Creating Sophisticated Development Tools with OmniBrowser ⋆

Alexandre Bergel [a] Stéphane Ducasse [b] Colin Putney [c] Roel Wuyts [d]

[a] *LERO & DSG, Trinity College Dublin, Ireland*

[b] *LISTIC – University of Savoie, France & University of Bern, Switzerland*

[c] *Wiresong*

[d] *IMEC, Belgium & professor at Université Libre de Bruxelles, Belgium*

Abstract

Smalltalk is not only an object-oriented programming language; it is also known for its extensive integrated development environment supporting interactive and dynamic programming. While the default tools are adequate for browsing the code and developing applications, it is often cumbersome to extend the environment to support new language constructs or to build additional tools supporting new ways of navigating and presenting source code. In this paper, we present the OmniBrowser, a browser framework that supports the definition of browsers based on an explicit metamodel. With OmniBrowser a domain model is described in a graph and the navigation in this graph is specified in its associated metagraph. We present how new browsers are built from predefined parts and how new tools are easily described. The browser framework is implemented in the Squeak Smalltalk environment. This paper shows several concrete instantiations of the framework: a remake of the ubiquitous Smalltalk System Browser, a coverage browser, the Duo Browser and the Dynamic Protocols browser.

⋆ We gratefully acknowledge the financial support of the french ANR project "Cook: Réarchitecturisation des applications industrielles objets" (JC05 42872) and of Science Foundation Ireland and Lero — the Irish Software Engineering Research Centre.

Email addresses: Alexandre.Bergel@cs.tcd.ie (Alexandre Bergel), stephane.ducasse@univ-savoie.fr (Stéphane Ducasse), cputney@wiresong.ca (Colin Putney), Roel.Wuyts@imec.be (Roel Wuyts).

Preprint submitted to Elsevier Science *14 December 2006*

1 Introduction

Smalltalk is an object-oriented language featuring a complete development environment supporting interactive and dynamic programming [GR83,Gol84]. While the default environment already supports advanced ways of navigating source code and fluid development since the eighties, new browsers have been developed over the years, such as the *Refactoring Browser* [FBB+99,RBJO96] [RBJ97] which was the first system browser supporting refactoring, the *Classification Browser* [Hon98] and the *StarBrowser* [WD04] that support smart grouping of objects, a browser for incremental development supporting visual feedback of undefined methods [SB04] or the *Whiskers* browser that shows multiple methods at the same time maximizing the screen space. StrongTalk [BG93], a more exotic Smalltalk version featuring optional typing, offers a glyph based browsing environment.

The problem when building all of these browsers is that they are always rebuilt from scratch because there hardly exists any domain models or frameworks for building such development tools. In fact, the current browsers in most Smalltalk environments are hard to extend for two reasons: (a) they are monolythic applications that are not really meant to be embedded elsewhere, and (b) the navigation and interaction of the end-user with the browsers is typically hardcoded in the browser UI elements, and is therefore hard to change or extend.

Note that some Smalltalk environments allow one to embed applications within each-other. VisualWorks for example has a notion of *subcanvases* which can be used to that end. This helps to reduce the problem (a) in the previous paragraph, but not problem (b) of the hardcoding of the the navigation and interaction in the browser UI elements. Other browsers (*Eclipse*, for example) are designed with a certain amount of customizability in mind, and are therefore easier to extend, but even those lack explicit descriptions of the navigation.

As was already reported by Steyaert et al. [SLMD96], we conclude that current visual application builders and application frameworks do not live up to their expectations of rapid application development or non-programming-expert application development. They fall short when compared to component-oriented development environments in which applications are built with components that have a strong affinity with the problem domain (*i.e.*, being domain-specific).

In this paper we present OmniBrowser, a framework to define and compose new browsers : graphical list-oriented tools to navigate and edit elements from an arbitrary domain. In the OmniBrowser framework, a browser is described

by a *domain model* that specifies the domain elements that can be navigated and edited, and a *metagraph* that specifies the navigation between these domain elements. Nodes in the metagraph describe states the browser is in, while edges express navigation possibilities between those states. The OmniBrowser framework then dynamically composes widgets such as list menus and text panes to build an interactive browser that follows the navigation described in the metagraph.

The contributions of this article are: (i) the description of a meta-driven framework to build browsers, (ii) the application of the framework to build three browsers and (iii) using navigation history to deduce dynamic information. The article is an extension of [BDPW06], refining the description of the OmniBrowser framework to be clearer and presenting more examples of browsers that use the OmniBrowser framework.

In Section 2 we describe difficulties and challenges to define states and flow between those states for a graphical user interface. In Section 3 we present the key entities of the OmniBrowser framework. In Section 4 we present the OmniBrowser-based system browser and in Section 5 we give a first illustration with the coverage code browser. Section 6 shows how multiple views on a same domain can be modeled with the Duo Browser, Section 7 illustrates rendering of dynamic informations. In Section 8 we discuss about properties of the OmniBrowser framework. In Section 9 we provide an overview of related work. In Section 10 we conclude by summarizing the presented work.

2 Defining and Maintaining the State of a Graphical User Interface

In this section we enumerate some of the problems encountered when building complex tools such as an advanced code editor.

The graphical user interface (GUI) of an application is constructed from individual widgets, such as lists, text panes or buttons. Users interact with the widgets (such as selecting list elements or by pressing buttons) to navigate, inspect or edit domain elements. When the GUI application is in a certain state (for example, showing a list of employees, with one employee selected), the user can do several things, such as deselecting the person, selecting anoother person, editing the name of address of the person shown, add a new person, remove a person, etc. As a result of this interaction the browser will be in a new state, and the user can continue interacting.

Note that depending on the state of the browser the user is able to do a certain number of things, while other ones will not be possible. For example, a list element can only be deselected when there is a selection in the first

place. Given the potential high number of different states of a GUI, validating what actions are allowed in what state is a challenging task. Let's illustrate this problem with the Smalltalk system browser, a graphical tool to edit and navigate into Smalltalk source code.

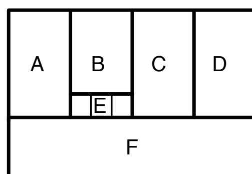

Fig. 1. The traditional Smalltalk System Browser roughly depicted.

Figure 1 depicts the different widgets of a traditional Smalltalk class system browser (see Figure 7 for a real picture). Without entering into details, A, B, C and D are lists that show class categories (groups of classes), classes, method protocols (groups of methods) and methods. E is a radio button composed of three choices and F is a text pane.

Pane A lists the categories in the system. Selecting a category in this list, makes the classes in that category appear in pane B. Selecting a class results in the protocols for that class being shown in pane C, and selecting a protocol lists the method names in pane D. Switch E controls whether the class or the metaclass is being edited, and therefore whether the protocols and methods shown are instance level or class level methods. Pane F is a text pane that gives feedback on whatever is selected in the top panes, always displaying the most specific information possible. For example, when a user has selected a method in a protocol in a class in a certain category, pane F shows the definition of that method (and not the definition of the class of that method).

The description of how the browser works shows a number of navigation invariants that need to be kept when implementing the browser. For example, the selections goes from left to right: it is not possible to have methods listed in pane E with pane D being empty.

Invariants such as the one given above need to be implemented and checked when building a browser. So we are dealing with writing an application that deals with a potentially very big number of states in which only certain transitions between states are allowed (the ones that correspond to navigations the user of the browser is allowed to do). Whenever a user clicks on widgets that make up the GUI of the browser, the state of one or more widgets is changed, and possibly new navigation possibilities are opened up (being able to select a method name, for example) while other ones will no longer be possible (not being able to select a method name when no protocol is selected). To deal with the fact that a widget can be in an inconsistent state, developers often rely on guards: the method performing an action in reaction of a user action

always has to check whether the state is actually correct or not nil.

In addition the state management is often spread over UI elements. This leads to code with complex logic (and often bugs). It makes tool elements difficult to extend and reuse in different contexts.

The problem when building a browser is in representing the mapping from the intended navigation model to the domain model and widgets. Even though graphical framework like MVC [Ree79,Ree] and Coral [SM88] offer ways to modularize the model and the graphical user interface, they do not provide means (i) to preserve consistency of the interface by restricting unexpected state transition to happen and (ii) to keep the widgets synchronized with each other [KP88].

In the next section, we describe a new framework to design browsers where the domain model is distinct from the navigation space. This latter being described by a metagraph. The state of a browser is defined by a path in this metagraph.

3 Defining a Browser: a Graph and a Metagraph

The domain of the OmniBrowser framework is *browsers*, applications with a graphical user interface that are used to navigate a graph of domain elements. When instantiating the OmniBrowser framework to create a browser for a particular domain, the domain elements need to be specified, as well as the desired navigation paths between them.

