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).

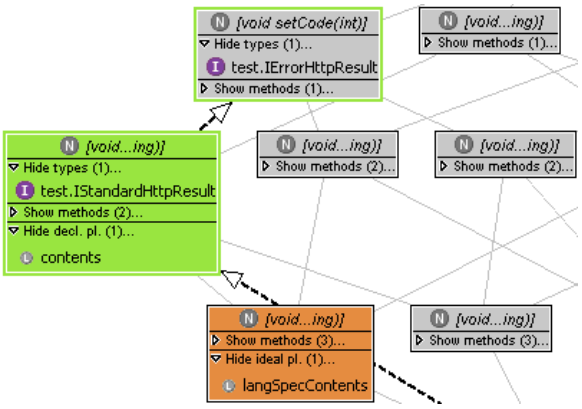

Figure 1: TAA Visual View

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.

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.

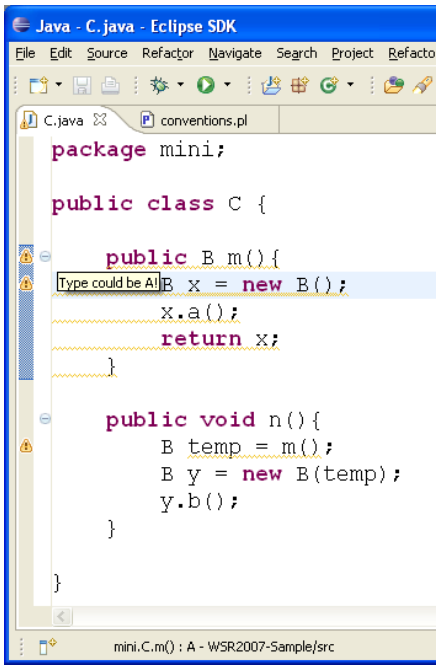

Figure 3. Tool tips indicating possible type generalizations detected by automated logic-based analysis.

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.

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.

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.

A Model of Refactoring Tool Use

Emerson Murphy-Hill
The University of British Columbia
emhill@cs.ubc.ca

Abstract

For the most part, refactoring tools have changed little since the Smalltalk Refactoring Browser. By continuing to mimic the Refactoring Browser's user interface, the community of tool builders may be not find new user interfaces that help programmers do their job more effectively. In this position paper, I put forward a general model of how programmers use Refactoring Browser-like tools. I argue that the model is useful for breaking away from the design of traditional refactoring tools and thinking about the design of new ones.

Categories and Subject Descriptors D.2.2 [*Design Tools and Techniques*]: Object-oriented design methods

General Terms Design

Keywords refactoring, tools, model, process

1. Origins of the Model

In 2008, Andrew Black and I published a paper at the International Conference on Software Engineering that discussed two problems with existing refactoring tools [8]. By doing a formative study where we observed several programmers in the lab performing EXTRACT METHOD refactorings, we showed that programmers have difficulty selecting code for refactoring and understanding error messages. We then attempted to implement solutions to both problems; two tools to help programmers select code, and one that displays error messages graphically. Our evaluation provided evidence that our techniques were more usable.

But how well do our observations about problems – or claims of usability of the improvements – generalize to other brands of refactoring tools, other than Eclipse? Also, how does it generalize to different types of refactorings, other than EXTRACT METHOD? At the time, we simply claimed that Eclipse's EXTRACT METHOD had a user interface that is typical of most refactoring tools. We had some confidence in this statement because we had painstakingly looked at every refactoring tool, at least a dozen at a time. In subsequent research, we found this process tedious and the argument somewhat unconvincing. Thus, we have found it useful to create a generalized model of how most refactoring tools work, then made the argument for generalizability based on congruence with the model. Moreover, we have found that the model is also useful for creating new, innovative user interfaces to refactoring tools.

In the next section, I discuss the model in detail (Section 2), and compare it to existing models (Section 3). I then discuss how the model can be used effectively (Section 4). My main contribution is the model itself, a model that I have found useful and believe other researchers and toolsmiths will find useful as well.

2. The Model

Figure 1 shows a model of how programmers use conventional refactoring tools, in the style of the Refactoring Browser [11]. Let me explain each step in turn, then explain the transitions between steps.

