# A Refactoring Tool for Smalltalk

Don Roberts, John Brant, and Ralph Johnson

University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Ave.
Urbana, IL 61801
e-mail: {droberts,brant,johnson}@cs.uiuc.edu

## Abstract

Refactoring is an important part of the evolution of reusable software and frameworks. Its uses range from the seemingly trivial, such as renaming program elements, to the profound, such as retrofitting design patterns into an existing system. Despite its importance, lack of tool support forces programmers to refactor programs by hand, which can be tedious and error-prone. The Smalltalk Refactoring Browser is a tool that carries out many refactorings automatically, and provides an environment for improving the structure of Smalltalk programs. It makes refactoring safe and simple, and so reduces the cost of making reusable software.

## 1. Introduction

Improving an object-oriented system often involves both redesign and extension. Programmers often do not distinguish between these two activities and intertwine them freely. We have found it useful to perform improvements by first transforming the design while preserving the program behavior, and then extending the better designed system [OBHS86][Cas91][Cas92]. This approach allows the designer to validate the new design separate from the extended functionality. Furthermore, if the designer uses automatic tools to transform the design while ensuring that program behavior is preserved, then the extensions are the only possible source of additional error and the only portion of the system that must be validated[1].

Behavior-preserving transformations are known as *refactorings* [Opd92]. Refactorings are changes whose purpose is to make a program more reusable and easier to understand, rather than to add behavior. Refactorings are specified as parameterized program transformations along with a set of preconditions that guarantee behavior preservation if satisfied. Figure 1 gives an example refactoring for adding a new class. This would be one of the first refactoring steps to create an abstract class.

A closely related field of research is that of object-oriented database schema evolution [BK87][Kim90][PS87][WP91][DZ91]. Many of the manipulations that are studied in schema evolution are similar to the refactorings that we are studying. A major difference between the two areas is that schema evolution must deal with the existence of "live" instances of evolving objects and therefore focuses much more on the data component of the objects. Our research does not consider live instances, but focuses much more on the manipulations of the methods of the objects.

Much of the prior work on refactoring has focused on developing a small, primitive set of refactorings that are provably behavior-preserving. These primitive refactorings often seem trivial, but they possess three important properties:

[1] Since the refactoring is behavior-preserving, if there were errors in the original code, there will be errors in the refactored code.

Name: AddClass

Parameters:

| | | |
|---|---|---|
| className | : | String |
| superclass | : | Class |
| subclasses | : | Set of Class |

Preconditions: $\forall c \in \text{classes}(\mathit{Program}).\mathit{className} \neq \text{name}(c)$
$\wedge\ \forall c \in \mathit{subclasses}.\text{superclass}(c) = \mathit{superclass}$

Transformation:

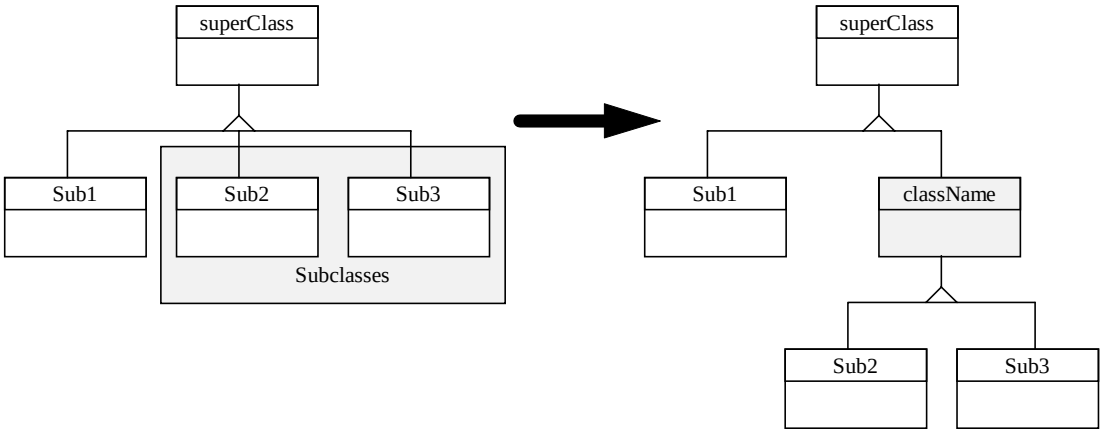


**Figure 1 - The AddClass Refactoring for Smalltalk**

1. They can be completely automated. This frees programmers from having to perform mundane restructuring tasks manually.

2. They are provably correct. This ensures that the restructuring process will not introduce new errors into the system. This is especially important in refactorings that affect code in many places, such as renaming methods and variables.

3. More complex refactorings can be created by composing primitive ones. Since each primitive refactoring preserves the behavior of the program, the entire composition is itself behavior-preserving.

