Inside the Refactoring Browser

Vassili Bykov

April 2006

1 August 2008 Preface

This document is based on the series of internal notes on the overall architecture of the Refactoring Browser I wrote in 2006. The current design of the RB is already somewhat different, and with time will move further away from what is described here. Still, I hope these notes will be useful despite the inaccuracies. Many thanks to Travis Griggs for helping me recover these bits and for the following comments:

1. *RBCommand stuff is not being moved forward.*
2. *I began evolving some newer terms in 7.6 (e.g. toolset, etc).*

2 Basic Anatomy

In this overview of basic browser anatomy we introduce the main players that together make the browser window work. We will see which classes are responsible for which parts of the UI, and what happens during the major browser state changes such as selecting a different tab or switching to a new browser view.

Figure 1 shows the browser in its freshly opened state. The prominent UI elements are the four lists grouped in twos inside the two tab controls at the top, and a text editor (usually) in the tab control at the bottom. The bottom half displays the details of whatever is selected in the top half. In this initial state with nothing selected, only one tab appears at the bottom showing the definition of the Smalltalk namespace. (However useful that might be).

Figure 2 shows the most important objects behind this window, organized to match the window layout. The white boxes correspond to the visible widgets of the browser: the four list views and the text editor. The boxes that contain them are the subapplications tying the views together. All boxes except NavigatorState are subclasses of ApplicationModel, or more specifically BrowserApplicationModel. We are going to walk though them from the outside in.

The instance of RefactoringBrowser is "the whole thing". It is the application model of the browser window. It has a number of parts, but the two main ones

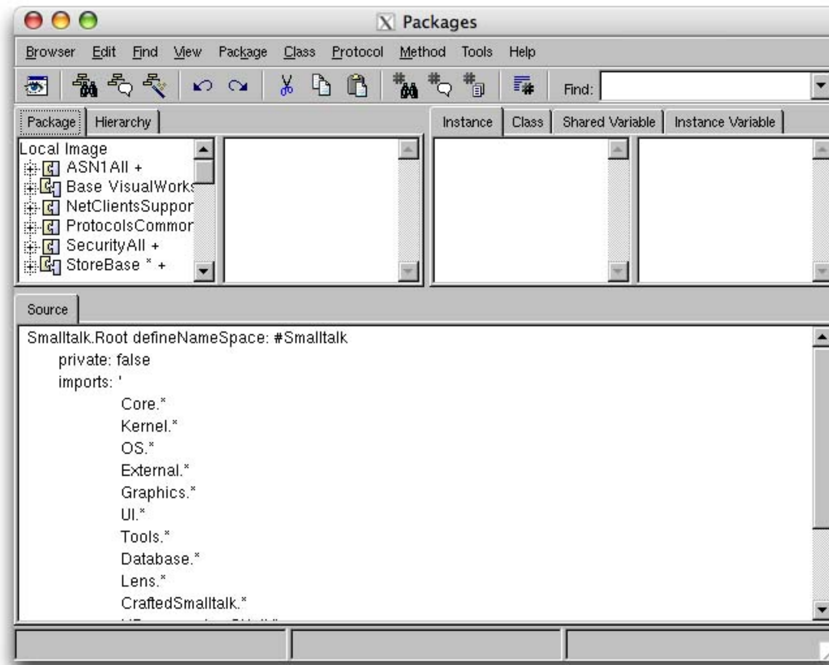

Figure 1: Refactoring Browser in the default state.

are the BrowserNavigator and the CodeModel. In general, the BrowserNavigator is responsible for the top half of the window and the CodeModel for the bottom half. In reality, either can occasionally control the the entire window–but most of the times this top/bottom separation holds.

2.1 The Top Half

BrowserNavigator is an invisible subapplication that controls the top half of the window. It orchestrates the user navigation through the hierarchy of system objects displayed by the browser.

