Chapter 6

Refactoring Tools

The goal of this research has always been to bring program transformation and analysis into the realm of real-world languages and programs. As part of this research, we developed the Refactoring Browser, a tool to refactor Smalltalk programs.[JBR95][RBJ97] Developing this tool has given me insight into the criteria for a succesful tool and given us a framework with which to experiment on more radical forms of program transformation tools.

6.1 The Refactoring Browser

There were at least two reasons for developing the Refactoring Browser. First, we wanted to create a *practical* refactoring tool that could be used in commercial software development. We felt that many of the ideas of evolutionary software development, such as continual refactoring and designing to the current set of requirements, would not be fully realized until we had produced such a tool. Secondly, we wanted a framework within which to test many of the newer refactoring ideas that we had. We feel that we have been successful on both fronts.

6.1.1 History

The first incarnation of the Refactoring Browser, then known simply as The Refactory, was a stand-alone tool separate from any of the other tools in the Smalltalk environment. It

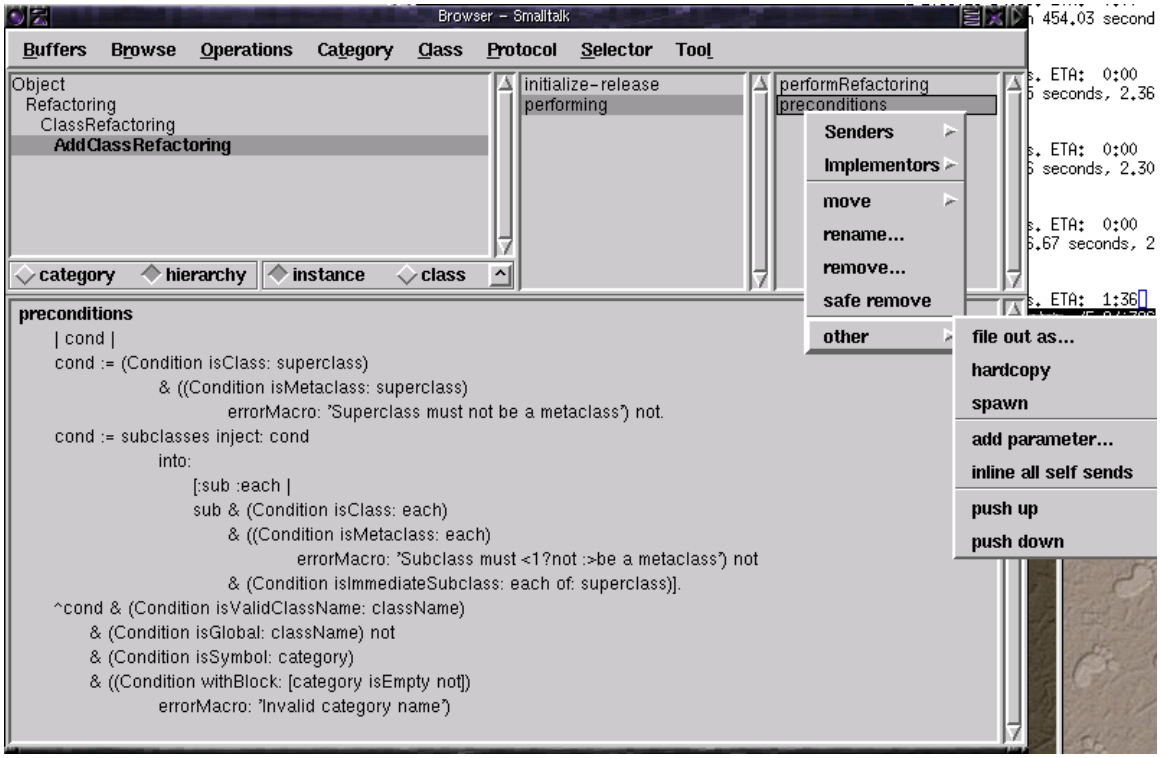

Figure 6.1: The Refactoring Browser

implemented many of the refactorings that exist in the current tool, but in a cruder fashion. While technically interesting, it was rarely used, even by ourselves. This led us to examine what the real criteria were to get programmers to use this tool. As a result, we created the Refactoring Browser.