At first glance, simply restructuring a system seems to be of limited utility. However, there are several areas of software development that can be aided by refactoring. One area is in program comprehension. Complex systems are often difficult to understand because they are more complicated than they need to be, with duplicated code and methods that perform more than one task. By using refactorings to reorganize the code, a programmer can gain better understanding as he factors out the various abstractions and eliminates redundancy. In fact, often important abstractions are discovered only after the code has been rearranged to make it simpler. Kent Beck refers to this phenomenon as “talking programs” [Beck96]. Abstract classes are a classic example of this. Often in the program development process, the designer finds that code is being duplicated in several classes. By creating a common superclass and moving the common methods and instance variables into it, the common abstraction often becomes evident. This syntactic manipulation reveals a design-level insight.

Even non-behavior-preserving changes can be aided by refactoring. Quite often a change can be made much more easily if the code to be changed can be brought into one place (class,

**Instance/Class Variable Refactorings**

add variable
rename variable
remove variable
push down variable into subclass(es)
pull up variable from subclass(es)
create accessors for a variable
change all variable refs to accessor calls
(abstract variable)

**Class Refactorings**

create new class
rename class
remove class

**Method Refactorings**

add method
rename method
remove method
push down method into subclass(es)
pull up method from subclass(es)
add parameter to method
move method across object boundary
extract code as method

**Table 1- Refactorings that have been automated**

method, etc.). By first localizing the code to be changed, the change can be made much more accurately since the programmer does not have to search through the entire system to determine which parts are affected by the change. An example of this is using the Strategy pattern to localize an algorithm within a single class, and then replacing the strategy object with a different one which encapsulates a different algorithm [GHJV95].

Since refactoring occurs at all levels within the software development life cycle, the ability to perform refactorings automatically is crucial to software evolution. This is especially true with the advent of design patterns. Due to the relatively recent development of design patterns, few existing programs use the flexible designs typified by them. Adding these designs to existing software can be a tedious process. Refactorings simplify this process by automatically handling the details of the code [TB95].

## 1.1 The Refactoring Browser

The goal of our research is to move refactoring into the mainstream of program development. The only way that this can occur is to present refactorings to developers in such a way that they cannot help but use them. To do this, the refactoring tool must fit the way that they work. This goal imposes the following design criteria on refactoring tools:

- **The refactorings must be integrated into the standard development tools**. In Smalltalk, the standard development tool is the browser. Initially, we simply added menu items to the browser for each refactoring. We eventually implemented an entirely new browser, with many enhancements, but even this new browser looks very similar to the standard system browser.
- **The refactorings must be fast**. Smalltalk programmers are used to being able to immediately see the results of a change. This is due to the dynamic nature of the Smalltalk environment. Therefore, refactorings that take a long time to perform an analysis will not be used. If a refactoring takes too long, a Smalltalk programmer will just do it by hand and live with the consequences.
- **Avoid purely automatic reorganization**. In Smalltalk, names are very important since that is one of the fundamental ways of determining what a class, method, or variable is used for. Automatic reorganization tools that have to create entities must name them. These names typically do not have any meaning in the problem domain and serve to obfuscate the code. Whenever we have to name something, we always prompt the user to provide a name.
- **The refactorings must be reasonably correct.** The programmers must have a degree of trust in the transformations. In a reflective environment such as Smalltalk, any change to the system can be detected by the system. Therefore, it is possible to write programs that depend on the objects being a particular size, or that call methods by getting a string from the user and calling **perform:** with it. Therefore, it is impossible to have totally correct, nontrivial refactorings. However, the refactorings in our system

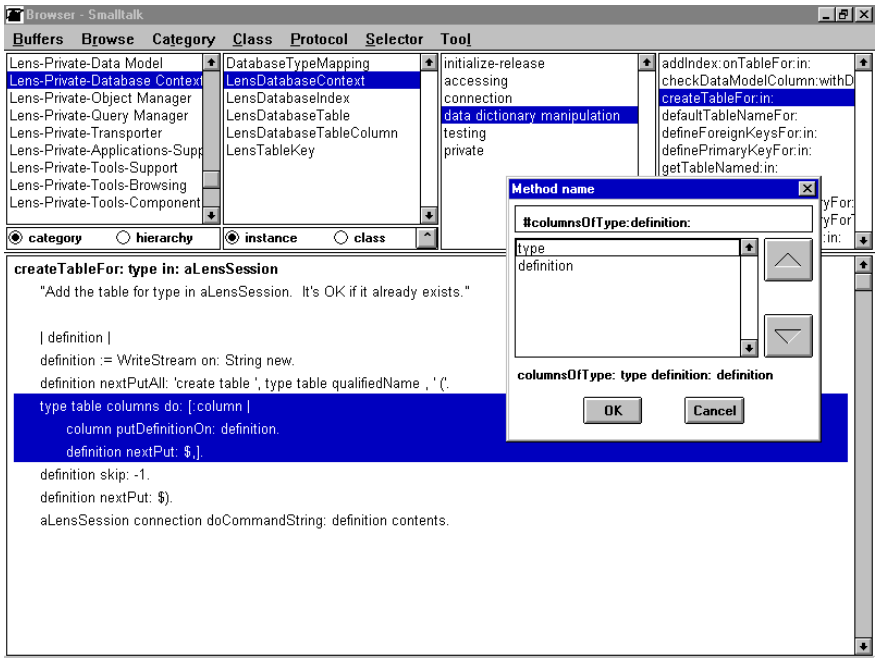


**Figure 2 - Screenshot of Refactoring Browser during extract code as method refactoring**

handle most Smalltalk programs, but if a system uses reflective techniques, the refactorings will be incorrect.