The OmniBrowser framework is structured around (i) an explicit domain model and (ii) a metagraph, a state machine, that specifies the navigation in and interaction with the domain model. The user interface is constructed by the framework, and uses a layout similar to the Smalltalk System Browser, with two horizontal parts. The top part is a column-based section where the navigation is done. The bottom half is a text pane.

Section 3.1 explains the major classes that make up the OmniBrowser framework. Section 3.2 shows a concrete instantiation to build a file browser. Section 3.3 goes in some more detail and describes the core behavior of the framework. Section 3.4 explains how the widgets are glued together.

The major classes that make up the OmniBrowser framework are presented in Figure 2, and explained briefly in the rest of this section.

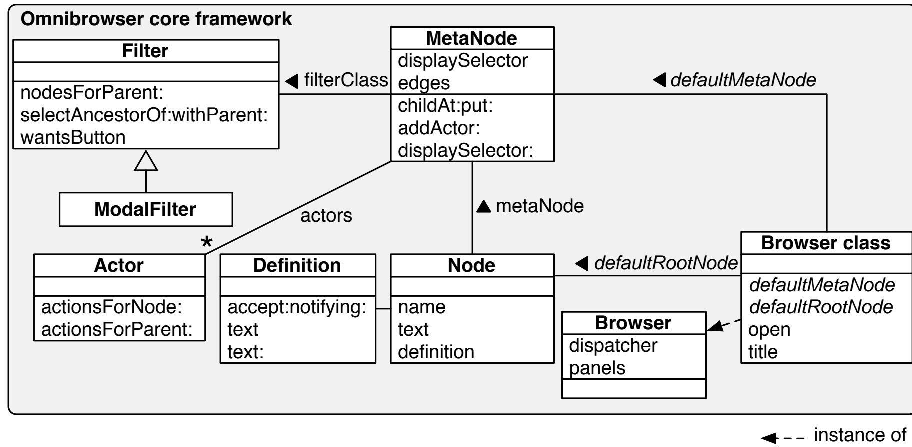

Fig. 2. Core of the OmniBrowser framework.

Browser. A *browser* is a graphical tool to navigate and edit a domain space. This domain has to be described in terms of a directed cyclic graph (DCG). It is cyclic because for example file systems or structural meta models of programming language (*i.e.*, packages, classes, methods...) contain cycles, and we need to be able to model those. The domain graph has to have an entry point, its root. The path from this root to a particular node corresponds to a state of the browser is defined by a particular combination of user actions (such as menu selections or button presses). The navigation of this domain graph is specified in a *metagraph*, a state machine describing the states and their possible transitions.

Node. A *node* is a wrapper for a domain object, and has two responsibilities: rendering the domain object in the browser, and returning domain nodes. Note that how the domain graph can be navigated is implemented in the *metagraph*.

Metagraph. A browser's *metagraph* defines the way in which the user may traverse the graph of domain objects. A metagraph is composed of metanodes and metaedges. A metanode references a filter (described below) and a set of actors. The metanode does not have the knowledge of the domain nodes, however each node is associated to a metanode. Transitions between metanodes are defined by metaedges. When a metaedge is traversed (*i.e.*, result of pressing a button or selecting an entry list), siblings nodes are created from a given node by invoking a method that has the name of the metaedge.

Actor. An *actor* is a basic unit of domain-related functionality. Actors are

attached to metanodes, and supply the *actions* used to interact with objects wrapped by nodes. For instance, *actors* are used to build context menus and buttons in the *browser*.

Action. An *Action* represents a Command [ABW98] for manipulating, interacting and navigating with the graph domain. Actions can be made available through menus or buttons in the browser. They carry information on how they should be presented to the user and are responsible for handling exceptions that can occur when they are triggered. Actions are created by actors.

Filter. When a metanode has several metaedges (modeling the fact that a user can navigate to several other nodes from the current node), the user somehow needs to choose just one of the possible navigation paths. Therefore the OmniBrowser framework renders all end nodes of the transitions that can be followed from the current node in the next column. When a user selects an element in this column, selecting in fact the transition to use, that transition actually happens, the pane definition is updated (and perhaps other panes such as button bars) and the next round of possible transitions is gathered.

Definition. While navigating in the domain space, information about the selected node is displayed in a dedicated text pane. If the selected node can be edited by the user, then a definition is necessary that can handle editing (*i.e.*, an *accept* in the Smalltalk terminology). A definition is produced by a node.

3.2 A Simple Example: A File Browser

To illustrate how the OmniBrowser framework is instantiated, we describe the implementation of a simple file browser supporting the navigation in directories and files [Hal05].

Figure 3 shows the file browser in action. A browser is opened by evaluating `FileBrowser open` in a workspace. The navigation columns in the case of a file browser are used to navigate through directories, where every column lists the contents of the directory selected in its left column, similar to the *Column View* of the Finder in the Mac OS-X operating system. Note that we can have an infinite numbers of pane navigating through the file system. The horizontal scrollbar lets the user browse the directory structure. A text pane below the columns displays additional properties of the currently selected directory or file and provides means to manipulate these properties.

Node definitions. Nodes wrap objects of the browsed domain. First the class `FileNode` a subclass of `Node` is created which represents a file. A file node is identified by a full path name, stored in a variable. The name of the node is

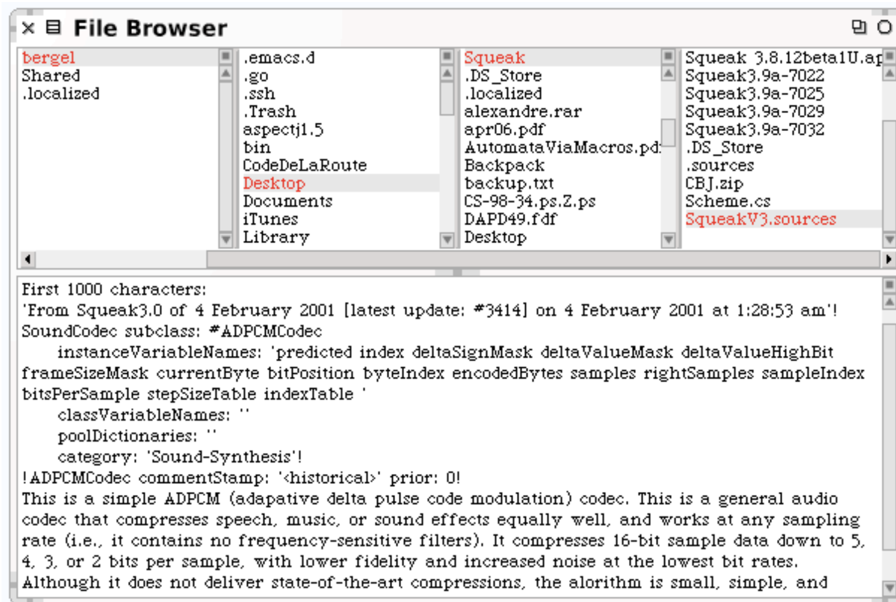

Fig. 3. A minimal file browser based on OmniBrowser.

simply the name of the file selected:

```
FileNode>>name
    ^ (FileDirectory directoryEntryFor: path) name.
```

A text containing information about the selected file is returned by the method text:

```
FileNode>>text
    ^ 'First 1000 characters: ', String cr,
     ((FileStream readOnlyFileNamed: path) converter: Latin1TextConverter new;
           next: 1000) asString
```

A directory node is a kind of file that contains directories and files. The methods files and directories are defined on the class DirectoryNode.

```
DirectoryNode>>directories
    | dir |
    dir := FileDirectory on: path.
    ^ dir directoryNames collect: [:each |
                        DirectoryNode new path: (dir fullNameFor: each)]

DirectoryNode>>files
    | dir |
    dir := FileDirectory on:  path.
    ^ dir fileNames collect: [:each |
       FileNode new path: (dir fullNameFor: each)]
```

The implementation shows the two responsibilities of a node: rendering itself (implemented in the text method), and calculating the nodes reachable from a node (in the directories and files methods).