In the default package/class/protocol/selector configuration, the navigator always knows what package, class, protocol and method are selected. When the user selects a different item at any of the levels of that hierarchy, the navigator is notified about the change. It then in turn notifies all the other components, which may update themselves to match this new state of navigation and notify the navigator about their change. For example, when the user clicks a new protocol in the protocol list, the navigator is notified and broadcasts a notice about the new selection. The part that manages the list of selectors sees the change in the selected protocol, updates the list to match the new selected protocol and notifies the navigator that no selector is now selected. The notifier

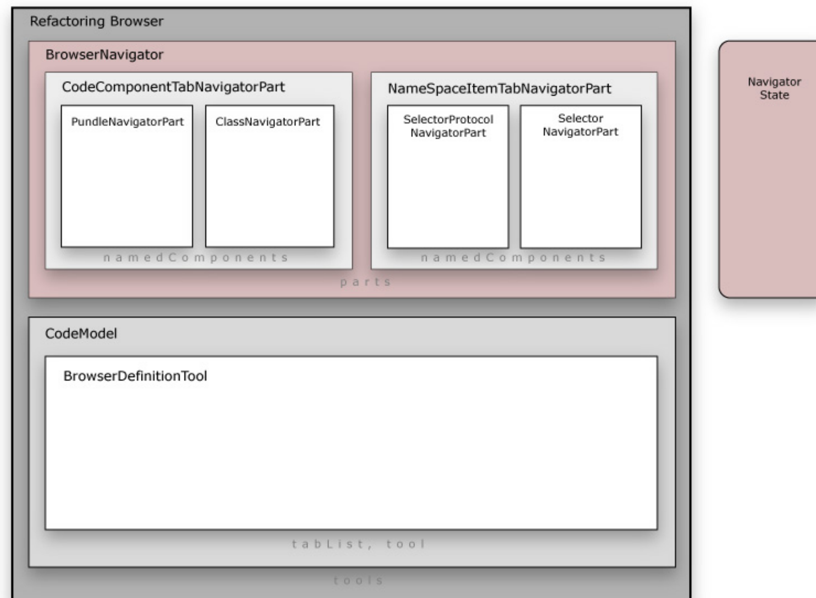

Figure 2: Core Refactoring Browser components.

now broadcasts this notification so that the part responsible for the method source sees the change and updates itself to show nothing.

Navigation state is such an important concept that it's captured as an object: an instance of NavigatorState. In the pattern lingo, NavigatorState is a Memento capturing the current state of the various selections in BrowserNavigator. The navigator can produce a NavigatorState that describes its current UI state, or set its UI state to match that encoded as a NavigatorState instance.

In a way, BrowserNavigator is the heart of RB. All the other subapplications you see in this diagram have a navigator instance variable that holds a direct link to the navigator. This is quite natural, considering that it's the navigator that keeps track of what exactly the browser is displaying at the moment, and all the other browser components need this information. Most (though not all) of the browser menu commands are implemented as methods of BrowserNavigator.

In the standard five-pane configuration, BrowserNavigator contains two main subapplications (also called *parts*), in its parts instance variable. They are the things that manage the tab controls. CodeComponentTabNavigatorPart is what displays the *Package* and *Hierarchy* tabs on the left, while NameSpaceItemTabNavigatorPart is responsible for the *Instance*, *Class*, *Shared Variables* and *Instance Variables* tabs on the right.

When the *Package* tab is selected, the CodeComponentNavigatorPart contains two subapplications as its namedComponents. PundleNavigatorPart is what populates and manages the tree of bundles and packages (just that one wid-

get). ClassNavigatorPart is responsible for the list of classes (also just that one widget).

When a different tab is selected, namedComponents are changed to match the new selection, so that new subapplications with different behavior are installed at the same spot of the UI. For example, when the *Hierarchy* tab is selected the CodeComponentTabNavigatorPart content changes to include a HierarchyPundleNavigatorPart and a HierarchyClassNavigatorPart.

In the same manner, NameSpaceItemTabNavigatorPart by default (when the *Instance* tab is selected) holds a SelectorProtocolNavigatorPart which manages the list of method protocols for the selected class, and a SelectorNavigatorPart which manages the list of method selectors.

