Rewrite Rule Editor

The rewrite rule editor allows you to create search and replace patterns that work at the methods structural level. Unlike simple string matching these patterns work at the parse tree level. This allows you to transform your code very rapidly.

The best way to learn how to use the rewrite rule editor is by an example. This section gives an example of how to convert `at:ifAbsent:` messages into `at:ifAbsentPut:` messages. In VisualWorks 2.5 an `at:ifAbsentPut:` method was defined so that code that was once written as:

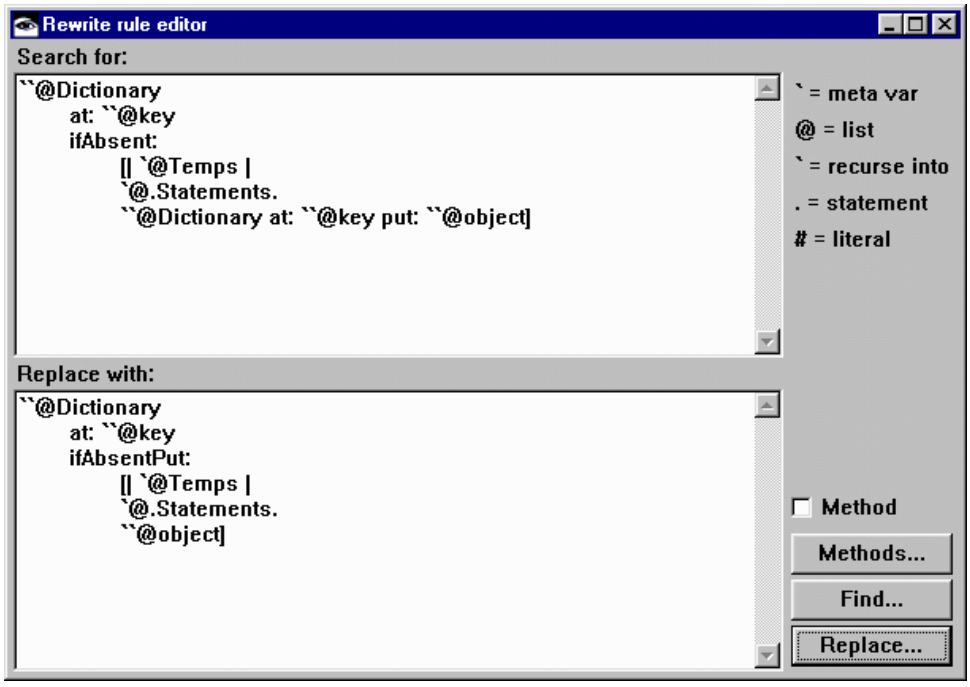
```
aDictionary at: key ifAbsent: [aDictionary at: key put: value]
```

can now be written as:

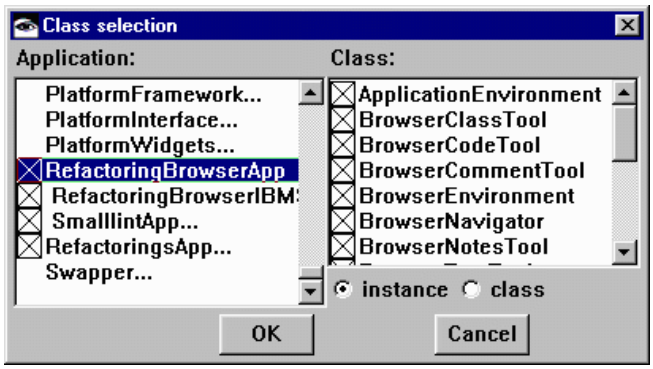
```
aDictionary at: key ifAbsentPut: [value]
```

Since the `at:ifAbsentPut:` variant more quickly describes what is occurring we would like to transform all occurrences of the old `at:ifAbsent:` variant to use the new `at:ifAbsentPut:`.

The first step in creating a rewrite rule is to select a piece of code that you wish to transform and then select "rewrite…" from the browser's menu. This will open a window with the selected text in the "Search for:" pane. You can also select the "Rewrite tool" from the ENVY pulldown menu.



The top pane of the editor is used to enter the pattern to search for. The bottom text pane is used to enter the pattern that will replace the found search pattern. The "Methods…" push button allows you to select which classes to look at when transforming the code. It opens a class selection dialog:



You will select which classes to inspect by using the pop-up menus on the category and class lists. Only classes with checks by them are selected (not the highlighted classes).

