
Why Every Smalltalker Should Use the Refactoring Browser

Don Roberts, John Brant, and Ralph Johnson

Introduction

Refactoring software is as much a part of the Smalltalk culture as the browser itself. As long as people have been writing Smalltalk programs, they have been re-writing Smalltalk programs. Because of this, Smalltalk programmers enjoy the most mature environments and class libraries in existence. One of the benefits of dynamic environments such as Smalltalk is that refactoring the code can be done quickly. Since there is no need for the programmer to worry about types or how many files will have to be recompiled, there is less inertia to overcome when a program could be improved by refactoring.

Refactoring is essential to the development of well-designed software. Ward Cunningham espouses the idea of “architectural substitution” in Episodes, his software development pattern language.^[PLoPD2] The idea is to change the architecture of a system if a new function does not fit well with the existing architecture. In his pattern language, this is not necessarily accomplished at the exact time that the function is added to the system. Quite often, the function must be added expediently to meet a deadline, and the system refactored later when there is a little headroom in the schedule.

In a recent column, Kent Beck states:

The best way I know to find a good design is to first get something, anything running (by which I mean producing the right answers), and then rearranging the code so it communicates clearly and satisfies the cardinal rule of good design, “say everything once and only once.” By not designing on speculation, I am able to focus on the critical parts of the design much more carefully, and I am much less likely to add extra features to the design that bring costs without benefits. The result is that I code faster and simpler.^[Beck96]

This “evolutionary” process of system design will yield a design that is flexible enough to handle the common changes that the particular system must deal with without adding too much flexibility, which unnecessarily increases the mental load that programmers must deal with when extending the system in the future.

In this model of software development, each time a change or extension is made to system, the programmer considers whether code is being duplicated or if a change to the design of the system would facilitate the change under consideration. If there is a way to simplify the change by refactoring the system, the refactoring is performed. For example, if a new algorithm is being added to a complex data structure, one approach is to add a piece of the algorithm to every node that makes up the data structure so each node “knows” how to execute algorithm on itself. This is typically the initial approach taken. This approach makes sense if new types of nodes are being added often and new algorithms are rarely added. However, if the types of nodes are fairly static, and new algorithms are introduced

regularly, it is more appropriate to encapsulate the algorithms in separate objects. This is an example of the Visitor pattern ^[GOF95]. Usually, systems are initially implemented with the algorithm distributed its parts, and later are refactored into Visitors as it becomes apparent that many algorithms are being implemented on the structure.

Smalltalkers have a particular manner of programming. This manner typically involves browsing the code that already exists in the system before and during the process of writing additional code. The Smalltalk environment, in particular the browser, has been designed to facilitate this type of programming behavior. However, refactoring programs is also a portion of this programmer behavior, but tool support for refactoring programs has been sadly lacking. Therefore, experienced Smalltalk programmers have developed mental scripts to perform refactorings. For example, here is the script for renaming a method:

- 1) Using the browser, perform an “all senders” on the method to be renamed.
- 2) Change the name of the method and accept it.
- 3) For each sender in the “all senders” browser, replace the old message send with a call to the new method name. If there are multiple implementers of the method, you have to decide which of the calls refer to the method being renamed, and which refer to the other method.
- 4) Delete the original method (since accepting in step 2 makes a copy with the new name).
- 5) Run the appropriate unit tests to make sure you didn't miss anything. This is especially critical in the case of multiple implementers, since it is very easy to overlook a call that should be renamed or to rename a call that should not have been renamed.

As you see (or already know from first-hand experience) this is a tedious process. It seems highly unusual that in an environment with as much programmer support as the Smalltalk environment, no one has developed a tool to assist with this process.

The Refactoring Browser

We have developed such a tool, the Refactoring Browser, shown in Figure 1, which is freely available at our web site at <http://st-www.cs.uiuc.edu/~brant/refactoringBrowser>. The Refactoring Browser integrates automatic refactorings with the standard program development environment. This article shows how to use the Refactoring Browser to refactor your code quickly and eliminate the testing step that is necessary when refactoring manually.

