this was a serious problem. As a generalization, the Eclipse Language ToolKit (LTK) provided a “participant” mechanism, allowing additional entities to register interest in a given type of refactoring, and participate in both checking pre-conditions and contributing to the set of changes required to effect the refactoring. Using this mechanism, breakpoints, launch configurations, and other artifacts outside the source itself can be kept in sync with source changes. As applications are increasingly built using multiple languages, this ability becomes critical to the applicability of automated refactoring to the mainstream developer.

Eclipse 3.1 included Infer Type Arguments [3], a migration refactoring that helps Java 5 developers migrate client code of libraries to which type parameters have been added. The migration is important because it can greatly increase static type safety. The necessary analysis, however, is subtle and pervasive enough that many developers might hesitate to perform the migration manually. Of particular interest was the Java 5 Collections library, which had been retrofitted with type parameters. In particular, the Infer Type Arguments refactoring infers the types of objects that actually flow into and out of the instances of these parametric types, and inserts the appropriate type arguments into the corresponding variable declarations as needed. In some sense, the underlying analysis reconstructs an enhanced model of the original application, recovering lost or implicit information that may be critical to maintenance or further development. As such, this kind of refactoring offers great benefits in maintaining or even "revitalizing" legacy code.

Before Infer Type Arguments can be applied, however, the library itself must be parameterized. Java 5 Collections were parameterized manually, but many other existing libraries would benefit from added type-safety and expressiveness, if they were converted to use generics. A recently described refactoring, Introduce Type Parameter [4], addresses this complex issue. With the addition of such a refactoring, the Eclipse JDT would support developers in a wide spectrum of generics-related maintenance tasks.

Eclipse 3.2 introduced a team-oriented innovation: storing API refactorings with the API library itself, along with a “playback” mechanism to automatically perform the necessary transformations on API client code when the new library is imported

[6,7]. Such tools help smooth the interactions amongst team members or even amongst teams, by automatically propagating the effects of changes from one component to another, or perhaps by automatically making the necessary changes implied by another. As software development becomes more and more distributed, we believe this sort of tooling will become increasingly vital.

Eclipse 3.3 offers an Introduce Parameter Object refactoring. Additionally, a great number of CleanUps that can also be applied to source files on save, for example Organize Imports, Format, or adding missing J2SE-5-style annotations.

With all of the functionality now in Eclipse, we are still a long way from achieving the benchmark of complete coverage of Martin Fowler's refactoring catalog [5]. Moreover, this is only a start; many more transformations are possible. The future promises more emphasis on parallelism in the form of multi-core platforms, clusters and massive machines consisting of thousands or even millions of processors. Concurrency-aware and concurrency-targeted refactorings will be important tools to speed the development of such highly parallel software. Additionally, most current refactorings effect changes on relatively fine-grained structures such as methods, fields, expressions, statements, and individual types; refactorings that manipulate coarser-grained structures (e.g., packages, entire type hierarchies, components etc.) could enable refining software at nearly the architectural level.

What do we need to deliver on the promise of such a rich suite of transformations? In Eclipse, a refactoring consists of several phases: precondition checking, detailed analysis, and source rewriting. We make three recommendations that, in our opinion, would ease the development of refactorings.

First, we need a simpler source rewriting mechanism to avoid writing painful imperative code that creates AST nodes one-by-one. Such a mechanism would be especially helpful if it provided assurances of correctness of the generated constructs, or at least performed run-time checks to help check correctness. The AST and import rewriters already shield refactoring implementers from low-level formatting issues, but higher level APIs would foster more reuse of "refactoring components".

Second, we need a better means of specifying the underlying analyses, which maps onto efficient and scalable implementations that permit the application of refactorings to large code bases. Third, much research is needed in understanding the semantics of and manipulating the increasingly prevalent mixtures of languages.

Additional avenues to pursue that would greatly expand the reach of our tooling include that of the Holy Grail of developer-specified refactorings, and that of more general (non-behavior-preserving) developer-specified transformations. The latter tooling could replace the dangerous and yet still prevalent language-oblivious macro processors like cpp or m4, giving developers static safety combined with the power to greatly reduce the difficulty of creating regular code structures.

With the combination of these meta-tools at our disposal, both Fowler's catalog and an even richer space of transformations could be within reach; without them, they are likely to take years to attain.

References

[1] F. Tip, A. Kiezun, and D. Baeumer. Refactoring for generalization using type constraints. In OOPSLA, pp. 13–26, Nov. 2003.
[2] J. Palsberg, and M. Schwartzbach, Object-Oriented Type Systems. John Wiley & Sons, 1993.
[3] R. M. Fuhrer, F. Tip, A. Kiezun, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. ECOOP, pp. 71–96, July 2005.
[4] M. Fowler, Refactoring. Improving the Design of Existing Code. Addison-Wesley, 1999.
[5] A. Kiezun, M. D. Ernst, F. Tip and R. M. Fuhrer. Refactoring for parameterizing Java classes. In ICSE, May 2007
[6] J. Henkel, A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution, In ICSE, pp. 274--283, 2005
[7] D. Dig. Using refactorings to automatically update component-based applications, In OOPSLA Companion, 2005
[8] F. Steimann, P. Mayer and A. Meissner. Decoupling classes with inferred interfaces, In SAC, pp.1404--1408, 2006

Product Line Variability Refactoring Tool

Fernando Calheiros
Meantime Mobile Creations
fernando.calheiros@cesar.org.br

Paulo Borba
Informatics Center - UFPE
phmb@cin.ufpe.br

Sérgio Soares
Computing Systems Department - UPE
sergio@dsc.upe.br

Vilmar Nepomuceno
Meantime Mobile Creations
vilmar.nepomuceno@cesar.org.br

Vander Alves
Lancaster University - UK
v.alves@comp.lancs.ac.uk

Abstract

With the growing academic and industrial interest in Software Product Lines (SPL), one area demanding special attention is tool support development, which is a pre-requisite for widespread SPL practices adoption. In this paper, we present FLiPEx, a code refactoring tool that can be used for extraction of product variations in the context of developing mobile game SPLs.

Keywords Refactoring, Software Product Lines

1. Introduction

The extractive and the reactive Software Product Line (SPL) [1] adoption strategies [4] involve, respectively, bootstrapping existing products into a SPL and extending an existing SPL to encompass another product. In both cases, product line refactorings [2][3] are useful to guide the SPL derivation process by extracting product variations and appropriately structuring them. They also help to assure the safety of the whole process by preserving SPL configurability [2] — the resulting SPL has at least the same instances than the initial set of independent products being bootstrapped or the initial SPL being extended.

In single system refactoring, ensuring safety and effectiveness of a refactoring process already requires automation, in practice, In the SPL context, where complexity increases due to the need to manage a high number of variants, such support becomes even more indispensable. In this context, we describe FLiPEx, a refactoring tool that implements code transformations [3] for extracting product variations from Java classes to AspectJ aspects. Aspects are used since we need a better modularization technique. The tool is built on top of Eclipse and interacts with the FLiPG tool, which integrates with Feature Model (FM) [5] tools for updating a SPL FM accordingly to code transformations, which, for example, might turn mandatory into optional features. FLiPEx has been designed and tested in the context of mobile game SPLs.

The main contribution of this paper is to describe our experience on designing and developing FLiPEx, its supported refactorings, and the associated user-centric view of the SPL refactoring process (Section 2). We also present and discuss FLiPEx's architecture (Section 3).

2. FLiPEx

FLiPEx is based on the Eclipse plugin platform and uses the Eclipse infrastructure to perform source code refactorings that extract product variations. The tool extracts code related to an application feature and modularizes it using AspectJ aspects. Besides refactoring source code, FLiPEx, interacting with FLiPG, also updates the SPL feature model and the configuration knowledge, for example by including new extracted features into the model and adding the aspect to the configuration knowledge so that when the feature is selected, the corresponding aspect will appear in the product.

We will illustrate the entire refactoring process with one of the implemented refactorings: *Move Extends Declaration to Aspect*, Consider the following example. `MainCanvas` is a class responsible for managing the graphical part of the application, the graphics depend on the API provided by each device. Depending on the API available at the devices, the `extends` clause will change.

```
import com.nokia.mid.ui.FullCanvas;
public class MainCanvas extends FullCanvas
{…}
```

or,

```
import javax.microedition.lcdui.Canvas;
public class MainCanvas extends Canvas {…}
```

The code snippet below shows the result of applying this refactoring for the first variation. The refactoring consists of checking the preconditions of the selected code above, which is represented with bold text, removing the original code, and generating the AspectJ code:

```
//core
public class MainCanvas {...}

//Nokia configuration
import com.nokia.mid.ui.FullCanvas;
public aspect NokiaCanvasAspect {
    declare parents: MainCanvas extends
         FullCanvas;
}
```

This aspect will insert the variation at the point from where it was extracted, preserving the behavior of the original product but offering the possibility of plugging in to the common code alternative variations of that feature.

The refactoring process is presented to the user in the form of a wizard that the user interacts with to provide all the information required to perform the refactoring. The user is presented with a list of suggested refactorings; after selecting the refactoring, s/he selects or creates the features to which the code to be extracted belongs, and then chooses the destination aspects and associates them with the selected features. In the previous example, all destination features will be alternative, and each aspect will define a `declare parents` clause for the `MainCanvas` class. The possibility of selecting several destination aspects to which the generated AspectJ construct is copied helps the user to develop different implementations of the same variation. When the user performs the extraction of a single feature spread throughout the code, FLiPEx provides an option to remember the features and aspects selection, thus simplifying the process of code extraction, which is minimized to selecting the desired refactoring.

The following list summarizes the refactorings provided by the FLiPEx tool in its current version.

- Extract Before Block
- Move Field to Aspect
- Move Import Declaration to Aspect (this uses an ABC[7] extension we've developed)
- Move Interface Declaration to Aspect
- Move Method to Aspect
- Move Extends Declaration to Aspect

The implementation of six more refactorings is planned, including a family of Extract After Block (call, execution, initialization and set) refactorings, and more refined ones such as a refactoring for extracting context and moving static block to before-initialization.

3. Architecture

FLiPEx is part of the FLiP tool suite, built upon the Eclipse plugin platform. Figure 1 shows the relation between FLiPex, FLiPG, the Eclipse framework, and a Feature Model tool, currently Pure::Variants [5].

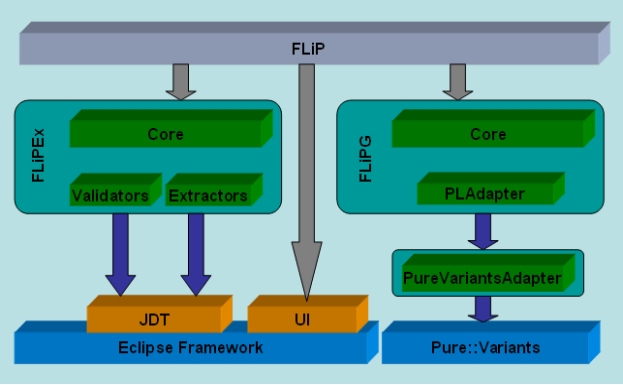

Fig. 1 Architecture

All parts of the FLiPEx schema are independent plugins, *Core* is the main plugin and gives support to the *Extractors* and *Validators*. It is responsible for acquiring the selected code from Eclipse's Java editor, creating an object that contains all the information needed to perform the validation and extraction, such as the beginning and final position of the selected code, the file from which it will be extracted, and the file into which the aspect code will be written.

Each refactoring has an *Extractor*, which is responsible for removing the code from the Java class and creating the AspectJ code that inserts the variation. Each *Extractor* has a corresponding *Validator*, which is responsible for analyzing the selected code to check if it meets all the preconditions of its refactoring [3]. When a user wants to extract some code, FLiPex runs all registered *Validators* through that code, returns a list of the extractors whose preconditions have been met, and a list of the extractors that cannot be used and the reasons their *Validators* failed.

FLiPEx already provides an abstract base extractor class that takes care of most of the peripheral tasks that all extractors need to perform, such as writing the Java and the aspect files, updating the imports, and removing the selected code from the Java source file. With this base extractor, the code of the concrete extractors is very small and only takes care of creating the aspect code that will be inserted in the aspect file. Additional refactorings can be added to the platform through the extractors and validators extension points that FLiPEx exports to plugin developers. Also additional feature model tools can be used with FLiP, since it exports an extension point for feature model tools adapters.

FLiPEx uses JDT's infrastructure, AST and visitors, to perform analysis on the preconditions that must be met by the Java classes and to remove the selected code from the Java class. Due to incompleteness of AJDT's [6] plugin infrastructure for AspectJ AST manipulation, code generation in FLiPEx is currently being done by string manipulation.

4. Conclusions

We have presented FLiPEx, a tool for SPL refactorings, introduced the refactorings it implements and a few that will be implemented in the future, and described its architecture. Eclipse's plugin infrastructure was of easy understanding, not only because of useful documentation, but also due to the available framework, which is easy to use. On the other hand, working with AJDT was difficult because its infrastructure for AST manipulation is not finished yet, so our code generation is hardcoded with string manipulation at this point.

As future work, we intend to develop more refactorings, integrate with other feature model tools, and improve our process of product line refactoring.

Acknowledgments

We gratefully acknowledge the other FLiP team members: Isabel. Wanderley, Geraldo. Fernandes, Andréa. Frazão and former members Jorge Pereira and Davi Pires. We would also like to thank Meantime Mobile Creations and C.E.S.A.R. for supporting the FLiP project. This research was partially sponsored by CNPq and MCT/FINEP/CT-INFO.

References

[1] P. Clements and L. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley, 2002.

[2] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring Product Lines. In *Proceedings of GPCE'06*, October 2006. ACM Press.

[3] V. Alves, P. Matos Jr, L. Cole, P. Borba, and G. Ramalho. Extracting and Evolving Mobile Games Product Lines. In *Proceedings of the 9th SPLC'05*. September 2005. Springer-Verlag.

[4] C. Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering*. Spain, October 2001.

[5] Pure::Variants. http://www.pureystems.com/Variant_Management.49.0.html, 2007

[6] AJDT: AspectJ Development Tools. http://www.eclipse.org/ajdt/, 2007.

[7] abc. The AspectBench Compiler. http://www.aspecbench.org, 2007.

AJaTS: AspectJ Transformation System

Roberta Arcoverde
Informatics Center - UFPE
rla4@cin.ufpe.br

Sérgio Soares
Computing Systems Department - UPE
sergio@dsc.upe.br

Patrícia Lustosa
Informatics Center - UFPE
plvr@cin.ufpe.br

Paulo Borba
Informatics Center - UFPE
phmb@cin.ufpe.br

Abstract

The interest in aspect-oriented software development naturally demands tool support for both implementation and evolution of aspect-oriented software, as well as refactoring current object-oriented software to aspect-oriented. In this paper, we present AJaTS – a general purpose AspectJ Transformation System, that supports AspectJ code generation and transformation. AJaTS allows the definition of specific transformations, providing a simple template-based language, as well as pre-defined aspect-oriented refactorings.

Keywords Refactoring, Aspect-Oriented Programming, Code Generation

1. Introduction

Aspect-Oriented programming intends to increase software modularity, by separating the implementation of concerns which generally crosscut the system. Therefore, AOP addresses some object-oriented programming issues, like tangled and spread code, usually related to the implementation of transversal requirements. AspectJ [3], an aspect-oriented extension to Java [2], allows the definition of separated entities called aspects, which implement crosscutting concerns. This separation improves software quality, since it increases its modularity and reuse.

Due to its power and simplicity, the implementation of aspect-oriented systems with AspectJ is becoming each day more common. Tool support for AspectJ transformations has therefore become very important. However, there are still few tools that provide AspectJ programs generation and transformation, as well as refactoring's definition support.

In this paper, we present AJaTS – a general purpose AspectJ Transformation System, that supports AspectJ code generation and transformation. The main contribution of this paper is to present AJaTS's application value, describing its functionalities, use scenarios and examples of aspect-oriented refactorings supported.

Section 2 presents an introduction to the AJaTS engine, including its functionalities, template's language and application examples. Next, we discuss the AJaTS's architecture and technical points and Section 5 offers our concluding remarks.