Currently, the Refactoring Browser is being used on several commericial programming projects around the world. The tool has performed reliably in many different contexts. Programmers have begun to realize the power that even simple automated refactorings give them. Nearly all of our feedback has been positive and most of the requests have been for additional features. Figure 6.1 shows a screenshot of the commericial tool.

6.1.2 The Research Framework

The Refactoring Browser has several components within it that are useful for research in program transformation. It has a custom-built parser for the Smalltalk language. This parser can accept an extended syntax to represent meta-nodes in parse trees. These trees can then be used as patterns for the parse tree rewriter component of the Refactoring Browser. The rewriter is the engine that actually performs the source-to-source transformations that implement the refactorings. There is also a framework for program analysis. The various conditions that are checked to verify the preconditions for the refactorings are reified as objects. These objects then perform the necessary analysis to determine if the refactoring is valid. These objects facilitate the implementation of the assertion propagation that this thesis describes to eliminate the computation of various properties of programs.

6.2 Success Criteria

Early versions of the refactoring tool were hardly used. The basic functionality for performing refactorings was present, but was insufficient for ensuring the success of the tool. As we developed later versions of the tool, we were forced to examine the issues that determined whether the tool was being used or not. This section examines the criteria that we identified as necessary for the success of a refactoring tool.

6.2.1 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 time-consuming even when automated and if we can eliminate that step, we can accelerate the refactoring process significantly. The technical requirements for a refactoring tool are those properties that allow it to transform a program while preserving its behavior. This ensures that the test suite will not have to be rerun after every little change. Even in the case of dynamic refactoring, which requires

running the test suite, testing is still reduced. When refactoring by hand, the programmer makes a small change in the code, and runs the test suite, if the test fails, he corrects the error and then reruns the test suite. With dynamic refactoring, the test suite needs to only be run once. Therefore time is saved both by eliminating the updating of cross-references by hand, and by only have to run the test suite once.

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. Tightly integrated environments such as Smalltalk constantly maintains this database. At any point, the programmer can perform a search to find cross references. The maintenance of this database is facilitated by 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 lexical 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 is. 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.

Abstract Syntax Trees (ASTs) 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 ASTs.

There are more sophisticated program representations that contain more information about the program such as control flow graphs [ASU88] and program dependency graphs[FOW87, KKL+81]. However, we have found the ASTs contain all the information that we need and are created extremely quickly. This agree with what Morgenthaler found in his work.[Mor97]

Accuracy The refactorings that a tool implements must reasonably preserve the behavior of programs. Total behavior-preservation is impossible to achieve. For example, a refactoring might make 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 a program constructs a String and then uses the reflective facilities to execute the method that the String names, renaming the method will cause the program to have errors.

However, refactorings can be made reasonably accurate for most programs. Most programs are well-behaved and have requirements that are simply a set of outputs for a given set of inputs. 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. One of the techniques that we have used to increase the accuracy of the tool is to let the programmer provide information that would be difficult to compute. This does introduce the possibility that the programmer is wrong, and will therefore introduce an error into the program. What

we have found, though, is that the programmer usually has a good understanding of the program, and can easily provide the information. Another benefit is that the programmer feels that he has more control over the process and trusts the tool more.

6.2.2 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. In the implementation of the refactorings the speed should always be considered. In the implementation of 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. Another 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 the refactoring to be performed quickly.

Undo As mentioned earlier, automatic refactoring allows an exploratory approach to design. You can push the code around and see how it looks under the new design. Since a refactoring is supposed to be behavior-preserving, the inverse refactoring, which undoes the original, is also a refactoring and behavior-preserving. Earlier version of the Refactoring Browser did not incorporate the undo feature. This made refactoring a little more tentative because undoing some refactorings, while behavior-preserving, was difficult to do. Quite often I would have to go find an old version of the program

and start again. This was annoying. With the addition of undo, yet another fetter was thrown off. Now I can explore with impunity, knowing that I can roll back to any prior version. I can create classes, move methods into them to see how the code will look, change my mind and go a completely different direction, all very quickly. The necessity of an undo operation in tools that are used in an exploratory manner has been observed by other researchers.[BG97]

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. 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.