A screen shot from the Refactoring Browser is shown in Figure 2. It currently implements the refactorings presented in Table 1. Several of these refactorings could be decomposed into several more primitive refactorings, but we are seeking a set that programmers actually use, rather than a minimal set.

To demonstrate the usefulness of the Refactoring Browser, we present four case studies of problems that have arisen in working Smalltalk systems and show how they can be easily solved with automatic refactorings.

# 2. **Renaming** translateBy:

One, seemingly simple, problem that arose involved renaming methods. In release 4.0 of ParcPlace Smalltalk, both **GraphicsContext** and most subclasses of **Geometric** defined methods named **translateBy:**. Although these methods shared a common name, their semantics were markedly different. Sending **translateBy:** to a **Geometric** created a new object that is offset from the original by the argument (a point). However, sending the same message to **GraphicsContext** did not create a new instance of **GraphicsContext**, but changed its state so that entities drawn on it were offset by a particular amount.

This situation was clearly undesirable. Therefore, in release 4.1 of ParcPlace Smalltalk, the method defined in **Geometric** and its subclasses was renamed to **translate*d*By:**, a name that more accurately reflects its semantics. Of course, all of the references to the new methods were updated within the image. The problem arose when converting existing applications from the original version of the image to the new version. All of the senders of **translateBy:** in the application had be examined by a programmer and the type of the receiver had to be determined. Since there may be many such locations, this was a very tedious, error-prone process as several hundred Smalltalk programmers that were given the job of upgrading existing applications can attest to.

In Smalltalk, the type of an object is effectively the set of messages that it responds to. Any object that responds to the same set of messages as another can replace it regardless of their inheritance relationship. This is unlike C++ where the inheritance hierarchy is also the type hierarchy. Therefore, in Smalltalk, renaming a method in a class is effectively changing the type of all of the instances of that class. This can get complicated when two classes that are not related by inheritance have methods that share a common name. It cannot be easily determined if these classes are used polymorphically, or if the names are purely coincidental. If they are polymorphic, then both methods must be renamed if either one is. If they are not polymorphic, then either method can be renamed independent of the other, but determining which method is being referred to at each call site in the program is difficult. This is the case with **translateBy:**.

## Solution

To port an application that uses the **Geometric** classes from Smalltalk R4.0 to R4.1, we must locate all of the sites that send **translateBy:** message to a subclass of **Geometric**. These sites must be modified to send the **translatedBy:** message. This takes the following steps with the Refactoring Browser:

1. Using the *rename method* refactoring, rename the **translateBy:** method in the **Geometric** class to **translatedBy:**. This will rename the method in the **Geometric** class and all of its subclasses. (All of the subclasses are renamed to preserve the common protocol that all **Geometrics** understand.)

2. Exercise the application on a test suite. This step is necessary because the *rename method* refactoring is a *dynamic refactoring*, which will be discussed in Section 6.3.

3. File out the application and file it into an Objectworks 4.1 image.

After completing these steps, all of the references to the **translateBy:** method in the **Geometric** classes will be renamed while the references to the method of the same name in class **GraphicsContext** will remain untouched.

Renaming methods at first seems to be one of the simplest of program transformations, but in a dynamically typed environment such as Smalltalk, where the type of an object is the set of methods that it responds to, renaming a method is changing the type of the object. Two objects that are polymorphic may not be after a method is renamed. Therefore, the Refactoring Browser prompts the user whenever the situation is ambiguous. There are two situations that may arise whenever a renaming is specified:

1. The user wishes to rename all of the methods with a given name in the image. In this case, all of the implementers and all of the senders are renamed. This can be done using only the functions in the base image for determining all senders and all implementers of the method. All of the classes must be checked for conflicts with the new name.

2. The user wishes to rename only a subset of the implementations of a method. This procedure requires runtime analysis of the program and is only behavior-preserving if the classes that implement the method are not used polymorphically with the class in which the renaming is occurring. If the classes are used polymorphically with the class in which the renaming occurs, then renaming the method in one class and not the others will make them no longer polymorphic, breaking the program.

# 3. Converting Values into ValueHolders

When ParcPlace released VisualWorks, creating user interfaces became much simpler. However, the granularity of what was considered a model in VisualWorks changed. Before, models were typically large, monolithic objects that implemented the entire functionality of the application. Under VisualWorks, every widget (e.g. button, input field) has its own model, which are typically very simple, containing a single value. These simple models are known as

```
foo
        "Getter method for instance variable foo"

        ^foo

foo: aValue
        "Setter method for instance variable foo"

        foo := aValue
```

**Figure 3 - Accessor methods for *foo***

*holders*. Holders are the same as variables except that they notify their dependents whenever their value changes. Therefore, to convert existing applications to use the new interface tools that are in VisualWorks, many of the values that are stored in domain models must be converted into **ValueHolders**.

## Solution

Converting values to **ValueHolders** by hand is tedious. Using the Refactoring Browser to perform this transformation is extremely simple and quick.