The "Find" button on the rewrite rule editor searches the selected classes for occurrences of the "Search for:" pattern, and the "Replace" button both searches and then transforms the occurrences of the "Search for:" pattern. After transforming the code, it opens an inspector on the methods so that you can confirm that it executed correctly before accepting the methods.

The next step in the at:ifAbsent: example is to convert all of the varying parts of the patterns into meta-variables. Each meta-variable must begin with a ` character. Immediately following the ` character, other character can be entered to specify what type of node this meta-variable can match. After all the special character have been entered, you must then enter a valid variable name. The special characters currently supported are listed in the following table:

Character	Type	Comment	Examples
`	recurse into	whenever a match is found, look inside this matched node for more matches	"``@object foo" matches foo sent to any object, plus for each match found look for more matches in the ``@object part
@	list	when applied to a variable node, this will match a literal, variable, or a sequence of messages sent to a literal or variable when applied to a keyword in a message, it will match a list of keyword messages (i.e., any message send) when applied with a statement character, it will match a list of statements	"| `@Temps | ..." matches list of temps "`@.Statements" matches list of statements "`@object" matches any message node, literal node or block node "foo `@message: `@args" matches any message sent to foo
.	statement	matches a statement in a sequence node	"`.Statement" matches a single statement
#	literal	matches only literal objects	"`#literal" matches any literal (#(), #foo, 1, etc.)

When matching a statement or statements, you will need to match a whole sequence node that includes both the temporaries in that sequence node as well as the other statement nodes.

For the `at:ifAbsent:` example, we need to make meta-variables for the dictionary, key, and the object added to dictionary, since these are the only parts that can vary. Since all of these objects can be a single variable or a sequence of messages sent to a variable, we need to use the @ character in their names. Also, since we should look for more matches inside each node, we also need to use the ` character. This results in the ``@ prefix added to each name (the first ` signifies a meta-variable). Using this prefix we can enter the search rule:

```
''@aDictionary at: ''@key
    ifAbsent: ['']@aDictionary at: ''@key put:''@value]
```

Now you might want to check that you have enter the pattern correctly. After selecting the "Methods..." to look at, you can "Find" all methods that match the pattern. This will open a browser on the methods that match the pattern. If you get a message that appears in the search pane, then you probably have an error in your formula. Correct it and rerun the "Find" command.

After looking at the found methods, you might notice that not all variants are present. In particular, there are some variants of the `at:ifAbsent:` message that have temporary variables and other statements in the ifAbsent: block. For example, a method such as:

```
@aDictionary at: @key
    ifAbsent:
      [self doSomeWorkHere.
       aDictionary at: @key put: @value]
```

is not found since the `doSomeWorkHere` message send doesn't match anything.

We need to extend our "Search for:" pattern to include other statements in ifAbsent: block. In addition to other statements, we should also allow for temporary variable definitions. We then modify our pattern to be:

```
``@aDictionary at: ``@key
      ifAbsent:
            [| `@Temps |
            ``@.Statements.
            ``@aDictionary at: ``@key put: ``@value]
```

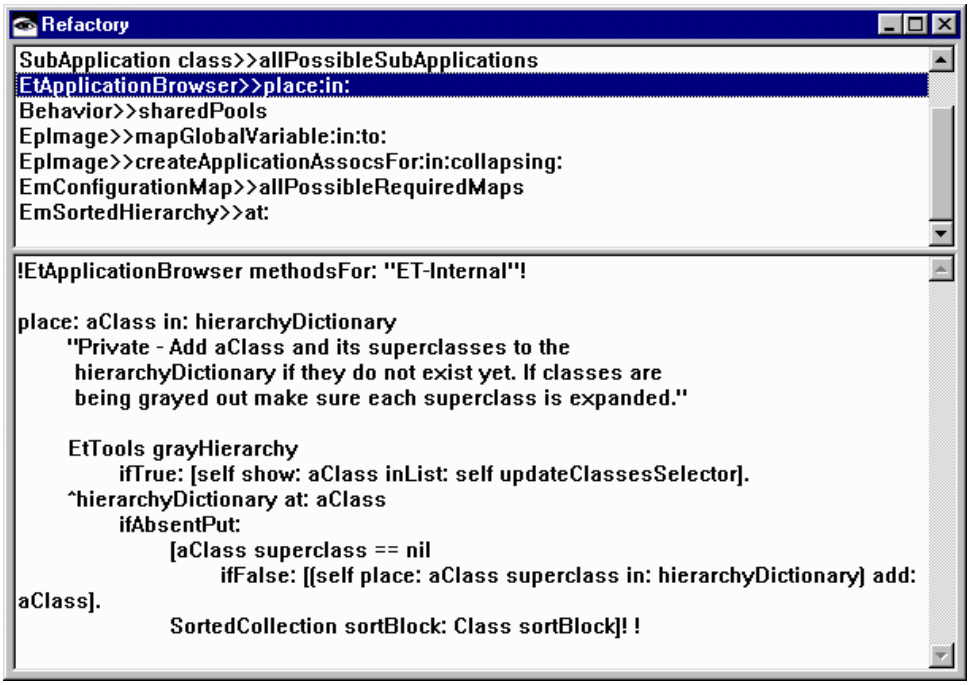
We match a list of temporaries with the `@Temps, and a list of statements using "``@.Statements". Since we do not need to recurse into the temporaries we don't need the extra ` character. We then rerun the "Find" command, and see that we are finding all of the methods.