Action Definitions. The user can perform some actions on selected files. Those are implemented in the class FileActor which inherits from Actor. Action are commands with user-interface information such as icon.

```
FileActor≫actionsForNode: aNode
    ^ {OBAction
            label: 'remove'
            receiver: self
            selector: #removeFile:
            arguments: {aNode}
            keystroke: $x
            icon: MenuIcons smallCancelIcon.
        OBAction
            label: 'rename'
            receiver: self
            selector: #renameFile:
            arguments: {aNode}}

FileActor≫removeFile: aNode
    "Remove the file designed by aNode"
    ...

FileActor≫renameFile: aNode
    "Rename the file designed by aNode"
    ...
```

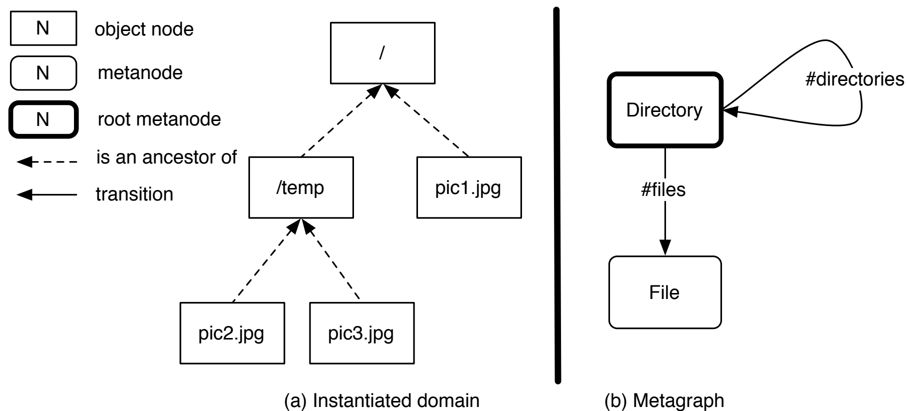

Fig. 4. A filesystem (as a graph) (a) and its corresponding metagraph (b).

Metagraph Definition. Figure 4 shows a metagraph describing a filesystem. Two metanodes, Directory and File, compose this metagraph. The navigation

between these nodes is defined by two transitions, files and directories. The starting point in a metagraph is designated by a root metanode.

The metagraph is implemented in the class FileBrowser. The methods defaultMetaNode and defaultRootNode are defined on the class side of FileBrowser. These methods define the metagraph and gives the root node, respectively:

```
FileBrowser class≫defaultMetaNode
    "returns the directory metanode that acts as the root metanode"

    | directory file |
    directory := OBMetaNode named: 'Directory'.

    file := OBMetaNode named: 'File'.
    file addActor: FileActor new.

    directory
        childAt: #directories put: directory;
        childAt: #files put: file;
        addActor: FileActor new.

    ^ directory

FileBrowser class≫defaultRootNode
    ^ DirectoryNode new path: '/'
```

When one of the two #directories and #files metaedges is traversed, the name of this metaedge is used as a message name sent to the metanode's node.

The complete source code of the file browser is freely downloadable[1].

3.3 Core Behavior of the Framework

The core of the OmniBrowser framework is composed of 8 classes. Figure 5 shows the framework and how it is used to define the file browser. We denote the Smalltalk metaclass hierarchy by a dashed arrow.

The metaclass of the class Browser is Browser class. It defines two abstract methods defaultMetaNode and defaultRootNode that return the root metanode and the root domain node, respectively. Subclasses override these methods, and they are called when a browser is instantiated. A browser is opened by sending the message open to an instance of the class Browser.

[1] http://www.squeaksource.com/FileOmnibrowser.html

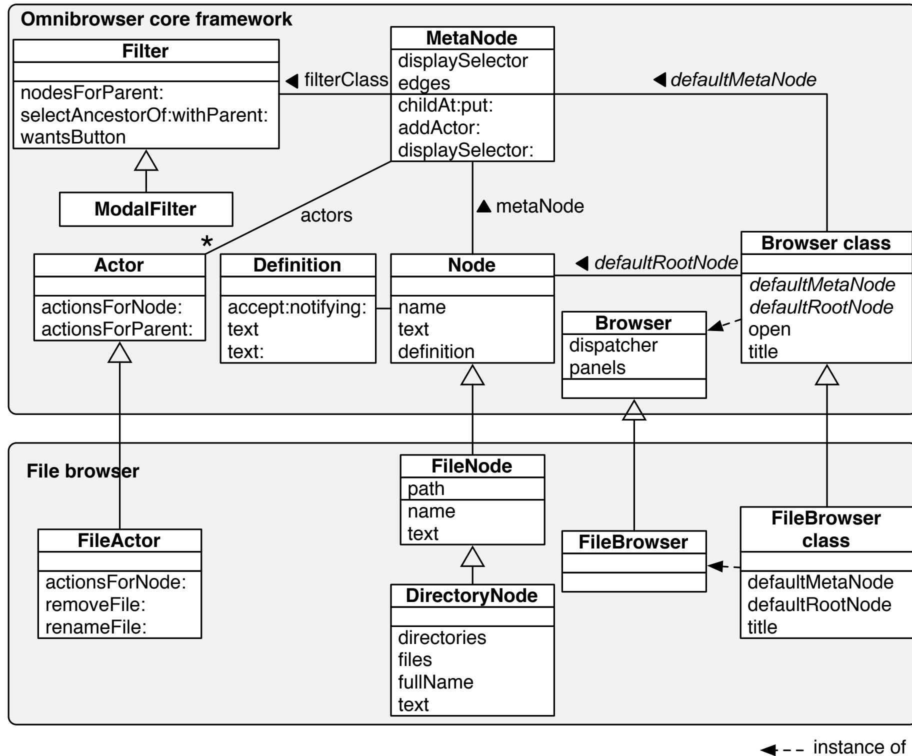

Fig. 5. Core of the OmniBrowser framework and its extension for the file browser.

The navigation graph is built with instances of the class MetaNode. Transitions are built by sending the message childAt: selector put: metanode to a MetaNode instance. This has the effect of creating a directed metaedge named selector from the metanode receiver of the message to metanode.

At runtime, the graph traversal is triggered by user actions (*e.g.*, pressing a button or selecting a list entry) which sends the metaedge's name to the node that is currently selected. Actors are attached to a metanode using the method addActor:. Rendering nodes is done by invoking the selector stored in the variable displaySelector in the metanode on the domain node.

The class Actor is normally instantiated by metanodes and is used to define node related actions. The method actionsForNode: may be overridden in subclasses to answer an ordered collection of actions. It is used to specify actions that are independent from any nodes. These actions are typically shown on a menu when no node is selected.

The class Node represents an element of the domain graph. Each node has a name. This name is used when lists of nodes are displayed in the navigation columns of the browser. When a node is selected in a list, information related to this node needs to be displayed in the bottom text pane. When the node is not supposed to be edited, the message text is sent to it, returning a string

displayed in the bottom pane. When it is editable, it is sent the message definition which needs to return an instance of a subclass of Definition. Note that the nodes do not need to be configured to be editable or not. When they implement a method definition, this will be used and the node will be editable. If that method is not present, then the method text is used.

When the browser is in a state where several transitions are available, it displays the possibilities to the user. From all the possible transitions, OmniBrowser framework fetches all the nodes that represent the states the user could arrive at by following those transitions and list them in the next column. Once a selection is made, the transition actually happens, the pane definition is updated and the process repeats.

As explained before, a filter or modal filter can be used to select only a number of outgoing edges when not all of them need to be shown to the user. This is useful for instance to display the instance side, comments, or class side of a particular class in the classic standard system browser (cf. Section 4). Class Filter is responsible for filtering nodes in the graph. The method nodesForParent: computes a transition in the domain metagraph. This method returns a list of nodes obtained from a given node passed as argument. The class Filter is subclassed into ModalFilter, a practical filter that represents transitions in the metagraph that can be traversed by using a radio button group in the GUI, as explained in the next section.

3.4 Glueing Widgets with the Metagraph

From the programmer's point of view, creating a new browser implies defining a domain model (set of nodes like FileNode and DirectoryNode, Figure 5), a metagraph intended to steer the navigation and a set of actors to define interaction and actions with domain elements. The graphical user interface of a browser is automatically generated by the OmniBrowser framework. The GUI generated by OmniBrowser framework is contained in one window, and it is composed of 4 kinds of widgets (lists, radio buttons, menus and text panes).

The layout of a browser can be redefined and use other widgets than the ones described above, but those are then not used by the metagraph. For instance, the OmniBrowser framework-based system browser uses a toolbar widget that allows a user to launch other kind of browsers like the variable and hierarchy browsers. We will not describe how to use other widgets, as this is outside the scope of this paper.