1. Use the *add method* refactoring to create a method that returns the initialized value holder.

2. Perform the *abstract instance variable* refactoring on the instance variable you wish to convert.

This refactoring does several things. First, it creates accessor methods for the variable if none exist. For example, if the variable is named *foo*, it creates the two methods shown in Figure 3. It then replaces all direct references to the variable with either the getter method, if it is a reference, or the setter method, if it is an assignment.

Now, to make the variable a value holder, simply change the accessor methods (using the browser) to use the value holder method created in step 1. The code will look like Figure 4. The **foo** and **foo:** methods act the same as the original methods. The **fooHolder** methods gives programmers access to the valueHolder object which can be passed to the user interface or any other object than needs to be dependent on the value of foo.

```
foo
        "Getter method for instance variable foo"

        ^self fooHolder value

foo: aValue
        "Setter method for instance variable foo"

        self fooHolder value: aValue

fooHolder
        ^foo isNil
                ifTrue: [foo := nil asValue]
                ifFalse: [foo]
```

**Figure 4 - Transformed accessor methods**

A common strategy for refactoring is to localize the scope of change. A major reason that maintenance is so expensive is that adding a single feature typically requires changes throughout a program. If the scope of the change can be limited to as few classes and methods as possible, errors are less likely to occur and are easier to track down. In this case, the variable accesses are scattered throughout the class and its subclasses. The automatic refactoring allows the programmer to rapidly localize all of these references into two methods where the necessary changes can be made simply. If this refactoring occurs often, the procedure can be entirely automated, eliminating the need for the programmer to manually change the accessor methods.

# 4. Creating an Abstract Class

One of the most common design changes that occurs when developing reusable software is creating an abstract class from several concrete classes [JF88]. The abstract class typically represents a common abstraction that several classes share, hence the name. Finding the correct abstractions for a particular problem is difficult. Typically, several concrete classes must be created before the correct abstraction becomes apparent [OJ93]. Therefore, abstract classes are created from a particular set of concrete classes by migrating variables and functions from the concrete classes into the new su-

perclass. When done correctly, this will not change the behavior of the system .

For example, the original code for the Refactoring Browser had a navigational part that allowed the user to choose classes and methods to edit. This was commonly displayed at the top of the browser. In addition to the navigational component, there was also a selection component that the user used to pick several classes and selectors so that actions could be performed on them (e.g., running a lint rule on the selected classes). This selection component was originally developed for a testing tool and was retrofitted into the Refactoring Browser. Since it was originally developed for a different application, it did not fit into the same hierarchy as the navigational component. As a result, many items were duplicated, and the selection component lacked some features implemented by the navigational component. This was fixed in later versions of the Refactoring Browser, by creating an abstract class that represented these commonalities.

## Solution

Creating an abstract class by hand takes several steps, most of which can be easily automated [Opd92][Opd93][OJ93]. The following steps create an abstract class using the Refactoring Browser (they do not necessarily have to be performed in exactly this order):

1. Create an empty superclass between the concrete classes and their current superclass. This is by using the *create new class* refactoring. This refactoring allows the users to insert an empty class into the hierarchy by specifying its superclass and which of the subclasses of the original superclass will be its subclasses.

2. Use the *rename instance variable* refactoring. Rename common instance variables to share a common name by using the rename instance variable refactoring. Often classes have instance variables that represent the same concept but have different names. This is especially true if the classes were developed by independent developers.

3. Move common instance variables from the concrete subclasses into the superclass. This is accomplished by using the *pull up instance variable* refactoring.

4. Methods with similar functions should be renamed to have identical names. This is accomplished with the *rename method refactoring*.

5. Often, methods in the subclasses will be identical except for a few pieces of code. These methods will have to have the differences factored into smaller methods. The *extract code as method* refactoring allows the user to select a portion of a method and have it extracted as a new method. Figure 5 shows an example of making two methods identical by factoring out differences. Before the extraction, every subclass must override the entire **getController** method, duplicating much code. After the extraction, the subclasses only need to define the one portion of the method that is actually different across subclasses.

   There are several complications that can arise when attempting to extract code into a separate method. The problems occur in code that uses variables local to the original method. If the variables are simply referenced and not assigned, they can be passed as parameters to the new method. If the variables are assigned and referenced exclusively within the extracted code, they can become temporaries of the new method. However, if the variables are assigned within the extracted code and are referenced in the original method, then the code cannot be extracted. Since Smalltalk passes all parameters by reference, there is no simple way to get the new values of the temporary variables back to the original method.

   In the Refactoring Browser, the code to be extracted is analyzed and the user is prompted to enter a selector with the appropriate number of arguments. If the

***Before Extraction:***

```
View>>getController
        controller == nil ifTrue: [self setController: StandardController new].
        ^controller

SpecialView>>getController
        controller == nil ifTrue: [self setController: MySpecialController new].
        ^controller
```

***After Extraction:***

```
View>>getController
        controller == nil ifTrue: [self setController: self defaultControllerClass new].
        ^controller

View>>defaultControllerClass
        ^StandardController

SpecialView>>defaultControllerClass
        ^MySpecialController
```

**Figure 5 - Extracting code to make similar methods**

code cannot be extracted, the user is informed and the process is aborted.