2.1 Steps in the Model

Finding code to be refactored is the Identify step, often called finding code smells [3]. This is often a step that programmers perform manually.

Typically you Select code in an editor, but it could also be done in derivative code representation (such as Eclipse's Outline View, Quick Fixes, or O'Connor and colleagues' graph representation [9]). Code can also be selected semi-automatically, such as by specifying queries in iXj [1].

To Initiate a refactoring tools is to choose the desired refactoring. Initiation can be done by choosing the desired refactoring name from a menu or with a hotkey.

To Configure a refactoring, you choose some options for the desired transformation, such as when you move a method, whether you want to delete the original method. This is often achieved with a dialog box.

To Execute a refactoring is to indicate to the refactoring tool that you are ready for the transformation to be applied, often by clicking an "OK" button.

To Interpret Results, you determine whether the refactoring that was applied was the refactoring that you desired.

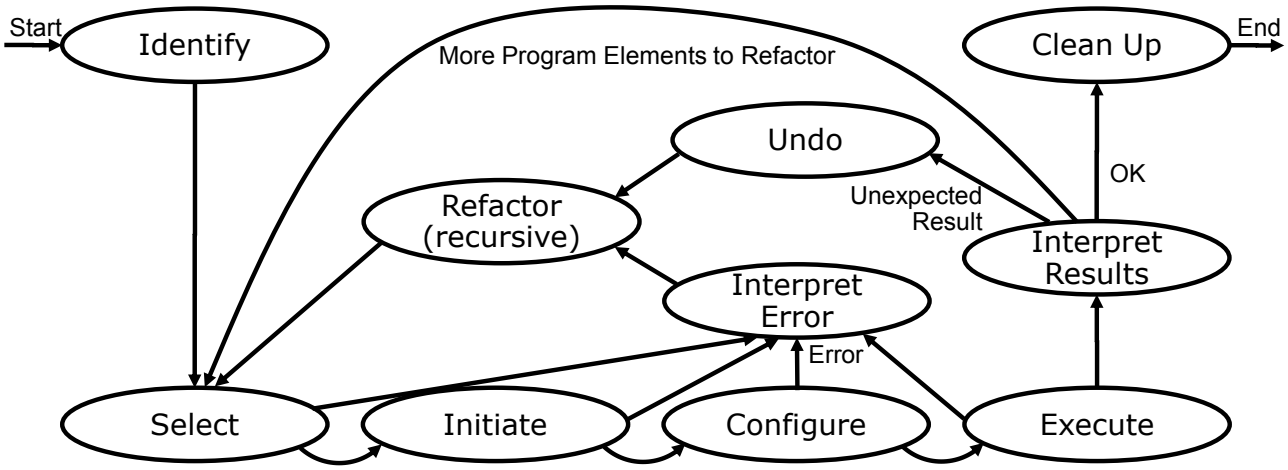

Figure 1. A model of how programmers use conventional refactoring tools.

When a precondition is violated, you typically must interpret an error message and choose an appropriate course of action (Interpret Error).

When an unexpected result is encountered, you may revert the program to its original state (Undo).

You may recursively perform a sub-refactoring (Refactor) in order to make the desired refactoring successful. For example, suppose you want to abstract the following code into a logging function, so you perform an EXTRACT METHOD on it with the Eclipse refactoring tool:

```
System.out.println("start server");
```

and you end up with:

```
log();
```

This is probably not what you intended. To produce the desired results using the Eclipse refactoring tool, you need to first apply EXTRACT LOCAL VARIABLE:

```
String s = "start server";
System.out.println(s);
```

then EXTRACT METHOD again:

```
String s = "start server";
log(s);
```

This is what I am calling recursive refactoring.

Relatedly, you may choose to perform some Clean Up refactorings to "unwind" the recursive step. Using the same example, after performing EXTRACT METHOD, you may want to INLINE LOCAL VARIABLE on `s` to undo the recursive EXTRACT LOCAL VARIABLE, ending up with:

```
log("start server");
```

2.2 Transitions in the Model

The ideal way to transition through the model is via the following steps; Identify, Select, Initiate, Configure, Execute, Interpret Results, and Clean Up. As mentioned, you may realize that the refactoring did not do what you want, so you may need to UNDO and REFACTOR recursively. A refactoring tool may produce an error at any of the Select, Initiate, Configure, or Execute steps, and thus the programmer would transition to the INTERPRET ERROR step. While not explicitly shown, you may abandon using the tool at any point, which corresponds to transitioning to a failure state from any step in the model.