Chapter 7

The Architecture of the Refactoring Browser

As we developed the Refactoring Browser, a reusable framework for browsing and transforming code emerged. We did not explicitly design this architecture from start, but rather allowed it to evolve. This chapter describes the components of that design.

7.1 Example Refactoring

To demonstrate how the portions of the Refactoring Browser fit together, we will use the RenameInstanceVariable refactoring. If the user wishes to rename an instance variable using the Refactoring Browser, they select a menu item from the Class menu, "Rename Instance Variable...". From the resulting dialog, the user selects the instance variable to rename and then types in the new name. The browser then checks that the preconditions are valid. If they are valid, it performs the refactoring. Otherwise, it alerts the user of the illegal condition and aborts the refactoring.

7.2 The Transformation Framework

There are essentially two parts to the transformation framework, the parser and the tree rewriter.

7.2.1 The Parser

As mentioned in section 6.2.2, the parser needed to be a standard parser for Smalltalk augmented with the ability to parse a metavariable syntax. Original versions of the Refactoring Browser were only developed in VisualWorks Smalltalk, so we simply co-opted some extended syntax from the Smalltalk parser that wasn't being used, and added the necessary abstract syntax tree nodes for metavariables. The syntax was particularly cumbersome, but it worked.

As we targeted other dialects of Smalltalk, some of which did not allow access to the parser, it became apparent that we needed to write our own parser. This is more important than it seems. One of the fundamental difficulties of source-to-source transformations that Opdyke identified was the preservation of non-semantic information about the program text (e.g., comments, whitespace, formatting, etc.).[Opd92] Traditional parsing literature does not deal with maintaining these features since parsers are often the front-end of compilers. Since this information has no effect on the code, they are ignored.

Another inspiration for creating a new parser and scanner for Smalltalk was the poor design of the original. In the original, Parser was a subclass of scanner. This was evidently so they could pass information between them as instance variables. This information was also stored in a non-object-oriented fashion. For example, token classes were represented by symbols stored in one instance variable along with the value of the token stored in another. We believe that the only reason that this design persisted for as long as it did was that the syntax of the language has changed very little in the past several years.

Our criteria for the parsing component were the following:

- It must use first-class objects not only for the AST nodes, but also for the individual tokens produced by the scanner. This allows us to attach additional information to the token itself regarding the actual source that produced the token.

- It needed to exhibit good object-oriented design principles. Conceptually, a scanner is

simply a stream of token objects. Therefore, a scanner should adhere to the standard Stream protocol. A parser should be an example of the Builder pattern[GHJV95] that takes the stream of tokens and builds an abstract syntax tree.

- It had to be nearly as fast as the existing parser. Transforming large systems (like the Smalltalk image) can potentially require parsing the source for the entire system. This had to be as fast as possible.

Based on these requirements, we decided to forgo parser or scanner generators and manually create a recursive-decent parser. This was relatively easy due to the simple syntax of Smalltalk. We were successful in achieving all three of our goals. The resulting parser was only a few percent slower than the stock VisualWorks parser, and twice as fast as the stock VisualAge parser. Additionally, by removing the dependency on any one dialect's parser, we have freed ourselves to expand into other languages such as Java in future developments and research.

Probably the biggest advantage of creating our own parser was that we were able to add syntax to the language to represent entire subtrees by metavariables. By parsing this extended Smalltalk code, we can create pattern trees that are used by a tree rewriter to transform code. Since the syntax for these patterns is primarily Smalltalk, we are able to expose this capability to the user to allow them to perform their own advanced searches and replaces. The next section will discuss the matching algorithm and give some examples of matches.

For the RenameInstanceVariable refactoring, the parser must parse all of the methods that reference the instance variable being renamed. The parse trees that it produces are passed to the parse tree rewriter for transformation.

7.2.2 Parse Tree Rewriter

The parse tree rewriter takes a template in the form of a Smalltalk AST, which may contain named metavariables, and a standard Smalltalk AST, and attempts to find a subtree that matches it. Alternatively, the rewriter can take a pair of pattern trees, which share a common set of metavariables, and replace every match of the first tree with the second with all of the metavariables replaced by the corresponding subtree from the matched tree.