6. Move identical methods from the subclasses into the superclass using the *pull up method* refactoring.

# 5. Retrofitting the Visitor Pattern

One of the key features of design patterns is that they balance a set of forces. The visitor pattern is a classic example of force-balancing. The visitor pattern is used in situations where an algorithm must be performed on a complex, heterogeneous data structure (e.g., generating machine code from a parse tree in a compiler) [GHJV95].

One solution is to have each node in the data structure implement the portion of the algorithm that deals with itself. For example, an assignment node in a parse tree could generate code for itself by asking its rvalue to generate code for itself and then generate the code to assign the results returned from the code fragment to the storage location represented by the lvalue. This solution is appropriate if the types of algorithm that operate on the data structure are fairly static. Extending existing algorithms to deal with additional types of nodes is simply a matter of adding the appropriate method to the new node for each algorithm. However, to add a new algorithm, a new method must be added to *every* node type. If adding new algorithms is a rare occurrence, then this is the correct way to implement the algorithm.

The visitor pattern addresses the situation where the types of nodes are fairly constant, but adding new algorithms or refining existing algorithms occurs relatively frequently. In the visitor pattern, algorithms are performed by a separate object known as a *visitor*. Each node type implements the **accept**: method, which takes an argument that is a visitor object. It then calls the method of the visitor that corresponds to the type of the receiver. For instance, an assignment node would implement the **accept: aVisitor** method by sending the **acceptFromAssignmentNode: self** method back to the visitor. This callback method will implement the details of the algorithm. In this solution, each algorithm is entirely encapsulated in a single visitor object. However, adding a new type of node to the data structure will force the programmer to add a method for the new node type to *every* visitor object.