Create New Subclass	Abstract Class Variable
Delete Class	Push Down Class Variable
Rename Class	Pull Up Class Variable
Add Instance Variable	Rename Method
Delete Instance Variable	Delete Method
Rename Instance Variable	Push Down Method
Push Down Instance Variable	Push Up Method
Pull Up Instance Variable	Rename Temporary Variable
Abstract Instance Variable	Move Temporary to Innermost Scope
Add Class Variable	Convert Temporary to Instance Variable
Delete Class Variable	Extract Code as Method
Rename Class Variable	

Table 1 - Refactorings implemented by the Refactoring Browser

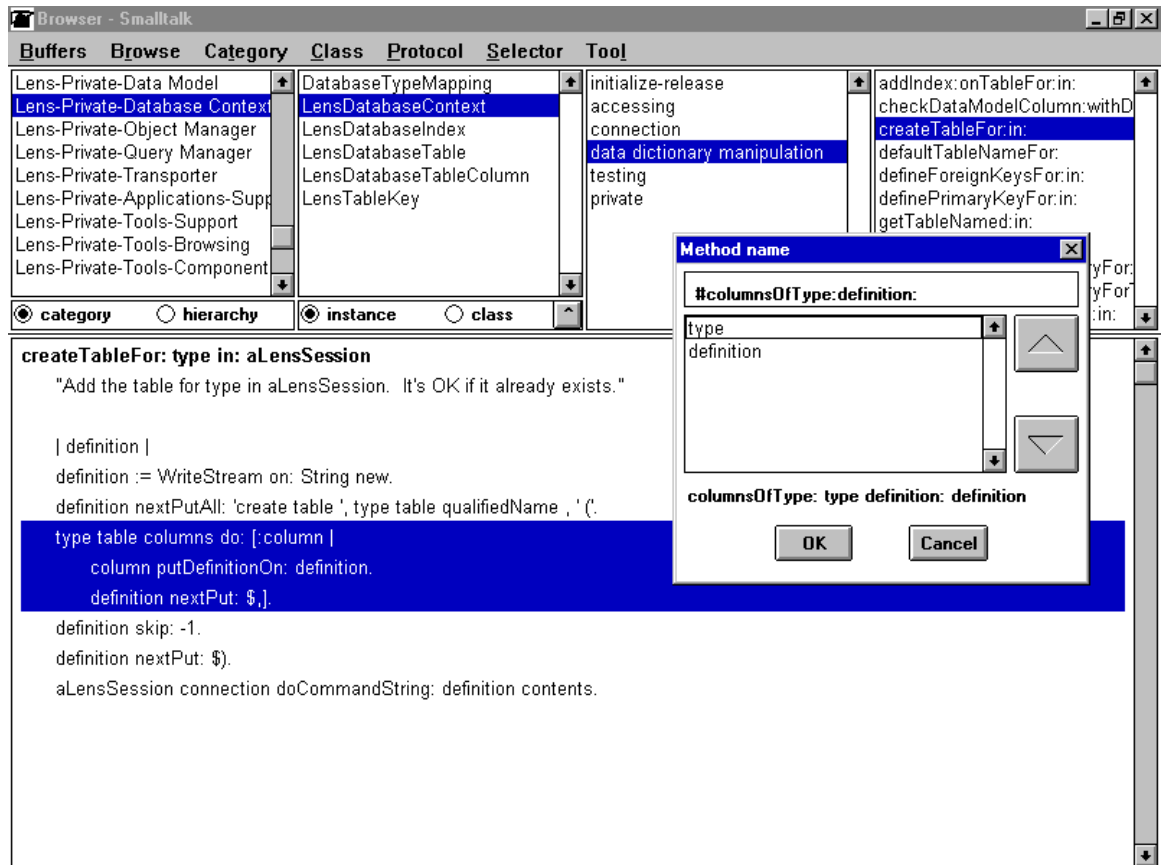


Figure 1 - Screenshot of the Refactoring Browser

The Refactoring Browser is currently available for VisualWorks 2.5, Envy/Developer 3.01 and IBM VisualAge 3.0. It is an enhanced browser. Several of the enhancements are user interface enhancements that streamline commonly performed operations. The key enhancements, though, are automatic refactorings. The refactorings in Table 1 are implemented in the current version of the Refactoring Browser.

What does the Refactoring Browser do for you? It provides many of the common refactorings that Smalltalk programmers typically perform manually. These refactorings are low-level, but since the behavior of the program is preserved after each transformation, they

can be composed to create design-level changes. Since these refactorings are automated, they can be performed both quickly and safely. Tools such as this remove the major barriers that keep programmers from adjusting the design when it becomes obvious that it is inadequate.

