Secrets of the Rewrite Tool

(How to roll your own refactorings)

John Brant
The Refactory, Inc.
brant@refactory.com

Don Roberts
The Refactory, Inc. / University of Evansville
roberts@refactory.com

The Refactoring Browser

- First released in 1994
- Core of the RB is rewriter
- A refactoring is a transformation with several preconditions

Why you should care

- Custom refactorings
- Bulk transformations
- Changing layers
- Migrations

Program Objects

- Programs are made of text
  - very naïve
- Programs are made of objects
  - Classes
  - Variables
  - Methods

Method Structure

- The body of a method is free-form text
- This text must be legal
- These rules are called a *grammar*

Example Grammar (partial)

Method: MethodName MethodSequenceNode

MethodName:
    UnaryMethodName | BinaryMethodName | KeywordMethodName

UnaryMethodName: <name>

BinaryMethodName: <binary_symbol> Variable

KeywordMethodName:
    <keyword> Variable
    | KeywordMethodName <keyword> Variable

MethodSequenceNode: Temporaries Primitives Statements

Derivation (Parse) Trees

- The rules that were applied to create a method form a tree
- The nonterminals (lhs) are the nodes
- But many of the nonterminals are similar
- Parse Nodes usually represent several similar
- The resulting tree is a parse tree

Example Parse Tree

```
add: anInteger
    ^self + anInteger
```

```
MethodNode('add:')
    VariableNode(anInteger)
    SequenceNode
        ReturnNode
            MessageNode('+')
                VariableNode(self)
                VariableNode(anInteger)
```

Parse Nodes in Smalltalk

- Classes represent the nodes
- RBProgramNode is the superclass
- Contain the attributes of the node
- Hold onto the children
- Implement several useful methods
  - parent, nodesDo:, isAssignment, references:

Unification

- Pattern matching is accomplished by the unification algorithm
- Input:
  - A tree to search
  - A pattern (contains pattern variables)
- Output:
  - False – no match found
  - True + a *set of assignments* if a match is found

Unification cont.

- The set of assignments contains the value of the pattern variables that makes the match work
- Example:
  - 'self X' matches 'self foo' if X='foo'
  - 'X Y' matches 'self foo' if X='self' and Y='foo'
  - 'X := X' does not match 'foo := bar' because X must be the same value in both places

The Rewrite Tool

- The rewrite tool is a parse-tree matcher
- The pattern trees are created by parsing a superset of Smalltalk into the pattern tree
- Therefore, writing patterns is just like writing Smalltalk code
- Caveat: This is not a textual search and replace, the pattern 'foo' will not match ('self foo') but will match ('**foo** := **foo** + 1')

Searching and Replacing

- Pattern can be used to simply look for matches
- However, the more powerful use is to actually change the code.

Simple Rule

- Smalltalk code with no pattern variables will match exactly that code
- The parse trees are identical

Pattern Variables

- Identified with a backquote (`)
- Have a name
  - Allow the same variable to occur multiple times in a pattern
  - Allow the transformed code to refer to the original code
- Example: `receiver

Single Variables

- `receiver foo
- Only matches single variables
- Matches:
  - self foo
  - x foo
  - OrderedCollection foo

Literal Variables

- `#lit size
- Matches literals (numbers, strings, literal arrays, etc.)
- Matches:
  - 3 size
  - 'foo' size
  - #(a b c) size

List Variables

- `@receiver foo
- Matches any subtree in the first position
- Matches:
  - self foo
  - self size foo
  - (self at: 1) foo
- The @ symbol is also used to denote lists of 0 or more items (shown later)

Recursive Search

- ``@receiver parent
- The second quote tells the rewriter to search deeper even if a top-level match is found
- Example:
  - x parent parent

Statement Variable

- `.Statement1
- Matches an entire statement node
- Example:
  - `.Statement1.`
    `.Statement1`
  - foo
    ```
    x := 1.
    x := 1
    ```

List of Statements

- Matching lists of statements can be tricky
- A statement list node can also have temporaries
- Example:

```
`@.Statements1.
`.Duplicate.
`.Duplicate.
`@.Statements2
```

List of Statements (cont.)

- Wouldn't match:

```
| x |
x := 1.
x := 1
```

- Correct rule:

```
| `@temps |
`@.Statements1.
`.Duplicate.
`.Duplicate.
`@.Statements2
```

Matching Method Nodes

- The pattern is an entire tree
  - Sometimes an expression (e.g. ``@receiver foo)
  - Sometimes a statement list (e.g. Previous example)
  - What if we want to match an entire method?
- Check the 'method' checkbox

Method Nodes (cont.)