Every metavariable must start with the backquote (‘) character to identify it. The simplest metavariable (e.g., `‘foo`) will match any single variable. An pattern that would match any code that incremented a variable is `‘foo := ‘foo + 1`. Since both metavariables in this pattern have the same name, they must be same variable in any code that matches this pattern. So the Smalltalk code, `x := y + 1` would not match this pattern.

Usually, we want a more general match than a single variable. The `@` prefix allows a match on multiple items in a position. For example, the pattern `‘@foo` will match any expression no matter how complex. As an example, the pattern `‘@receiver add: ‘@item` will match all senders of the `add:` message.

With the syntax already discussed, only portions of statements can be matched. Often, we need to match a statement or statements. To specify a metavariable that matches a statement, use the period (`.`) prefix. If used in conjunction with the `@` prefix, the variable will match multiple statements. So, `‘.foo` will match any single statement, while `‘.@foo` will match any number of statements. The pattern

```
‘@test ifTrue: [ ‘.firstStatement.
                 ‘.@trueStatements.]
       ifFalse: [ ‘.firstStatement.
                  ‘.@falseStatements.].
```

will match any conditional statement that has the same first statement in both branches.

It possible to restrict a metavariable to only match a literal by using the `#` prefix. In

‘	Metavariable prefix
@	Match multiple items in this position
.	Match an entire statement
#	Match a literal
‘	Recurse into subtree on a successful match

Table 7.1: Metavariable Prefixes

```
name → self name
name := ‘@obj → self name: ‘@obj
```

Figure 7.1: Transformation rules for abstract instance variable name.

Smalltalk, a literal is a number, string, character, symbol, or literal array. The pattern `‘@collection at: ‘#literal` will match any code that is accessing a hard-coded location within a collection.

By default, the tree matching algorithm returns the topmost node that matches its pattern without looking at any of the subtrees of the match. So, for example, the pattern `‘@rec at: ‘@index` when matched against the Smalltalk code array at: (otherArray at: 3)) would return one match corresponding to the first at:. Obviously, in certain transformations, we will want to recurse into the subtrees of a match. To specify this, an addition backquote (‘) is prefixed to the variable. If we wanted to find *all* occurrences of the at: message, the correct pattern would be `‘‘@rec at: ‘‘@index`. Table 7.1 gives the prefix characters that identify various types of meta-nodes.

To transform code, a target tree has to be specified using the same syntax. All of the metavariables in the target tree must be present in the source tree. Figure 7.1 shows the transformation rules for abstracting all references to the variable name by replacing direct access with calls to getter and setter functions.

The rewriter is an instance of the Visitor pattern.[GHJV95] The Visitor pattern takes an algorithm that would normally be distributed across the various types of nodes in a data structure, and centralizes it within the visitor class. Each node then implements the

```
BRAssignmentNode>>match: aNode inContext: aDictionary
    aNode class == self class ifFalse: [↑false].
    ↑(variable match: aNode variable inContext: aDictionary)
        and: [value match: aNode value inContext: aDictionary]
```

Figure 7.2: Matching Code for Assignment Nodes

```
BRMetaVariableNode>>match: aNode inContext: aDictionary
    self isAnything ifTrue: [↑(aDictionary at: self ifAbsentPut: [aNode]) = aNode].
    self isLiteral ifTrue: [↑self matchLiteral: aNode inContext: aDictionary].
    self isStatement
        ifTrue: [↑self matchStatement: aNode inContext: aDictionary].
    aNode class == self matchingClass ifFalse: [↑false].
    ↑(aDictionary at: self ifAbsentPut: [aNode]) = aNode
```

Figure 7.3: Matching Code for Metavariable Nodes

accept: method that accepts the visitor as a parameter and calls a method on the visitor that corresponds to its type with itself as an argument. This is known as *double dispatching*. The rewriter first separates the nodes into argument nodes and other nodes. It then sends the match:inContext: to the node. The context that is passed to this method is the current assignment to the metavariables. Figure 7.2 shows the matching code for an assignment node. Figure 7.3 shows the matching code for a metavariable.

