Refactoring Tool Challenges in a Strongly Typed Language

Christopher Seguin
Independent Researcher
150 Torington Way
(302) 631-7561

seguin@acm.org

ABSTRACT

This poster examines the challenges of developing a refactoring tool for a weakly typed language such as Smalltalk as opposed to a strongly typed language such as Java. To explore this, we will compare the push up field refactoring in each language. This refactoring was selected because it is relatively simple conceptually, but difficult to implement for Java.

In a weakly typed language such as Smalltalk, push up field is simple. The user simply determines that the parent class needs the variable. The refactoring tool moves the field to the parent class. Then the tool searches all the subclasses of the parent class, if the classes have a variable with the same name, the refactoring tool removes the variable from the subclass. The task is complete.

For Java, a description of classes and types is necessary. Let's start with a base class A. A has a number of child classes, B, C, D, E, F, and G. Each of B-G has a single instance variable named var. The only difference between the classes is the type of var. B and C both have a variable named var with type X. D has a variable named var with a type Y. E has a variable named var with type W. F has a variable named var with a type of Z. And G has a variable named var with the type int. W, X, Y, and Z are classes. W is the base class, X is a subclass of W, and Y is a subclass of X. Z is unrelated to all of the other classes. Since all subclasses of A have a variable named var, a programmer might suspect that they could reduce the amount of code by moving the variable into the parent class A.

Let's move the field named var from class B into class A. Like in Smalltalk, the Java refactoring tool can remove the variable var from B and C, since the var variable in both classes have the same type. The source file for A would get the declaration of type X named var. The source file for A might also gain an import for the type X. Let's postpone the discussion of the scope of the variable var.

The refactoring tool can remove var from D since the storage in A is sufficient. However, the impact on the source code depends on whether there is a difference between the interface of X and Y. If there are methods or fields added in Y that are not in X, then by removing the variable var from D, the refactoring tool must be sure to cast var back to a Y in code that refers to var in a way that accesses the differences. For instance, if X and Y have a method m, then in D the invocation var.m() remains unchanged. If Y has a method n and no similar method is present in X or a parent class of X, then D must change the code that looks like var.n(); to ((Y) var).n();

The refactoring tool cannot remove var from E, since a variable of type W can not be stored in a class X. A refactoring tool might change the type of the object in class A to W. The effect of casting var to an X or a Y type would be more widespread.

To remove var from F, the refactoring tool would need to change the variable var's type to Object, since Object is the base class for all classes thus would be a common storage type for Ws and Zs. Everywhere that var is used in any source files, it must be cast to the appropriate type, eliminating the benefit of compile time type checking.

Removing var from G is even more difficult, since a primitive type such as int does not derive from Object, the refactoring tool must find a way to store var for G. Java provides a series of immutable types that store the primitive types, i.e. java.lang.Integer. To access the variable's value, "var" would be replaced with "var.intValue()". To set var, the tool would create a new instance of the Integer class, and assign it to var. Since multi-threaded programming is encouraged, any operation that replaces an atomic action with a series of steps must be considered very carefully. In this instance, the assignment of a value to var is atomic, but now includes the creation of an Integer object followed by an assignment. Object instantiation is not atomic, therefore it might be necessary to wrap the assignment of var in an appropriate synchronized block. Synchronized blocks are computationally very expensive, and should not be used indiscriminately. Therefore, an automated tool should not remove the variable var from G as a result of pushing var from B to A. Instead, a refactoring tool might recommend renaming the variable var in G.

The analysis of pushing a field into a child class is also colored by the scope of the field. The first factor is the original scope of the variable var in each of the subclasses where var was removed. It can be no less protective than the least protective scope. For instance, if the variable var in B has public scope, then the variable var in A must also have public scope. The scope must also be sufficient to allow the child classes to access var. The scope of var depends on how dispersed the classes that consider A to be an ancestor are. If all the children of A are located in the same package, then var can have package or default scope. If even one subclass of A is in a different package, then the scope of the variable var must be protected.

If some parent class of A defined a variable var with appropriate scope, and some child class of A does not define a variable var, then the refactoring becomes more complex for both Smalltalk and Java.

The issues related to types arose during the development of the JRefactory tool, an open source refactoring tool for Java. This tool can be downloaded at http://users.snip.net/~aseguin/chrisdown.html. It implements the simplest version of this refactoring where the scope of the variable is changed to protected and only fields where the type is identical are removed. An interesting future research project might be to decide what level the variable based on the number of object casts were required per file.

A number of other refactoring tools and manual refactoring processes are described in the literature. Opdyke[4] described a set of refactorings where the transformations were proved to change the structure but maintain the behavior. This was further described by Foote and Opdyke[1]. Fowler[2] describes a set of steps to perform various refactorings in Java. While sufficient for manual transformation, these steps are insufficient by themselves to automate the refactorings. Korman[3] describes a refactoring tool for finding and replacing duplicated code for the Java programming language. Roberts, Brant, and Johnson[5] describe a refactoring tool for Smalltalk. Seguin[6] describes a previous version of the JRefactory tool that moves classes between packages.

1. REFERENCES

[1] Brian Foote and William F. Opdyke, "Lifecycle and Refactoring Patterns that Support Evolution and Reuse." *Pattern Languages of Program Design*, vol 1, ed. James O. Coplien and Douglas C. Schmidt, Addison-Wesley, 1995.

[2] Martin Fowler, *Refactoring*, Addison-Wesley, 1999.

[3] Walter Fred Korman. *Elbereth: Tool support for refactoring java programs*. Master's thesis, University of California, San Diego, 1998.

[4] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[5] Don Roberts, John Brant, and Ralph Johnson, "A Refactoring Tool for Smalltalk", *Theory and Practice of Object Systems*, 3(4), 1997.

[6] Christopher Seguin, "Package Refactoring in Java from UML Class Diagrams", *OOPSLA 1999 supplement*, page 69-70, 1999.