Lists. Navigation in OmniBrowser framework is rendered with a set of lists and triggered by selecting one entry in a list. Lists displayed in a browser are ordered and are displayed from left to right. Traversing a new metanode, by

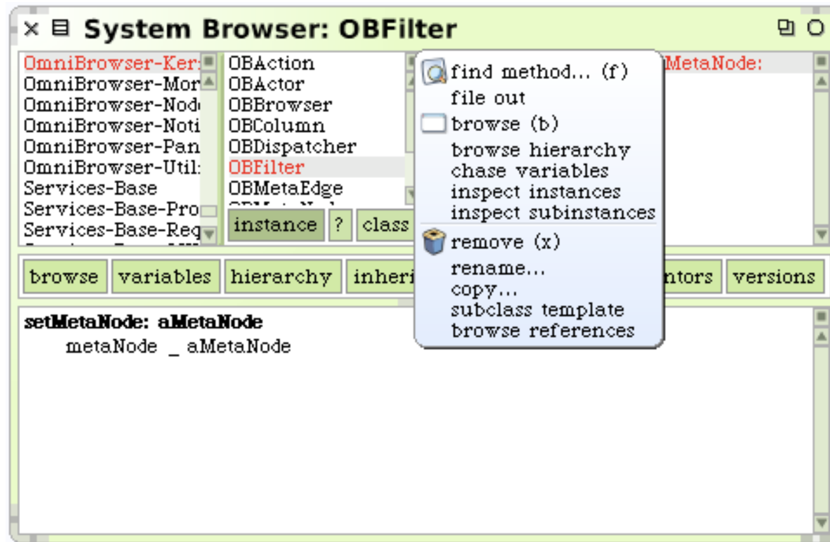

Fig. 6. Example of menu in the OmniBrowser framework system browser.

selecting a node in a list A, triggers the construction of a set of nodes intended to fill a list B. List B follows list A.

The root of a metagraph corresponds to the left-most list. The number of lists displayed is equal to the depth of the metagraph. The depth of the system browser metagraph (Figure 9) is 4, therefore the system browser has 4 panes (Figure 7). Because the metagraph of a filesystem may contain cycles (*i.e.*, a directory may contain directories, as shown in Figure 4), the number of lists in the browser increases for each directory selected in the right-most list. Therefore a horizontal scrollbar is used to keep the width of the browser constant, yet displaying a potentially infinite number of lists in the top half.

Radio buttons. A modal filter in the metagraph is represented in the GUI by a group of radio buttons, where each edge leading away from the filter is represented as one radio button in the group. Only one button can be selected at a time in the radio button group, and the associated choice is used to determine the outgoing edge. For example, the second list in the system browser contains the three buttons instance, ? and class as shown the transition from the environment to the three metanodes class, class comment and metaclass in Figure 7.

Menus. A menu can be displayed for each list widget of a browser. Typically such a menu displays a list of actions that can be executed by a browser user. These actions enable interaction with the domain model, however they do not allow further navigation in the metagraph.

Figure 6 shows an example of a menu offering actions related to a class. These correspond to the list of actions returned by the method actionsForNode: in the class ClassActor.

Text pane. When a node is selected in a list, some information related to this node is displayed in a text pane. Committing a change in the text pane sends the message accept: newText notifying: aController to the definition shown in this pane. A browser contains only one text pane.

4 The OmniBrowser-based System Browser

In this section we show how the framework is used to implement the traditional class system browser.

4.1 The Smalltalk System Browser

The System Browser is probably the most important tool offered by the Smalltalk programming environment. It enables code navigation and code editing. Figure 7 shows the graphical user interface of this browser, and how it appears to the Smalltalk programmer.

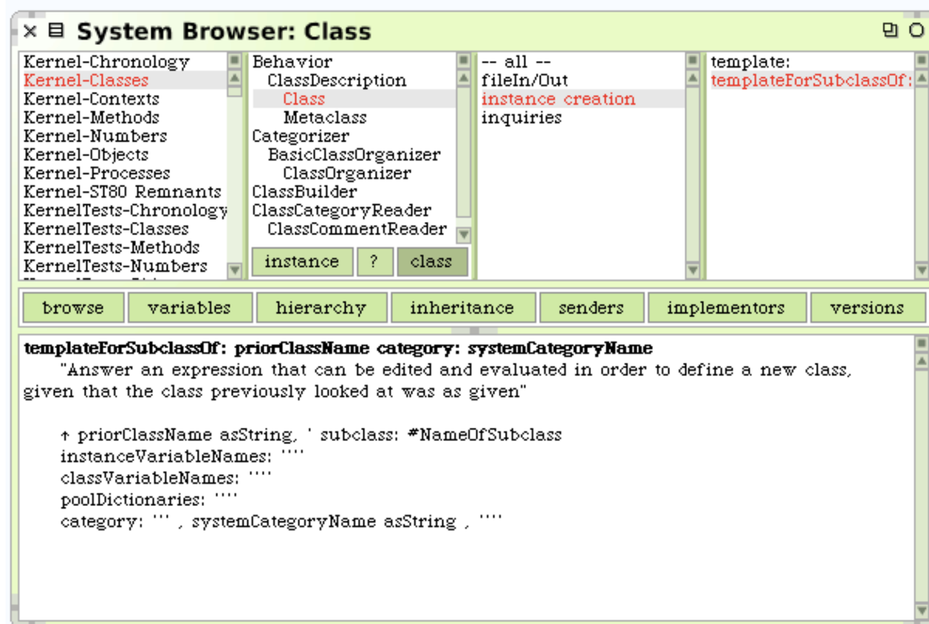

Fig. 7. OmniBrowser framework based Smalltalk system browser.

This browser replicates the traditional four panes system browser discussed in Section 2. The system browser is composed of four lists (upper part) and a text pane (lower part). From left to right, the lists represent (i) class categories, (ii) classes contained in the selected class category, (iii) method categories defined in the selected class to which the – all – category is added, and (iv) the list of methods defined in the selected method category. On Figure 7, the class

named Class, which belongs to the class category Kernel-Classes is selected. Class has three methods categories, plus the – all – one. The method templateForSubclassOf:category contained in the instance creation method category is selected.

The lower part of the system browser contains a large text pane that displays information about the current selection in the lists. Selecting a class category makes the render display a class template intended to be filled out to create a new class in the system. If a class is selected, then this pane shows the definition of this class. If a method is selected, then the definition of this method is displayed. The text contained in the pane can be edited to create or redefine a class or a method.

In the upper part, the class list contains three buttons (titled instance, ? and class) to let one switch between different "views" on a class: the class definition, its comment and the definition of its metaclass. Just above the pane, there is a toolbar intended to open more specific browsers like a hierarchy browser and a variable access browser.

4.2 System Browser Internals

The Omnibrowser-based implementation of the Squeak system browser is composed of 19 classes (2 actors, 2 classes for the browser, 3 classes for the definitions of classes, methods and organization, 10 classes defining nodes and 2 utility classes with abstractions to help link the browser and the system) and 220 methods. Figure 8 shows the classes in the OmniBrowser framework that need to be subclassed to produce the system browser. Note that the two utility classes are not represented on the picture.

Compared to the default implementation of the Squeak System Browser, the OmniBrowser-based implementation uses less code and has a better structure. In addition other code-browsers can freely reuse these parts.

Figure 9 depicts the metagraph of the system browser. The metanode environment contains information about class categories. The filter is used to select what has to be displayed from the selected class (*i.e.*, the class definition, its comment or the metaclass definition). A class and a metaclass have a list of method categories, including the – all – method category that shows a list of methods.

Widgets notification. Widgets like menu lists and text panes interact with each other by triggering events and receiving notifications. Each browser has a dispatcher (referenced by the variable dispatcher in the class Browser) to pass events between widgets. The vocabulary of events is the following one:

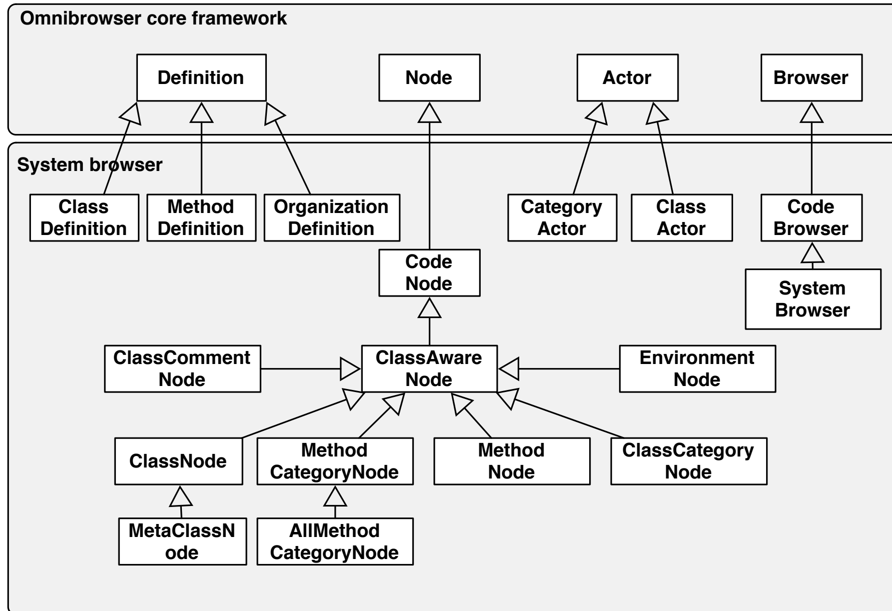

Fig. 8. Extension of the OmniBrowser framework to define the system browser.