In the next section, we will briefly describe each of the refactorings implemented in the Refactoring Browser and discuss why they are safe. To demonstrate the power of automated refactorings, we will show an example of a design-level change that can be accomplished using the Refactoring Browser.

The Refactorings

Each of the refactorings in the Refactoring Browser tests certain preconditions to ensure that the transformation is safe. This section will describe the preconditions of each of the refactorings. Even though we specify these preconditions and check that they hold, these checks *can* be defeated. For example, when attempting to find all the senders of a method, there is no way to detect if a program constructs a selector by concatenating strings together and then executing a **perform:** on the result converted to a symbol. Therefore, although we go to great lengths to try to make the refactorings safe, if your code uses one of these techniques, you will have to use your best judgment whether a given refactoring is safe within your system, or not. Quite often, you will want the Refactoring Browser to perform a transformation, even if it isn't safe, and change the "unsafe" cases by hand.

Create New Subclass

The standard Smalltalk tools support creating a new subclass. The only checks are that the class name is legal. However, the standard Smalltalk browser does not support inserting a class into the middle of a hierarchy. The Refactoring Browser allows a new subclass to be inserted by allowing the user to specify which of the subclasses of the original class will become subclasses of the inserted class.

Add Instance/Class Variable

These refactorings are extremely simple; they only check that the variable being added has a legal name and that a variable with the same name does not already exist in the scope of the new variable.

Delete Instance/Class Variable, Delete Method, and Delete Class

All of the deletion refactorings are similar. The only check is that the element being removed is no longer referenced. It is possible for a system to defeat this check in several ways. For example, using `instVarAt:` to reference an instance variable, or only referring to a class or method within an external file that is loaded at runtime. For example, a system might have a configuration file that is loaded on startup. This file might contain symbols that are performed to initialize the system. There is no way to correctly detect if these methods are not referenced, since the references occur outside of the Smalltalk image. If your system uses techniques such as these, don't blindly trust the delete refactorings.

Rename Instance/Class/Temporary Variable and Rename Class

These renaming refactorings are also very similar to each other. Each checks that the new name will not conflict with existing elements within the same scope. If the new name is valid, it then finds every reference to the element and renames the reference. Again, if it is impossible to find some references (e.g., `instVarAt:`), these refactoring could break your code.

Rename Method

This rename was not lumped in with the other renames because methods with the same name can exist in the same scope. This is known as polymorphism and is one of Smalltalk's most powerful features. The problem with renaming is best illustrated with an example. Suppose you have a drawing package and have implemented the `add:` method on your `Drawing` class. You decide that it should be named `addFigure:`. However, when you do an all senders on the `add:` method, you are amazed to find hundreds of calls to `add:`, only a fraction of which actually refer to your method.

Therefore, the rename method refactoring has two cases. If there is only a single implementer of a method, that method is renamed along with all calls. If there are multiple implementers, the only way to make sure that the renaming is safe is to rename *all* of the implementers. Of course, the Refactoring Browser will prompt the user for confirmation before taking such drastic action.

Abstract Instance/Class Variable

The Abstract Instance/Class Variable refactorings do two things. First, they create accessors for the variable, if they don't already exist. Second, they replace every direct reference to the variable with a call to either the getter or the setter. This allows code that was not written in the accessor style to be quickly converted. The code to detect accessors looks for methods that simply set or assign to the particular variable, regardless of the name of the methods. It only detects simple getters and setters, so techniques such as lazy initialization will elude this detection and new, simple accessors will be generated.

Push Down Instance/Class Variable

The Push Down Variable refactoring takes an instance or class variable, removes it from the class it is defined in and defines it in all of the subclasses of the original class that refer to it. This is allowed if there are no references to the variable in the original class. Class variables have the additional restriction that only one subclass references it. One thing to be aware of is that if instances of the class or its subclasses exist, these variables will become nil, possibly changing the behavior of an already executing program.

Pull Up Instance/Class Variable

The Pull Up Variable refactoring takes an instance or class variable, defined in a subclass removes it from the class it is defined in and defines it in the original class. In this refactoring, instance variable and class variables have two different requirements. In the case of instance variables, if an instance variable with the same name is defined in any sibling classes, that variable is also removed. In the case of class variables, if a class variable with the same name is defined in any sibling classes, the refactoring is illegal and not per-

formed. This is because what was two separate pieces of storage would be merged into one.

