Refactoring Tools

By Don Roberts and John Brant

One of the largest barriers to refactoring code has been the woeful lack of tool support for this critical area of software development. In languages where refactoring is part of the culture, such as Smalltalk, much can be attributed to powerful environments that support many of the features necessary to refactor code. Even there, though, the process has been only partially supported until recently and most of the work is still being done by hand.

Refactoring with a Tool

Refactoring with automated tool support has a much different feel to it than manually refactoring with the protection of a test suite. Even with the safety net in place, refactoring by hand is still time consuming. This simple fact will always prevent programmers from making refactorings that they know they should, simply because they cost too much. By making refactoring as inexpensive as adjusting the format of your code, cleanup work can be done in a similar manner to cleaning up the look of your code. However, this type of "cleanup" can have a profound, positive impact on the maintainability, reusability, and understandability of your code. Kent Beck says,

> [The Refactoring Browser] completely changes the way you think about programming. All those niggling little "well, I should change this name but..." thoughts go away, because you just change the name because there is always a single menu item to just change the name.
>
> When I started using it, I spent about two hours refactoring at my old pace. I would do a refactoring, then just kind of stare off into space for the five minutes it would have taken me to do the refactoring by hand, then do another, stare into space again. After a while, I caught myself and realized that I had to learn to think bigger refactoring thoughts, and think them faster. Now I use probably half and half refactoring and entering new code, all at the same speed.

With this level of tool support for the refactoring activity, it becomes less and less a separate activity from programming. Very rarely do I say, "Now I'm programming" and "Now I'm refactoring." It's more like, "extract this portion of the method, push it up to the superclass, then add a call to the new method in the new subclass that I'm working on."Since I don't have to test after the automated refactorings, the activities flow into each other and the process of switching hats becomes much less evident, although it is still occurring.

As Martin has pointed out, Java needs tools to support this kind of behavior by programmers. We wanted to point out some of the criteria that such a tool must have to be successful. While we have included the technical criteria, we feel that the practical criteria are much more important.

Technical Criteria for a Refactoring Tool

The main purpose of a refactoring tool is to allow the programmer to refactor his code without having to retest the program. Testing is a time-consuming process even when automated and if we can eliminate that step, we can accelerate the refactoring process by a significant factor. This section briefly discusses the technical requirements for a refactoring tool that are necessary to allow it to transform a program while preserving its behavior.

Program Database

One of the first requirements that was recognized was the ability to search for various program entities across the entire program. For example, given a particular method, finding all calls that can potentially refer to the method in question. In tightly integrated environments such as Smalltalk, this database is maintained constantly. At any point, the programmer can perform a search to find cross references. This is mainly due to the dynamic compilation of the code. As soon as a change is made to any class, it is immediately compiled into bytecodes and the database is updated. In more static environments such as Java, the code is entered into text files. Updates to the database must be performed explicitly. These updates are very similar to the compilation of the Java code itself. Some of the more modern environments such as IBM's VisualAge for Java mimic Smalltalk's dynamic update of the program database.

