# Meta-Driven Browsers

Alexandre Bergel[1], Stéphane Ducasse[2], Colin Putney[3], Roel Wuyts[4]

[1] DSG, Trinity College Dublin, Ireland,
[2] Language and Software Evolution – LISTIC, Université de Savoie,
[3] Wiresong,
[4] Lab for Software Composition and Decomposition, Université Libre de Bruxelles,

**Advances in Smalltalk**
**Proceedings of 14th International Smalltalk Conference (ISC 2006),**
**LNCS, vol. 4406, Springer, 2007, pp. 134-156**

**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, and a coverage browser.

**Keywords**: Tools, MetaModeling, UI, Browsers, Squeak

## 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: the *Refactoring Browser* [FBB+99,RBJO96,RBJ97] which was the first system browser supporting refactoring, the *StarBrowser* [WD04] which supports smart groups, a browser for incremental development supporting visual feedback of undefined methods [SB04] and the *Whiskers* browser that shows multiple methods at the same time maximizing the screen space. StrongTalk, a more exotic Smalltalk version featuring optional typing, offered 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 included 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 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. In OmniBrowser framework, a browser is a graphical list-oriented tool to navigate and edit an arbitrary domain. The most common representative of this category of tools is the Smalltalk system browser, which is used to navigate and edit Smalltalk source code. In OmniBrowser framework, a browser is described by a domain model and a metagraph which specifies how the domain space is navigated through. Widgets such as list menus and text panels are used to display information gathered from a particular path in the metagraph. Although widgets are programmatically composed, the OmniBrowser framework framework supports their interaction.

The contributions of this article are: the description of a metadriven framework to build system browsers and the application of the framework to build some tools. 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 OmniBrowser framework. In Section 4 we present the OmniBrowser-based system browser and in Section 5 we describe the coverage code browser. In Section 6 we discuss about properties of the OmniBrowser framework. In Section 7 we provide an overview of related work. In Section 8 we conclude by summarizing the presented work.

## 2 Defining and Maintaining the State of a Graphical User Interface

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

The state of a graphical user interface (GUI) is defined as a collection of the states of the widgets making up the interface. The state of a widget refers to the state the widget is in. It is modified whenever an end-user performs an action on this widget such as clicking a button or selecting an entry in a menu. Therefore, a GUI has a high number of different states. Asserting the validity for each of these states is crucial to avoid broken or inconsistent interfaces.

Given the potential high number of different states of a GUI, asserting the validity of a GUI is a challenging task. Let's illustrate this situation with the Smalltalk system browser, a graphical tool to edit and navigate into Smalltalk source code.

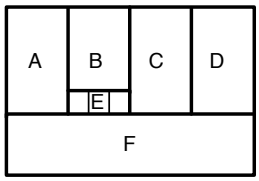


**Fig. 1.** The traditional Smalltalk System Browser roughly depicted.

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 need to be 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 open 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 an user action always checks whether the state is actually correct or not nil.

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

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.

### 3.1 Overview of the OmniBrowser framework

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

**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

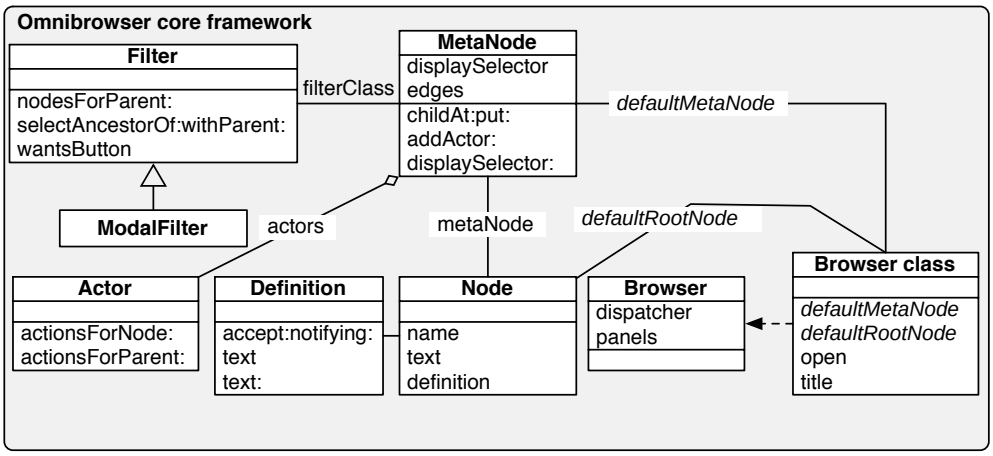


**Fig. 2.** Core of the OmniBrowser framework.

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, 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.** The metagraph describes a state machine. When the browser is in a state where there are two transitions available. The user is the one that decides which transition to follow. To allow that to happen OmniBrowser framework 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. Note that the transition is not actually make yet, and the definition pane is still displaying the class definition. Once a click is made, the transition actually happens, the pane definition is updated (and perhaps other panes such as button bars) and it gathers the next round of possible transitions.