2.2 The Bottom Half

CodeModel is the subapplication that shows the list of tabs you see in the bottom half of the browser, such as *Source*, *Comment*, *Rewrite* and so on. It is also what really implements the "views", previously known as "buffers", of the browser.

A RefactoringBrowser holds its CodeModels in a collection called tools. When you first open the browser, there is only one CodeModel in that collection, also contained in the currentBuffer value holder.

When you select the *New View* menu item, the browser adds a second code model to the collection and puts the new code model into the currentBuffer value holder. Even though visually it seems that switching to a different browser view changes the whole content of the browser window, what really happens is the BrowserNavigator loads its state from the new NavigatorState stored in the CodeModel we are switching to, and sets that new CodeModel as the subapplication of the bottom half of the browser.

Inside the code model are a number of CodeTools such as the BrowserDefinitionTool, which is the subapplication behind the *Source* tab when a namespace or a class is selected. Each tool gets a tab in the current lineup of tabs in the RB, so the tabList SelectionInList in the current CodeModel is what holds all of the currently available tools. The currently selected tool is also stored in the tool variable of the codeModel.

2.3 Other Configurations

To summarize what has just been said about the RB interface, in the standard five-pane configuration it has the following structure:

```
RefactoringBrowser
  BrowserNavigator
    CodeComponentTabNavigatorPart (*)
      PundleNavigatorPart (+)
      ClassNavigatorPart (+)
    NameSpaceItemTabNavigatorPart (*)
      SelectorProtocolNavigatorPart (+)
```

```
    SelectorNavigatorPart (+)
CodeModel (*)
  BrowserDefinitionTool (+)
```

The names marked with (*) are what we see as tab controls. The names marked with (+) are pluggable subapplications that are replaced as tab selection changes. Here is the summary of subapplications line-ups for common browser configurations:

```
* CodeComponentTabNavigatorPart (the top left)

      o Package tab selected
            + PundleNavigatorPart
            + ClassNavigatorPart

      o Hierarchy tab selected
            + HierarchyPundleNavigatorPart
            + HierarchyClassNavigatorPart

* NameSpaceItemTabNavigatorPart

      o Instance or Class tab selected
            + SelectorProtocolNavigatorPart
            + SelectorNavigatorPart

      o Shared Variable tab selected
            + SharedVariableProtocolNavigatorPart
            + SharedVariableNavigatorPart

      o Instance Variable tab selected
            + InstanceVariableNavigatorPart
            + SelectorNavigatorPart

* CodeModel, commonly used tabs

      o Source tab - one of the following,
        depending on current selection
            + BrowserDefinitionTool
            + BrowserCodeTool
            + SharedVariableCodeTool
            + SUnitCodeTool

      o Properties tab
            + CodeComponentPropertiesTool
```

```
o Comment tab
     + BrowserCommentTool
```

3 Navigator and its Parts in More Detail

Figure 3 shows navigator parts (subapplications managing the top half of the browser) and how they are related.

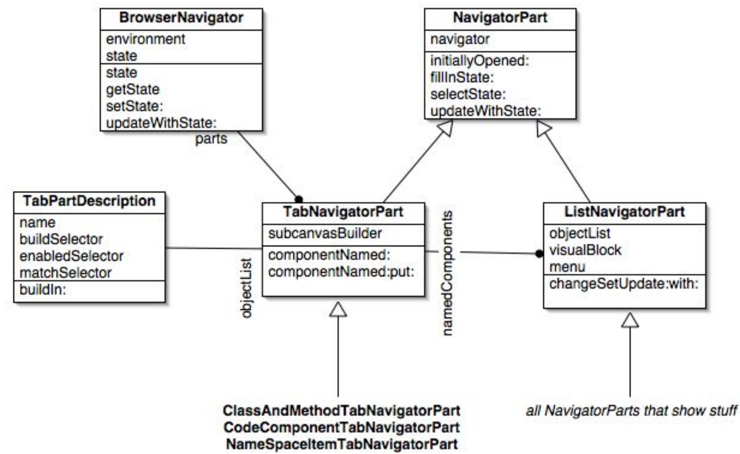

Figure 3: BrowserNavigator structure.