2. AJaTS

AJaTS – AspectJ Transformation System – was conceived as a general purpose AspectJ Transformation System that supports AspectJ code generation and transformation. The main concept in AJaTS transformations is the capability of enable the user to specify templates for matching and code generation. Such templates are defined in a simple transformation language, similar to the target language. Such similarity makes AJaTS transformations easier to define and to understand. This feature allows the implementation of refactorings in a declarative way using a language, rather than hard coding refactorings in programs that manipulate AST or source code. This makes easier to write, to understand, and to evolve refactorings with AJaTS.

We show examples of both matching and generation templates below:

```
//matching template
public aspect #ASPECT_NAME { }
//generation template
public aspect #ASPECT_NAME {
    private String newField;
}
```

The matching template will match the source code, defining which classes/aspects will be transformed, as well as which structures will be saved in AJaTS variables. The generation template defines the transformation itself.

The basic constructs of the template's language are the AJaTS variables (i.e.: `#ASPECT_NAME`), used as information placeholders in a transformation. These variables have well defined types that can vary since a simple identifier until a whole set of methods of a class or aspect. The AJaTS variables are preceded by a '`#`' character. AJaTS template's language also offers more complex constructs, like conditional control (`#if, #else`) and loops (`forall`).

The AJaTS engine allows the user to define general transformation templates and applying them to any aspect-oriented project. Likewise, it also allows the generation of specific aspects, refactoring object-oriented software to aspect-oriented.

Besides allowing any developer to write their own transformation templates, AJaTS also brings some pre-defined useful transformations, which can be automatically applied to any Java/AspectJ project. One of these transformations is the Distribution Concern implementation [5]. It generates aspects that provide distribution, by modifying the system's façade, business entity classes and adding some auxiliary classes to the specified project. The details of this implementation are extensively explained elsewhere [5]. An example of how this transformation affects the system's code is shown above.

```
public class Facade {
  fds
  cds
  mds
}
//generated aspect
public aspect FacadeServerSideAspect {
  declare parents: Facade implements IFacade;
  declare parents : entities implements
    java.io.Serializable;
  ...
}
```

In this example, `entities` represents a list of business entity classes, automatically filled through user's input. This transformation example provide distribution through RMI, but it would be possible to use another distribution technology.

To make the facade instance remote, AJaTS generates an aspect called Server-side Aspect. It modifies the facade class (`Facade`) to implement the following remote interface (`IFacade`), also generated by AJaTS, which is demanded by the RMI API [7].

```
//generated interface
public interface IFacade implements
    java.rmi.Remote { mds' }
```

AJaTS also applies some pre-defined recommended refactorings to AspectJ code. The *Extract Pointcut* refactoring [4], for example, is demonstrated below.

```
//source code
aspect A {
    before() : exp { ... }
    after() : exp { ... }
}
//transformed code
aspect A {
    pointcut pc() : exp;
    before() : pc() { ... }
    after() : pc() { ... }
}
```

In this example, the pointcut `pc` is derived from the replicated expressions `exp`. All these transformations are implemented through templates, using the AJaTS template's language. The templates that perform these transformations are available at the project homepage (http://www.cin.ufpe.br/~jats/ajats).

Next section describes the architecture and implementation issues of AJaTS engine. It also presents the AJaTS plug-in, designed as an Eclipse IDE extension.

3. Architecture

The AJaTS Transformation Engine was conceived as an extension to a previously developed Java Transformation System, i.e., JaTS [1]. Whereas it reuses JaTS mechanisms to perform code generation and transformation, we still had to extend JaTS language and engine in order to support the manipulation of AspectJ code.

In this way, the JaTS parser had to be extended, including AspectJ syntax support. There were also included nodes to represent AspectJ constructs, and their respective meta-variables. These modifications allowed JaTS to create, identify and modify AspectJ syntax trees, performing transformations also in AspectJ programs.

In order to increase modularity and abstract JaTS's code modifications, AJaTS was designed as an aspect-oriented system itself. The visitors responsible for manipulating the AST, performing the engine operations, for example, were extended with methods inter-type declarations (an aspect-oriented construct), defined in separated aspects. Thus, we use AspectJ aspects to integrate AJaTS's code to the JaTS engine – making it easier to maintain. Figure 1 summarizes the AJaTS's extensions over JaTS's architecture: the addition of AspectJ nodes, and extension of the visitors and the parser.

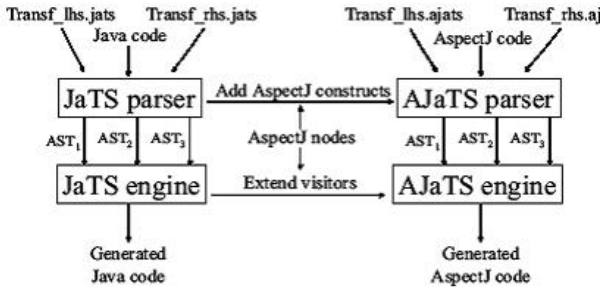

Figure 1 – JaTS x AJaTS architecture

We are currently improving an AJaTS Eclipse IDE plug-in. It integrates AJaTS main functionalities, such as refactorings definitions support, to Eclipse editor. This AJaTS implementation allows the application of its refactorings by code selection directly, using the Eclipse project explorer and the AspectJ editor provided by AJDT plug-in [6].

4. Conclusions

The elaboration of this work has shown some of AJaTS's limitations. Whereas it is clearly possible to define complex refactorings, they might require some extra processing, still not supported by the transformation engine itself. The *Extract Method Calls* [3], for example, is a well-known refactoring that involves Java code removal after its application. In order to realize it, several code comparisons are needed, which cannot be achieved with current's AJaTS version.

As a future work possibility, we propose an AJaTS improvement, which allows code analysis in a lower granularity level, to support the definition of such comparisons within the transformation templates. Another valuable contribution to this work is the implementation of a context-sensitive approach that allows the definition of much richer refactorings. Such approach is currently being developed.

Acknowledgments

We would like to thank the members of Software Productivity Group (www.cin.ufpe.br/spg) for all their technical contributions and support, in particular to Adeline Sousa. This research was partially sponsored by CNPq.

References

[1] F. Castor and P. Borba. A language for specifying Java transformations. In *V SBLP* , Brazil. May, 2001.

[2] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification. Java Series. Addison-Wesley, 2th Edition, 1996.

[3] G. Kiczales et al. Getting started with AspectJ. Communications of the ACM, 44(10):59-65, October 2001.

[4] R. Laddad. Aspect-Oriented Refactoring Series. TheServerSide.com, December 2003.

[5] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In Proceedings of OOPSLA 2002. ACM Press, 2002.

[6] AspectJ Development Tools. http://www.eclipse.org/ajdt/, 2007.

[7] Sun Microsystems. Java Remote Method Invocation (RMI).

TOWARDS A CHANGE SPECIFICATION LANGUAGE FOR API EVOLUTION

JÜRGEN REUTER AND FRANK PADBERG

Abstract. An important application of distributed refactoring is automated restructuring of client source code in response to a revised library API. However, the standard refactorings are insufficient to express all cases of API changes, mainly because refactorings by definition preserve semantics. We are developing a language of transformation rules for metaprogramming at the AST level. The transformation rules specify changes required to adapt client source code to library changes. In particular, our language will facilitate specifying new refactorings.

1. Introduction

Adapting client code to a library with a revised API is a tedious and error prone work if performed manually. Library vendors often try to leverage this work by providing informal documentation or tools for updating client code. Informal documentation is of limited help, since it does not immediately help automatizing the process of updating. Typical tools for automatic update that are bundled with revised libraries often catch only very simple cases that are covered by regular or context-free pattern matching and replacing, but fail as soon as semantic analysis is required for correct matching. For example, if in an object-oriented language a library method is renamed, scoping rules as well as method overloading must be considered in order to update only those method calls in the client code that refer to the method with the changed name.

Current implementations of refactorings offered by IDEs such as `Eclipse`[1] or NetBeans IDE[2] use the underlying compiler tools in the IDE to perform code transformations directly on the abstract syntax representation. This way, they can also utilize results of the semantic analysis of the underlying compiler. Increasingly more IDEs implement recording and replaying of refactorings to support distributed refactoring. An obvious application of distributed refactoring is to record refactorings on the library API and replay them at the client code in order to automatically update it. Henkel and Divan[3] follow this approach. They observe that many API changes can be expressed as refactorings. Dig and Johnson[4] claim that about 80% of the API changes of some real-world APIs that they examined can be expressed by refactorings. Still, we think that refactorings as viewed and implemented today are unsatisfactory to specify how to adapt client code to library changes for a number of reasons.

- API evolution may lead to semantic changes, whereas refactorings by definition preserve behavior.
- Revised libraries most often retain old API for backwards compatibility, rather than replacing obsolete code elements, as refactorings typically do.
- For a library user, the library code may be available in binary form only. Hence, the description of changes must be independent from any references to the library source.
- The set of refactorings implemented by popular IDEs is not tailored to API evolution. For example, changes must be applied to the client code only, but not to the library code.
- Internal changes to the library code that do not have a visible effect on its interface are irrelevant and must be ignored when adapting the client code.

Rather than expressing API changes by standard refactorings, we think that code restructurings should be expressed on a metaprogramming basis. The `Jackpot`[5] scripting language provides metaprogramming facilities. However, its pattern matching operates on regular expressions of flat token sequence patterns that do not consider AST node types. As a result, patterns tend to match wrong locations in the code.

Balaban, Tip and Fuhrer[6] use type constraints to check if a class can be replaced by a different

Key words and phrases. API Evolution, Automated Software Adaptation, Refactoring, Metaprogramming.

class without affecting type correctness or program behavior. Our work focuses on the actual program transformation.

Scripting languages like `JunGL`[7], that operate on a graph rather than tree representation of the program, encompass static program semantics and thus are very powerful and flexible. However, the script author has to operate on the much more complicated graph structure and manually extract static program semantics from the graph. The authors ignore object-oriented features like inheritance and visibility.

Our *transformation rules language* operates on the simpler syntax-only tree view of the program and provides static semantics through built-in functions, thus loosely following the approach of Chow and Notkin[8]. Chow and Notkin observe that semantic analysis is crucial, but they consider only a special case (resolving method overloading for a simple signature). While they operate on the *concrete* syntax, we rely on the metaprogramming framework `recoder`[9] that provides an *abstract* syntax tree (AST) view of the code. `recoder` supports full-fledged scoping and type resolution based on tree node attribution.

```
RULE swapTwoParams(MethodDecl M) {
  FOREACH LOC(MethodRef: REF(M))
    ;; for all invocations of M do
  RULE {
    (P=Expression, Q=Expression)
      ;; find params P, Q by looking
      ;; for sequence of expressions
    =>
    (Q, P)
      ;; replace (P,Q) by (Q,P)
  }
}
```

FIGURE 1. Rule for Adapting Methods in Response to Swapped Parameters

Our approach is to make `recoder`'s semantic analysis features available as functions in our rules language. For example, given a method declaration M, in our rules language we provide a locator function LOC(MethodRef : REF(M)) that will represent all syntax tree locations that have a method reference that refers to M according to `recoder`'s semantic analysis. Imagine that in the declaration of a method M with two formal parameters, the parameters have been swapped. The rule in Fig. 1 will adapt all references to this method accordingly.

Our rules language is under development and currently provides

- hierarchical node type matching, such as matching *any* expression node or the more specific expression statement nodes,
- generating collision-free identifiers for introducing new variables, methods, etc.,
- semantic checks as built-in Boolean functions, and
- semantic dereferencing as built-in locator functions to be used in FOREACH constructs.

API changes and refactorings are on the same level of abstraction, such as the example of renaming a method or moving a field shows. Therefore, we expect that our rules language will also be useful to express new refactorings in a handy way, rather than implementing each new refactoring from scratch over and over again.

REFERENCES

[1] The `Eclipse` Foundation, "The `Eclipse` project." `http://www.eclipse.org/`, 2007.

[2] Sun Microsystems, Inc., "NETBEANS IDE." `http://www.netbeans.org/`, 2007.

[3] J. Henkel and A. Diwan, "Catchup! capturing and replaying refactorings to support api evolution," in *Proceedings of the 27th international conference on Software engineering (ICSE'05)*, pp. 274–283, May 2005.

[4] D. Dig and R. Johnson, "How do APIs evolve? A story of refactoring," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006.

[5] Sun Microsystems, Inc., "NETBEANS IDE `Jackpot`." `http://jackpot.netbeans.org/`, 2007.

[6] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," *SIGPLAN Not.*, vol. 40, no. 10, pp. 265–279, 2005.

[7] M. Verbaere, R. Ettinger, and O. de Moor, "Jungl: a scripting language for refactoring," in *ICSE '06: Proceeding of the 28th international conference on Software engineering*, (New York, NY, USA), pp. 172–181, ACM Press, 2006.

[8] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," in *Proceedings of the 1996 International Conference on Software Maintenance (ICSM '96)*, (Washington, DC, USA), pp. 359–368, IEEE Computer Society, 1996.

[9] A. Ludwig, *Automatische Transformation großer Softwaresysteme*. Dissertation, Universität Karlsruhe (TH), Fakultät für Informatik, Karlsruhe, Germany, Dec. 2002.

IPD, UNIVERSITÄT KARLSRUHE, GERMANY
E-mail address: {`reuter,padberg`}`@ipd.uka.de`

Holistic Semi-Automated Software Refactoring

Erica Mealy
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia, Qld 4072, Australia
{erica}@itee.uq.edu.au

Abstract

Post-deployment maintenance and evolution can account for up to 75% of the cost of developing a software system. Software refactoring can reduce the costs associated with evolution by improving system quality, but although refactoring can yield benefits, the process includes potentially complex, error-prone, tedious and time-consuming tasks. It is these tasks that automated refactoring tools seek to address. However, although the refactoring process is well-defined, current refactoring tools do not support the full process. In this paper we present a study of refactoring in terms of automation and situation awareness in order to propose an ideal allocation of tasks between the user and automated refactoring support environments. This allocation defines an ideal level of automation designed to remove from the user unnecessary and undesirable processing tasks whilst still maintaining the user's understanding of the system they are refactoring. The developed ideal was compared to four sample refactoring support tools to identify where improvement is needed to better support users.

1 Introduction

Software refactoring is the process of internal improvement of software without change to externally observable behaviour [3, 4]. Tourwé and Mens [6] identify three phases associated with the process of refactoring:

1. Identification of when an application should be refactored (code-smells).
2. Proposal of which refactorings could be applied where.
3. Application of the selected refactorings.

Software refactoring presents several challenges for computer-based tool support. During refactoring, users must synthesize and analyse large collections of data (code) to identify inappropriate or undesirable features (such as duplicated code), propose solutions to discovered issues, and perform potentially complex, error-prone and tedious transformations to rid their systems of these undesirable features. Throughout the process, the user must maintain the existing behaviour of their system in addition to maintaining or improving their own understanding (mental model) of the system being refactored. This requires that the level of automation the tool exhibits must also take care not to negatively affect the user's understanding. Through the application of research and theory from situation awareness [1, 7] and function/task allocation[5, 2], we aim to improve the quality of support provided for software refactoring.

2 Automation

Automation is often introduced to aid users by reducing effort and cognitive load [1, 7]. In the context of software refactoring, automation allows the ability to introduce more thorough, complex or subtle code-smell detection mechanisms, better matching of refactoring transformations to code-smell instances, and more thorough and correct refactoring transformations.

Although automation can bring benefits, the addition or increase of automation in any system has inherent problems including mistrust (reliability and calibration) and over-trust (complacency) [1, 7]. For complicated applications, in which computer-based support is used to assist the user operating on and understanding a system, over-automation can have the further problem of interfering with the user's understanding [1]. Thus, the addition or increase of automation in a refactoring tool requires careful analysis and consideration.

Sheridan [5] identifies eight levels of automation (presented in Figure 1) which can assist in classifying the level of automation a process exhibits. Sheridan identifies four general stages in which automation can occur: 'acquire', 'analyse', 'decide' and 'implement'. For the three-stage process of software refactoring, code-smell detection maps to the acquire and analyse stages, the proposal of the appropriate refactoring transformations maps to the analyse and decide stages, and the application of refactoring transformations maps to the implementation stage.