A filter provides a strategy for filtering out some of the nodes from the display. If a node is the starting point of several edges, a filter is needed to filter out all but one edges to determine which path has to be taken in the metagraph.

**Definition.** While navigating in the domain space, information about the selected node is displayed in a dedicated textual panel. If edition is expected by the browser user, then a definition is necessary to handle commitment (*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 panel 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 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
```

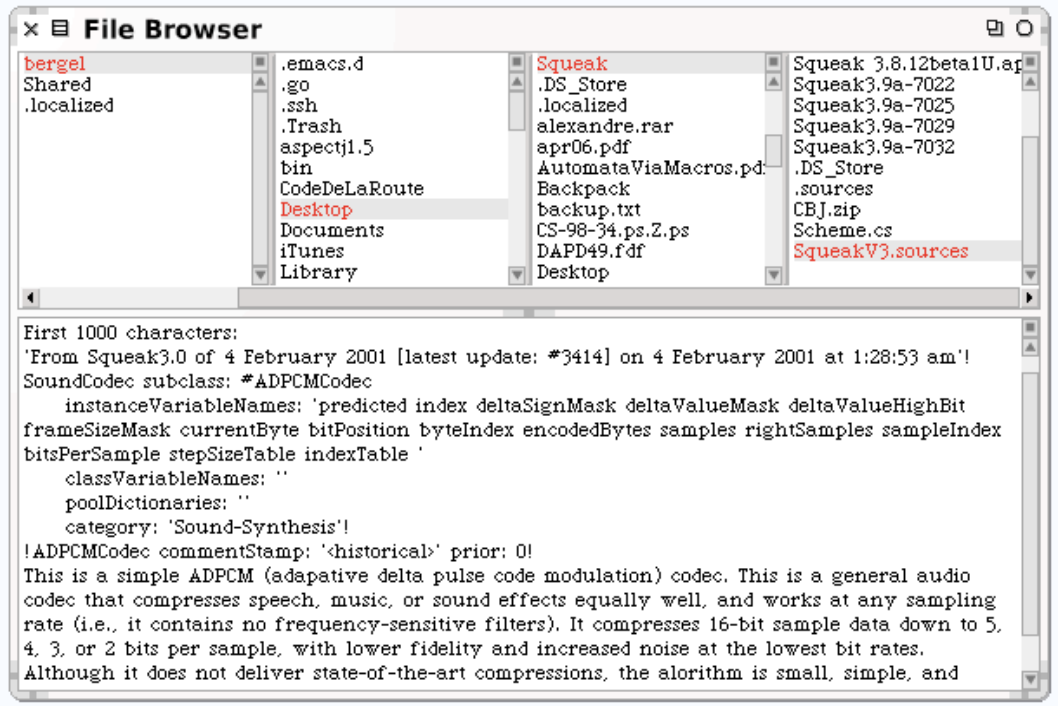


**Fig. 3.** A minimal file browser based on OmniBrowser.

```
    ^ '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"
    ...
```

**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.
```