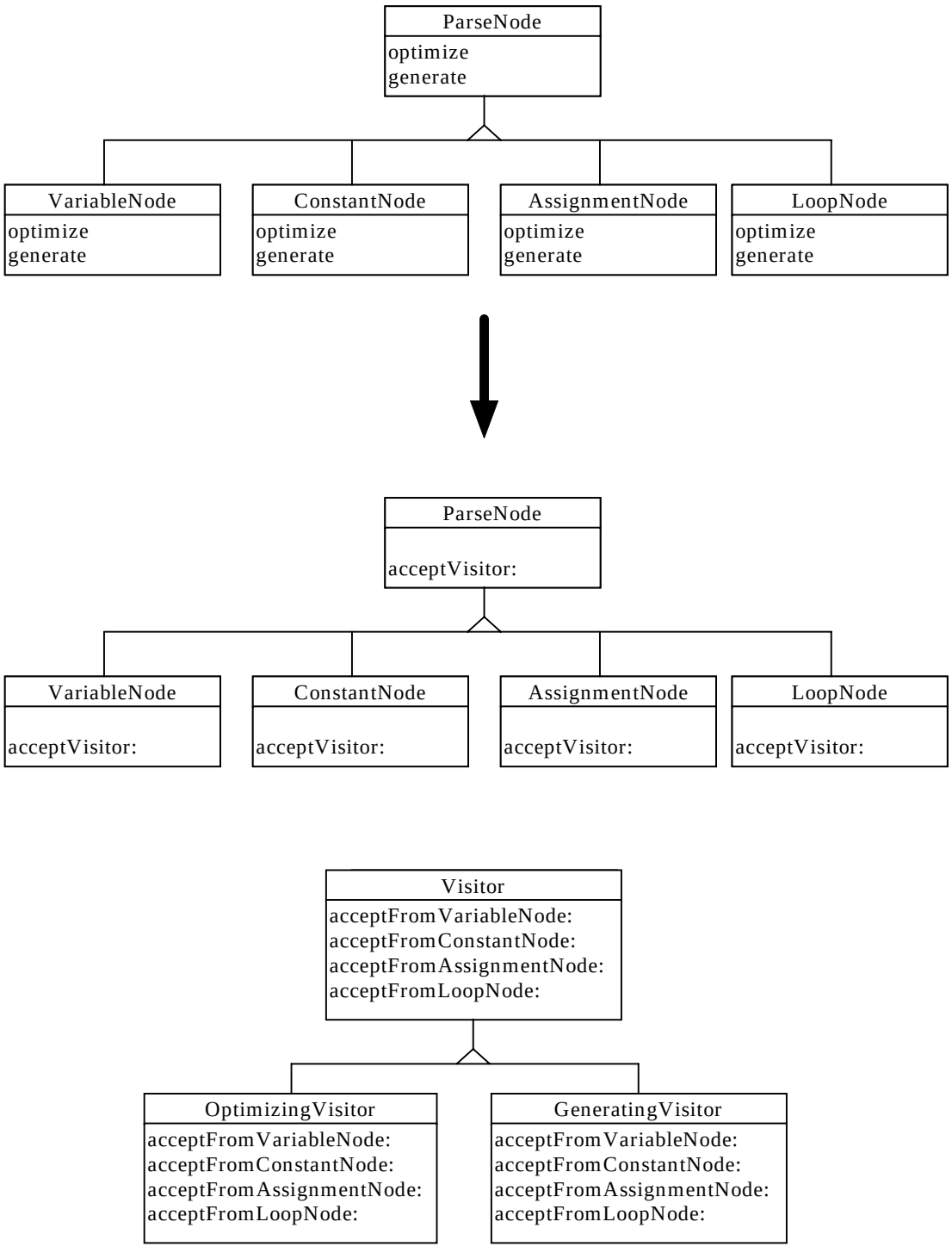


**Figure 6 - The visitor pattern**

```
ParseNode>>optimize

        self optimizeWithVisitor: OptimizingVisitor new
```

**Figure 7 - New *optimize* method**

```
AssignmentNode>>optimizeWithVisitor: aVisitor

        aVisitor acceptFromAssignmentNode: self
```

**Figure 8 - New *optimizeWithVisitor:* method**

In the initial design process, it is usually not apparent which is the correct method of implementation. Only after the system has been running and several extensions have been made will it become apparent whether to use a visitor or a distributed algorithm. Therefore, the ability to easily refactor a design from one method to the other is essential.

## Solution

Figure 6 shows a classic example of a situation that is a candidate for the visitor pattern. This is a parse tree from within a compiler that implements both an **optimize** and a **generate** method. The steps to implement the visitor pattern are as follows:

1. Create an abstract visitor class with the *create new class* refactoring.

2. Add methods for each type of node to be visited with the *add method* refactoring. The body of the method corresponding to the root of the node hierarchy will just be **self subclassResponsibility** since this is an abstract class. The body of the methods corresponding to each subclass will simply call the method corresponding to its superclass. In the parse tree example, the method **acceptFromAssignmentNode:** will have **^self acceptFromParseNode:** as its body. This arrangement handles the case where the **optimize** method is not implemented on every node. It does this by simulating the methods that would be called by inheritance.

3. Extract the body of the **optimize** method to the **optimizeWithVisitor** method using the *extract code as method* refactoring.

4. Now all of the **optimize** methods have the same body (i.e**., self optimizeWithVisitor**) and can be consolidated. Use the *pull up method* refactoring to combine them into one **optimize** method in the **ParseNode** class.

5. Next, use the *add parameter to method* to add a parameter containing an **OptimizingVisitor** to all of the **optimizeWithVisitor** methods. You must specify a default value to passed. In this case, the default value is **OptimizingVisitor new**. This refactoring causes the **optimize** in class **ParseNode** to look like Figure 7.

6. To move the code into the visitor, use the *move method across boundary* refactoring to move the body of each **optimizeWithVisitor:** method into the appropriate visitor method. Figure 8 shows the code for the **AssignmentNode**.

7. Finally, since the **optimizeWithVisitor:** has nothing specific to optimization, use the *rename method* refactoring to change its name to **acceptVisitor:**.

Adding the code for the generating visitor is similar to above process. However, the **acceptVisitor:** will already be implemented.

# 6. Implementing the Refactoring Browser

This section briefly discusses some of the implementation details of the Refactoring Browser. One of the key design criteria was to create a tool that could refactor real Smalltalk programs with the same interactive style that Smalltalk developers are used to. Therefore, one of our guiding principles is that automatic refactorings should take less time than performing them by hand. To achieve this goal, analyses that take longer than a few seconds were unacceptable. This led to the framework for analysis and transformation discussed in this section.

Another of the key design criteria is that we assume that there is an intelligence directing the refactoring process. We do not attempt to automatically refactor code. It has been our experience that systems that perform purely automatic code reorganization are of limited utility in production programming environments since they typically have no understanding of the problem domain. For example, simply creating abstract classes everywhere there is duplicated code rarely yields meaningful designs. Especially since the algorithms assign names to the new classes in some automatic fashion. This ignores the fact that abstract classes, while eliminating duplicated code, should also represent some abstraction in the problem domain. Divining this abstraction from the results of an automatic reorganization is often more difficult than finding it in the original code due to the obfuscation introduced by the naming system.

However, the algorithms used in automatic reorganization systems such as Ivan Moore's [Moore96], can be valuable when augmented with user input. For instance, rather than automatically generating names, the system could prompt the user for names for the new abstractions that it discovers. Our approach is similar to this in that we provide a tool that will search for places in your program where code is duplicated, or unused, and point them out. This allows a human to make the final call whether or not the code should be consolidated and to provide a semantically meaningful name.

One of the key features of Smalltalk that allows us to perform the sorts of analyses and source code transformations necessary for refactoring is that it has *reflective* facilities, that is, the ability for the language to examine and modify its own structures. For instance, the Refactoring Browser examines the context stacks of executing programs to determine runtime properties such as callers of a particular method or cardinality relationships between components and aggregates. Refactoring can be performed in the absence of reflective facilities, but then requires a separate, metalanguage that can be used to manipulate the original program. Having a single language simplifies the process.