Push Down/Push Up Method

The Push Down and Push Up Method refactorings remove a method from the current class and define it in all of the subclasses or the superclass of the current class, respectively. The push down is legal only if the current class is an abstract class and no subclasses perform a super send of this method. To check if a class is abstract, the Refactoring Browser checks if there are any methods defined as **subclassResponsibility** or if there are no references to the class. The Push Up Method refactoring is legal as long as none of the class's superclasses define the same method or if the superclass defines the method and is abstract. In this case, we copy the method defined in the abstract superclass down into all of its subclasses that don't define it and then move the original method up into the abstract superclass.

Move Temporary to Innermost Scope

The Move Temporary to Innermost Scope refactoring moves a temporary to the variable list of the innermost block that encompasses its entire lifetime. This refactoring allows the Smalltalk compiler to generate an optimized block, rather than a full block. Creating an optimized block is an order of magnitude faster than creating a full block.

Convert Temporary to Instance Variable

The Convert Temporary to Instance Variable refactoring removes a temporary from the current method and defines it as an instance variable in the current class. This transformation is fairly rare, but is a technique for eliminating parameters to methods being used internally within a class, and is also key to implementing Kent Beck's *Method Object* pattern [Beck96b]. This refactoring is legal as long as an instance variable with the same name does not already exist within the scope of the original variable and the variable is written before being read. This refactoring can be defeated if the method is recursive, since each recursion of the method generates a new variable, but an instance variable would be common across all recursions.

Extract Code as Method

The Extract Code as Method Refactoring allows the user to select a portion of a method and create a new method with that code as its body and replace the original code with a self send of the new message. There are many cases where this refactoring will be unable to extract the code and a complete discussion of them is beyond the scope of this article. Basically, you can only extract code that would stand alone as a complete method body. This refactoring will examine the code being extracted and determine which parameters and temporaries from the original method must be passed as arguments. The user is prompted for a method name with the appropriate number of keywords for the number of arguments being passed. If a method with identical structure to the extracted code exists, the users will be prompted if he or she wants to use it instead.

Creating Abstract Classes

One common design change that occurs in Smalltalk systems is to introduce an abstract class. These classes usually represent domain-level concepts, but they are often first discovered in the process of eliminating duplicate code.

Here is a typical scenario. You create a concrete class for your system. You create another concrete class. However, during the implementation of the second class, you realize that the two classes share some common code and instance variables. At this point, you should create an abstract class that collects the common parts of the two concrete subclasses. Even in the traditional Smalltalk environment, this is tedious.

There are three conceptual steps to creating an abstract class. They are:

- 1) Create an empty superclass above the concrete classes.
- 2) Find the common instance variables and move them into the new superclass.
- 3) Find the common methods or parts of methods in the concrete classes and move them into the new superclass.

While these steps are apparently simple, there are several subtleties that arise in each step (Those who have done this manually, know what they are).

Although creating a new class in Smalltalk is simple, inserting a class into the middle of an inheritance hierarchy is not. The programmer must manually find all of the classes that are to be subclasses of the new class and change their definitions and reaccept them.

Finding the common instance variables is relatively straightforward, except when instance variables that represent the same concept in the subclasses have different names. In this case, the programmer must choose which name is the better one and rename the other instance variable and of its references.

Step 3 can be quite difficult. Instance variables tend to have similar names, but methods with identical functionality can have widely different names depending on what the programmer was thinking when he implemented it. Also, methods are often identical except for a small portion. If the nonidentical portion is factored out into a separate method, the remaining, common method can then be pushed up into the superclass.

Example

By way of example, consider the development of a framework for playing board games. Suppose that the first application developed was a program to play checkers. In that application, the class representing a move is defined as:

```
Object subclass: #CheckersMove
  instanceVariables: 'player from to' ...

CheckersMove>>applyMoveTo: aBoard
(aBoard pieceAt: from) isKing
  ifTrue:[ ...
... checking logic ...
```

```
aBoard clearPosition: from.  
aBoard addPiece: to color: player
```

Furthermore, suppose that the second application is a program to play Go¹. The class representing a move is defined as:

```
Object subclass: #GoMove  
  instanceVariables: 'color position' ...  
  
GoMove>>applyMoveTo: aBoard  
  (aBoard pieceAt: position) isEmpty  
  ifFalse: [ self error: ...].  
  aBoard addPiece: position color: color
```

A small example such as this one may seem contrived, but I have seen code similar to this in many existing systems. Quite often the various classes were developed independently and duplicated code was written.