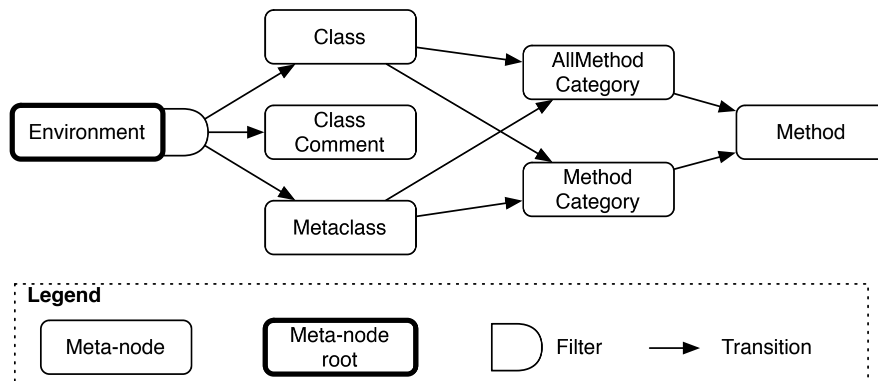

Fig. 9. Metagraph of the system browser.

- refresh is emitted when a complete refresh of the browser is necessary. For instance, if a system change occurs (for example, when a class is added to the system), this event is triggered to order a complete redraw.
- nodeSelected: is emitted when a list entry is selected with a mouse click.
- nodeChanged is emitted when the node that is currently displayed changes. This typically occurs when one of buttons related to the class is selected. For example, if a class is displayed, pressing the button instance, class or comment triggers this event.
- okToChangeNode is emitted to prevent loosing changes to the content of a text pane while editing. This might happen when a node is being edited, but the user navigates to another node before accepting (*i.e.*, compiling) the changes.

Each graphical widget making up a browser can function both as listener and as emitter of events. Creation and registration of widgets as event listener and emitter is completely transparent to the end user.

State of the browser. Contrary to the original Squeak system browser where each widget state is contained in a dedicated variable, the state of a OmniBrowser framework-based browser is defined as a path in the metagraph starting from the root metanode. Each metanode on this path is associated to a domain node. This preserves the synchronization between different graphical widgets of a browser.

5 The Coverage Browser

The coverage browser is an extension of the system browser to show the coverage of code by unit tests. It extends the system browser in two ways. First of all it appends the percentage of elements covered by tests to the elements in the lists making up the browser. Secondly it adds a fifth pane that lists the unit tests that test a selected method. A screenshot is shown in Figure 10. It shows us that 39% of the class UUID is covered by tests, and that the method initialize is covered at 100% by the tests shows in the right-most pane. One of these test is testCreation.

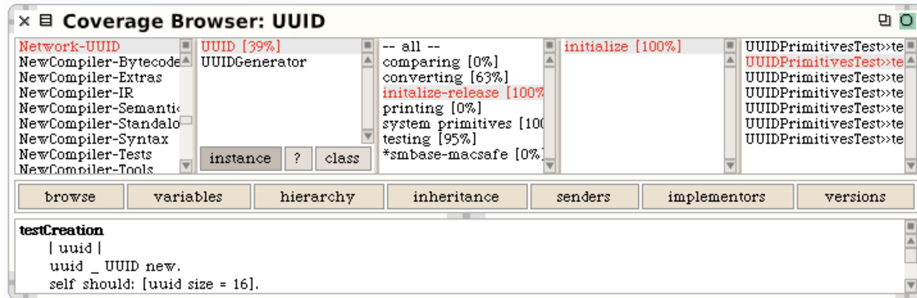

Fig. 10. Screenshot of the coverage browser.

The coverage browser is composed of 11 classes (1 class for the browser, 5 actors and 5 nodes). Figure 11 illustrates how classes in OmniBrowser and in the system browser are extended to define this new browser. The metagraph is depicted in Figure 12 and is identical to the system browser except with a new Method Coverage metanode. The depth of the graph, which is 5, is reflected in the number of list panes the browser is composed of.

The Coverage browser is freely available[2].

[2] www.squeaksource.com/Coverage.html

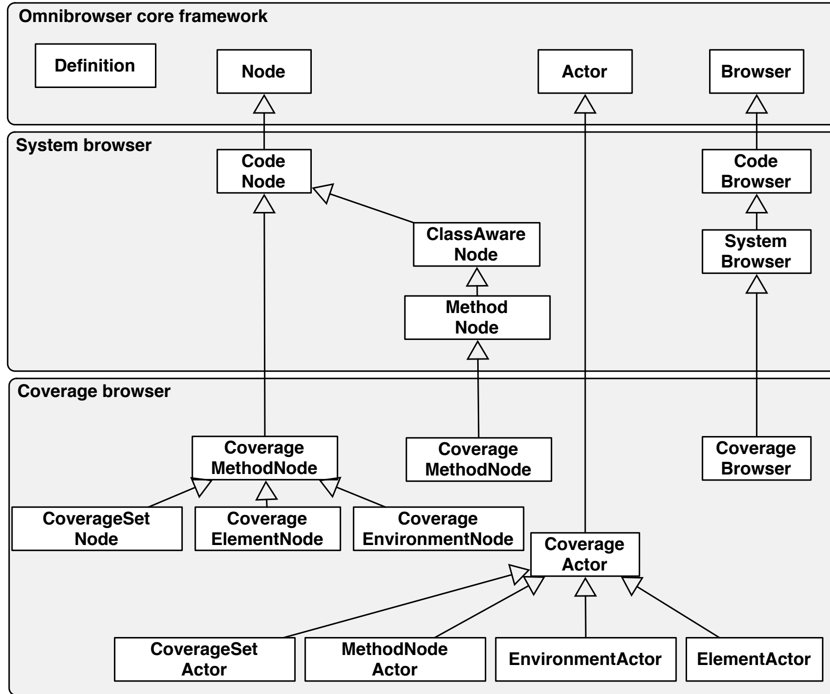

Fig. 11. Extension of Omnibrowser and system browser to define the coverage browser.

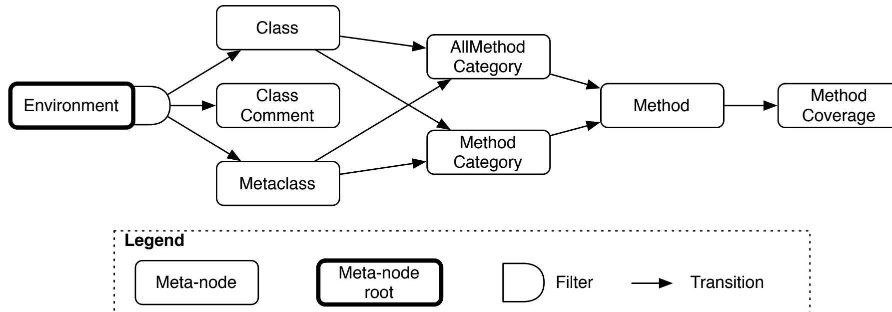

Fig. 12. Metagraph for the coverage browser.

6 Multiple Views with the Duo Browser

Previous sections described a file browser, a system code browser and the coverage browser. Those browsers use a single view to browse a domain. For example, the system code browser displays the list of class categories in the left-most pane, and a list of classes for the selected category in the second pane. The hierarchy browser on the other hand shows an inheritance hierarchy in the left-most pane, and the implementation of the selected class in the rest of the panes. This makes it easy to browse the implementation of methods in the superclasses, which is frequently needed in non-trivial inheritance hierarchies.

The system code browser and the hierarchy browser are very useful since they offer a dual view on the implementation of a system, respectively one that lets a developer work on her classes, and one that shows in detail the inheritance relationships.

While both views are needed, they are offered in separate browsers, and users frequently need to open a hierarchy browser from a full system browser and vice versa. The Duo Browser integrates both views in a single browser. This idea in itself is not new, and was pioneered by the System Browser in the VisualWorks Smalltalk distribution. The Duo Browser uses the OmniBrowser framework to implement the core idea of this browser, and was implemented in a matter of days.

The principle of the Duo Browser is simple: it offers to show either the packages in the system and the classes in each package, or the inheritance hierarchy and the packages for each of the classes in the inheritance hierarchy. Two buttons allow to switch between these views. Crucial to the functioning of the browser is that when switching views, the selections of the old view are used to determine the contents in the other view.