Reflection is currently a hot topic of research, but from the standpoint of this project we are not concerned with many of the theoretical aspects of reflection but are simply users of reflection. Specifically, we used three reflective facilities when implementing the browser

```
'varName := `@anything' -> 'self varName: `@anything'

'varName' -> 'self varName'
```

**Figure 9 - Matching patterns to abstract instance variable *varName***

## 6.1 Parse Tree Matching

The fundamental facility that any program transformation system requires is access to an easily manipulable representation of the program. VisualWorks provides access to the Smalltalk compiler and its intermediate structures such as parse trees. The source code transformations that refactorings require are much easier to implement as parse tree to parse tree transformations rather than string to string transformations. Other dialects of Smalltalk do not provide this level of access to the system, which is one of the major reasons the Refactoring Browser has not been ported to other environments.

Originally, the refactorings were implemented by using visitors that would iterate the nodes of a parser tree and convert one parse tree into another depending on the refactoring being performed. As we developed more refactorings, developing the enumerators became tedious. To simplify this process, we developed a tree matching system.

The matching system uses specially annotated parse trees that are created by extending the Smalltalk language. There are annotations that support matching variables, message sends, any object (A variable followed by a list of message sends), a statements, and a list of statements. Once a match is found, a block of code is executed for that match.

The parse tree matcher can also be used to modify the parse trees. In addition to supplying an annotated Smalltalk expression to match against, another expression specifies the conversion once a match is found. For example, to rename the **size** message send to **length**, you only need to supply a search string such as **`@anyObject size** and its conversion string **`@anyObject length** (where **`@anyObject** is a

wildcard that can match any series of message sends to a variable). Once the parse tree has been modified, it is parsed back into Smalltalk code and compiled. Figure 8 shows the abstract instance variable refactoring as implemented using the tree matcher.

### 6.2 Compilation Framework

Arbitrary program transformations are not behavior-preserving. To ensure that the behavior of a program does not change, every refactoring has associated with it a set of preconditions that must be met in order to apply the transformation. Often these preconditions are as simple as "no class with this name already exists in the system." The static properties of the program must be analyzed to determine if these preconditions are satisfied before performing a refactoring.

Fortunately, the compilation framework in VisualWorks already provides many of the checks that are necessary for correct refactorings. The framework uses classes as first class objects to manage variables, methods, and class hierarchies. Many of the analysis and transformational parts of a refactoring simply require a couple of message sends to a class. For example, to add a method to a class, you must check that the method name does not conflict with a method defined in the class or its superclasses. This can easily be checked by sending the **canUnderstand:** message the class with the selector in question as an argument.

Since we reused many of the existing static checks present in the compilation framework, the refactoring framework is much simpler than it would be if we had implemented all of these checks ourselves.

### 6.3 Dynamic Analysis and Method Wrappers

Some preconditions of refactorings are not simple to compute from the static program text. With Smalltalk's dynamic typing, determining the type of a particular variable can be extremely difficult and time consuming. Moreover, there are analyses that defy static analysis in any language, such as cardinality relationships between objects.

Since, performing these types of analyses statically is difficult or impossible, the Refactoring Browser computes some of the preconditions by performing the analysis dynamically rather than statically. By observing the actual values that occur, the Refactoring Browser can perform correct refactorings. As an example, consider the *rename method* refactoring. To correctly rename a method, all calls to that method must be renamed. This is difficult in an environment that uses polymorphism to the extent that Smalltalk does. Smalltalk also allows dynamically created messages to be sent via the **perform:** message. If an application uses this approach, any automatic renaming process has the potential of failure. Under these conditions, guaranteeing the safety of a rename is impossible.

The Refactoring Browser uses *method wrappers* to collect runtime information. These wrappers are activated when the wrapped method is called and when it returns. The wrapper can execute an arbitrary block of Smalltalk code. To perform the *rename method* refactoring dynamically, the Refactoring Browser renames the initial method and then puts a method wrapper on the original method. As the program runs, the wrapper detects sites that call the original method. Whenever a call to the old method is detected, the method wrapper suspends execution of the program, goes up the call stack to the sender and changes the source code to refer to the new, renamed method. Therefore, as the program is exercised, it converges towards a correctly refactored program.

This type of "lazy" behavior occurs quite often in many branches of computer science. CLOS and other object systems based on lisp have a lazy instance conversion scheme [Steele90]. In fact, virtual memory can even be considered a lazy allocation scheme where pages are only loaded as needed.

The major drawback to this style of refactoring is that the analysis is only as good as your test suite. If there are pieces of code that are not executed, they will never be analyzed, and the refactoring will not be completed for that particular section of code.

## 7. Future Work

The Refactoring Browser does not yet support undo. This feature is desirable in that it

further encourages the exploration of program designs by allowing the programmer to try it one way and easily change his mind. Since Smalltalk provides a continuous change history, it should be fairly simple to extend this to implement unlimited undo. More interesting, though, is allowing the programmer to undo refactorings in any order, and the system will determine if there are any later refactorings that depend upon the undone refactoring and undo them also.

We are always searching for new refactorings to add to the system. Our main resource for refactorings up to this point is William Opdyke's Ph.D. thesis [Opd92]. In general, we prefer to add primitive refactorings that can be composed into larger, more complex refactorings. This composition is a topic that we are examining most closely.