3. Related Work

The model is not without precedent, so I discuss prior work here.

Tourwé and Mens described three phases of refactoring [12]: detecting when the refactoring should be applied, identifying which refactorings to apply, and performing the refactoring. The steps in this simpler model correspond to the Identify, Initiate, and Execute steps of my refactoring model, respectively.

The documentation for Apple's Xcode refactoring tools contains a strikingly similar workflow to my model [4, p. 11], including steps corresponding to the Select, Initiate, Configure, Execute, Interpret Results, and Error Interpretation steps. Their workflow also includes steps for running unit tests after a refactoring is complete.

Kataoka and colleagues' introduced a 3-step model [5]: identification of refactoring candidates, validation of refactoring effect, and application of refactoring. This corre-

sponds to the Identify, Interpret Results, and Execute steps of our refactoring model, respectively.

4. Using the Model

It is easy to argue that the model does not capture how every refactoring tool is used. But to make such an argument is to miss the point of the model in the first place; it is precisely the ways that it can be changed that makes it useful. Indeed, as Petre has observed, innovation does not simply spring from the minds of insightful individuals; instead, it happens as deliberate methods are employed by designers, such as the systematic variation of constraints [10]. It is my hope that the model, combined with the following methods, will help toolsmiths systematically explore the design space of refactoring tools.

Here I identify 6 different methods for modifying the model, methods that afford several advantages. For each method, I sketch how the method is used, an existing example of what the method entails (when possible), and a new example of what the method entails.

4.1 Enrichment

One way to use the model is to enrich steps by analyzing how one step in the model might be improved by adding something to it.

For example, you might notice that there is little support for identifying opportunities for refactoring in most development environments (the identify step). Using the enrichment method on the model could entail the creation of a "smell detector," such as JCosmo [2]. The advantage of such a tool would be to automate a process that is sometimes difficult for programmers to perform manually [6].

Taking another example, we might see how we could enrich the recursive refactor and clean up steps. I know of no current support for these steps. However, you can imagine a tool that maintains a stack of refactorings for you. Suppose a refactoring tool invocation (again, say EXTRACT METHOD) does not work out as you intended because you need to perform EXTRACT LOCAL VARIABLE first. If so, you can push that *refactoring object* onto a stack without applying it, perform your EXTRACT LOCAL VARIABLE refactoring, and finally, automatically pop the refactoring object off the stack and apply it to your code. The advantage is that you would not have to remember what or how you were refactoring in the first place; the tool would remember for you.

4.2 Depletion

The opposite of enrichment is depletion, meaning removing something from a step, making it optional, or removing a step entirely.

Suppose that you eliminate the configuration step. Then you would have, for example, Eclipse's more streamlined refactoring tools, which perform the refactoring immediately. The advantage is that you do not have to wade through a dialog just to perform a simple refactoring.

To take another example, suppose you depleted the interpret error step by building a tool that did not show errors that would cause the code to stop compiling. Thus, a programmer would be allowed to perform a refactoring that introduced compilation errors. The advantage to this is two-fold: first, I would argue that programmers sometimes want to break code for some higher purpose, and second, programmers already know how to fix compilation errors, so having them fix compilation errors should be easier than fixing unfamiliar refactoring tool errors.

4.3 Reordering

Changing the order of steps in the model can be useful as well.

For example, suppose that you switched select and initiate. Black and I did this in our refactoring cues tool [7], so that you chose what refactoring you want before selecting code. The advantage to initiating the refactoring first is that you can select multiple program elements for refactoring at one time, something that is more difficult if you select first, then initiate.

As another example, suppose you executed first, then configured the refactoring. Essentially, you would change your code with default settings, then change how the refactoring was performed. For example, you could EXTRACT METHOD first, then choose where you want the method inserted or what the parameter names would be (both Eclipse and Refactor! can do variants of this). The advantage is that interpreting what the refactoring tool did would be more incremental, and thus, I would argue, easier for the programmer to understand.