It is clear that there should be an abstract class, **Move**, that these two classes inherit from. To perform this refactoring using the Refactoring Browser, we would perform the following steps:

- 1) Use the “Create New Subclass” refactoring to create a new subclass of object named **Move**. When prompted, select the classes **GoMove** and **CheckersMove** to be subclasses of the new class.
- 2) Since a **CheckersMove** requires two positions and a **GoMove** requires one position, we can pull up one of the position instance variables into the superclass, however, the instance variable all have different names. Therefore, we will apply the “Rename Instance Variable” refactoring to rename the “position” instance variable in the **GoMove** class to “to”. This also renames all of the references within the class.
- 3) Now that both of the subclasses have a common instance variable defined, namely “to”, we can apply the “Pull up Instance Variable” refactoring to move the common instance variable into the Move class.
- 4) The **applyMoveTo:** methods for both **GoMove** and **CheckersMove** are very similar. The only difference is the first few lines. We can apply the “Extract Code as Method” refactoring to pull this varying code into another method named **validateMoveFor:**. This refactoring determines that the method requires one argument and allows us to enter a keyword selector that takes the appropriate number of arguments. Here is the code for these methods after the extraction:

```
CheckersMove>>applyMoveTo: aBoard  
  self validateMoveFor: aBoard.  
  aBoard addPiece: to color: player
```

```
CheckersMove>>validateMoveFor: aBoard  
  (aBoard pieceAt: from) isKing  
  ifTrue: [ ...  
  ... checking logic ...  
  aBoard clearPosition: from.
```

¹ In the game of Go, pieces are just added to the board and never moved unless captured.

```
GoMove>>applyMoveTo: aBoard
self validateMoveFor: aBoard.
aBoard addPiece: to color: color
```

```
GoMove>>validateMoveFor: aBoard
(aBoard pieceAt: to) isEmpty
ifFalse: [ self error: ...].
```

- 5) At this point, you can see that the **applyMoveTo:** methods are identical. Therefore, we will apply the “Push Up Method” refactoring to push the **applyMoveTo:** method into the Move class. When we do this, this Refactoring Browser prompts us that there are structurally identical methods in other subclasses, do we want to remove them? The answer is yes.

The above process is repeated for all of the duplicate code we can find. In the end, we are left with an abstract class that factors out all of the duplicated code in the two concrete subclasses.

Why Every Smalltalker Cannot Use the Refactoring Browser

Since I'm sure you are convinced that every Smalltalker should use the Refactoring Browser, why *can't* every Smalltalker use it? The answer is simply that it is not available on every platform. We developed the Refactoring Browser first for VisualWorks Smalltalk. Many people told us they needed it for ENVY and IBM Smalltalk, so we made a version of those platforms, also. We created these versions by building a portability layer, the main part of which is our own parser. Therefore, the main effort of porting this system to another dialect of Smalltalk goes into porting the GUI.

We believe strongly that for refactoring to become integrated into the software development life cycle, it must be integrated into the standard development tools. Our interests lie in developing new and better refactorings, therefore, it is highly unlikely that we will port the refactoring browser to additional environments. However, we know that there are experts (and vendors) who would like to see the Refactoring Browser exist for their particular brand of Smalltalk. Therefore, we strongly encourage those who would like to see a port, to port it themselves. We will provide technical support and as much assistance as we can, especially with the internals of the refactorings.

It is our opinion that every platform (and language, for that matter) should have a tool such as this to support programmers. We hope that some of you will help up make that dream a reality.

References

- [Beck96] Beck K. Make it run make it right: Design Through Refactoring. *The Smalltalk Report*,. 6(4):19–24, January 1997, edited by John Pugh and Paul White. SIGS Publications.
- [Beck96b] Beck K. *Smalltalk Best Practice Patterns, Volume 1: Coding*. Prentice-Hall, 1996.
- [GOF95] Gamma E, Helm R, Johnson R, Vlissides J, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [PLoPD2] Cunningham W. Episodes: A pattern language of program development. In *Pattern Languages of Program Design 2*, edited by J.M. Vlissides, J.O. Coplien, and N.L. Kerth. Addison-Wesley, 1996.