3 Situation Awareness

A user's *situation awareness* is defined in terms of their perception and comprehension of a system's state, and the projection to future states and actions relevant to the completion of a particular task [1]. For refactoring, situation awareness applies to the developer's understanding of the system, including its structure and conventions, and projections of how actions will affect their ability to maintain and re-design the system in the future. Endsley [1] argues that to minimise the negative effects from the introduction

1. Computer offers no assistance: human must do it all.
2. Computer suggests alternative ways to do the task.
3. Computer selects one way to do the task.
4. Computer selects one way to do the task, and executes that suggestion if the human approves.
5. Computer selects one way to do the task, and executes that suggestion if the human approves, or allows the human a restricted time to veto before automatic execution.
6. Computer selects one way to do the task, and executes automatically, then informs the human.
7. Computer selects one way to do the task, and executes automatically, then informs the human only if asked.
8. Computer selects, acts, and ignores the human.

Figure 1. Sheridan's levels of automation [5]

of automation, systems should be designed to maximise user involvement whilst reducing the load that would have been associated with doing the task manually.

4 Ideal Automation for Refactoring

In order to appropriately allocate the sub-tasks of software refactoring between computers (automated) and users (manual), we have used Fitts' MABA-MABA (Men [sic] are better at-Machines are better at) List [2]. The items from Fitts' List relevant to software refactoring are for users *Reasoning inductively* (generalisation) and *Exercising judgement*, and for automation *Reasoning deductively* (specialisation).

Code-smell detection requires both inductive and deductive reasoning and cannot by wholly automated as not all code-smells can be quantified. The proposal and selection of appropriate refactoring transformations to remedy code-smells similarly requires both inductive and deductive reasoning. When considered in context, the partial automation of the proposal stage leads to the integration of the whole refactoring process within a single tool. The software refactoring task that requires judgement is the decision on what refactoring transformations will be applied to remedy an identified code-smell instance. This need for judgement is due to the subjective nature of the goal of refactoring. Using Sheridan's levels, the ideal level of automation for refactoring is 6-6-2-4 (Figure 2).

5 Existing Automation in Refactoring tools

To identify areas in which current automation is not ideal, we studied four sample refactoring tools to ascertain their levels of automation. The tools selected were: Eclipse 3.2, Condenser 1.05, RefactorIT 2.5.1, and Eclipse 3.2 with the Simian UI 2.2.12 plugin. These tools were selected as representative of available automated refactoring tools, with the inclusion of Eclipse (open source) and RefactorIT (commercial) due to their reputation as premiere refactoring transformation tools, and Condenser (command-line) and Simian (GUI) as representative of code-smell detection tools.

The graph in Figure 2 shows the levels of automation for the four refactoring tools studied compared to the proposed ideal level presented in Section 4. From this graph it can be seen that none of the studied tools provided the ideal

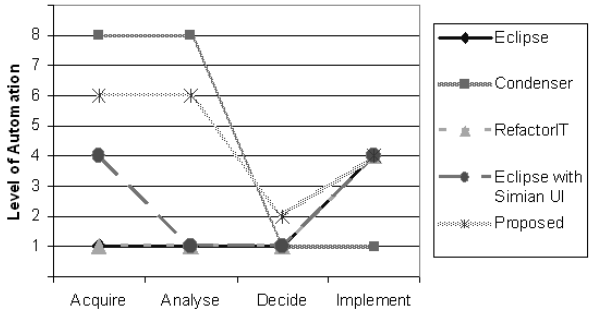

Figure 2. Automation of refactoring tools

level of automation. The tools exhibiting automation closest to the ideal are Eclipse with the Simian UI plugin and Condenser, however these tools still over-automated and under-automated key parts of the refactoring process. Importantly, none of the tools studied attempted to automate the proposal stage, and as such none of the tools exhibited a holistic approach to supporting the refactoring process.

6 Conclusion

This paper presented the case for a holistic approach to automating software refactoring. This approach was designed to balance automation with user involvement to meet the aim of removing the burden of complex, error-prone, tedious and time-consuming tasks, whilst still supporting user involvement and allowing maximum program understanding and situation awareness. An ideal level of automation was presented, and when compared against automation levels found in existing refactoring tools, it was found that there is still work required to produce an ideal automated software refactoring support environment.

References

[1] M. Endsley. Automation and situation awareness. In R. Parasuraman and M. Mouloua, editors, *Automation and human performance: Theory and applications*, pages 163–181, 1996.

[2] P. Fitts. Human engineering for an effective air navigation and traffic control system. Technical report, Ohio state University Foundation Report, Columbus, OH, 1951.

[3] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[4] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Department of Computer Science, 1992.

[5] T. Sheridan. Function allocation: Algorithm, alchemy or apostasy? *International Journal of Human-Computer Studies*, 52(2):203–216, 2000.

[6] T. Tourwé and T. Mens. Identifying refactoring opportunities using logic meta programming. In *Proceedings of 7th European Conference on Software Maintenance and Reengineering*, pages 91–100. IEEE Computer Society, 2003.

[7] C. Wickens, S. Gordon, and Y. Liu. *An Introduction to Human Factors Engineering*. Addison-Wesley Longman, 1998.

Engineering usability for software refactoring tools

Erica Mealy
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia, Qld 4072, Australia
{erica}@itee.uq.edu.au

Abstract

The goal of refactoring tools is to support the user in improving the internal structure of code whilst maintaining its existing behaviour. As a human-in-the-loop process (i.e. one that is centered around a user performing a task), refactoring support tools must aim to meet high standards of usability. In this paper we present an initial usability study of software refactoring tools. During the study, we analysed the task of software refactoring using the ISO 9241-11 standard for usability. Expanding on this analysis, we reviewed 11 collections of usability guidelines and combined these into a single list of 34 guidelines. From this list and the definition of refactoring, we developed 81 usability requirements for refactoring tools. Using these requirements, four sample refactoring tools were studied to analyse the state-of-the-art for usability of refactoring tools. Finally, we have identified areas in which further work is required.

1 Introduction

Software refactoring is a software development process designed to reduce the time and costs associated with software development and evolution. Refactoring is defined as the process of internal improvement of software without change to externally observable behaviour [1, 5].

Usability of software refactoring tools is an area in which little research has been performed. In general, the production of and research into software development tools often overlooks the issue of usability [6]. In the area of refactoring, usability related research has focused on understandability of error messages and assisting the user in the selection of text with a mouse cursor prior to the application of an automated refactoring transformation [3]. We believe that there are more important aspects affecting the usability of existing refactoring tools and we wish to identify and address these issues. To identify areas in which usability can be improved, we have developed usability requirements for refactoring tools and have used these requirements to analyse the state-of-the-art for refactoring tools. To achieve this, we sought usability design guidelines which could yield a set of usability requirements when combined with a definition of the process of refactoring using the ISO 9241-11 interface design standard [2].

In general, the process of usability engineering consists of the identification of the intended user group for the tool, and the tailoring of the tool's design specifically for that user group. The implementation of estabilished standards, norms (etc.), both general and domain-specific, that make human-computer interaction more efficient, productive and desirable, are also included in 'designing for the intended user group'. Nielsen [4] defines the *Usability Engineering Life Cycle* as a framework for more consistently and formally addressing the issue of designing or engineering for usability. In this paper we will present the results of the application of the pre-design and design phases of this life cycle.

2 Defining Refactoring for usability

To design refactoring support with maximum usability, we used the ISO 9241-11 specification as a framework to define the process of refactoring in terms of goals, users, tasks, environment and equipment (which maps to stage 1 of Nielsen's Usability Engineering Life Cycle). For example, the goals of refactoring tools are defined as 'assisting a software developer to perform software refactoring in the most efficient and effective means possible', and 'not hindering the developer's ability to understand and reason about the software system being refactored and developed'. This specification of refactoring is reusable and is available online[1].

3 Usability guidelines

Guidelines have been used for both the design and evaluation of user interfaces since the early 1970s. In looking at usability guidelines, we found many different sets of guidelines, rules, heuristics, maxims, etc., yet no one set was complete. During our study we collected 126 guidelines from 13 sources from 1971 to 2000. To manage the number of individual guidelines, we collated and

[1] http://www.itee.uq.edu.au/ erica

categorised the lists based on fundamental groupings that were evident across the initial 13 sources. Duplicate guidelines and those addressing a similar or closely-related concept became more prevalent as the categorised list became larger. We distilled the categorised list of 126 initial guidelines into 34 to provide a more usable list.

The results of this study of usability guidelines yielded a single, published list of guidelines (available online[1]) that are applicable to not just the development of software refactoring tools, but also general software systems. These guidelines are particularly useful for application in Usability Engineering Life Cycle stage 'Guidelines and Heuristic Analysis'.

4 Usability requirements

To improve the usability of software refactoring tools, we developed a set of usability requirements. These requirements can be used in the design of new software refactoring tool support as well as to evaluate existing tools to identify issues and improve usability in subsequent iterations. These requirements were developed through a process of refinement using the 34 distilled usability guidelines and the definition of refactoring using ISO 9421:11. This process yeilded 81 usability requirements which are available online [1]. An example of the refinement of a guideline into a requirement using the ISO 9241-11 specification of refactoring is Requirement 3 "Make refactoring tool interface work and look same as code editors and related tools". This requirement was derived from guideline C1 "Ensure things that look the same act the same and things that look different act different" and the refactoring ISO 9241-11 equipment specification of a software development environment. A similar requirement is derived mandating the use of operating system standards and norms.

5 Usability Analysis

To analyse the current level of usability in existing refactoring tools, we evaluated four refactoring tools using the 81 usability requirements we developed. The tools evaluated were: Eclipse 3.2, Condenser 1.05, RefactorIT 2.5.1, and Eclipse 3.2 with the Simian UI 2.2.12 plugin. These tools were selected as representative of available automated refactoring tools, with the inclusion of Eclipse (open source) and RefactorIT (commercial) due to their reputation as premiere refactoring transformation tools, and Condenser (command-line) and Simian (GUI) as representative of code-smell detection tools. This study aimed not to focus on particular issues exhibited by these tools, but to instead identify trends across the tools that would allow us to determine usability issues requiring further work.

Overall, our evaluation found that there is much work to be done on the usability of refactoring tools. The area in which tools performed best was consistency with existing operating system and environment standards. The requirements that the tools performed most poorly on were related to user control, i.e. the ability to define new, and modify or delete existing code-smells, code-smell-to-transformation proposals and transformations. Providing feedback about code-smell instances to the user within the user's regular development view and the integration of support for the whole refactoring process are other areas identified by the evaluation. Another area is the user's control over the level of automatic investigation, i.e., whether refactoring tools act reactively (i.e. only when the user instructs) or actively (such as with incremental compilation) which is vurrently is not supported by any of the studied tools. It is feedback to the user, integration of the stages of refactoring and the level of automatic investigation that is the focus of our research.

6 Conclusion

This paper has presented a summary of three contributions to the area of usability of refactoring tools. The first contribution is a set of collated and distilled usability guidelines to aid in the development of usable software tools in a general, as well as refactoring tool context. The second contribution is a set of 81 usability requirements for software refactoring environments. The final contribution is a usability analysis of four existing refactoring tools, baselining the level of usability that exists for refactoring tools. This analysis has identified areas in which further work is necessary to develop better and more usable software refactoring tools.

References

[1] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] International Standards Organisation & International Electrotechnical Commission, Geneve, Switzerland. *International Standard ISO 9241-11 Ergonomic requirements for office work with visual display terminals (VDTs) Part 11: Guidance on Usability*, 1998.

[3] E. Murphy-Hill. Improving refactoring with alternate program views. Technical Report TR-06-05, Portland State University, Department of Computer Science, 2006.

[4] J. Nielsen. The usability engineering life cycle. *IEEE Computer*, 25(3):12–22, 1992.

[5] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Department of Computer Science, 1992.

[6] M. Toleman and J. Welsh. Systematic evaluation of design choices for software development tools. *Software – Concepts and Tools*, 19:109–121, 1998.

Automated Testing of Eclipse and NetBeans Refactoring Tools*

Brett Daniel Danny Dig Kely Garcia Darko Marinov
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{bdaniel3, dig, kgarcia2, marinov}@cs.uiuc.edu

ABSTRACT

This position paper presents our experience in automated testing of Eclipse and NetBeans refactoring tools. Test inputs for these tools are Java programs. We have developed ASTGen, a framework for automated generation of abstract syntax trees (ASTs) that represent Java programs. ASTGen allows developers to write *imperative generators* whose executions produce ASTs. ASTGen offers a library of generic, reusable, and composable generators that make it relatively easy to build more complex generators. We have developed about a dozen of complex generators and applied them to test at least six refactorings in each tool. So far, we have found 28 unique, new bugs and reported them, 13 in Eclipse Bugzilla and 15 in NetBeans Issuezilla. This is ongoing work, and the numbers are increasing.

We advocate the importance of automated testing—not only automated execution of manually written tests (using JUnit or XTest) but also *automated generation of test inputs*. We have developed several oracles that programmatically check whether a refactoring tool correctly made some program transformations (or gave warning that a specific refactoring should not apply to the given input program).

We hope that this paper motivates developers of refactoring tools to incorporate such generation and oracles into their tools. While most refactoring tools are already quite reliable, we believe that the use of such generation would further increase reliability, to the benefit of all users of refactoring tools. Moreover, we argue that such generation can be useful for testing other related tools that take (Java) programs as inputs. To encourage collaboration and enable others to try out ASTGen, we have made our ASTGen code and all experimental results publicly available at the ASTGen web page, `http://mir.cs.uiuc.edu/astgen`

1. WHY AUTOMATED GENERATION?

Testing involves several activities, including generation of test inputs (and expected outputs), execution of test inputs, and checking of obtained outputs. For a refactoring tool, each input consists of a program and a refactoring to apply, and each output is either a refactored program or a warning if the specific transformation might change the program's semantics.

*This paper is based on the work [1] to be presented at the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007) in Dubrovnik, Croatia.

It is often said that manual testing is tedious and error-prone. Indeed, developers of refactoring tools automate a large portion of testing. For instance, we have counted 2,673 JUnit tests for the major refactorings in Eclipse version 3.2. JUnit automatically executes these tests and checks the obtained outputs. However, JUnit does not automatically generate test inputs, and to the best of our knowledge, Eclipse developers manually wrote their JUnit tests.

Automated generation of test inputs has one significant benefit: it makes it easier to generate a large number of test inputs, which hopefully results in a more thorough testing and enables finding bugs before they are encountered in production runs. However, automated generation of test inputs, especially for refactoring tools, poses several challenges. We discuss these challenges and our solution.

1.1 Generation of Input Programs

How does one automatically generate valid Java programs to give as inputs to a refactoring tool? There is no obvious answer. While simpler test inputs, say one integer or a sequence of integers, can be generated even randomly, it is unclear how one could randomly generate sequence of characters (or abstract syntax trees) that satisfy the syntactic and semantic constraints for a valid Java program. Moreover, even if one could generate programs randomly, how would one ensure that these programs have properties relevant to the refactoring under test?

We have developed the ASTGen framework to generate a large number of relevant Java programs. ASTGen is not fully automatic. It requires that the developer write a class (which we call *generator*) that can produce test inputs relevant to a specific refactoring. ASTGen provides a library for generation of simple AST nodes. This library makes it easy to build more complex combinations of AST nodes. We have written several generators that produce Java programs for testing refactoring engines. More details are in the conference paper [1]. We point out that the generators do not always produce programs that compile. (The column "CI" in Figure 1 shows how many of the generated inputs compile and are thus valid inputs for a refactoring.)

1.2 Execution of Refactorings

How does one automatically run a refactoring tool on the automatically generated input programs? This is seemingly an easy task: just develop a piece of code that (efficiently) runs a specific refactoring on each of the generated programs. However, we have encountered a number of problems while developing this piece of code, in both Eclipse and

Refactoring	Generation: Generator	TGI	Time [min:sec]	CI	Oracles: WS Ecl	WS NB	DNC Ecl	DNC NB	C/I	Diff	**Bug Reports** Ecl	**Bug Reports** NB
Rename(Method)	MethodReference	9540	89:12	9540	3816	0	0	0	0	5724	0	0
Rename(Field)	FieldReference	3960	28:20	1512	0	0	0	304	0	40	0	1
EncapsulateField	ClassArrayField	72	0:45	72	0	0	48	0	0	48	1	0
	FieldReference	3960	15:19	1512	0	0	320	432	14	121	4	3
	DualClassFieldRef.	14850	41:45	3969	0	0	187	256	100	511	1	2
	DualClassGetterSetter	576	8:45	417	216	0	162	162	18	216	2	2
PushDownField	DualClassFieldRef.	4635	10:56	1064	760	380	152	228	0	380	2	2
	DualClassParentDecl.	360	6:50	270	246	168	18	90	0	78	1	2
PullUpField	DualClassChildDecl.	60	1:14	44	0	18	10	6	0	44	1	1
MemberToTop	ClassRelationships	70	0:36	51	0	0	0	2	0	2	0	1
	DualClassFieldRef.	6600	29:04	2824	0	0	353	507	0	2824	1	1
										Total Bugs:	**13**	**15**

Figure 1: Refactorings tested and bugs reported, Ecl = Eclipse, NB = NetBeans TGI = Total Generated Inputs, Time in [min:sec], CI = Compilable Inputs, WS = WarningStatus, DNC = DoesNotCompile, C/I = Custom/Inverse, Diff. = Differential

NetBeans. These problems have been partly due to certain design decisions in these refactoring tools.