Furthermore this setup makes it easy to properly integrate class extensions in the browser. A class extension is a method that is defined in a package, but that belongs to a class that is not defined in that package. This can be used to add and even replace methods of existing classes.

Figure 13 gives a screenshot of the Duo Browser. The left-most pane lists all Monticello packages in the image, with the package DuoSystemBrowser3.9 currently being selected. The second pane shows a hierarchical list with the classes in this package, as well as the class extensions (when the classes are in italic). The class extension OBClassAwareNode is currently selected.

Switching views is done with the package and hierarchy buttons below the first pane. Figure 14 shows the result when clicking the hierarchy button with the browser being in the state shown in Figure 13. The left-most pane now contains the class hierarchy that includes the class OBClassAwareNode, and lists all classes until the root class. The second pane shows all packages that either define the class selected in the left pane, or that provide extensions (the italic entries).

From a usability point of view we stress that it is important that the selections are taken into account when switching views. We saw this in the example: after switching, the left pane is centered around the class OBClassSwareNode. When we would press the package button we would again be in the same state as shown in the first figure.

As depicted in Figure 15, use of filters in the metagraph enables a browser

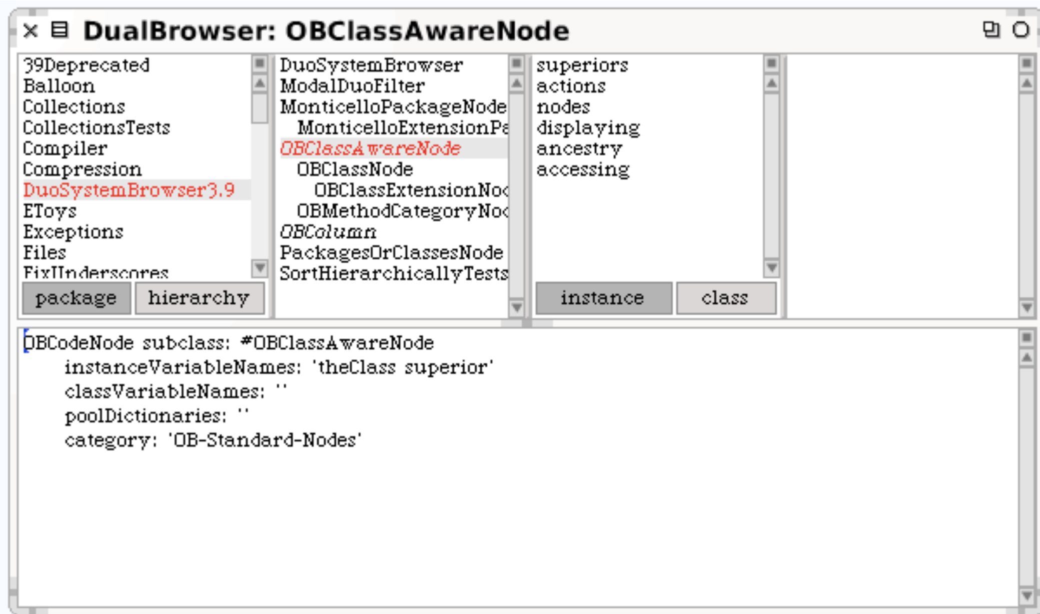

Fig. 13. The Duo Browser shows on the left-most pane a list of packages. The second pane shows the list of classes belonging to the DuoSystemBrowser3.9.

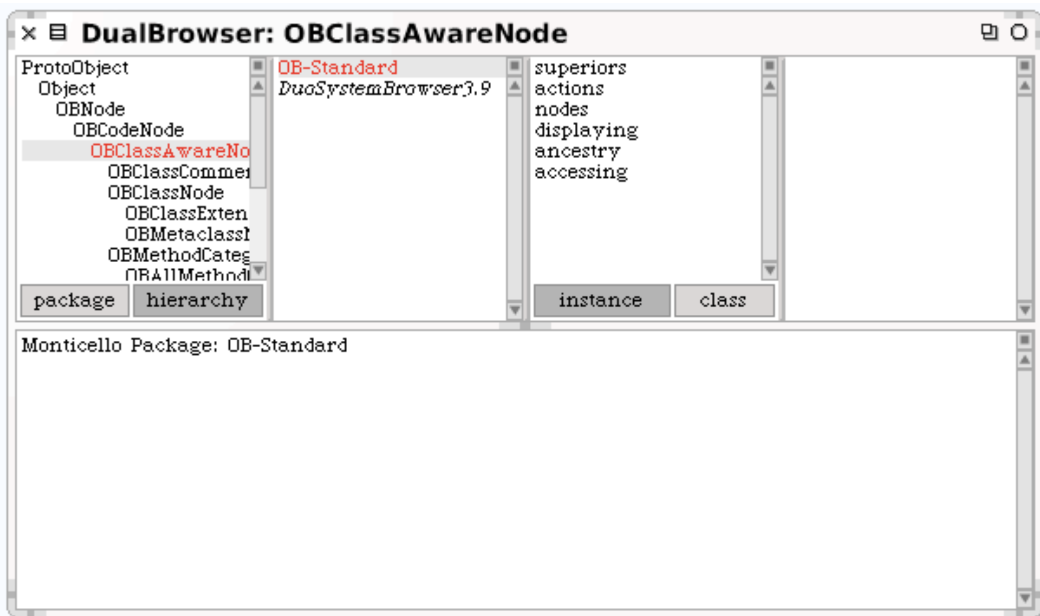

Fig. 14. The class hierarchy containing the class OBClassAwareNode is displayed on the left-most pane. The second pane lists the packages that define or extend this class.

to offer different views to navigate through a domain. Views can be switched using the filter installed on the metagraph root.

The implementation required one extension to the OmniBrowser framework. As was explained before, the OmniBrowser framework layouts the panes from

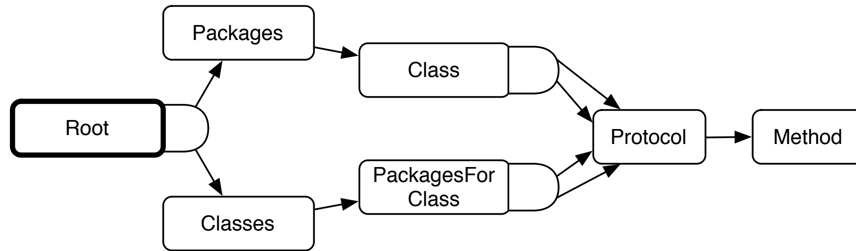

Fig. 15. The Duo Browser metagraph reflects the ability of having multiple views by using filters.

left to right, where a selection in one pane is used to show the content of a pane that is put to the right of that pane. A consequence of this approach is that each pane has a well-defined kind of items that does not change. In the System Browser, for example, the left-most pane will always display classes.

This is not true for the Duo Browser: the left-pane can display either Monticello packages or classes, and the same holds for the second pane. Moreover, pressing the button results in swapping the contents of these panes, while keeping the selections. Implementing this with the OmniBrowser framework required two tricks:

- a node class was added (PackagesOrClassesNode) that can play the role of either classes or packages. So it can be used in either pane and will work correctly. This is also used as the root node.
- a new filter class was added (ModalDuoFilter) that has as responsibility to keep track of the selected elements in both panes and, when switching views, re-selecting the right elements again in the switched panes.

On the one hand these tricks show some shortcomings of the framework (that are further discussed in Section 8), but on the other hand it also showed that by slightly extending the framework (adding a switch class) and properly understanding how it works, this different behaviour could easily be accomodated. This shows the simplicity and the openness of the OmniBrowser framework.

The Duo Browser is freely downloadable[3].