4.4 Parallelization

While the refactoring process is generally linear, it may also pay off to think about what would happen of you were to parallelize some steps in the process.

Taking our refactoring cues tool as an example again, we parallelized select and configure; once the refactoring is chosen, the programmer can configure using a non-modal palette next to the editor, or select code to be refactored in the editor itself. The advantage is allowing for implicit flexibility when using the tool.

To take another example, suppose that you made interpret results parallel with configure. Then, as you changed options in your refactoring, you could immediately see the results. The advantage of this immediate feedback is that you would not have to go back and forth between configure and interpret results to see the effect of your changes.

4.5 Splitting

One step may usefully be replaced by several steps via splitting.

For example, suppose that you split the initiation step done with a system menu (the menu at the top of the IDE) into two steps, by first filtering the list of applicable refac-

torings, then initiating one of those refactorings. Thus, you have context menus, which are filtered based on what is selected in the editor.

To take another example, suppose you split the interpret results step into two steps, one before configuration and one after. The advantage would be that you could determine whether you wanted to perform any configuration at all, or if the default options were sufficient.

4.6 Merging

On the other hand, one may take several steps and merge them into one.

For example, you might merge initiate and configure together usefully. Suppose that you gestured to initiate a refactoring, such as gesturing up with the mouse to initiate PULL UP METHOD [7]. Suppose further that the distance that you gestured up indicates how far up the inheritance hierarchy the method should go. So if you gesture up, say, less than one centimetre, the method would go into the direct superclass, and if you gestured further, it would go into other superclasses. The advantage would be that you could specify some configuration parameters while initiating the tool.

5. Conclusion

Refactoring tools have the potential to go far beyond what the original toolsmiths intended, but toolsmiths must consider alternative designs. In this paper, I have presented a model for how people use refactoring tools, a model that I have argued can be employed to generate new designs for refactoring tools. I have found this model useful in my own research; I hope that other researchers will find it useful as well.

Acknowledgments

Thanks to Andrew P. Black, who helped develop the model discussed here. Thanks to the National Science Foundation for partially funding this research under grant CCF-0520346.

References

[1] Marat Boshernitsan, Susan L. Graham, and Marti A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI '07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 567–576, New York, NY, USA, 2007. ACM. doi: doi.acm.org/10.1145/1240624.1240715.

[2] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, pages 97–106, Washington, DC, USA, 2002. IEEE Computer Society.

[3] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[4] Apple Inc. Xcode refactoring guide, 2009. Technical Document, `http://developer.apple.com/DOCUMENTATION/DeveloperTools/Conceptual/XcodeRefactoring/Xcode_Refactoring.pdf`.

[5] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *ICSM '02: Proceedings of the International Conference on Software Maintenance*, pages 576–585, Washington, DC, USA, 2002. IEEE Computer Society.

[6] Mika V. Mäntylä. An experiment on subjective evolvability evaluation of object-oriented software: explaining factors and interrater agreement. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 287–296, November 2005. doi: 10.1109/ISESE.2005.1541837.

[7] Emerson Murphy-Hill and Andrew P. Black. High velocity refactorings in Eclipse. In *ETX '07: Proceedings of the 2007 OOPSLA workshop on Eclipse Technology eXchange*, pages 1–5, New York, NY, USA, 2007. ACM. doi: doi.acm.org/10.1145/1328279.1328280.

[8] Emerson Murphy-Hill and Andrew P. Black. Breaking the barriers to successful refactoring: observations and tools for extract method. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 421–430, New York, NY, USA, 2008. ACM. doi: doi.acm.org/10.1145/1368088.1368146.

[9] Alexis O'Connor, Macneil Shonle, and William Griswold. Star diagram with automated refactorings for Eclipse. In *ETX '05: Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, pages 16–20, New York, NY, USA, 2005. ACM. doi: doi.acm.org/10.1145/1117696.1117700.

[10] Marian Petre. How expert engineering teams use disciplines of innovation. *Design Studies*, 25(5):477 – 493, 2004. ISSN 0142-694X.

[11] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997. ISSN 1074-3227.

[12] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. *European Conference on Software Maintenance and Reengineering*, 0:91 – 100, 2003. doi: 10.1109/CSMR.2003.1192416.