Two key problems that we encountered were (1) how to reduce the dependency of the refactoring under test from the rest of the IDE and (2) how to efficiently execute the refactorings. We still have not solved the first problem satisfactorily in Eclipse. Namely, our testing of refactorings requires that we run Eclipse in the GUI mode, which not only slows down the execution but also disallows using (fast) servers with a text-only connection. We still have not solved the second problem satisfactorily in NetBeans. Namely, our testing does not release all the resources after each refactoring. (Specifically, it creates a new project for each input program.) This results in an increasing memory usage over time and requires that we rerun NetBeans several times, splitting a large number of input programs into several smaller batches that can each fit into one run. We hope that developers of refactoring tools can provide better "hooks" for running automatically generated inputs programs.

1.3 Checking of Outputs

How does one automatically check the outputs that a refactoring tool produces for the automatically generated inputs? While this problem is related to checking correctness of compilers [2] (the output program should be semantically equivalent to the input program), in addition, the refactored program should have the intended changes.

We have developed a variety of oracles for programmatic checking of refactoring tools. The simplest oracle checks that the refactoring tool does not throw an uncaught exception, but we have not found such a case in either Eclipse or NetBeans. The WarningStatus (WS) oracle checks whether the tool produces a warning or a refactored program. The DoesNotCompile (DNC) oracle checks whether the refactored program compiles. The Custom/Invertible (C/I) oracle checks specific structural properties (e.g., moving an entity should indeed create the entity in the new location) or invertibilty (e.g., renaming an entity from A to B and then from B to A should produce the same starting input program). The Differential (Diff) oracle [2] gives the same input program to both Eclipse and NetBeans and compares whether they produce the same output.

1.4 Experimental Results

When is testing with automatically generated inputs applicable? There are at least two benefits of manually written tests. First, in test-driven development, the tests are written even before writing the code, and thus such tests help in designing the code. Second, in regression testing, when developers want to get a quick feedback about the code changes they are making, it is better to use a smaller number of tests manually written (or previously manually selected from some automatically generated tests) than to use a large number of automatically generated tests. However, we claim that when developers can run tests for longer time or want to exercise their code more thoroughly, it is appropriate to use automatically generated tests (in addition to manually written tests).

Figure 1 shows some of our experimental results that support the above claim. (The full results are available in the conference paper [1].) The "Time" column shows the total time required to generate the input programs and to run them in Eclipse. Running is over an order of magnitude slower than generation. Testing each refactoring takes less than an hour and a half (on a a dual-processor 1.8 Ghz machine), and the entire suite can be run overnight. The benefit is finding new bugs, as shown by a total of 28 new bugs in Eclipse and NetBeans.

2. BEYOND REFACTORING TOOLS

We believe that automated testing based on ASTGen generators is useful beyond refactoring tools. In principle, any tool that operates on programs (or abstract syntax trees) could benefit from ASTGen. The main question is how easy/hard it is to write the generators that produce interesting programs that satisfy the required constraints. We plan to investigate this in new application domains, e.g., in other parts of Eclipse and NetBeans IDEs.

3. REFERENCES

[1] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *ESEC/FSE 2007*, Dubrovnik, Croatia, Sept. 2007. (To appear.).

[2] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1), 1998.

Refactoring in Erlang, a Dynamic Functional Language*

László Lövei, Zoltán Horváth, Tamás Kozsik, Roland Király,
Anikó Víg, and Tamás Nagy
Eötvös Loránd University, Budapest, Hungary

Abstract

Refactoring in object-oriented languages has been well studied, but functional languages have received much less attention. This paper presents our ideas about refactoring in Erlang, a functional programming language developed by Ericsson for building telecommunications systems. The highlights of our work is dealing with the strong dynamic nature of Erlang and doing program manipulations using a relational database.

1 The Erlang programming language

Erlang/OTP [1] is a functional programming language and environment developed by Ericsson, designed for building concurrent and distributed fault-tolerant systems with soft real-time characteristics (like telecommunication systems). The core Erlang language consists of simple functional constructs extended with message passing to handle concurrency, and OTP is a set of design principles and libraries that supports building fault-tolerant systems.

Erlang is a functional language which means that a program is run by successively applying functions. Branches of execution are selected based on pattern matching of data and conditional expressions, and loops are constructed using recursive functions. Variables are bound a value only once in their life, they cannot be modified. Most constructs are side effect free, exceptions are message passing and built-in functions (BIFs).

*Supported by GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK, Ericsson Hungary, ELTE CNL, and OMAA-ÖAU 66öu2.

The speciality of Erlang is its strong dynamic nature. Variables are dynamically typed, there is no compile time type checking. The identifiers of functions are of a special data type called *atom* and they can be generated at run-time and passed around in variables. Execution threads are also created at run time, and they are identified by a dynamic system.

The challenge in building an Erlang refactoring tool is to cover as wide area of language constructs as possible by static (compile-time) analysis, and to identify the exact conditions when we can guarantee behaviour-preserving transformations.

2 Refactoring in Erlang

While refactoring in object-oriented languages has been well studied [3], functional languages have received much less attention, and most work is oriented towards pure functional languages with a strict type system. Our work has been focused on those refactorings that are applicable in Erlang as well and help us to develop a framework that makes implementation of other refactorings easy.

2.1 Transforming expressions

Expressions are the basic building blocks of functional programs, and many of the refactorings move, restructure, or modify expressions. We found that to preserve the behaviour of an expression, the most important thing is to maintain the binding structure of its variables. We defined the binding structure using the concepts of variable scope and visibility. Another expression-related concept is whether an expression is side effect-free.

We have studied the *rename variable* and *extract function* refactorings that use only these concepts.

2.2 Tracking function references

The most frequently used expression is the function application, and refactorings that transform a function call, must transform the function definition accordingly. Unfortunately, finding the relation between function calls and function definitions is not always possible by static analysis. Remember that the identifier of a function is really a data tag. Most function calls include this tag as a constant, but it is possible to create the tag at run-time, and there are built-in functions that call a function with an argument list constructed at run-time.

We classified these constructs as directly supported (e.g. constant name and static argument list), partially supported (e.g. static name and dynamic argument list) and not supported (e.g. name read from standard input) calls, and plan to cover a broader range using data flow analysis (e.g. function name stored in a variable, or lambda expressions).

Refactorings that use only this kind of information are *rename function* and *reorder function arguments*, and *generalisation* needs the binding structure and function reference tracking as well.

2.3 Restructuring data types

Erlang has no static type information attached to variables, but types exist in the language, and they are strictly checked at run-time. Available compound types are lists, tuples, and records, these can be used to build more complex data structures. Sometimes the transformation of such a data structure is desired, but it is hard to describe what changes are to be made, and usually data flow analysis is required to find the expressions that manipulate the data.

Our most recent work is the analysis of such a transformation, when a record is introduced to store the elements of a tuple. This refactoring transforms the expressions that work with the same (or slightly modified) tuple, and these expressions can be found by a kind of data flow analysis. Tracking the way of a piece of data is easy when there are no side effects, the complicating factors are function references and constructs where more than one type of data is handled.

A simpler refactoring on data structures we dealt with is *tuple function arguments*.

3 Implementation

Our approach to refactoring is that we express the side conditions and code transformations by graph manipulation. We build a semantic graph starting from the abstract syntax tree of the source code and extending it with edges that represent semantic relations between nodes. Semantic concepts like variable scoping or function references are encapsulated into the graph this way.

A working prototype software is built using these concepts, written in Erlang. Building on previous experiences with Clean refactoring [2], we decided to represent the semantic graph in a relational database, and use SQL to describe the manipulations. Every node type has a table that contains the attributes of the nodes and the links to other nodes (represented by their unique ID). A nice feature of this representation is that fixed length graph traversals can be expressed by joining tables.

Refactoring Erlang programs is a joint research with the University of Kent, building on experiences with Haskell and Clean. While we are sharing ideas and experiences, they are investigating a completely different implementation approach using traversals on annotated abstract syntax trees [4].

References

[1] J. Armstrong, R. Virding, M. Williams, and C. Wikstrom. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[2] P. Diviánszky, R. Szabó-Nacsa, and Z. Horváth. Refactoring via database representation. In *The Sixth International Conference on Applied Informatics (ICAI 2004)*, volume 1, pages 129–135, Eger, Hungary, 2004.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[4] H. Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg, and T. Nagy. Refactoring Erlang programs. In *The Proceedings of 12th International Erlang/OTP User Conference*, Stockholm, Sweden, November 2006.

Operation-based Merging of Development Histories

Tammo Freese
Department of Computing Science
University of Oldenburg
Germany
tammo.freese@informatik.uni-oldenburg.de

ABSTRACT

In development teams, refactorings can lead to merge problems if text-based merging algorithms are used. This paper describes a system which aims at reducing these problems by an operation-based merging of development histories captured in the development environments. In the case of library development, the merged development histories can later be used for automated migration of the libraries' clients.

1. INTRODUCTION

In development teams, changes are typically applied to local copies of the source code. Other workspaces remain unaffected by refactorings applied locally.

Text-based merging of the changes in different workspaces can lead to problems when refactorings have been applied: refactorings in one workspace may conflict with changes in the other, or they may not make some necessary changes, as they are not re-applied to the code of the other workspace.

This paper describes a system which aims at reducing the merge problems by using development histories, i.e. the sequence of edits and refactorings from the personal workspaces, as the foundation for an operation-based merging.

2. CAPTURING DEVELOPMENT HISTORIES

To capture a development history, the IDE needs to be extended so that changes to the program, refactoring information and compile results are captured.

The recorded data is then compacted by removing refactoring information for refactorings which start or end at a non-compilable state and summarizing subsequent edit steps to one edit step. With the additional requirement that the code compiles at begin and end of the capturing, the result is a sequence of edit and refactoring steps/operations, where the code compiles before and after each of them. We call this sequence a development history.

3. MERGING DEVELOPMENT HISTORIES

In merging we have a program p, a local development history $L = l_1, \ldots, l_n$ which changes p to $p_L = (l_1; \ldots; l_n)(p)$ and a development history $R = r_1, \ldots, r_m$ from the version control repository which changes p to $p_R = (r_1; \ldots; r_m)(p)$. The merge result should be adapted steps $L' = l'_1, \ldots, l'_k$ and $R' = r'_1, \ldots, r'_s$ so that applying the adapted local steps to the program from the repository results in the same program as applying the adapted repository steps to the local program: $(l'_1; \ldots; l'_k)(p_R) = (r'_1; \ldots; r'_s)(p_L)$.

To merge the development histories, the first local and the first repository step are merged, resulting in adapted local and repository steps. Then the adapted repository step(s) are merged with the second local step. This is repeated for all local steps, and then the whole process is repeated for all repository steps.

4. MERGING TWO OPERATIONS

Two operations l and r commute on program p iff $p' = (l; r)(p) = (r; l)(p)$ and p' compiles, otherwise they conflict. When merging two operations, it is in many cases possible to rule out a conflict, or to detect and resolve conflicts without inspecting the program source.

Refactorings only conflict if one invalidates the preconditions of the other, otherwise, they can be applied in any order. As an example, the refactorings *rename method `a` to `b` in class `X`* and *rename class `Y` to `Z`* commute.

An edit operation σ conflicts with a refactoring operation ρ if it either invalidates ρ's preconditions, or changes the places in the code which ρ changes. Again, conflicts can often be ruled out. For example, *rename class `A` to `B`* and an edit operation that changes class `C` which does not include any reference to the names `A` and `B` commute.

Two edit operations obviously conflict if they change the same places in the code. But even when changes are applied to different files, conflicts can occur, such as one developer making method `a()` private, while the other introduced a call of method `a()` in another class. Ruling out such conflicts without needing to compile the merge result involves an analysis of the added, removed and changed names in both edits.

5. RELATED WORK

The system described here is based on the author's earlier work [4, 5], which uses the idea of operation-based merging. The author is aware of two other systems using refactoring and edit operation in merging. Both operation-based merging and the two other systems use different approaches to merging than presented here.

5.1 Operation-based Merging

In 1992, Lippe and van Oosterom described *operation-based merging* of two sequences of transformations which are applied to a snapshot of an object management system (OMS) [7, 6].

Their merging approach mixes the transformations from both sequences to a sequence of blocks. Only blocks may contain conflicting transformations. Conflicts are resolved by ordering, editing or deleting transformations. The merge result is retrieved by applying the transformations from each block to the snapshot.

To build the blocks, the merge algorithm relies on global commutation information which is retrieved using unique object identifiers. In the context of this research, unique identifiers are not available, so the merge algorithm cannot be applied here.

Furthermore, the algorithm proposed by Lippe and van Oosterom gives no hint how the merge result is included in the OMS so that subsequent mergings work. Assume that developer A checked out version 3 and developer B checked out version 4. The current version is 6. Developer A merges his local changes with the changes from version 3 to version 6, generating the new version 7. However, as the transformations are mixed, and some may be edited and deleted, there is no sequence of steps from version 4 to version 7 which developer B may use to merge his changes.

5.2 Refactoring-Aware Versioning

Ekman and Asklund presented *refactoring-aware versioning* [3]. They built a version control system which stores the abstract syntax tree (AST) of the program. Each of the AST's nodes is identified by a unique id, so that refactorings can be described independent to parallel, non-conflicting refactorings.

Refactorings and edits are recorded by an extension to the Eclipse IDE [2]. In merging, refactorings and edits are reordered: edits are applied first, then the refactorings. However, this cannot work in all cases. As an example, one developer may have inlined method `a` and then edited of the places where the method has been inlined, while the other developer changed the body of `a`. Applying the inline method refactoring last would result in a successful merge, but there should be a conflict, since the code which has been edited locally is not the code that is inlined when the refactoring is applied last.

As in operation-based merging, it is not shown how subsequent mergings can be applied to the merge result.

5.3 MolhadoRef

Dig et.al. presented *MolhadoRef*, a refactoring-aware configuration management for object-oriented programs [1]. As Ekman and Asklund's refactoring aware-merging, MolhadoRef integrates in the Eclipse IDE [2]. For merging, only refactorings are recorded.

The input for the merge algorithm consist of the base version of the program, two changed versions, and the refactorings which were applied while creating each of the changed versions. API and code edits are detected by 3-way differencing of the three program versions (step #1). Then, conflicts of API edits and refactorings are detected and resolved by the user—either by editing or by deleting a refactoring (step #2). After that, the refactorings are inverted and applied to the changed programs (step #3), so that in the next step, the changes in the two versions can be merged textually (step #4). The last step reorders the refactorings and applies them to the textual merge result (step #5).

Compared to the merge algorithm presented in this paper, MolhadoRef's should be much faster, as it does not merge each operation combination. However, undoing the refactorings may lead to unnecessary conflicts or problems. As an example, assume that one developer edited method `a`, while the other renamed it to `b` and introduced a new method `a` that delegates to `b`. In this situation, one would expect that the merge result is to simply apply the changes of the second developer to the changes of the first.

According to MolhadoRef's description, the refactoring will either be removed/edited in step #2, or it will be inverted in step #3. However, the refactoring cannot be inverted, as a method with the old name `a` already exists, and removing/editing refactoring would mean that the merge result is not the expected one.

6. CURRENT STATUS

A prototype of the system is under development for the Eclipse IDE [2] and the Subversion version control [8]. At the time of writing, capturing the development history works, and the basic merge algorithm is implemented. Some mergers for two operations are realized as well.

7. REFERENCES

[1] D. Dig, K. Manzoor, R. Johnson, and T. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Proceedings of ICSE '07*, 2007.