Everything that appears as a subapplication inside the BrowserNavigator is a descendant of the abstract class NavigatorPart. The hierarchy of NavigatorPart includes two main branches: TabNavigatorPart and ListNavigatorPart.

3.1 TabNavigatorParts

Subclasses of TabNavigatorPart are subapplications that are elements of the parts collection of a BrowserNavigator. They are responsible for tab controls in the top half of the window.

Each tab of the control is represented by an instance of TabPartDescription contained in the objectList SelectionInList of the tab part. When selected, the tab part sends the message with the selector from its buildSelector instance variable to its containing TabNavigatorPart. The tab part then rebuilds itself to show the lineup of ListNavigatorParts corresponding to the newly selecting tab.

The enabledSelector message is sent to a TabPartDescription to check whether the tab should appear in the current browser state, and matchSelector determines whether it should be the default selected part.

3.2 ListNavigatorParts

Subclasses of ListNavigatorPart manage the lists (and also the trees) you see in the top half of the browser. They are usually contained inside TabNavigatorParts.

Any ListNavigatorPart has an objectList, which is a SelectionInList (or MultiSelectionInList) with whatever the part shows: classes, protocol names, selectors, or the tree of bundles and packages. It also holds onto a visual block and the menu used by its widget, and provides a number of helper methods to work with the object list.

3.3 BrowserNavigator

One of the main things a BrowserNavigator holds onto is the environment object. The environment (an instance of BrowserEnvironment) defines the scope visible in the browser. The environment of the default system browser reveals the entire system, while the environment of a browser spawned on a package shows only that package.

Another important object of the browser is its state, an instance of NavigatorState, already discussed in Basic Anatomy.

The diagram shows the four main state accessing methods. Here is how they compare:

- state - a simple getter of the state variable.
- getState - recomputes the current browser state, saves it as the new value of the state variable and answers it.
- setState: - sets the argument as the value of the state variable and tells the parts to update themselves to reflect the new state.
- updateWithState: - same as setState: plus signals the general "self changed".

4 Understanding ListNavigatorPart

Browser lists showing the "real stuff" are descendants of ListNavigatorPart, which makes the latter one of the most important classes to understand in order to change anything about those lists.

ListNavigatorPart holds its content in an instance variable called objectList, which is a SelectionInList or a MultiSelectionInList. On the class side, basicWindowSpec defines its default appearance which includes a list view hooked up to objectList.

A ListNavigatorPart thius deals with three kinds of things that all could informally be referred to as "the list".

- The list widget.
- SelectionInList held by the objectList instance variable (the aspect of the list widget).

- The list of objects inside that SelectionInList, which is what we see in the list widget.

To avoid confusion, we will refer to these things by their full name, such as "the list widget", "the objectList" and "the list of objects" whenever the intended meaning is not clear from the context.

4.1 Populating the List

How the list is filled with items is determined by concrete subclasses of ListNavigatorPart. Usually a subclass defines the method fillInListFor: which takes a NavigatorState, computes a new list of objects to display and sends updateListWith: to self with the computed list of objects.

4.2 Working with the List

Navigator parts don't provide an extensive public API exposing the list because this is not how they are used. A navigator part is responsible for populating the list widget to match a given navigator state, or the other way around, and for managing the list selection—again, to match a navigator state or vice versa. The primary communication channel inside the browser are NavigatorState instances passed around by the BrowserNavigator. Most actions that operate on the currently selected object are implemented by BrowserNavigator, which retrieves the selection from the current NavigatorState. Thus the navigator has no need to query navigator parts for their current selection or their content, leaving ListNavigatorParts rather solitary objects communicating with the outside world mostly through NavigatorStates.