7 Rendering Dynamic Information with Dynamic Protocols

Up to now we have been describing browsing tools that navigate through a static domain (*e.g.*, composed of classes and methods). This section illustrates an extension of the system code browser to render some dynamically

[3] http://www.squeaksource.com/DuoSystemBrowser.html

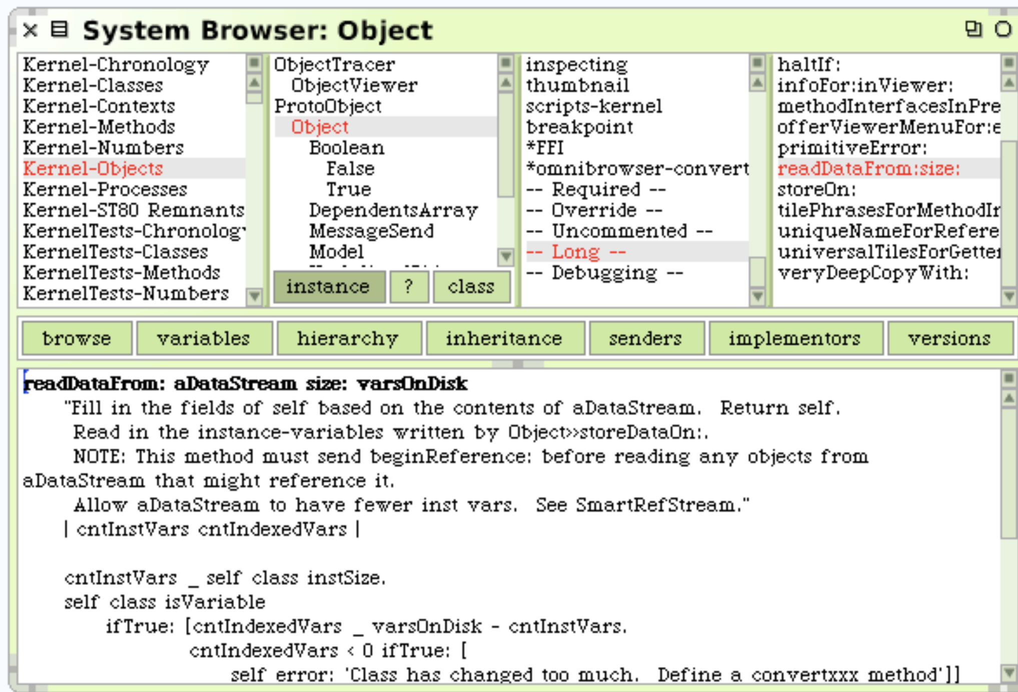

Fig. 16. Dynamic information is accessible from the third upper pane.

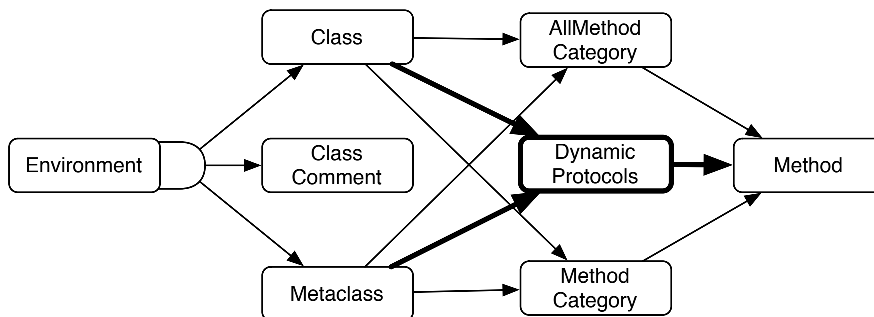

Fig. 17. Dynamic protocols extends the metagraph of the system code browser. Extension are shown in bold.

computed information like methods that are uncommented, super calls contained in methods and methods for which their source code is considered too long.

A list of dynamic protocols is displayed in the third pane in the upper pane of the system code browser. Figure 16 shows a screenshot of it. On this figure, the dynamic procotol -- Long -- is selected. The effect is that the right-most upper pane lists those methods for which their source code is considered too long (*e.g.*, greater than 20 lines of code in our case).

The metagraph of the system code browser is extended as the following (extensions are written in bold):

```
OBCodeBrowser≫ defaultMetaNode
    ...
    metaclass childAt: #allCategory put: allMethodCategory;
        childAt: #categories put: methodCategory;
        childAt: #dynamicProtocols put: protocols; "Added line"
        ... .
    class childAt: #allCategory put: allMethodCategory;
        childAt: #categories put: methodCategory;
        childAt: #dynamicProtocols put: protocols; "Added line"
        ... .
```

The resulting new metagraph is illustrated in Figure 17. The method dynamicProtocols is defined on the class OBClassNode, which in essence contains the following code:

```
OBClassNode≫ dynamicProtocols
    ↑ self allActivatedProtocols
        collect: [:dpClass | dpClass on: aClass]
        thenSelect: [:dp | (dp getSelectorsFor: aClass) notEmpty].
```

The method allActivatedProtocols returns a list of classes that implement different dynamic protocols. Those protocols are represented by a subclass of the DPAbstract class. The method getSelectorsFor: returns the list of methods that have to be displayed on the fourth upper pane of the browser. For instance this method is redefined in the class DPLongMethod that returns the list of methods having more than 20 lines of source code.

Dynamic protocols is freely available [4].

8 Evaluation and Discussions

Several other browsers such as a browser specifically supporting traits [DNS+06] have been developed using OmniBrowser framework demonstrating that the framework is mature and extensible [RJ97]. Figure 18 shows some browsers that are based on OmniBrowser framework. We now discuss the strengths and limitations of the OmniBrowser framework.

8.1 Strengths

Ease of use. Like with any good framework, instantiating the OmniBrowser framework following the framework's intention makes it easy to specify ad-

[4] http://www.squeaksource.com/DynamicProtocols.html

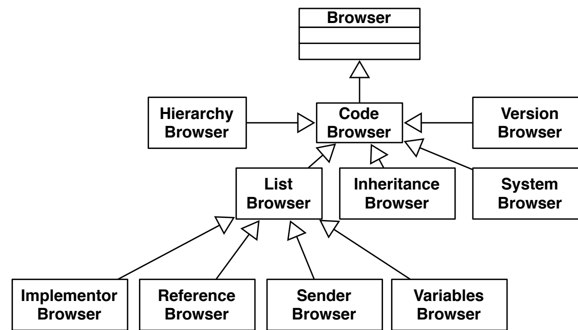

Fig. 18. Some code browsers developed using OmniBrowser framework.

vanced browsers. The fact that the browser navigation is explicitly defined in one place lets the programmer understand and control the tool navigation and user interaction, and removes the burden to explicitly create and glue together the UI widgets and their specific layout. Additional decorating widgets such as extra-menu are possible and are defined independently. Still the programmer focuses on the key domain of the browser: its navigation and the interaction with the user.

Explicit state transitions. Maintaining coherence among different widgets and keeping them synchronized is a non-trival issue that, while well supported by GUI frameworks, is often not well used. For instance, in the original Squeak browser, methods are scattered with checks for nil or 0 values. For instance, the method classComment: aText notifying: aPluggableTextMorph, which is called by the text pane (F widget) to assign a new comment to the selected class (B widget), is:

```
Browser≫classComment: aText notifying: aPluggableTextMorph
    theClass := self selectedClassOrMetaClass.
    theClass
        ifNotNil: [ ... ]
```

The code above copes with the fact that when pressing on the class comment button, there is no warranty that a class is selected. In a good UI design, the comment class button should have been disabled however there is still checks done whether a class is selected or not. Among the 438 accessible methods in the non Omnibrowser-based Squeak class Browser, 63 of them invoke ifNil: to test if a list is selected or not and 62 of them invoke ifNotNil:. Those are not isolated Smalltalk examples. Code that uses JHotDraw [JHo] uses a pattern to check for a nil value of variables that may reference graphical widgets.

Such situations do not happen in OmniBrowser framework, as metagraphs are declaratively defined and each metaedge describes one action the user can do in the browser, and all states a browser can be in are explicitly and fully described.

Separation of domain and navigation. The domain model and its navigation are fully separated: a metanode does not and cannot have a reference to the domain node currently selected and displayed. Therefore both can be reused independently.

8.2 Limitations

Hardcoded flow. As any framework, OmniBrowser framework constrains the space of its own extension. OmniBrowser framework does not support the definition of navigation not following the left to right list construction (the result of the selection creates a new pane to the right of the current one and the text pane is displayed). For example, building a browser such as Whiskers that displays multiple methods at the same time would require to deeply change the text pane state to keep the status of the currently edited methods.

Currently selected item. The OmniBrowser framework does not easily support the building of advanced browsing facilities such as those found in the VisualWorks standard browser. In VisualWorks, it is possible to select a package, then select one class of this package and then to see the inheritance hierarchy of this class within the context of the previously selected package. The problem is that conceptually the selected item is not part of the state representation. It is possible using UI events between the widgets to implement such functionality, but such support has to be implemented manually.

9 Related Work

MVC. The Model-View-Controller [KP88,Ree,Ree79] promotes a distinction between three important roles (namely data, output and interaction) that should be reflected in the design of a user interface framework. Those roles were reflected in three abstract superclasses: Model, View, Controller. Still for system browsers, developers consider the model as the entities of the domain and do not have explicit or meta entities describing the navigation within the domain model. Note also that a controller in MVC captures the interaction of users with a widget, and passes this information to the model. The level of abstraction, however, is lower than what is offered by the *Actor* in the OmniBrowser framework, which is not programmed in terms of a widget but in terms of the domain entities.

HotDraw. The state transitions between the possible tools in HotDraw [Joh92] are driven by an explicit state machine and follow an explicit transition structure. There is a graphical editor (constructed with HotDraw itself) to construct

the view and edit the state machine. The goal of the state machine is similar to the goal of the metagraph in the OmniBrowser framework: to make navigation explicit.