[2] Eclipse. http://www.eclipse.org/.

[3] T. Ekman and U. Asklund. Refactoring-aware versioning in Eclipse. *Electronic Notes of Theorical Computer Science*, 107, 2004.

[4] T. Freese. Towards software configuration management for test-driven development. In *Software Configuration Management ICSE Workshops SCM 2001 and SCM 2003*, volume 2649 of *LNCS*, 2003.

[5] T. Freese. Refactoring-aware version control: Towards refactoring support in api evolution and team development. In *Proceedings of ICSE '06*, pages 953–956, 2006.

[6] E. Lippe. *CAMERA – Support for distributed cooperative work*. PhD thesis, Utrecht University, 1992.

[7] E. Lippe and N. van Oosterom. Operation-based merging. *Software Engineering Notes*, 17(5):78–87, 1992.

[8] Subversion. http://subversion.tigris.org/.

Improving Method Extraction: A Novel Approach to Data Flow Analysis Using Boolean Flags and Expressions

1st Workshop on Refactoring Tools, Berlin, 2007

Nicolas Juillerat, Béat Hirsbrunner

Pervasive and Artificial Intelligence Research Group,
University of Fribourg, Switzerland
nicolas.juillerat@unifr.ch, beat.hirsbrunner@unifr.ch

1. Motivation

Method extraction is a complex refactoring: it has to transform statements of source code and preserve semantics. While many recent development environments provide a semi-automated implementation of this refactoring, most of them are still flawed. The part of the process we want to discuss is the so called *data flow analysis*. Its purpose is to *determine the arguments and the results* of the method to extract.

Various approaches to this problem have been proposed: graph based, scripting languages, and direct analyses.

Graph-based approaches [1] are the most widely spread techniques. The problem with graph-based approaches is that the construction of the graph itself from the source code is difficult: the common representation of statements is not a graph, but the Abstract Syntax Tree (AST). While these approaches have shown to be good for analyses, their use in transformation is usually limited to the class graph which does not model statements. Their use for method extraction is hence limited.

Scripting languages [2] have been used to express program transformations in a way that is simpler than an implementation in a language such as Java. While they can provide a high-level and simple description of various refactorings, these languages are hiding the actual implementation, for which they do not provide any help.

A look on the source code of Eclipse revealed that the implementation of method extraction was *neither* based on graphs, *nor* on scripting languages. It is just a "plain" implementation based on the AST.

Why does Eclipse use a "plain" implementation when high level formalisms are available? We give a partial answer to this question, by proposing a novel and simpler formalism for the problem of data flow analysis. Our approach differs from previous work in two aspects. First, it only uses very simple data structures and algorithms: tables of flags and boolean expressions. Second, it is based on the common representation of the source code: the AST.

2. Our New Approach to Data Flow Analysis

The idea behind our approach is to maintain, for each local variable, a table of boolean flags. Each flag captures some information about the variable, which is potentially relevant to identify arguments and results. More precisely, we define four *facts* and three *regions*, yielding 12 different flags per variable.

The four *facts* are: whether the variable is read (***R***), whether it is always written (***W***), whether it is conditionally written (***w***), and whether it is "live" (***L***). A variable is live if a read access (***R***) occurs before a certain write access (***W***).

The three *regions* are the regions before (***b***), within (***f***) and after (***a***) the range of statements we want to extract. Using these facts and regions, a flag can be identified by two letters, such as "$\boldsymbol{R_b}$" for a read access occurring before the extracted statements. Assuming a parser that produces an AST out of the source code, gathering these flags for each variable by traversing the AST is straightforward to do, as long as the code does not contain control statements and loops. We now detail how to deal with these two constructs.

2.1. Dealing with control statements

Our solution basically has to identify "multiple paths" and "merge points". "Multiple paths" occur whenever the execution flow can vary. An **if-then-else** construct, for instance, contains two paths; one for the "then" block and one for the "else" block. If there is not "else" block, we still have two paths, but one of them is empty (it covers no statement). The same approach applies to a while loop: one path traverses the loop body, and the other one skips the loop and is empty.

"Merge points" are the locations in the source code in which multiple paths are merged. Typically, in a conditional, the paths for the "then" and the "else" block are merged just after the conditional. But this is not always the case. If one of the blocks contains a **break** statement, then its merge point is after the loop that is escaped.

Every **break** and **return** statement generates two paths within a control statement: one path ends at the location of the statement and has its merge point at the end of the escaped block. The other path continues normally and has its merge point at the end of the current block.

When multiple paths are encountered, the current table of flags is stored and each path is analyzed separately with new tables of flags. When the end of the control statement is reached, the table stored at the beginning of the control statement is merged with every table whose merge point is located at this place. The merging logic can be expressed as follows. We use the "***k***" index to identify the flags of the paths to merge and no index to identify flags of the table stored at the beginning of the control statement:

$\boldsymbol{L} = \boldsymbol{L}\ OR\ (NOT\ \boldsymbol{W}\ AND\ \cup(\boldsymbol{L^k}))$. A variable gets live if it already was or if it is live in any path and was not previously written.

$\boldsymbol{R} = \boldsymbol{R}\ OR\ \cup(\boldsymbol{R^k})$. A variable gets read if it already was, or if it is read in *any* paths.

$W = W \; OR \; \cap(W^k)$. A variable gets certainly written if it already was, or is certainly written in *all* paths.

$w = w \; OR \; \cup(W^k \; OR \; w^k)$. A variable gets possibly written if it already was, or is written in *any* paths.

These flags are gathered separately for each region of the source code: before, within and after the fragment of statements we want to extract in a new method.

2.2. Dealing with loops

If the statements to extract are within a loop, an additional step is required. More precisely, the flags of the region after the fragment to extract must be combined with those of a special region, the "entrance". The "entrance" is defined as the region of code from the beginning of the top-most enclosing loop to the end of the fragment to extract.

Intuitively, because a loop can iterate again and again, variable accesses in the "entrance" must also be considered like the accesses occurring after the region to extract. Because the loop may eventually execute only once, the "entrance" must only be considered in a conditional way. Our implementation actually uses the merging logic discussed in the previous section in the following way: the region after the fragment is merged with *two* paths; the first one corresponds to the "entrance" region and the second one is an empty one.

2.3. Putting it all together

Once the flags have been gathered for all variables, getting arguments and results is straightforward. Arguments are all variables for which the "L_f" flag is true, that is, all variables that are "live" in the fragment to extract. Results are all variables for which the "$L_a \; AND \; (W_f \; OR \; w_f)$" expression is true. In other words, the variable must be "live" after the fragment to extract ("L_a"), and must be written (conditionally or always) within the fragment itself ("$W_f \; OR \; w_f$").

3. Preliminary Results

We have implemented the "extract method" refactoring using our new approach for the data flow analysis. The implementation is an Eclipse plugin and uses the "**jdt**" (Java Development Tools) library provided by Eclipse [3]. It is implemented in 10 classes totalizing less than 1000 lines of code (excluding any user interface). About two thirds of the code is language-independent. The rest basically deals with each different Java statement and dispatch to the corresponding methods of the language-independent part.

The implementation already performs correctly on a small subset of the Java language. Some constructs such as **switch** statements are still missing though and exception handling is incomplete. A more promising result is that we already found cases in which it outperformed existing implementations.

One case is the following:

```
int test(int y) {
  int x;
  if (y > 0) {      // extract from here
    x = 1;
    y = y + x;
  }                 // to here
  x = y;
  y = y + x;
  return y;
}
```

NetBeans 5.5 and Visual Studio 2005 development environments are each producing a result that fails to compile. NetBeans incorrectly identifies the **x** variable as an argument and Visual Studio incorrectly identifies it as a result. In both case a compilation error occurs afterwards because the variable is then used (as an argument or result) before it is initialized. Our version produces the correct result (the **y** variable is identified as the only argument and result).

A second case is the following:

```
void test(int x, int y) {
  while (x < 0) {
    doStuff(--x);  // extract from here
    y++;           // to here
    x = y - 1;
  }
}
```

Eclipse 3.2 and NetBeans 5.5 refuse to extract a method, because they incorrectly identify two required results (the **x** and **y** variables) where only one is actually required. Our version produces again the correct result (only the **y** variable is identified as a result and both **x** and **y** are identified as arguments).

The latter case is not trivial: if the last assignment of the **x** variable was removed for instance, both the **x** and **y** variables would be required results, as they are both potentially reused in the next loop iteration(s). The notion of "entrance" discussed in section 2.2 would properly capture this fact.

4. Conclusion

We have briefly presented an algorithm to identify arguments and results of a method to extract. Unlike traditional dataflow analyses [4], our approach provides only the information that is relevant for method extraction. As a result, the process is drastically simplified: it is not iterative and can be based only on tables of flags and boolean expressions. Because of its simplicity compared to previous approaches coming from the field of compiler optimization, we believe that our approach can be of interest to people that are implementing refactoring tools.

As a future working direction, we are investigating more complex AST-based refactorings such as forming a template method [5].

References:

1. Tom Mens: *On the Use of Graph Transformations for Model Refactoring*, Pre-proceedings of GTTSE, pp. 67–98, 2005
2. Tom Mens, Tom Tourwé: *A Survey of Software Refactoring*, IEEE Transactions on software engineering, vol. 30, no. 2, pp. 126–139, 2004
3. Leif Frenzel: *The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs*, Eclipse Magazin, vol. 5, 2006
4. John D. Morgenthaler : *Static Analysis for a Software Transformation Tool*, PhD Thesis at the University of San Diego, California, 2003
5. N. Juillerat, B. Hirsbrunner: *Toward an Implementation of the "Form Template Method" Refactoring*, 7th IEEE International Working Conference on Source Code Analysis and Manipulation, 2007

Refactoring-Based Support for Binary Compatibility in Evolving Frameworks

Ilie Şavga and Michael Rudolf

Institut für Software- und Multimediatechologie, Technische Universität Dresden, Germany, {is13|s0600108}@inf.tu-dresden.de

Abstract. Framework refactoring may invalidate existing plugins — modules that used one of its previous versions. We use traces of refactoring to generate an adaptation layer that translates between plugins and the framework. For each supported refactoring operator we formally define a *comeback*—a refactoring used to construct adapters. For an ordered set of refactorings that occured between two framework versions, the backward execution of the corresponding comebacks yields the adaptation layer.

Motivation. Our industrial partner is developing a .NET framework. A new version of the framework is released every six months, potentially invalidating existing plugins. Either plugin developers (which are third-party companies) are forced to manually adapt their plugins or framework maintainers need to write update patches. Both tasks are usually expensive and error-prone. We want to achieve *binary compatibility* of existing plugins—they must link and run with new framework releases without recompiling.

Solution. Refactorings are the major cause of binary incompatibility comprising more than 80% of problem-causing changes. We use the trace of framework refactoring to create an adaptation layer between the framework and plugins. The adapters then shield the plugins by representing the public types of the old version, while delegating directly to the new version (see Fig. 1). For each sup-

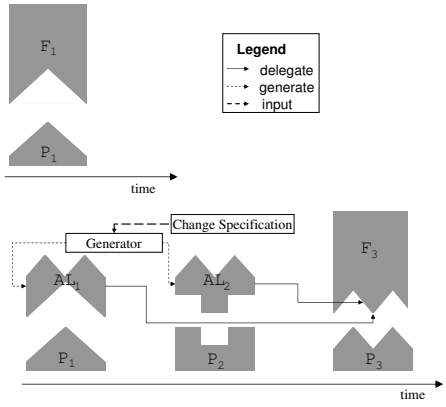

Fig. 1. Plugin adaptation in an evolving framework. In the upper part, the framework version F_1 is deployed to the user, who creates a plugin P_1. Later, two new framework versions are released, with F_3 as the latest one. While new plugins (P_3) are developed against the latest version, binary compatibility of existing ones (P_1 and P_2) is preserved by creating adapter layers AL_1 and AL_2 (the lower part).

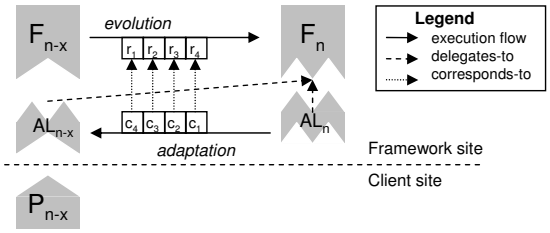

Fig. 2. Adaptation workflow. To a set of refactorings (r_1-r_4) between two framework versions (V_{n-x}, $V_n, n > x > 0$) correspond comebacks (c_4-c_1). Comebacks are executed on the adaptation layer AL_n backwards to the framework refactorings. The resulting adaptation layer AL_{n-x} delegates to the new framework, while adapting plugins of version P_{n-x}.

ported refactoring we define a *comeback*—a behavior-preserving transformation that defines how a compensating adapter is constructed. For an ordered set of refactorings that occured between two framework versions, the backward execution of the corresponding comebacks yields the adaptation layer.

Figure 2 shows the workflow of refactoring-based plugin adaptation. First, we create the adaptation layer AL_n (the right part of the figure). For each public class of the latest framework version F_n we provide an adapter with exactly the same name and set of method signatures. An adapter delegates to its public class, which becomes the adapter's delegatee. Once the adapters are created, the actual adaptation is performed by executing comebacks backwards with respect to the recorded framework refactorings, where a comeback is derived using the description of the corresponding refactoring. When all comebacks for the refactorings recorded between the last F_n and a previous F_{n-x} framework version are executed, the adaptation layer AL_{n-x} reflects the old functionality, while delegating to the new framework version.

Tool Validation. We are evaluating our concept in an operational environment using a logic programming engine. For a set of refactorings we specified the corresponding comeback transformations as Prolog rules and developed a parser for the CIL code (as stored in .NET assemblies) in order to extract meta-information about API types. This meta-information is used to create a fact base, on which a Prolog engine then executes the required comebacks. Once all comebacks are executed, the fact base contains the necessary information to create adapters and is serialized back to assemblies.

The LAN-simulation: A Refactoring Lab Session

Serge Demeyer, Bart Du Bois, Matthias Rieger, Bart Van Rompaey
Lab On Re-Engineering, University Of Antwerp

Abstract

The notion of refactoring —transforming the source-code of an object-oriented program without changing its external behaviour— has been studied intensively within the last decade. Despite the acknowledgment in the software engineering body of knowledge. there is currently no accepted way of teaching good refactoring practices. This paper presents one possible teaching approach: a lab session (called the LAN-simulation) which has received positive feedback in both an academic as well as industrial context. By sharing our experience, we hope to convince refactoring enthusiasts to try the lab-session and to stimulate teachers to incorporate refactoring lab sessions into their courses.

Introduction

Refactoring is widely recognized as one of the principal techniques applied when evolving object-oriented software systems. The key idea is to redistribute instance variables and methods across the class hierarchy in order to prepare the software for future extensions. If applied well, refactoring improves the design of software. As such, refactoring has received widespread attention within both academic and industrial circles, and is mentioned as a recommended practice in the software engineering body of knowledge [1].

The success of refactoring implies that the topic has been approached from various angles. As a result, refactoring research is scattered over different software engineering fields, among others object-orientation, language engineering, modeling, formal methods, software evolution and reengineering (see [2] for an overview of refactoring research). However, to actually teach refactoring in a classroom setting, we should distill the essence of what is good refactoring in a teachable format.

At the University of Antwerp[1], we have adopted an "active learning" approach, where students are asked to refactor a small system during a session in the computer lab. During the lab, the students are confronted with several of the most common refactoring operations (e.g., renaming, moving, extracting) to resolve a number of typical code-smells (e.g., duplicated code, nested conditionals, navigation code). In this paper, we share our experience with the lab-session.