They do, however, have private methods for working with the list. These methods are good to know if you need to implement a navigator part of your own.

- sortBlock answers the block used to sort the items in the list (used by a number of methods that follow).
- list answers the list of objects displayed by the part.
- list: sets a new list of objects to be displayed by the part. The list is set "as is" without any attempts to sort it.
- updateListWith: sets a new list of objects, after sorting it using sortBlock.
- addToList: adds a new item to the list, maintaining their sort order.
- mergeIntoList: adds a collection of new items to the list, maintaining their sort order.
- select: selects a collection of items in the list (not to be confused with the standard select: of Collection classes).

4.3 Item Properties

List item appearance is defined by the following methods:

- iconFor: is called with a list object and should return an icon to display for an object, or nil if there is no icon to show.
- displayTextFor: is called with a list object and should return the text (a String or a Text) to show for that item.
- listLabelFor: is the sender of the the above two messages. Usually there is no need to override it as it constructs a proper Label using the results of iconFor: and displayTextFor:.

Warning: these methods are called in "real time" as the list widget is repainted, every time it is repainted. The results are not cached anywhere. For this reason, these methods should not do anything time-consuming as it will visibly slow down the browser.

4.4 Responding to User Input

Here is how a ListNavigatorPart gets notified about user actions in the list widget:

- changeRequest is sent when the user clicks a different item in the list. This is a hook that allows the part to veto the change. If the method returns false, the list selection is left unchanged.
- changed is sent after the selection changes.
- doubleClickItem is sent when a list item is double-clicked.

Most list parts don't redefine changeRequest and changed in any part-specific way. Instead, they rely on the stock implementations of those methods provided by NavigatorPart.

Menu actions are handled by the menu framework and are usually implemented by BrowserNavigator.

4.5 Menus

A navigator part holds its menu in an instance variable menu and supplies it to the list widget to use as the context menu. The following methods provide access to it:

- menu returns the current menu, building it if needed by sending defaultMenu to self.
- menu: changes the context menu of both the navigator part and the associated list widget.

5 How to Add a Menu Item

One of the common questions (and points of aggravation) of would-be RB enhancers is "how do I add a menu item to ..."? First of all you need to know what class is responsible for the menu you want to extend.

5.1 Where are the menus?

RB menu definitions are scattered over several classes. To add an item to a menu, you should use a pragma method instead of changing the base definition, so that your extensions can coexist with others. Listed below are the browser menus, the location of their resource, and the menu key a pragma method extending the menu should use.

- *The Menu Bar* The menu bar is built by the (instance) method RefactoringBrowser>>menuBar.
- *Browser* RefactoringBrowser class>>browseMenu. Name key: #browserMenu.
- *Edit* The mainMenu methods on the class side of the various CodeTool subclasses.
- *Find* RefactoringBrowser class>>findMenu. Name key: #findMenu.
- *View* RefactoringBrowser class>>viewMenu. Name key: #viewMenu.
- *Package* BrowserNavigator class>>pundleMenu. Name key: #pundleMenu.
- *Class* BrowserNavigator class>>classMenu. Name key: #classMenu.
- *Protocol* BrowserNavigator class>>protocolMenu. Name key: #protocolMenu.
- *Method* BrowserNavigator class>>selectorMenu. Name key: #selectorMenu.
- *Tools* RefactoringBrowser class>>toolsMenu. Name key: #toolsMenu.
- *Help* RefactoringBrowser class>>helpMenu. Name key: #helpMenu.
- *The Toolbar* RefactoringBrowser class>>toolbarMenu. Name key: #toolbarMenu.

5.2 Adding menu items using pragma methods

To add a new a menu item to the browser as an extension pragma method, you add it to the class that defines the menu you are interested in. For example, to add an item to the *Tools* menu you would add a method to RefactoringBrowser.

In the menu method's pragma you identify the menu you extend by including the name key from the list above into the menu: parameter of the pragma. The parameter is a one-element array with the key if you want the item right on the menu, or an array with two or more elements to target a submenu of the menu.