The transformation that must be performed to implement the RenameInstanceVariable refactoring is quite simple. The matching pattern is just the name of the variable to be rename and the rewrite pattern is just the new name. No metavariables are necessary for this particular refactoring.

In addition to implementing refactorings, we have used the searching capability of the tree matcher to create a tool to detect various types of common Smalltalk coding and style errors. By creating sets of patterns that correspond to the way that programmers commonly write the wrong thing, we are able to detect many errors that have never been found by test cases or users.

7.2.3 Conditions

Early versions of the refactorings had their preconditions hard-coded into methods. As we developed more refactorings, we began to see common tests that were required by several of them. Additionally, I began to see the value of asserting conditions to eliminate program analysis. As a result, we created Condition objects to represent the various analyses that the refactorings need to perform. These correspond to the analysis functions described in this thesis. Each Condition, when initialized is supplied with its parameters. If the condition needs to be evaluated against the program, the **execute** method is called.

Reifying the conditions in this manner allows for two important enhancements to the Refactoring Browser. First, it allows for the assertion of postconditions. By simply adding the asserted condition objects to a collection, later refactorings can see if their preconditions have been previously asserted, and avoid having to perform the analysis directly on the program. Not only can assertions be created this way, but hard-to-compute analysis functions, such as ClassOf can be either be provided by earlier refactorings, or if computed, they can be cached, resulting in a speedup. Second, it provides components that, given a good user-interface, could provide for user-defined refactorings. Users could compose new refactorings by combining the appropriate condition object together with a program transformation specified in the extended Smalltalk syntax.

In addition to simple condition objects, we created conditions to represent the logical combination of simpler conditions. By creating *and*, *or*, and *not* conditions, we are able to construct the complete precondition for a refactoring and evaluate it all at once. Figure 7.4 shows the code that creates the precondition objects for the AddClass refactoring.

For the RenameInstanceVariable refactoring, the preconditions are fairly simple. Basically, the new variable name must be valid and not be defined anywhere in the hierarchy. Figure 7.5 shows the code that performs these checks.

```
AddClassRefactoring>>preconditions
    | cond |
    cond := (Condition isClass: superclass)
            & ((Condition isMetaclass: superclass)
                errorMacro: 'Superclass must not be a metaclass') not.
    cond := subclasses inject: cond
            into:
                [:sub :each |
                sub & (Condition isClass: each)
                    & ((Condition isMetaclass: each)
                        errorMacro: 'Subclass must <1?not :>be a metaclass') not
                    & (Condition isImmediateSubclass: each of: superclass)].
    ↑cond & (Condition isValidClassName: className)
        & (Condition isGlobal: className) not
        & (Condition isSymbol: category)
        & ((Condition withBlock: [category isEmpty not])
                errorMacro: 'Invalid category name')
```

Figure 7.4: Precondition code for the AddClass refactoring

```
RenameInstanceVariableRefactoring>>preconditions
    ↑(Condition isValidInstVarName: newName for: class)
        & (Condition definesInstVar: varName in: class)
        & (Condition hierarchyOf: class definesVar: newName) not
        & (Condition isGlobal: newName) not
```

Figure 7.5: Precondition code for the RenameInstanceVariable refactoring

7.2.4 Change Manager

Another important part of the Refactoring Browser's transformation framework is the change manager. The change manager is responsible for recording which refactorings are performed. This simple function allows the implementation of features such as undo, logging, and composition. Each refactoring in the system knows how to undo itself. By keeping a list of the refactorings that are performed, the user can undo the refactorings at will. The logging of refactorings provides interesting information for our research regarding which refactorings are used most often. The change manager also knows how to deal with composite refactorings. Composite refactorings behave as atomic refactorings in terms of application and undo, but are constructed from multiple, primitive refactorings.

7.3 The Code-Browsing Framework

After the failures of the initial version of the refactoring tool, we decided that the best strategy for getting refactoring technology into the hands of real programmers was to develop a browser that eliminated many of the shortcomings of the existing browsers and include the refactorings. This "Trojan horse" approach unexpectedly led to the development of an entire framework for browsing code. This framework has been used as the front-end for such diverse software tools as profilers, style checkers, and testing frameworks. Figure 7.6 shows a graphical representation of the relationships between the classes in the code browsing framework.