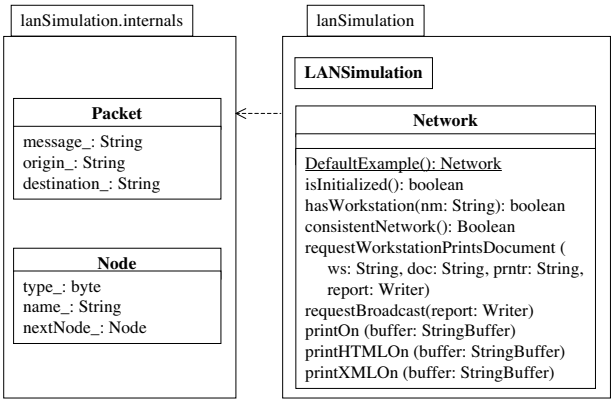

Figure 1. The design at the beginning of the lab — refactor the god class NETWORK**.**

The LAN-Simulation

We use a simulation of a Local Area Network (LAN) as the basis for the refactoring lab. The LAN-simulation is small enough to be refactored during a single lab-session (typically 3 hours), yet contains the necessary ingredients (new requirements, an implementation containing code-smells, a test suite that covers just the basics, ...) to be representative for what happens in realistic situations. The main goal of the lab is to have a practical hands-on experience with refactoring tools and relate it with other software reengineering skills, such as redesign, problem detection and regression testing. The lab requires students with good knowledge of object-oriented design principles and which are able to evaluate several design alternatives.

[1]This work has been sponsored by Eureka Σ 2023 Programme; under grants of the ITEA project if04032 entitled "Software Evolution, Refactoring, Improvement of Operational and Usable Systems" (SERIOUS).

Table 1. Overview of the use cases for the LAN-simulation.

1.0	BASIC DOCUMENT PRINTING	A workstation requests the token ring network to deliver document to a printer.
	REPORT NETWORK STATUS	Print a report of the network status as ASCII, HTML or XML
1.1	LOG PACKET SENDING	Each time a node sends a packet to the next one, log it on a log file.
1.2	POSTSCRIPT PRINTING	A packet may start with "!PS" to invoke a PostScript printing job.
1.3	PRINT ACCOUNTING	Printers register author and title of document being printed for accounting.
1.4	BROADCAST PACKET	A special type of packet "BROADCAST'" is accepted by all nodes.
2.0	READ FROM FILE	Read network configuration and network actions from a file.
2.1	GATEWAY NODE	Introduce a special "gateway" node, which can defer packets with an addressee outside the current subnetwork.
2.1	COMPILE LIST OF NODES	Gateway uses BROADCAST PACKET to periodically collect all addresses on the subnetwork.
3.0	SHOW ANIMATION	Have a GUI showing an animation while the simulation is running.

The refactoring lab starts from a requirement specification (a set of use cases — see Table 1), an initial design (a class diagram; see Figure 1) and a first increment which has been implemented in five iterations (1.0 — 1.4), both in Java and C++. To give an idea of the size: iteration 1.4 in Java consists of 4 classes totaling 677 lines of code and 280 lines of test-code. The use cases, design and code plus the lab assignments for the students can be downloaded from the "Artefacts" page at HTTP://WWW.LORE.UA.AC.BE/.

To illustrate a realistic refactoring situation, the 1.4 release is done in a procedural style. Most of the functionality is implemented in a single class NETWORK and the other classes mainly serve as data-holders – in refactoring parlance such a single class monopolizing control is called a "god class". Obviously, this class shows some typical code-smells (duplicated code, nested conditionals, navigation code) that warrant attention. Moreover, the use-cases we are expected to implement in the second release will even enlarge the god class, so it is best to refactor before starting increment 2.0. The tests should be used to demonstrate that the system does not regress during the refactoring process. However, students must assess whether the provided tests are an adequate safety net (i.e. sufficient coverage of use cases 1.0 — 1.4).

Of course there are several ways a software engineer can refactor this procedural design. During the lab session, the students are suggested to follow a path based on a number of reengineering patterns [3]: (a) EXTRACT METHOD to remove some duplicated code (release 1.5); (b) MOVE BEHAVIOUR, to move methods close to the data they operate upon (release 1.6); (c) ELIMINATE NAVIGATION CODE to reduce the coupling between some classes (release 1.7); and finally (d) TRANSFORM SELF TYPE CHECKS to change switch statements into polymorphism (release 1.8). After those refactorings, a lot of the code in NETWORK will be moved onto the class NODE and its newly created subclasses PRINTER and WORKSTATION; some code will be moved onto PACKET as well.

Teaching Experience

The lab was used by students from three universities. We did not conduct any formal questionnaires, but informal feedback indicates that the lab session is able to positively influence and change the way that students think about reengineering in general and refactoring in particular.

The lab was also applied in training sessions with professionals. In the latter case we provide step-by-step instructions, to allow trainees to redo the lab session in their office. We encourage trainees to pass the word to their colleagues and have several reports of persons who downloaded the package from the web-site and executed the lab individually with good results.

Conclusion

We presented a lab-session demonstrating a typical refactoring scenario that mimics realistic circumstances. The lab session has been used within university classes as well as industrial training sessions, and many trainees have responded positively to this approach to teach the do's and don'ts of refactoring. Based on this result, we invite practitioners who want to learn more about refactoring to try the lab. We also encourage teachers to incorporate this refactoring lab in their classes as it will help them to demonstrate the subtle craft of evolving existing software.

References

[1] P. P. C. I. C. Society. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2003.

[2] T. Mens and T. Tourwé. A survey of software refactoring. *Transactions on Software Engineering*, 30(2), 2004.

[3] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

A Heuristic-Based Approach to Code-Smell Detection

Position Paper

Douglas Kirk, Marc Roper, Murray Wood
Department of Computer and Information Sciences
University of Strathclyde
Glasgow, UK
{doug,marc,murray}@cis.strath.ac.uk

ABSTRACT

Encapsulation and data hiding are central tenets of the object oriented paradigm. Deciding what data and behaviour to form into a class and where to draw the line between its public and private details can make the difference between a class that is an understandable, flexible and reusable abstraction and one which is not. This decision is a difficult one and may easily result in poor encapsulation which can then have serious implications for a number of system qualities.

It is often hard to identify such encapsulation problems within large software systems until they cause a maintenance problem (which is usually too late) and attempting to perform such analysis manually can also be tedious and error prone. Two of the common encapsulation problems that can arise as a consequence of this decomposition process are data classes and god classes. Typically, these two problems occur together – data classes are lacking in functionality that has typically been sucked into an over-complicated and domineering god class. This paper describes the architecture of a tool which automatically detects data and god classes that has been developed as a plug-in for the Eclipse IDE.

The technique has been evaluated in a controlled study on two large open source systems which compare the tool results to similar work by Marinescu, who employs a metrics-based approach to detecting such features. The study provides some valuable insights into the strengths and weaknesses of the two approaches.

1. DATA CLASSES AND GOD CLASSES

Data classes can be described as "dumb data holders" [1]. In the extreme case they have methods for getting and setting the data and nothing else. Data classes are a problem as they typically provide poor encapsulation of their data and lack significant functionality. God classes are often a corollary to data classes and frequently represent an attempt to capture some central control mechanism. Riel [3] describes a god class as one that, "performs most of the work, leaving minor details to a collection of trivial classes" - these trivial classes being data classes. This relationship between god and data classes captures a situation where the behaviour within a system has become misplaced. Instead of being evenly distributed amongst the classes, the behaviour has somehow gravitated from the data classes into the god class. This is clearly an undesirable situation within a system and impacts upon a range of attributes of the design.

2. THE TOOL

The smell finder tool has been developed as a plug-in for Eclipse. In general terms the tool is based on the automatic identification of a set of heuristics that are symptoms of poor design – patterns of interaction or elements within the source code that are indicative of encapsulation problems and may be identified statically. It uses a static analysis of Java source files to detect these heuristics and from this infer the presence of god and data classes. Eclipse provides a rich environment for the development of such a tool because it provides project management functions and frameworks to enable the static analysis of project artefacts.

The tool has been developed to assess the utility of our detection strategies. As it currently stands it is a relatively basic implementation which only provides a minimum amount of functionality suitable for experimental purposes. In practice any such tool would require a more sophisticated front end for specifying queries and a more detailed mechanism for reporting and visualising the problems that it discovers.

2.1 Data Class Detection.

Our approach identifies a data class if the class exposes public state or if it contains one or more data methods. Public state is recognised by looking at the access modifiers for each FieldDeclaration Node in a class, any node which responds positively to an isPublic() query is considered public state. Data Methods are slightly more complex. In our approach they are detected by searching for the presence of a return statement, which returns class state or an assignment statement which assigns a value from a method parameter to the class state. The detection of class state is too elaborate to describe fully here.

2.2 God Class Detection.

In our approach god classes are detected by the presence of calls from a suspect class to a data class. This is detected in the tool by looking at the body of each method in the project searching for method calls to known data methods or field accesses to public fields (excluding self references or calls to inherited methods or fields). This test must also consider the affect of polymorphism and inheritance which can effect the physical location of the method we wish to inspect. For inherited behaviour it is relatively easy to search up the hierarchy until a suitable definition is found but polymorphic calls are not quite as simple. There may be several compatible definitions of the method and it is not possible statically to determine which of these definitions will be invoked at runtime. The tool therefore takes a conservative position of

checking each permissible definition that might be invoked and if any of those are found to be a data method then the polymorphic call is also considered to be a data method.

3. EVALUATION

We have found performing an objective comparison of this tool with others difficult to perform due to the ambiguity surrounding the definitions of data and god classes. For these reasons we have initially configured the tool to adopt a conservative approach where any data leaks are reported to be evidence of a data class and any use or abuse of such leaked data is regarded as evidence of a god class.

Clearly this is an extreme position to adopt but it is our view that any breaches of encapsulation should be identified and reported as even the most minor violation may be indicative of the start of a more serious problem. It is also our plan to augment these detectors with additional ones that sniff out evidence of the more serious problems (for example by using information such as the proportion of data leaks, the number of accesses and the way in which this data is manipulated). For this reason we have chosen to compare our system with the metrics implementation developed by Marinescu [2], not from a competitive point of view, but for the purposes of evaluating the strengths and weaknesses of the two approaches.

3.1 Results

The tool was evaluated in a comparison with the metrics-based approach of Marinescu based on two open-source case-studies - BeautyJ and JEdit[1]. The results of this comparison are shown in table 1. This shows the number of total, unique, and common problems identified using both approaches. Looking at the data there is a considerable difference in the number of problems identified by each technique. Marinescu's approach detects significantly fewer problems than our technique. This immediately raises a number of questions regarding the accuracy of the two approaches.

Table 1. Metric and heuristic results for the two systems

System		BeautyJ		JEdit	
	# classes	**Data**	**God**	**Data**	**God**
Metrics	**Total**	10	10	70	46
	Unique	2	1	56	15
	Common	8	9	18	31
Heuristics	**Unique**	63	83	161	114
	Total	71	92	179	145

A manual examination of the data class results shows that both techniques were accurate in their detection (a small number of false positives (12) were found from our technique but in relation to the number of problems identified (163) this is quite insignificant). This suggests that the difference lies with the numbers of false negatives detected by each approach. The table also shows that, despite quite different approaches to analysis, both techniques discover a surprisingly large number of data and god classes in the two case studies. Our approach suggested that about 35% of the classes in both BeautyJ and JEdit had some symptoms of broken encapsulation through data classes and 46% of the classes in BeautyJ and 29% of the classes in JEdit had some symptoms of god class breaches of encapsulation. Marinescu's top-level results were much less than this, identifying 5% data classes and 5% god classes in BeautyJ, and 14% data classes and 9% god classes in JEdit. However, Marinescu heavily filters his results, only showing the worst examples of data and god class breaches of encapsulation. If the filters are removed then Marinescu's results are of the same order as those produced by our approach. The main differences that remain result from the key differences in the two approaches - Marinescu relying on method naming conventions and requiring data methods to be relatively small and simple for data class detection, and the consideration of inheritance and polymorphism in our analysis of god classes and the consideration of cohesion and complexity in Marinescu's analysis of god classes. It is our contention that although a metrics-based approach exhibits certain strengths and is attractive in its relative simplicity, our AST-based analysis suffers from fewer weaknesses and has the potential to perform a more detailed and accurate analysis particularly when it comes to the more sophisticated design flaws we plan to study (such as "Feature Envy", "Primitive Obsession" and "Data Clumps" [1]).

[1] It must be stressed that this study is in no form intended as a criticism of either system and we are indebted to the authors for making their source available.

4. CONCLUSIONS AND FUTURE WORK

A central tenet of object-oriented design guidance is information hiding that encapsulates data and functionality together in a balanced set of cooperating classes. However, achieving this design goal in practice is extremely challenging, especially for large systems that are developed and maintained iteratively over a long period of time. This paper has described an automated approach that detects data and god class violations of encapsulation. The main contribution of this work is the demonstration that data and god class violations of encapsulation in object-oriented programs can be detected automatically by a heuristic-based analysis of the abstract syntax tree representation of the code. In applying the technique to two-open source case studies a surprising number of data and god classes were detected. - if these two case studies are representative of software that is currently being developed then data and god class breaches of encapsulation are extremely common. Further work is required to investigate whether this is really the case and to determine whether breaking encapsulation in this way is having the major impact on maintenance that conventional software design wisdom would have us believe.

5. REFERENCES

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, "Refactoring: Improving the Design of Existing Code", Addison Wesley, Reading Mass. 1999.

[2] R. Marinescu, "Detecting Design Flaws via Metrics in Object-Oriented Systems", Proc. 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39), IEEE Computer Society, 2001. pp. 173-182.

[3] A. J. Riel, "Object-oriented Design Heuristics", Addison Wesley, Reading Mass. 1999.

Using Java 6 Compiler as a Refactoring and an Analysis Engine

Jan Bečička
Technical Lead for the NetBeans Refactoring Engine
Sun Microsystems, Inc.
Jan.Becicka@sun.com

Petr Zajac
Sun Microsystems, Inc.
PhD student at VŠB - TU Ostrava
Petr.Zajac@sun.com

Petr Hřebejk
Engineering Manager
Sun Microsystems, Inc.
Petr.Hrebejk@sun.com

Abstract:

One of the cool features available in Java 6 is set of three related compiler APIs: JSR-199: Java™ Compiler API, JSR 269: Pluggable Annotation Processing API, and the Tree API (com.sun.source.tree and com.sun.source.util). These APIs provide a read-only model of Java source code. NetBeans IDE puts all these three APIs together with Jackpot transformation engine and provides full model of Java language offering read/write model capable of model transformation and formatted source rewriting. These four API work together to define a toolkit to create just about any Java language-aware tool.

This presentation will show these APIs in action. We will talk about using Java 6 compiler for Java Refactoring features and for Code Analysis features (Codeviation project).

Presentation Summary:

Our presentation will have two parts. First we will talk about Javac APIs and their integration into NetBeans APIs as part of Refactoring Engine and than we will talk about Codeviation project – code analysis tool utilizing javac to measure various quality criteria of the source code.

NetBeans Refactoring Engine

Refactoring is the technique of restructuring an existing body of code, altering its internal structure without changing its external behavior. This presentation is a practical instruction on how to create product-quality refactoring features in NetBeans IDE 6.0. The audience is expected to have prior understanding of the theoretical aspects of refactoring. NetBeans IDE 6.0 supports the creation of refactorings in two ways: through a new language-independent Refactoring API and through project Retouche - the integration of Java 6 compiler into NetBeans IDE. The new Refactoring API is an infrastructure that allows uniform handling of transformations to various file types (Java, XML, C++ etc). It can be viewed as the unification of various pluggable language-specific refactorings. Retouche (French for touching up images) is a framework that specifically provides for refactoring Java sources. It consists of the following components:

1. **JSR 199** - The Javac API: Retouche uses the Javac API to obtain a read-only Tree view of the Java Abstract Syntax Tree. This way, Retouche reuses the compiler and stay up to date with the latest language features.
2. **JSR 269** - The Pluggable Annotation Processing API: Retouche exposes a standard API from JSR 269 to refer to types and elements. This allows tools written around Retouche to use the javax.lang.model interfaces and be independent of the platform provider.
3. **Tree API & Jackpot technology**: While the Tree API gives an immutable view of the Java sources, Retouche uses a Jackpot-based code generator to perform model-based transformations of the code. This way, Retouche ensures that the modified code is formatted as per the user's specification, and performs an in-place modification of the source to avoid making changes to unaffected part of the source file.

This presentation will show the above technologies in action. The talk will then proceed to demonstrate some API examples showing how new refactorings can be integrated into NetBeans IDE. Such

refactorings can eventually become contributions to the NetBeans open-source project.

Codeviation Project

The Codeviation project permits for computing metrics and looking for potential bugs in software written in Java. It uses modified Javac compiler in order to measure arbitrary project. The data can then be connected to other sources of information about the project e.g. bug tracking systems, version control systems etc. Connecting all the information together makes it possible to prove and disprove hypothesis about the software development or to evaluate the usefulness of given metrics.