One result from composing refactorings is that in addition to having a precondition that must be satisfied, each refactoring has a postcondition that is true after the transformation. For example, in the visitor pattern, we added a parameter to the **optimize** method that was initialized with a new **Visitor** object. A postcondition of this refactoring is that the type of the variable is a **Visitor**, a fact that would otherwise require extensive analysis to determine. Keeping track of postconditions can be used to satisfy preconditions of later refactorings without requiring extensive, complex analysis of the source code. Therefore, refactorings are sometimes easier to perform in a composition than they are independently. Developing a system that incorporates these batch refactorings is currently being researched.

Another area that we are researching is using the tool to allow packages of refactorings to be released with new versions of a framework or Smalltalk image. The refactorings would be applied to old code filed into a new image. For example, the problem in section 2 would be solved by specifying the renaming refactoring. Then whenever any old application was filed into a new ObjectWorks 4.1 image, all of the references to the old message would be renamed during file in. These refactoring packages would allow the developer of the framework to provide a set of refactorings that would transform systems derived from the original framework to the new framework.

# 8. Conclusions

Refactoring is a common operation in the software life cycle and the Refactoring Browser provides automatic support for many of the common transformations that come up in Smalltalk development. The Refactoring Browser is a practical tool in that it can perform correct refactorings on nearly all Smalltalk programs. In fact, we regularly use the Refactoring Browser on itself. That is, we use the tool to refactor its own source code. Additionally, the Refactoring Browser has been used to help develop a wide range of frameworks from the HotDraw framework for graphical editors[BJ94], to financial models being developed by Caterpillar, to prototypes and models for a major telecommunications company.

The Smalltalk Refactoring Browser can be obtained by anonymous ftp at st.cs.uiuc.edu under the directory:

*/pub/Smalltalk/st80_vw/RefactoringBrowser*

or on the World Wide Web at:

*http://st-www.cs.uiuc.edu/~brant/Refactory/.*

# 9. Acknowledgments

This research funded in part by a fellowship from the Fanny and John Hertz Foundation and by a grant from the Union Bank of Switzerland.

# 10. References

[Beck96] Kent Beck. *Smalltalk Best Practice Patterns, Volume 1: Coding*. Prentice-Hall, 1996.

[BJ94] K. Beck and R. Johnson. Patterns Generate Architectures. In *Proceedings of the 8th European Conference, ECOOP '94*. Lecture Notes in Computer Science, Vol. 821, pages 139–149, 1994

[BK87] E. Blake and S. Cook. On including part hierarchies in object-oriented languages, with an implementation in smalltalk. In *Proceedings of ECOOP '87, Special Issue of BIGRE*, pp 45–54, June 1987.

[Cas91] Eduardo Casais. *Managing Evolution in Object-Oriented Environments: An Algorithmic Approach.* PhD thesis, University of Geneva. 1991.

[Cas92] Eduardo Casais. An Incremental Class Reorganization Approach, in *Proceedings ECOOP '92*, ed. O. Lehrmann. Madsen, LNCS 615, Springer Verlag, 1992, pp. 114–132.

[DZ91] Christing Delcourt and Roberto Zicari. The design of an integrity consistency checker for an object-oriented database system. In *Proceedings ECOOP '91*, ed. Pierre America, LNCS 512, Springer Verlag, 1991, pp. 97–117.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Gri91] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis. University of Washington. August 1991.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*. June/July 1988.

[JO93] Ralph E. Johnson and William F. Opdyke. Refactoring and Aggregation. In *Proceedings of ISOTAS '93: International Symposium on Object Techniques for Advanced Software*. November 1993.

[Kim90] Won Kim. *Introduction to Object-Oriented Databases*. MIT Press, 1990.

[Moore96] Ivan Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proceedings of OOPSLA'96*, 1996, pp. 235-250

[OBHS86] Tim O'Shea, Kent Beck, Dan Halbert, and Kurt J. Schmucker. Panel on: The learnability of object-oriented programming systems. In *Proceedings of OOPSLA '86*. pages 502–504. November 1986.

[OJ93] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In *ACM 1993 Computer Science Conference*. February 1993.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992. Available at: ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z

[Opd93] Ralph Johnson and William Opdyke. Refactoring and Aggregation, In *Object Technologies for Advanced Software*, LNCS 742, Springer Verlag, 1993, pp. 264-278

[PS87] D. Jason Penney and Jacob Stein. Class modifications in the GemStone object-oriented dbms. In *Proceedings of OOPSLA '87*, 1987.

[Steele90] Guy L. Steele, Jr. *Common Lisp: The Language*, Digital Press, 1990.

[TB95] Lance Tokuda and Don Batory. Automated Software Evolution via Design Pattern Transformations. In *Proceedings of the 3rd International Symposium on Applied Corporate Computing*, Monterrey, Mexico, October 1995. Also TR-95-06, Department of Computer Sciences, University of Texas at Austin, February 1995.

[WP91] Francis Wolinski and Jean-Francois Perrot. Representation of complex objects: Multiple facets with part-

whole hierarchies. In *Proceedings ECOOP '91*, ed. Pierre America, LNCS 512, Springer Verlag, 1991, pp. 288–306.