7.3.1 RefactoringBrowser

The class RefactoringBrowser is a glue class whose principal responsibility is to hold onto the other components of the browsing framework. It also provides the interface for creating a new Refactoring Browser by implementing the open method on the class side. RefactoringBrowser holds onto a Navigator that displays the current item being browsed. It also holds onto one or

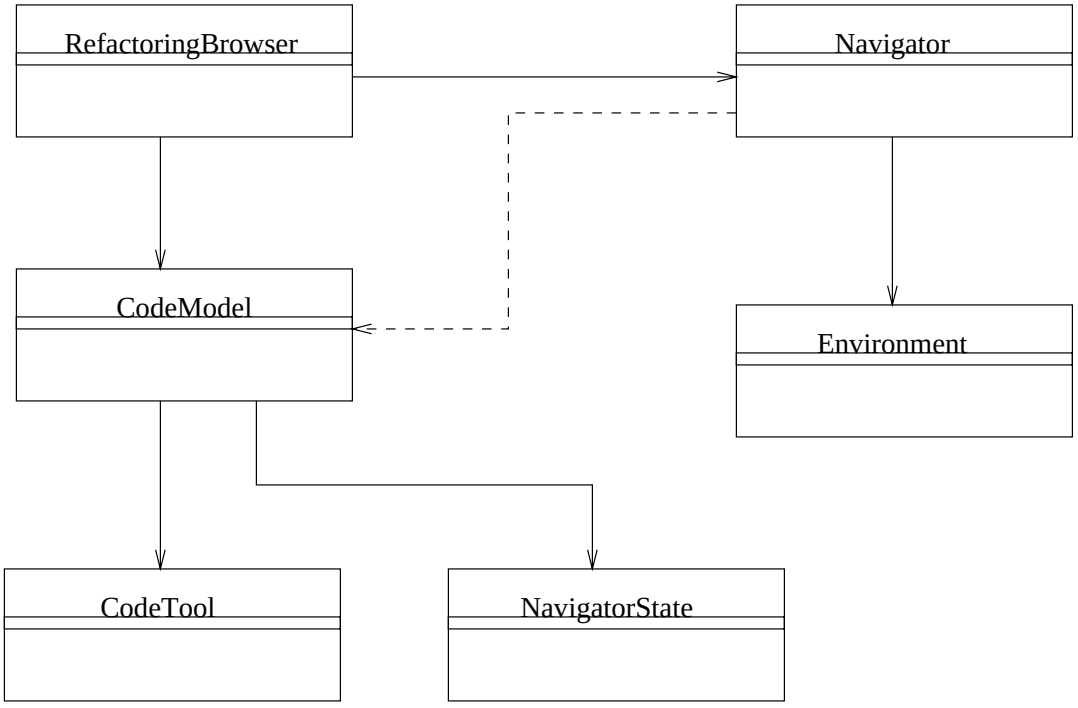

Figure 7.6: The architecture of the Refactoring Browser

more CodeModels. One of the features that we added to the standard browser is the concept of **buffers**. Smalltalk programmers that use the standard system browsers find that they must open several windows to accomplish their tasks. To alleviate this problem, we allow multiple buffers to be created within the same window. These serve the same purpose as multiple windows, but with less clutter.

7.3.2 Environments

The fundamental data structure in the code-browsing framework is known as an *environment*. An environment is an arbitrary collection of classes and methods. The common superclass of all environments is BrowserEnvironment. A new BrowserEnvironment contains all of the classes and methods of the system. All other types of environments are subclasses of LimitedEnvironment. A LimitedEnvironment has an instance variable, environment, that points to the environment that it is limited. This is an example of the Decorator pattern.[GHJV95]

We have specialized subclasses of LimitedEnvironment that represent some of the more common queries. These are CategoryEnvironment, which represents all of the classes and methods within a set of VisualWorks categories; ClassEnvironment, which represents all of the methods within an arbitrary set of classes; and ProtocolEnvironment, which represents all of the methods within a set of VisualWorks protocols. The class RestrictedEnvironment represents an arbitrary collection of classes and selectors. These environments are not usually created by sending a creation method to the class itself. There are methods on BrowserEnvironment to create the appropriate environment for a query. For instance, to create a ClassEnvironment on a set of of classes, you would use the statement: env := BrowserEnvironment new forClasses: classes.