For example, here is a method that adds a *Copy Name and Version* item to the bottom of the *Package* menu. The item copies the name and the version of the current package to the clipboard:

```
BrowserNavigator>>copyNameAndVersion
    <menuItem: 'Copy Name and Version'
            nameKey: nil
            menu: #(#pundleMenu)
            position: 100.1>
    self pundle ifNotNil:
        [:pundle |
        ParagraphEditor currentSelection: pundle itemString]
```

As soon as you accept this method, the item should appear and be functional in all the currently open browsers. (In production code you might want to use a three-element array as the item label to make it localizable, something like: `#(#CopyNameAndVersion #browser 'Copy Name and Version')`).

Other pragmas you can use to extend menus are visible as the list of methods in the 'generating' protocol of MenuAutomaticGenerator . Other pragmas allow more interesting options, such as specifying a selector to determine whether the item should be enabled or disabled. To enable the *Copy Name and Version* item only when a bundle or a package is selected, we would rewrite the pragma in the method as:

```
    <menuItem: 'Copy Name and Version'
            nameKey: nil
            enablement: #isPundleSelected
            indication: nil
            menu: #(#pundleMenu)
            position: 100.1>
```

The method isPundleSelected we rely on is defined by BrowserNavigator (or more exactly by its superclass Navigator).

As another example, to add a *Close* menu item to the Browser menu we would add the following method to the RefactoringBrowser class.

```
RefactoringBrowser>>closeWindow
    <menuItem: 'Close'
        nameKey: nil
        menu: #(browserMenu)
        position: 100.1>
    self closeRequest
```

The *Edit* menu is special because instead of the browser or the navigator, it is managed by the tool currently selected in the bottom half of the browser.

In order to extend it we first need to find the CodeTool subclass responsible for the tool. BrowserCodeTool is the most common case.

For CodeTool subclasses, the menu: parameter of the pragma will always be `#(mainMenu)`—or `#(mainMenu ...more IDs...)` if the item is added to a submenu.

6 How to Change a Menu Item

What if instead of adding a new menu item we want to change the implementation of an already existing one? To do that we need to find the method responsible for the item. That can be tricky because unlike items added by extension methods, the relationship between the items defined by menu resource and their implementation methods is indirect.

Here is how to find the implementation method of a menu item in five easy steps.

1. First of all find the menu definition—that is, the resource method that defines the menu. Refer to the list in *RB - How to Add a Menu Item* for the location of all the menus you commonly see in the browser.

2. Once you've found the menu resource, open it in the Menu Editor, find the item you want and look at its properties. Usually the *Value* property will be action: and the ID will be a sentence-like symbol–something like `#'Browse References to Class'`. That means that the item is managed by the RB action framework and you need to go through that framework to find the implementing method.

3. Evaluate the following expression:

   ```
   RBActionEditor open
   ```

   This will open a window with a list labeled *Action* on the left and a number of fields on the right. Find the ID you saw in the menu editor in the list of actions and select it.

4. The right half of the window will update to show the properties of the selected action. Since this is just a "how to" and not a detailed description of the action framework, we are not going to discuss all the properties and their meaning. The one property you care about is the *Action* (yes, both the list on the left and the property on the right are labeled the same though they mean different things). Note that the label of the Action property is a button.

5. Click that button, and a browser will open on the method associated with the menu item.

6.1 What if the item isn't there

If at step 2 you open the menu and don't see the item you are interested in, it means the item is not part of RB proper and was added by an extension. In that case, the rest of this *How To* does not apply and you have to find the extension pragma method. Here is one way to do it.

You already know the class responsible for the menu, and since you were just looking at its menu resource method, the class is already selected in your browser. Switch to the instance side and select the *Hierarchy* view to see the list of packages defining and extending the class. Any package that *doesn't* begin with *Browser-* is an extention that likely adds some menu items. Select it and show all methods. Usually there is only a handful to look through before you find the one you are looking for.