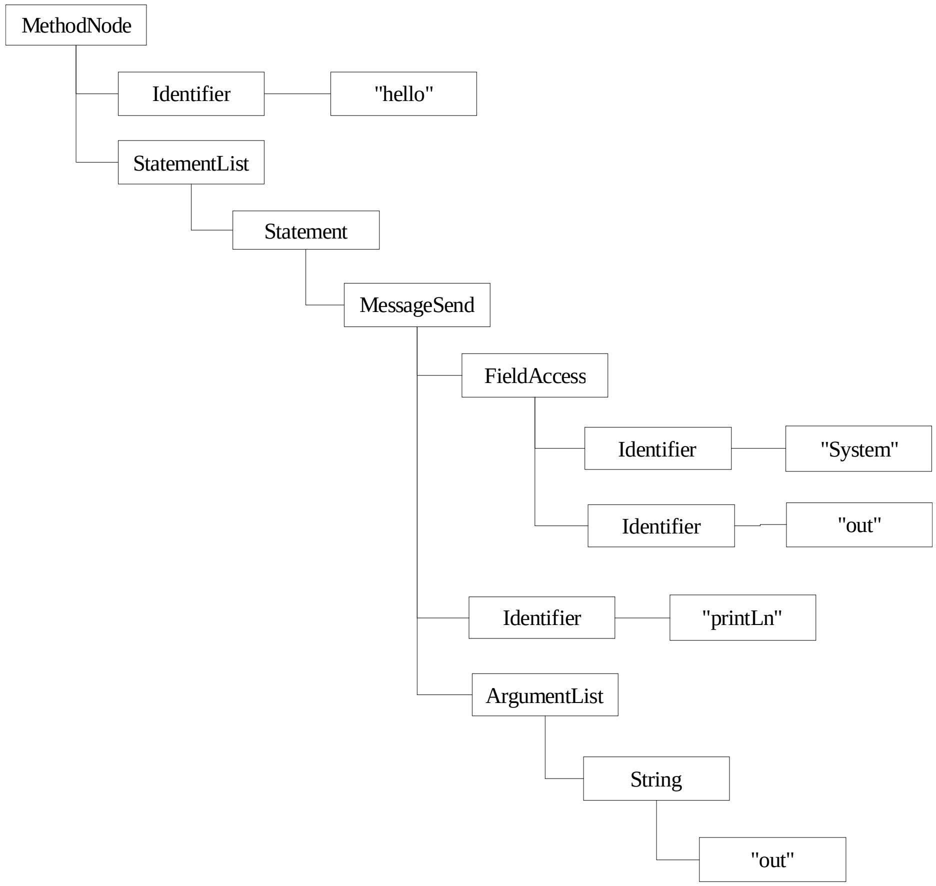
A naïve approach to this is to use textual tools such as grep to do the search. This breaks down quickly because it cannot distinguish between a variable named *foo* and a function named *foo*. Creating a database requires using semantic analysis (i.e. Parsing) to determine what "part of speech" every token in the program belongs to. This must done at both the class definition level, to determine instance variable and method definitions, and at the method level, to determine instance variable and method references.

Parse Trees

Most refactorings have to manipulate portions of the system that are below the method level. These are usually references to program elements that are being changed. For example, if an instance variable is renamed (simply a definition change), all references within the methods of that class and its subclasses must be updated. Other refactorings are entirely below the method level, such as extracting a portion of a method into its own, stand-alone method. Any update to a method needs to be able to manipulate the structure of the method. To do this requires parse trees. A parse tree is a data structure that represents the internal structure of the method itself. As a simple example, consider the following method:

```
void hello()
{
     System.out.printLn("Hello World\n");
}
```

The parse tree corresponding to this would look like:



Accuracy

The refactorings that a tool implements must reasonably preserve the behavior of programs. Total behavior-preservation is impossible to achieve. For example, what if a refactoring makes a program a few milliseconds faster or slower? Usually, this would not affect a program, but if the program requirements include hard real-time constraints, this could cause a program to be incorrect. Even more traditional programs can be broken. For example, if your program constructs a String and the uses the Java Reflection api to execute the method that the String names, renaming the method will cause the program to throw an Exception that the original did not.

However, refactorings can be made reasonably accurate for most programs. As long as the cases that will break a refactoring are identified, programmers that use those techniques can either avoid the refactoring or manually fix the parts of the program that the refactoring tool cannot fix.

Practical Criteria for a Refactoring Tool

Tools are created to support a human in a particular task. If a tool does not fit the way a person works, they will not use it. The most important criteria are the ones that integrate the refactoring process with other tools.

Speed

The analysis and transformations that are required to perform refactorings can be time consuming if they are very sophisticated. The relative costs of time and accuracy must always be considered. If a refactoring takes too long, a programmer will never use the automatic refactoring, but will just perform it by hand and live with the consequences. In the implementation of the refactorings the speed should always be considered. In the process of developing the Refactoring Browser, we have several refactorings that we have not implemented simply because we could not implement them safely in a reasonable amount of time.

An approach to consider if an analysis would be too time consuming is to simply ask the programmer to provide the information. This puts the responsibility for accuracy back into the hands of the programmer while still allowing it to be performed quickly. Quite often the programmer knows the information that is required. Even though this approach is not provably safe since the programmer can give erroneous information, the responsibility for error rests on them. Ironically, this actually make people more likely to use the tool since they are not required to rely on some program's heuristic to find information.

Integrated with Tools

In the past decade the Integrated Development Environment has been at the core of most development projects. The IDE integrates the editor, compiler, linker, debugger, and any other tool necessary for developing programs. An early implementation of the Refactoring Browser for Smalltalk was a separate tool from the standard Smalltalk development tools. What we found was that no one used it. In fact, we did not even use it ourselves. Once we integrated the refactorings directly into the Smalltalk Browser, we used them extensively. Simply having them at your fingertips made all the difference.