Two additional environments, NotEnvironment and AndEnvironment provide for the logical combination of queries. All environments can be combined using the logical operators &, —, and *not*. These operations create a combination of AndEnvironments and NotEnvironments to represent the query. The semantics of the NotEnvironment is to return every class and method in the system that is not in the environment that it is placed on. We create the *or* operation by using DeMorgan's law that states $A \vee B = \neg(\neg A \wedge \neg B)$.

An environment can be used to restrict the scope of the system which is searched during various queries such as finding all senders of a message. They have also been used as filters for displaying results from a style tool. They can also be used as a basis for metrics tools. They provide methods such as numberSelectors and numberClasses that return counts of the different entities within them. As of now, refactorings ignore environments and perform their transformations and analysis on the entire system. This has primarily been in the interest of safety. It is possible to use them to restrict the scope of transformations and with the recent introduction of namespaces in several dialects of Smalltalk, we will probably use them to restrict the scope of refactorings to single namespaces.

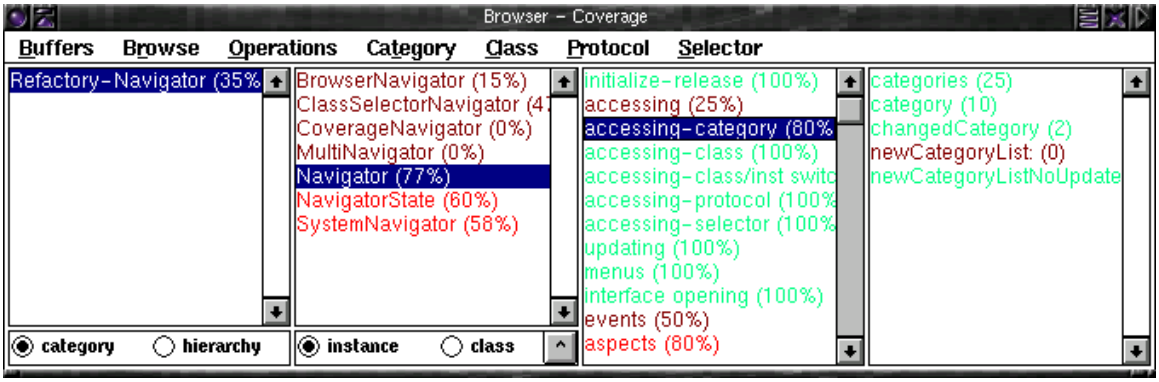

Figure 7.7: Navigator customized as front-end to a coverage tool

7.3.3 Navigator

One of the fundamental activities that occurs while browsing code is selecting which class and method to look at. The user interface to an environment that facilitates browsing is known as a *navigator*. The navigator is subclassed to change the way the tool displays the elements of the environment or to add commands to operate on the elements of the environment. Figure 7.7 shows a screenshot of a navigator that has been customized to be the front end of a coverage tool. These commands take the form of menu items that appear in the GUI. There are some standard menu items such as *Find Class...* and *Find Method...* that are standard.

7.3.4 CodeModel and NavigatorState

The CodeModel class is fairly simple. It is responsible for determining which CodeTool to use when displaying the current selection within the Navigator. It does this by being a dependent of the Navigator. Whenever the selection in the Navigator changes, the CodeModel is notified and it takes the appropriate action. It also holds onto an instance of NavigatorState. NavigatorState is an example of the Memento pattern[GHJV95]. It contains all of the information necessary to specify the state of the current selection within the Navigator. It is used to implement buffers. Whenever the user switches to a buffer, the corresponding

NavigatorState object is passed to the Navigator, which causes it to change the selection to match the one stored in the NavigatorState object.