The system also produces historical data. It is possible to measure the project in different development phases (e.g. milestones or releases). This helps to understand how given project evolves. Last but not least the system is capable of detecting hot-spots of the bad quality in given software. Such hot-spots are good candidates for applying refactoring to them. It should also be possible to evaluate whether applying a refactoring to given class or set of classes really helped the quality of the whole application.

This presentation will give brief overview of the system and its architecture, list experiments already done using it and list other ideas which experiments and usages we envision for the future. We also will give a short demo of the project in action., showing how a new project and new metric can be added and how the resulting data can be visualized in the web.

Audience is supposed to have basic knowledge of the Java programming language and of metrics for object oriented languages.

Making Programmers Aware Of Refactorings

Peter Weißgerber Benjamin Biegel Stephan Diehl
University of Trier, 54286 Trier, Germany
{weissger, diehl}@uni-trier.de benjamin.biegel@gmail.com

Abstract

Modern integrated development environments, such as ECLIPSE, provide automated or semi-automated refactoring support. Despite this support, refactorings are often done manually — either because the developer is not aware of the tool support or because he is not aware that what he did was a refactoring. Moreover, as we found in [7] programmers tend to do a bad job in documenting the refactorings they have performed or even do not document them at all.

In this paper we present the design rationale of a plug-in for the ECLIPSE development platform that informs programmers about the refactorings they have performed manually and provides hyper-links to web sites describing these. The plug-in is currently under development. Finally, it should support the developer in documenting refactorings by appending an exact description of each performed refactoring to the CVS/SVN log message. For such refactorings that have been done manually, but can be performed automatically using ECLIPSE, our tool should inform the developer that this tool support exists and it is much safer to use it than to implement the refactoring manually.

1. Introduction

In evolving software systems, refactoring tasks are virtually essential too keep the code maintainable and the code structure understandable, and thus, are part of the daily work of a developer [6]. However, if done manually, refactorings can be error-prone. Thus, Fowler's book [3] contains a large catalog of refactorings and for each of it a description how it is implemented correctly. An additional problem is that, as we found in earlier work [7], for a project such as TOMCAT3 less than 10% of all refactorings are documented in the log messages of the software repository.

Thus, a tool that makes the programmer aware of his refactorings, provides links to web pages explaining these, and helps to document these, would certainly be helpful.

2. Refactoring Identification

This section gives a short overview on our refactoring detection approach, which is described in detail in [7]. In summary, this approach works in three phases:

Preprocessing: Find out which code blocks (classes, fields, methods; all identified by their fully-qualified signature) have been added and deleted compared with an earlier version of the software (e.g., the latest version in the repository).

Signature-based Identification: Compare the added and removed code-blocks using a signature-based approach to find refactoring candidates. E.g., if the method `doComputation(int, int):double` has been removed and the method `compute(int, int):double` added to the same class, we detect a candidate for a Rename Method refactoring.

Ranking and Filtering: Rank the candidates using code clone detection on the old resp. new body of the block. Filter out all candidates below a certain rank.

The remaining candidates are presented to the user. Obviously, the quality of these candidates strongly depends on the exact configuration of the clone detection and the filter. However, discussing these configuration details is beyond the scope of this paper. Instead, we focus on how to leverage the information about identified refactorings to make programmers aware of refactorings, improve the documentation of refactorings in log messages, and help to prevent errors.

Currently, we detect move, as well as rename refactorings, changes to the visibility of a symbol, and parameter additions/deletions to/from methods. Additionally, we want to record all refactorings that are done using the ECLIPSE refactoring functionality.

3. Integration in ECLIPSE

In the following we describe how we intend to integrate our refactoring identification approach into ECLIPSE.

A Refactoring View in ECLIPSE

As the space in this paper is very limited, we do not describe each feature separately. Instead, we illustrate how our tool could look like and how it can be used by a programmer by means of an example.

Figure 1 shows a mock-up how the user interface of our ECLIPSE plug-in could look like. The list of identified refactorings is presented in its own view next to the list of problems, declarations, the console, etc.. The first line contains a summary of how many refactorings have been identified, and how many of those have been performed manually respectively automatically (using the ECLIPSE refactoring tool).

In the table below the summary, each line shows information about a single identified refactoring. The first column of each line contains the refactoring kind, while the second column contains a detailed description of the refactoring. The third column indicates whether this refactoring has been done automatically or manually. Next, links to web pages with additional information on this kind of refactoring are listed. For example, we can link to the particular sub-page below `www.refactoring.com` (the web page of the book [3] which also includes the refactoring catalog). The last

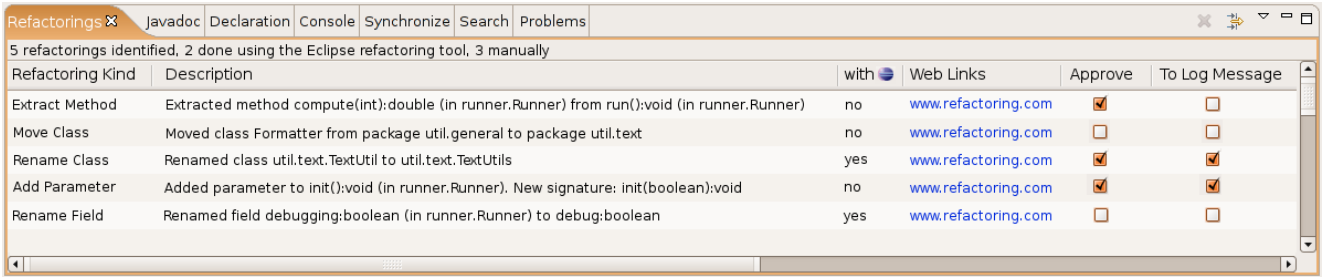

Figure 1. Example how the Eclipse integration could look like

two columns provide check-boxes to approve the refactoring and to automatically add a description of it to the log message when it is committed to the repository.

To make programmers really aware of new refactorings we also intent to show a pop-up every time new refactorings have been identified. Obviously, such a pop-up can annoy developers who are experienced enough to watch the refactoring list anyway or who are not interested in this functionality. Thus, this pop-up should be enabled by default but may be disabled easily.

Options

In the next paragraphs, we discuss technical issues respectively design options.

When to update and show the refactoring list? There are several options when the refactoring list can be updated: Our first idea was to update the list *on request*, i.e. every time the user pushes a particular button. The advantage would be that the computation of the refactoring candidates (which may cause latency on slow systems) is only done when the user requests it. However, requiring the programmer to push a button conflicts with our goal to make him aware of refactorings. Thus, we dismissed this approach and decided to update the list automatically *on save*, that means each time a file is saved to the disk. If the pop-up option is enabled, also a pop-up window opens then, provided a new refactoring is detected. Finally, the third option is that the list of refactorings is updated and presented when the developer tries to commit his changes to the repository. If only this option (and not "on save") is used, the problem is that the developer is informed about his changes quite late, but at least before they get available to other developers. We decided to implement this option additionally to "on save": if the pop-up is enabled, it should be shown again before the commit operation is executed.

How to get the changes? As explained in Section 2 we need to compare removed code block with added code blocks in order to identify refactorings. Thus, obviously the current version of the software (as shown in the workspace) has to be compared with some older version. The first approach would be to take the older version from the local history of the changes within ECLIPSE [1]. This approach should work quite fast and be easy to implement because no access to the software repository (which could be a SCM system like CVS or SUBVERSION) is required. However, as the memory to store this local change history is limited within ECLIPSE, old changes in a session can get lost. Furthermore, the local history is cleared whenever ECLIPSE is restarted. Thus, retrieving the older version from the repository enables us to identify more refactorings under certain conditions.

[1] The local history is an ECLIPSE feature that allows to browse through the n latest changes performed in an ECLIPSE session.

4. Related Work

While there exists tool support for performing refactorings in most of today's programming IDEs [5] only few researchers have tried to identify refactorings automatically [2, 7, 1].

Henkel and Diwan have shown that is useful to record automatically performed refactorings [4]. Their CATCHUP tool records refactorings performed with the ECLIPSE tool and allows to reapply these to client code. In contrast, we additionally take identified refactorings into account and help the programmmer keeping track of the refactorings he has done.

5. Conclusion

In the introduction we have motivated why an ECLIPSE plug-in that makes programmers aware of refactorings and helps to document these, would be a helpful tool. We explained how the refactoring identification works and presented how it can be integrated reasonably into ECLIPSE.

While the refactoring detection algorithm has already been implemented and evaluated [7], we have just recently started to implement the ECLIPSE integration. We are very confident, that we will be able to present a first prototype at the workshop provided that the paper is accepted. This prototype should at least perform the refactoring identification and present a list of the identified refactorings including web links, each time a developer saves his changes.

References

[1] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proc. Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000)*.

[2] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Finding Refactorings via Change Metrics. In *Proc. European Conference on Object-Oriented Programming (ECOOP 2006)*.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2001.

[4] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *Proc. International Conference on Software Engineering (ICSE 2005)*.

[5] E. Mealy and P. Strooper. Evaluating software refactoring tool support. In *Proc. Australian Software Engineering Conference (ASWEC 2006)*.

[6] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions On Software Engineering*, 30(2), 2004.

[7] P. Weißgerber and S. Diehl. Identifying Refactorings from Source-Code Changes. In *Proc. International Conference on Automated Software Engineering (ASE 2006)*.

Why Don't People Use Refactoring Tools?

Emerson Murphy-Hill and Andrew P. Black
Portland State University
{emerson,black}@cs.pdx.edu

Abstract

Tools that perform refactoring are currently under-utilized by programmers. As more advanced refactoring tools are designed, a great chasm widens between how the tools must be used and how programmers want to use them. In this position paper, we characterize the dominant process of refactoring, demonstrate that many research tools do not support this process, and initiate a call to action for designers of future refactoring tools.

1. Refactoring Tools are Underutilized

Since the original Refactoring Browser [11], programming environments have seen a remarkable integration of tools that perform semi-automatic refactoring. Programmers have their choice of refactoring tools in most mainstream languages such as Java and C#.

However, we believe that people just aren't using refactoring tools as much as they could. During a controlled experiment, we asked 16 object-oriented programming students whether they had used refactoring tools – only two said they had, reporting using them only 20% and 60% of the time [7]. Of the 31 users of Eclipse 3.2 on Portland State University computers in the last 9 months, only 2 users logged any refactoring activity. In a survey we conducted at the Agile Open Northwest 2007 conference, 112 people self-reported on their use of refactoring tools. When available, they chose to use refactoring tools 68% of the time when tools were available, an estimate which is likely optimistically high. Murphy and colleagues' data on Eclipse usage characterizes 41 programmers who were early tool adopters, and who used Eclipse for a significant amount of Java programming [6]. According to this data, over a mean period of about 66 hours per programmer, the median number of different refactoring tools used was just 4, with Rename and Move as the only refactorings practiced by the majority of subjects.

While it is difficult to tell when people are using refactoring tools and when they *could* be using refactoring tools, this second hand evidence leads us to believe that refactoring tools are currently not used as much as they could be.

2. When do Programmers Refactor?

We believe that explaining when programmers refactor also explains why programmers don't use refactoring tools, especially tools produced by researchers.

There are two different occasions when programmers refactor. The first kind occurs interweaved with normal program development, arising whenever and wherever design problems arise. For example, if a programmer introduces (or is about to introduce) duplication when adding a feature, then the programmer removes that duplication. Fowler originally argued strongly for this kind of refactoring [1], and more recently, Hayashi and colleagues [3] and Parnin and Görg [8] stated they believed this was a common refactoring process. This kind of refactoring, done frequently to maintain healthy software, we shall call **floss refactoring**.

The other kind of refactoring occurs when time is set aside. For example, a programmer may want to remove as much duplication as possible from an existing program. This sort of refactoring has been described by Kataoka and colleages [4], Pizka [9], and Borquin and Keller [1]. This kind of refactoring, done after software has become unhealthy, we shall call **root canal refactoring**.

Floss refactoring appears to be more effective, currently more widely used, and likely to be more widely used in the future. Both Pizka [9] and Borquin and Keller [1] note distinct negative consequences when performing root canal refactoring. Over the history of 3 large open-source projects, Weißgerber and Diehl were surprised to find that development contained no days of only refactorings [13]; if a day contained only refactorings, it would have indicated root canal refactoring was taking place. Likewise, Eclipse usage data from Murphy and colleagues [6] show that on only one occasion out of thousands did a programmer perform only refactoring iterations between version control commits. Furthermore, because floss refactoring is a central part of Agile

methodologies, as more programmers become Agile, we expect more programmers to adopt floss refactoring.

3. Tool Support for Floss Refactoring

Even though floss refactoring appears to be a more popular strategy than root canal refactoring, many (if not most) tools for refactoring described in the literature are built for root canal usage.

Smell detectors, fully automated refactoring tools, and refactoring scripts are examples of refactoring tools are typically built for root canal refactoring. For instance, jCosmo takes a significant amount of time and reports system-wide smells [12], making it inappropriate for floss refactoring. Guru restructures an entire inheritance hierarchy without regard to what a programmer is having trouble modifying or understanding [5], making this tool unsuitable to floss refactoring as well. Refactoring Browser scripts [10] may be too viscous for a programmer to use to perform an impromptu restructuring during floss refactoring.

While we are only able to point out a few examples due to space constraints, we believe that the majority of tools described in the literature are designed for root canal refactoring. Some exceptions do exist, such as Hayashi and colleagues' tool, which suggests refactoring candidates based on programmers' copy and paste behavior [3].

4. Future Work

We suggest that future work on refactoring tools should pay more attention to floss refactoring. Many refactoring tools can be built in a way that supports either floss or root-canal refactoring; we suggest tool builders be cognizant of which one their tool supports.

A good way to determine what kind of refactoring your tool supports is to conduct user studies. These studies can be as simple as having a few undergraduates try to refactor some open-source code. In our research, we have found that such studies are invaluable in determining the preferred usage and the limitations of our tools.

5. Acknowledgements

This research supported by the National Science Foundation under grant number CCF-0520346.

6. References

[1] F. Bourquin and R. Keller, "High-Impact refactoring based on architecture violations," Proceedings of CSMR 2007.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code: Addison-Wesley Professional, 1999.

[3] S. Hayashi, M. Saeki, and M. Kurihara, "Supporting Refactoring Activities Using Histories of Program Modification," IEICE Transactions on Information and Systems, 2006.

[4] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A quantitative evaluation of maintainability enhancement by refactoring," presented at International Conference on Software Maintenance, 2002.

[5] I. Moore. "Automatic inheritance hierarchy restructuring and method refactoring," In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications. ACM Press, New York, NY, 235-250, 1996.

[6] G. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Eclipse IDE?," IEEE Software, 2006.

[7] E. Murphy-Hill. Improving Refactoring with Alternate Program Views. Technical Report TR-06-05, Portland State University, Portland, OR, 2006.

[8] C. Parnin, G, and C. Görg, "Lightweight visualizations for inspecting code smells," Proceedings of the 2006 ACM Symposium on Software Visualization, 2006.

[9] M. Pizka. "Straightening Spaghetti Code with Refactoring." In *Proc. of the Int. Conf. on Software Engineering Research and Practice - SERP*, pages 846-852, Las Vegas, NV, June 2004.

[10] D. Roberts and J. Brant, "Tools for making impossible changes - experiences with a tool for transforming large Smalltalk programs," IEE Proceedings - Software, vol. 151, pp. 49-56, 2004.

[11] D. Roberts, J. Brant, and R. Johnson, "A refactoring tool for Smalltalk," Theor. Pract. Object Syst., vol. 3, pp. 253-263, 1997.

[12] E. Van Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells," Proceedings of the Ninth Working Conference on Reverse Engineering, 2002.

[13] P. Weißgerber, and S. Diehl, "Are refactorings less error-prone than other changes?," presented at MSR '06: Proceedings of the 2006 international workshop on Mining software repositories, 2006.

Automating Feature-Oriented Refactoring of Legacy Applications

Christian Kästner, Martin Kuhlemann
School of Computer Science
University of Magdeburg
{ckaestne,mkuhlema}@ovgu.de

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

Abstract

Creating a software product line from a legacy application is a difficult task. We propose a tool that helps automating tedious tasks of refactoring legacy applications into features and frees the developer from the burden of performing laborious routine implementations.