- Example:

```
`keyword1: `arg1 `keyword2: `arg2
    | `@temps |
    `@.Statements1
```

- Matches all 2 argument methods

Programmatic Matching

- Sometimes you want additional criteria for matching
- e.g. Only find classes that start with “RB”
- use `{:node | smalltalk code that returns a boolean} for matching
- Use the methods available on RBProgramNodes

Programmatic Example

```
`{:node | node isVariable
  and: ['RB*' match: node name]}
```

Decision Tree for Writing Patterns

- Parse Tree Search:
  - If method -goto- **Method Search**
  - If multiple statements
    or contains temporary variable definition
    or sequence node with only one statement
    -goto- **Sequence Node Search**
  - otherwise -goto- **Single Node Search**

Method Search

- -evaluate- **Message**
- -evaluate- **Sequence Node Search**

Message

- If known selector
  - enter directly
    - at: `@arg1 put: `@arg2
    - use the single node search for the arguments
- If unknown number of arguments
  - ``@methodName: ``@args
  - (args must be a list type)

Message (II)

- If known number of arguments
  - `selector1: ``@arg1 `selector2: ``@arg2
  - at: ``@arg1 `selector: ``@arg2

Sequence Node Search

- -evaluate- **Search Temporaries** -evaluate- **Search Statements Search**

Temporaries

- If unknown number of temps
  - | `@args |
- If contains a known temporary with unknown # of temps
  - | `@args1 myArgument `@args2 |
- if known number of arguments
  - | `arg1 `arg2 |
- if no arguments
  - leave blank

Search Statements

- If any statements
  - ``@.Stmts'
- If contains at least one statement
  - ``.Stmt. ``@.Stmts

Search Statements (II)

- If contains particular statements in "one location"
  - ``@.Stmts1.
    self myStatement.
    self myOtherStatement.
    ``@.Stmts2'
  - (use the Single Node search for the "self myStatement" and "self myOtherStatement")

Search Statements (III)

- Contains particular statements in two or more locations
  - ``@.Stmts1.
    self myStatement.
    ``@.Stmts2.
    self myOtherStatement.
    ``@.Stmts3'
  - (use the Single Node search for the "self myStatement" and "self myOtherStatement")

Single Node Search

- Match anything
  - ``@x
- Exact code
  - x := x + 1
- Assignment
  - `x := `@value
  - (use Single Node Search for variable and value)

Single Node Search (II)

- Return
  - ^`@value
  - (use single node search for the value)
- Block with fixed number of arguments
  - [:`var1 | "**Sequence Node Search**"]
  - (use variable search for each argument)

Single Node Search (III)

- Block with variable number of arguments
  - [:`@vars | "**Sequence Node Search**"]
- Message send
  - -evaluate- **Single Node Search** for receiver and evaluate message for the selector and arguments

Single Node Search (IV)

- Cascade with known # of messages
  - -evaluate- receiver with **Single Node Search**, and -evaluate- each message send with **Message** separated by ;

Single Node Search (V)

- Cascade with unknown # of messages
  - -evaluate- a **Message** with (un)known selector and then enter
    a cascaded message with the `@;cascade: `@cascadedArgs
  - e.g., ``@rcvr `@msg1: `@args1; `@;cascade: `@cascadeArgs

Single Node Search (VI)

- Known literal
  - enter value directly
- Unknown Literal
  - `#literal Variable

Search Code Rules

- use `{:node | smalltalk code that returns a boolean} for matching
- can optionally take a second argument that contains a matching dictionary
- also can refer to other pattern variables that occur earlier in the matching process

Search Code and Replacing0

- on replacing, use `{ smalltalk code that returns a RBValueNode }
- can refer to all pattern variables in the search
- can have optional argument for the matching dictionary

Restricting the Scope of a Rewrite

- When rewrites are applied, only the selected classes and methods are searched
- Can be used to restrict the scope of a transformation
- Be careful! You can break your program big time

Building your Own Tools

Smalllint Rules

Change Objects

- The RB changes the system via *change objects*
- RefactoryChange is the superclass
- Provides several nice features:
  - Atomicity – can be grouped into composite objects and performed all at once
  - Undo – all change objects can undo themselves
- All tools should use these to change the system
- Can be used independently of the RB

Custom Refactorings

- A refactoring is simply a transformation along with a set of conditions under which it is behavior-preserving
- Often a one-off refactoring isn't worth programming...simply use the rewriter

Scripted Transformations

- Creating sets of transformations to be performed all at once.
- Can be used to perform a mass transformation (e. g., replacing an entire layer)

Migrations

- Converting from one Smalltalk to another or from one library to another is mainly a problem of rewriting
- Example: (VW to Dolphin)

```
``@a numArgs → ``@a argumentCount
 Timestamp → TimeStamp
``@Exception
    raiseWith: ``@object errorString: ``@string →
``@Exception
    signal: ``@string with: ``@object
```

Contact Info

- Web: http://www.refactory.com/RefactoringBrowser
- Email: {brant,roberts}@refactory.com