7.3.5 CodeTools

The Smalltalk Browser has always had a particular form. It consisted of several small panes at the top that are used to find the class and possibly the method to be browsed, and a lower portion that actually displays the code. In the Refactoring Browser, this lower portion is replaced by a CodeTool. A CodeTool is an arbitrary view that appears based on the selection within the navigator. It is an example of the Strategy pattern[GHJV95]. By allowing arbitrary views, we can provide more specialized views on the code than the text-based views that were provided by the original browser. For example, when viewing a method that defines an icon, rather than displaying the literal array that encodes the icon, we display the icon itself by using an IconViewer, a subclass of CodeTool. We can also allow the user to select different views for the same entity. For example, when a class is selected, the user can view the class definition, the class comment, or a graphical representation of the hierarchy. Figure 7.8 shows an example of a code tool that displays a graphical view of the class hierarchy.

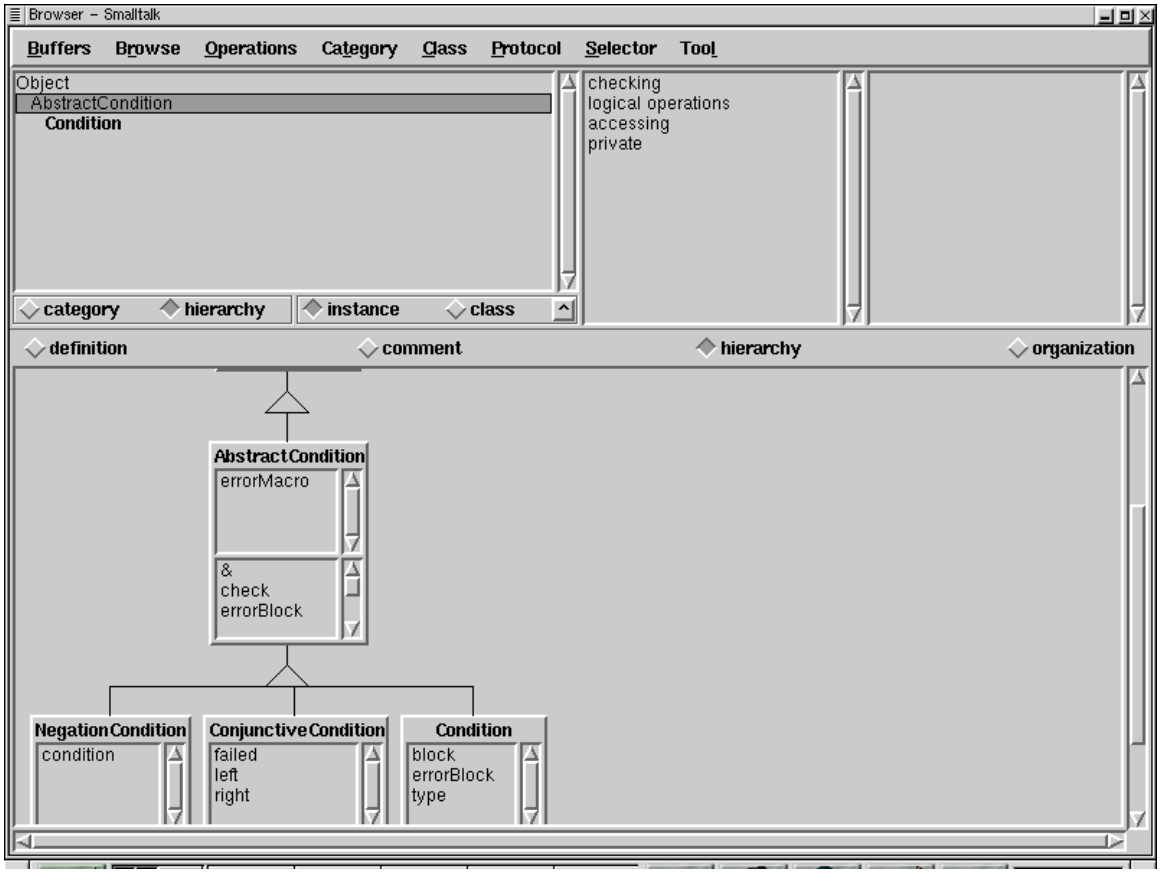

Figure 7.8: Graphical hierarchy view code tool