1. Introduction

A *software product line (SPL)* aims at creating highly configurable programs from a set of features. To reduce costs and risks, developers often take an extractive approach for creating the SPL by refactoring and decomposing one or more legacy applications into features [2]. In prior case studies, we detached optional features like transactions, statistics, or caches from database systems, and experienced that refactoring legacy applications manually is a complex and difficult task containing many routine operations [7].

When decomposing a legacy application into features, the developers focus is on identifying the feature code, i.e., classes, methods, fields, or statements associated with a certain feature. In contrast, the actual refactoring, consisting of removing code fragments and reintroducing them in feature modules, is a routine task that can be automated with a tool.

Previously, we built a tool called *ColoredIDE* to identify and mark feature code in a legacy Java application. Now, we use this marked code base to refactor a legacy application into a software product line with multiple features.

Features in an SPL can be implemented in different ways. Current research suggests to implement features as mixin layers [12] or aspects [5, 8], but other implementation approaches are possible. In the prototype of our tool we investigated refactorings into feature modules implemented with *Jak* [1] and *AspectJ* [8]. In this paper we focus on AspectJ as target language and assume a basic knowledge of it.

2. Refactoring

The input of our refactoring tool is a list of features and a marked version of the source code, where fragments are associated to these features. In Figure 1 we show an example class *Stack* with a feature *Locking* whose code is underlined. Technically, features are associated to elements in the *abstract syntax tree (AST)* of the source code, e.g., the AST node for the method *lock* (Line 8) and the statements in Lines 3 and 5 are marked with the *Locking* feature (underlined).

```
1 class Stack {
2   void push(Object o) {
3     Lock lock = lock(o);
4     elementData[size++] = o;
5     lock.unlock();
6   }
7
8   Lock lock(Object o) { /*...*/ }
9 }
```

Figure 1. Marked Legacy Code

Our tool now creates a new project for the SPL with directories for every feature and a directory for the base code. The base code contains the original program without any feature code. The feature directories contain aspects that reintroduce the feature code. In Figure 2 we show the resulting class of the base code and the aspect implementing the *Locking* feature for our example. The SPL can now be configured by selecting the directories to include in the compilation process.

For the implementation of the AspectJ refactoring we follow proposals for refactorings like *Extract Introduction* [6], *Move Field from Class to Inter-type* [10], or *Extract Advice* [6].

3. Advanced Topics

Generally, our tool uses more sophisticated rewrites than shown in the example above. For instance, when the feature

```
1  class Stack {
2    void push(Object o) {
3      elementData[size++] = o;
4    }
5  }
```

```
6  aspect Synchronization {
7    void around(Stack stack, Object o) :
8      execution(void Stack.push(Object)) && args(o) &&
           this(stack) {
9      Lock lock = stack.lock(o);
10     proceed(stack);
11     lock.unlock();
12   }
13   Lock Stack.lock(Object o) { /*...*/ }
14 }
```

Figure 2. Refactored SPL code

code is not placed at the beginning or end of the method. AspectJ does not support statement level join points [11]. In some cases it is possible to advise a method call that is located next to the feature code, in other cases we have to create artificial join points by preparing the base code. For example, we introduce calls to empty hook methods [11] or perform a preliminary *Extract Method* refactoring [4].

Furthermore, code can be associated with multiple features. Such code is usually a result of feature interactions, e.g., when one feature calls a method introduced by another feature. To refactor such cases we use the derivative model by Liu et al. [9] and create separate modules containing aspects for these code fragments. Again our tool automates the creation of the additional modules and the refactorings.

Finally, our tool initially refactored every marked code fragment individually. That means that advanced AspectJ mechanisms, e.g., pattern expressions for homogeneous pointcuts, were not employed. However, our tool combines pointcuts where advice statements have equal bodies with automated *Extract Pointcut* refactorings [3]. Thus, our refactoring tool takes advantage of AspectJ's capabilities and reduces code replication automatically.

4. Conclusion

Refactoring a legacy application into features to create a SPL is a difficult and laborious task. It consists of detecting features in the legacy code and of their refactoring. While detecting features is an interactive procedure the refactoring can be automated completely. We propose a refactoring tool which generates an SPL implemented in *Jak* or *AspectJ* based on marked legacy code.

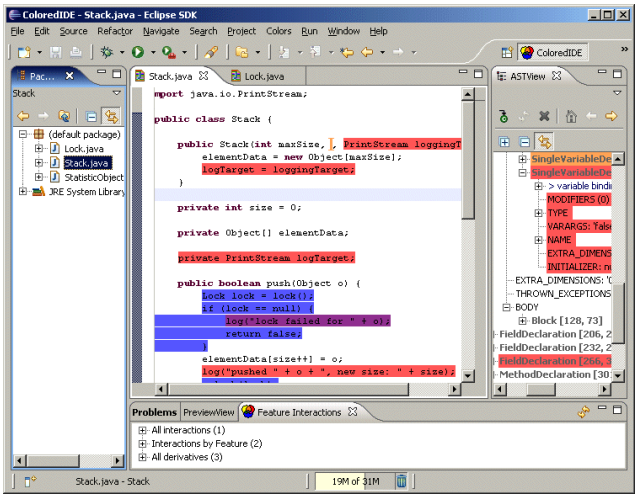

Figure 3. ColoredIDE Screenshot

References

[1] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6), 2004.

[2] P. Clements and C. Kreuger. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software*, 19(4), 2002.

[3] L. Cole and P. Borba. Deriving Refactorings for AspectJ. In *Proc. Int'l Conf. Aspect-Oriented Software Development*, 2005.

[4] M. Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[5] M. L. Griss. Implementing Product-Line Features by Composing Aspects. In *Proc. Int'l Software Product Line Conference*. 2000.

[6] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of Aspect-Oriented Software. In *Proc. Net.ObjectDays*, 2003.

[7] C. Kästner. Aspect-Oriented Refactoring of Berkeley DB. Master's thesis, University of Magdeburg, Germany, 2007.

[8] G. Kiczales et al. An Overview of AspectJ. In *Proc. Europ. Conf. Object-Oriented Programming*. 2001.

[9] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *Proc. Int'l Conf. on Software Engineering*, 2006.

[10] M. P. Monteiro and J. M. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In *Proc. Int'l Conf. Aspect-Oriented Software Development*, 2005.

[11] G. C. Murphy et al. Separating Features in Source Code: an Exploratory Study. In *Proc. Int'l Conf. on Software Engineering*. 2001.

[12] Y. Smaragdakis and D. Batory. Mixin Layers: an Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2), 2002.

An Adaptation Browser for MOF*

Guido Wachsmuth

Humboldt-Universität zu Berlin
Unter den Linden 6
D-10099 Berlin, Germany
guwac@gk-metrik.de

1 Metamodel Adaptation

Refactoring improves the structure of code in a behaviour-preserving manner. MOF compliant metamodels describe the structure of models. Thus, behaviour-preservation properties do not characterise metamodel refactoring accordingly. In [1], we define semantics-preservation properties in terms of modelling concepts of a metamodel and its set of possible instances. In these terms, metamodel refactoring improves the structure of existing metamodels in a semantics-preserving manner.

Like other software artefacts, metamodels evolve over time due to several reasons: During design, alternative metamodel versions are developed and well-known solutions are customised for new applications. During maintenance, errors in a metamodel are corrected. Furthermore, parts of the metamodel are redesigned due to a better understanding or to facilitate reuse.

Metamodel evolution is usually performed manually by stepwise adaptation. Our tool provides automated adaptation steps listed in Table 1. Each step is classified according to its semantics- and instance-preservation properties. Metamodel adaptation goes beyond pure refactoring. *Construction* increases the instance set of a metamodel. In contrast, *destruction* decreases the instance set.

Models need to co-evolve in order to remain compliant with the metamodel. Without co-evolution, these artefacts become invalid. Like metamodel evolution, co-evolution is typically performed manually. This is an error-prone task leading to inconsistencies between the metamodel and related artefacts. These inconsistencies usually lead to irremediable erosion where artefacts are not longer updated [2]. The adaptation browser comes with automatic co-evolution steps deduced from well-defined evolution steps. This *co-adaptation* prevents inconsistencies and metamodel erosion.

2 Implementation

Our tool is build upon *A MOF 2 for Java* [3]. This tool offers a non persistent but fast model storage, a Java language mapping that allows type-safe programming without type-casts, and an integrated OCL processor. Furthermore, it supports property subsetting and other advanced redefinition features. The adaptation browser is implemented as an Eclipse plugin bundle. Thereby, we rely on Eclipse's *Language Tool Kit*. This provides us with undo/redo support, adaptation history, and scripting facilities.

3 Applications

The tool facilitates a well-defined stepwise metamodel design. Starting from

* This work is supported by grants from the DFG (German Research Foundation, Graduiertenkolleg METRIK).

Table 1. Metamodel adaptations provided by the adaptation browser.

Adaptation	Semantics preservation	Inverse
Refactoring		
rename element	strictly preserving	rename element
move class	preserving modulo variation	move class
move property	preserving modulo variation	move property
extract class	preserving modulo variation	inline class
inline class	preserving modulo variation	extract class
association to class	preserving modulo variation	class to association
class to association	preserving modulo variation	association to class
Construction		
introduce class	introducing	eliminate class
extract superclass	introducing	flatten hierarchy
generalise property	increasing	restrict property
introduce property	increasing modulo variation	eliminate property
pull property	increasing modulo variation	push property
dissociate properties	increasing modulo variation	associate properties
Destruction		
eliminate class	eliminating	introduce class
flatten hierarchy	eliminating	extract superclass
restrict property	decreasing	generalise property
eliminate property	decreasing modulo variation	introduce property
push property	decreasing modulo variation	pull property
associate properties	decreasing modulo variation	dissociate properties

basic features, new features are introduced by construction. Scripts of consecutive adaptation steps document design decisions. By changing particular steps, metamodel designers can alternate designs.

Like other software, metamodels are subject to maintenance. Metamodel maintenance also benefits from a transformational setting. Erroneous features can be corrected by construction and destruction. Refactoring provides for reengineering a metamodel design without introducing defects. Construction and destruction assist adjustment to changing requirements.

Often, language knowledge resides only in language-dependent tools or semi-formal language references. Language recovery is concerned with the derivation of a formal language specification from such sources. For grammar recovery, a transformational approach already proved to be valuable [4]. In a similar way, our tool assists metamodel recovery.

References

1. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In Ernst, E., ed.: ECOOP'07. Volume 4609 of LNCS., Springer (2007) 600–624 (to appear).
2. Favre, J.M.: Meta-model and model co-evolution within the 3D software space. In: ELISA'03. (2003) 98–109
3. Scheidgen, M.: A MOF 2.0 for Java. (http://www.informatik.hu-berlin.de/sam/meta-tools/aMOF2.0forJava)
4. Lämmel, R.: Grammar adaptation. In Oliveira, J.N., Zave, P., eds.: FME '01. Volume 2021 of LNCS., Springer (2001) 550–570

Refactoring Functional Programs at the University of Kent

Simon Thompson, Chris Brown, Huiqing Li, Claus Reinke, Nik Sultana
Computing Laboratory, University of Kent, UK

S.J.Thompson@kent.ac.uk

http://www.cs.kent.ac.uk/projects/refactor-fp/
http://www.cs.kent.ac.uk/projects/forse/wrangler/doc/

We have developed tools for refactoring functional programs written in both Haskell and Erlang. As well as giving brief demonstrations of our tools, we will

- discuss the particularities of refactoring functional programs, such as verification of refactorings;
- examine the differences in building refactoring support for the two languages; and
- ask the question of how to design tools that will be taken up by the working software developer.

Haskell and Erlang

Both Haskell and Erlang are general-purpose functional programming languages, but they also have many differences. Haskell is a lazy, statically typed, purely functional language featuring higher-order functions, polymorphism, type classes, and monadic effects.

Erlang is a strict, dynamically typed functional programming language with built-in support for concurrency, communication, distribution, and fault-tolerance. In contrast to Haskell, which arose from an academic initiative, Erlang was developed in the Ericsson Computer Science Laboratory, and has been actively used in industry both within Ericsson and beyond.

Refactoring Haskell

Our project *Refactoring Functional Programs*, has developed the Haskell Refactorer, HaRe, providing support for refactoring Haskell programs. HaRe is a mature tool covering the full Haskell 98 standard, including "notoriously nasty" features such as monads, and is integrated with the two most popular development environments for Haskell programs: Vim and X/Emacs. HaRe refactorings apply equally well to single- and multiple-module projects. HaRe is itself implemented in Haskell.

Haskell layout style tends to be idiomatic and personal, especially when a standard layout is not enforced by the program editor, and so needs to be preserved as much as possible by refactorings. HaRe does this, and also retains comments, so that users can recognise their source code after a refactoring. The current release of HaRe supports a wide variety of refactorings, and also exposes an API for defining Haskell refactorings and program transformations.

The refactorings supported by HaRe fall into three categories: structural refactorings affecting the names, scopes and structure of the entities defined in a program; module refactorings affecting the imports and exports of modules and the definitions contained in them; and data-oriented refactorings of data types. All these refactorings have been successfully applied to multiple module systems containing tens of thousands of lines of code.

Refactoring Erlang

Wrangler is an Erlang refactoring tool that supports interactive refactoring of Erlang programs. The current snapshot (Wrangler 0.1) is a prototype, made available so that potential users can experiment with refactoring support for Erlang programs, and feed back on their experience; nonetheless, this release incorporates features elicited from the user community (e.g. support for macros).

Wrangler 0.1 supports a number of basic Erlang refactorings, including renaming variable/function/module names and generalisation of a function definition. Built on top of the functionalities provided by the Erlang syntax-tools package, Wrangler is embedded in the Emacs editing environment, and makes use of the functionalities provided by Distel, an Emacs-based user interface toolkit for Erlang, to manage the communication between the refactoring tool and Emacs.

All the implemented refactorings are module-aware. In the case that a refactoring affects more than one module in the program, a message telling which files have been modified by the refactorer will be given after the refactoring has been successfully effected. Since there is no formal notion of 'project' in Erlang, we provide a 'customize' command in the refactorer to allow the user to specify the boundary of the program for the purposes of the transformation. Undo is supported by the refactorer. Applying undo once will revert the program back to the status right before the last refactoring performed.

Wrangler makes use of functionalities provided by the epp_dodger module from Erlang SyntaxTools to parse Erlang source code, and the refactorer is able to refactor Erlang modules containing preprocessor directives and macro applications, as long as these are syntactically well-formed, otherwise the refactorer will give a syntax error message.

Publications

Refactoring Erlang programs. Huiqing Li, Simon Thompson, Laszlo Lovei, Zoltan Horvath, Tamas Kozsik, Anik Vig, and Tamas Nagy. In The Proceedings of 12th International Erlang/OTP User Conference, Stockholm, Sweden, November 2006.

A comparative study of refactoring Haskell and Erlang programs. Huiqing Li and Simon Thompson. In Massimiliano Di Penta and Leon Moonen, editors, Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), pages 197-206. IEEE, September 2006.

Formalisation of Haskell Refactorings. Huiqing Li and Simon Thompson. In Marko van Eekelen and Kevin Hammond, editors, Trends in Functional Programming, September 2005.

Refactoring Functional Programs. Simon Thompson. In Varmo Vene and Tarmo Uustalu, editors, Advanced Functional Programming, 5th International School, AFP 2004, volume 3622 of Lecture Notes in Computer Science, pages 331-357. Springer Verlag, September 2005.

The Haskell Refactorer: HaRe, and its API. Huiqing Li, Simon Thompson, and Claus Reinke. In John Boyland and Grel Hedin, editors, Proceedings of the 5th workshop on Language Descriptions, Tools and Applications, 2005.

Progress on HaRe: the Haskell Refactorer. Huiqing Li, Claus Reinke, and Simon Thompson. Poster, International Conference on Functional Programming, Snowbird, Utah. ACM, 2004.

Tool support for refactoring functional programs. Huiqing Li, Claus Reinke, and Simon Thompson. In Johan Jeuring, editor, ACM SIGPLAN 2003 Haskell Workshop. Association for Computing Machinery.

A case study in refactoring functional programs. Simon Thompson and Claus Reinke. In Roberto Ierusalimschy, Lucilia Figueiredo, and Marcio Tulio Valente, eds., VII Brazilian Symposium on Programming Languages, Sociedade Brasileira de Computacao, 2003.