Now we're ready to enter the replace pattern. We just need to enter the pattern that will replace the found pattern. For the at:ifAbsent: example, we can enter a pattern such as:

```
``@aDictionary at: ``@key
      ifAbsentPut:
            [| `@Temps |
            ``@.Statements.
            ``@value]
```

Whenever you enter a meta-variable in the replace pattern, this means to copy the value from the found expression and put it in the replaced tree. (Note: all meta-variables used in the replace pattern must also be defined in the search pattern)

After entering the replace pattern, we perform the "Replace" option. This opens a window on the possible replacements. It does not immediately compile the replacement, but instead opens a window for you to review/remove the changes before compiling them:



From this window you will be able to compile single method changes, compile all changes, and remove "faulty" changes.

Congratulations, you have just converted all of the old at:ifAbsent: messages to their newat:ifAbsentPut: variants.

Other Examples:

Here are several other examples. They are not fully explained, but it is hoped that they will help answer any questions you may have.

Search for	Replace with	Explanation
`| `@Temps |` `` ``@.Statements.`` `` ``@Boolean ifTrue: [^false].`` `^true`	`| `@Temps |` `` ``@.Statements.`` `^``@Boolean not`	Return the value of the boolean negated, instead of using a conditional to return the opposite value.
`` ``@object not ifTrue: ``@block``	`` ``@object ifFalse: ``@block``	Eliminate an unnecessary not message.
`| `@Temps |` `` ``@.Statements1.``	`| `@Temps |` `` ``@.Statements1.``	Move the same statement that ends both the true and false blocks outside

``@Boolean ifTrue: [`@.TrueStatements. `.Statement] ifFalse: [`@.FalseStatements. `.Statement]. ``@.Statements2	``@Boolean ifTrue: [`@.TrueStatements] ifFalse: [`@.FalseStatements] `.Statement. ``@.Statements2	the blocks.
`@methodName: `@methodArgs | `@Temps | `@Condition ifTrue: [`.Stmt1. `.Stmt2. `@.Statements]	`@methodName: `@methodArgs | `@Temps | `@Condition ifFalse: [^self]. `.Stmt1. `.Stmt2. `@.Statements	Eliminate ifTrue: guard clauses. With this rule you'll need to check the "Method" check box. This signifies that we are matching the whole method, and not its parts.

Limitations:

The rewrite rule editor is a very powerful system that can be used to change a lot of code quickly. As a result, it is also somewhat dangerous since it can "change a lot of code quickly." Here are some limitations of the current system as well as some hints on creating good rewrite rules:

- You can enter some rules that create new sources that cannot be compiled. This happens when you are searching for a variable and replacing it with a non-variable object. For example, if you are trying to replace references to the variable "object" to send the message "self getObject", you will transform code such as an assignment "object := aVariable" into "self getObject := aVariable" which obviously will not compile. Currently, the only way around this limitation is to use the rewrite engine directly without the interface. This way you can specify other rules that will eliminate the false matches.
- It is best to enter rules that do not change the receiver of a message if it could be part of a cascaded message. For example, if you entered a rule such as "``@aStream print: ``@anObject" -> "``@anObject printOn: ``@aStream", this will cause problems with a statement such as "readStream print: anObject; cr" since you are changing the receiver of the cascaded message.
- Since the rules only look at the names, you can change code that shouldn't be changed. For example, if we have another class that defines the at:ifAbsent: message, but not theat:ifAbsentPut: variant, then our rewrite rule about would break all uses of this other class since it is only going by the message name at:ifAbsent: not the type of the object that receives the message.

If you have comments or suggestions, Contact Us.