HotDraw and the OmniBrowser framework differ in the way that users instantiate the framework. HotDraw makes extensive use of meta programming, and users therefore need to know what methods to implement on what classes in order to successfully use the framework. For example, every tool offered by the editor to manipulate graphical figures has a button on the toolbar. Therefore the editor has to offer icons to use as buttons for every tool name mentioned in the method toolNames on the metaclass of the editor. These icons are returned by methods that reside on the class side of the editor meta class. However, there has to be some mapping between the name of the tool, and the name of the method used to provide the icon. This mapping is actually a naming convention which is all but implicit. Hence the toolbar is constructed by enumerating all the tool names provided by the toolNames method, and retrieving the icon for this tool using the naming convention. Using meta programming for instantiating the framework is sometimes quite cumbersome. Instantiating the OmniBrowser framework is done by subclassing, which makes it easier to understand what needs to be done.

HyperCard. Conceptually, a HyperCard [Goo98] application is a stack of cards. Each card contains some information and links to other cards in the same or other stacks. The information on the cards is shown using text and graphics. The links to other cards are presented as buttons, typically completed with an icon representing the destination card. A user of HyperCard browses the cards of a stack using the link button. Only one card of a stack is displayed at a time. Clicking a link button results in the display of the destination card. When a stack has not only information to be displayed, but also has to exhibit an active behavior, the stack designer has to develop cards by means of a scripting level, on which programming in the dedicated language HyperTalk is supported. There is no concept like the metagraph in OmniBrowser that describes the overal navigation of a domain graph: every card locally describes its links with other cards.

ApplFLab. Steyaert *et al.* defined the notion of reflective application builder [SHDB96] with as explicit goal to be able to construct and reuse (parametrizable) user interface components. ApplFLab was used to construct several domain specific user interfaces, including browsers in development environments [Wuy96].

ApplFLab structures a software program using four distinct kinds of components:

- a *user interface component* controls the display and the user interaction of

a particular piece of information, supplied by the domain model. Note that this component is parametrized by the domain model, and therefore can be reused across different domains.

- an *application model* manages the global behavior of group of interface components. It is responsible for the user interface logic and controls user interface. A same application model can be reused on different domain models and a domain model can have several application models in parallel.
- a *domain model* models the overall functionality of the problem domain and maintains user interface independent constraints.
- a set of *aspects* is needed to separate the domain model from the user interface component.

Interaction between these four components is based on emitting events and receiving notifications. There are three kinds of event: *display*, *notify* and *control*.

The advantage of ApplFLab lies in its notion of parametrized user interface component. A user interface component consists of a GUI description, and parameters to link the component to the domain or to specify other information when it is used in an application. The components are plugged together to form applications. One could for example build a list component, and parametrize it with categories, classes, protocols and selectors to get the four top elements that make a System Browser (as shown in Section 4.1). Combine it with a Text component and the System Browser is complete.

While both ApplFLab and the OmniBrowser make it easy to build browsers, there are some differences. The OmniBrowser is a domain specific approach for building browsers, while ApplFLab is general. So when using ApplFLab to build browser, browser specific components need to be built first, for example to get the left-to-right selection behavior that is built-in with OmniBrowser. ApplFLab also had a steeper learning curve, since building a good reusable component (be it a visual one or a regular one) remains fairly difficult. On the other hand, OmniBrowser offers more built-in behavior which makes it easier to use but also forces certain behavior that might not always be wanted.

ThingLab. Freeman-Benson and Maloney [FB89] wrote ThingLab II, an object-oriented constraint system for a *direct manipulation* user interface implemented in Smalltalk-80. In ThingLab II, user-manipulable entities are collections of objects know as *Things*. ThingLab II provides a large number of primitive Things equivalent to the operations and data structures provided in any high-level language: numerical operations, points, strings, bitmaps, conversion, etc.

A thing is constructed from things objects and constraint objects. Higher-level things can be built out of the lower-level ones. Constraints are either satisfied

or they are not satisfied, and they are simple declarative declarations that do not hold state. Browser navigation can be expressed as constraints on the different elements that compose a browser. But there is no explicit distinction between the domain and its navigation.

10 Conclusion

Smalltalk is known for its advanced development environment, featuring advanced browsers that let developers navigate and change code relatively easily.

Building browsers, however, is a daunting task. The main problem is that every navigation action performed by a user in a widget changes the state of that (and possibly other) widgets. Given the high number of possible navigation actions, the complexity of managing the navigation by managing the states of the browser is a very complex task. This can be seen in most current browser implementations, which are complex and hard to extend because the navigation is implicitly encoded in the management of the state of the widgets.

To make it easier to build and extend browsers, this paper introduces a framework for building browsers that is based on modeling user navigation through an explicit graph. In this framework, browsers are built by modeling the domain with *nodes*, expressing the navigation with a *metagraph* and describing the interaction between the browser and the domain through *actors*. The framework uses these descriptions to construct a graphical application. The top half of the application uses lists that allow the user to navigate the described domain. The bottom half of the pane allows to visualize and edit nodes selected in the top half.

The framework is implemented in Squeak Smalltalk through the OmniBrowser framework. The paper showed three concrete instantiations of the framework: a file browser to navigate a file system,a reimplementation of the ubiquitous Smalltalk System Browser, and a code coverage browser. There are more instantiations of the browser that we have not discussed in this paper but that are available. The validation shows that the goals of the frameworks are met. Building the System Browser with the OmniBrowser framework shows that the code is lots simpler. The Code Coverage browser shows that it is easy to extend an existing browser.

For future work we plan to enhance the OmniBrowser framework with the ability to have multiple text panes to be part of a browser. We also plan to extend the framework to support more and richer widgets (such as toolbars and flaps). Last but not least we want to investigate how we can extend the metagraph to look at other ways of navigating it.

Acknowledgment. We would like to thank Damien Cassou for his precious review on an early draft version. Thanks also to Niklaus Haldimann and Stefan Reichnart for their use of the OmniBrowser framework.

References

[ABW98] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998.

[BDPW06] Alexandre Bergel, Stéphane Ducasse, Colin Putney, and Roel Wuyts. Meta-driven browsers. In *Proceedings of the International Smalltalk Conference (ISC 2006)*, LNCS. Springer, 2006. To appear.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 215–230, October 1993.

[DNS+06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, March 2006.

[FB89] Bjorn N. Freeman-Benson. A module mechanism for constraints in Smalltalk. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 389–396, October 1989.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[Gol84] Adele Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984.

[Goo98] Danny Goodman. *The Complete HyperCard 2.2 Handbook*. iUniverse, 1998.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[Hal05] Niklaus Haldimann. A sophisticated programming environment to cope with scoped changes. Informatikprojekt, University of Bern, December 2005.

[Hon98] Koen De Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. PhD thesis, Vrije Universiteit Brussel,Departement of Computer Science, Brussels — Belgium, December 1998.

[JHo] Jhotdraw: a java gui framework for technical and structured graphics. http://www.jhotdraw.org.

[Joh92] Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings OOPSLA '92*, volume 27, pages 63–76, October 1992.

[KP88] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August 1988.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[RBJO96] Don Roberts, John Brant, Ralph E. Johnson, and Bill Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.

[Ree] Trygve M. H. Reenskaug. The model-view-controller (mvc) – its past and present. JavaZONE, Oslo, 2003.

[Ree79] Trygve M. H. Reenskaug. Models - views - controllers, December 1979. http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf.

[RJ97] Don Roberts and Ralph E. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design 3*. Addison Wesley, 1997.

[SB04] Nathanael Schärli and Andrew P. Black. A browser for incremental programming. *Computer Languages, Systems and Structures*, 30(1-2):79–95, 2004.

[SHDB96] Patrick Steyaert, Koen De Hondt, Serge Demeyer, and Niels Boyen. Reflective user interface builders. In Chris Zimmerman, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection*, pages 291–309. CRC Press — Boca Raton — Florida, 1996.

[SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proceedings of OOPSLA '96 (International Conference on Object-Oriented Programming, Systems, Languages, and Applications)*, pages 268–285. ACM Press, 1996.

[SM88] Pedro Szekely and Brad Myers. A user interface toolkit based on graphical objects and constraints. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, volume 23, pages 36–45, November 1988.

[WD04] Roel Wuyts and Stéphane Ducasse. Unanticipated integration of development tools using the classification model. *Journal of Computer Languages, Systems and Structures*, 30(1-2):63–77, 2004.

[Wuy96] Roel Wuyts. Class-management using logical queries, application of a reflective user interface builder. In I. Polak, editor, *Proceedings of GRONICS '96*, pages 61–67, 1996.