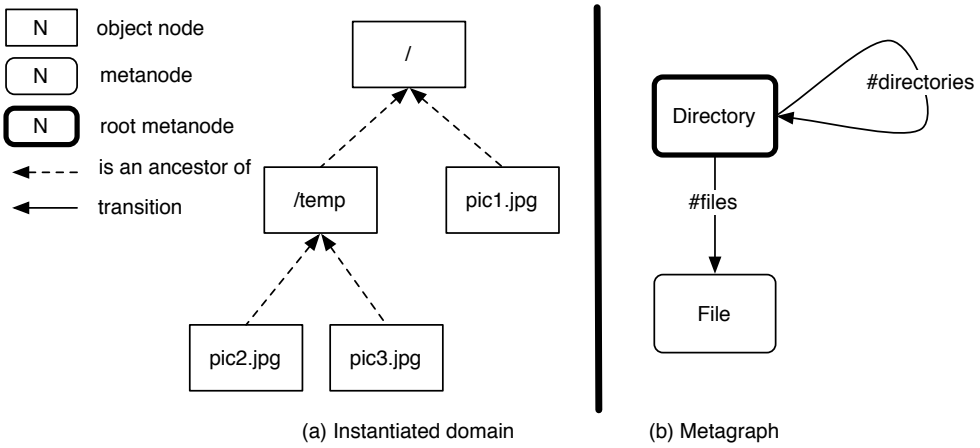


**Fig. 4.** A filesystem (as a graph) (a) and its corresponding metagraph (b).

```
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.

### 3.3 Core Behavior of the Framework

The core of the OmniBrowser framework is composed of 8 classes (Figure 2). 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. These methods are abstract, they therefore need to be overridden in subclasses. These methods are called when a browser is instantiated. The methods defaultMetaNode and defaultRootNode returns the root metanode and the root domain node, respectively. A browser is opened by sending the message open to an instance of the class Browser.

The navigation graph is built with instances of the class MetaNode. Transitions are built by sending messages childAt: selector put: metanode to a MetaNode instance. These has the effect to create a metaedge named selector leading away the metanode receiver of the message and metanode.

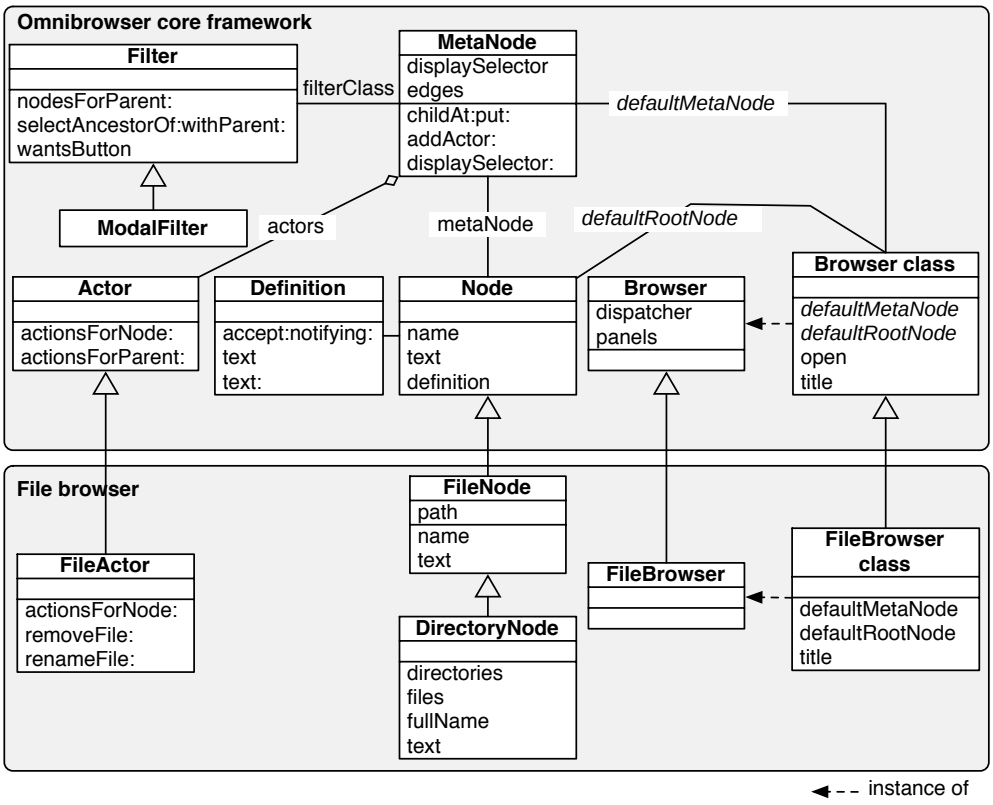


**Fig. 5.** Core of the OmniBrowser framework and its extension for the file browser.

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:. The rendering of a node is performed by invoking on the domain node the selector stored in the variable displaySelector in the metanode.

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. The method actionsForParent: 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 handy filter that represents transitions in the metagraph that can be traversed by using a radio button in the GUI.

### 3.4 Glueing Widgets with the Metagraph

From the programmer point of view, creating a new browser implies defining a domain model (set of nodes like FileNode and DirectoryNode), 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 then 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 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 radio button. Each edge leading away from the filter is represented as a button in the radio button. Only one button can be selected at a time in the radio button, and the associated choice is used to determine the outgoing edges. 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.

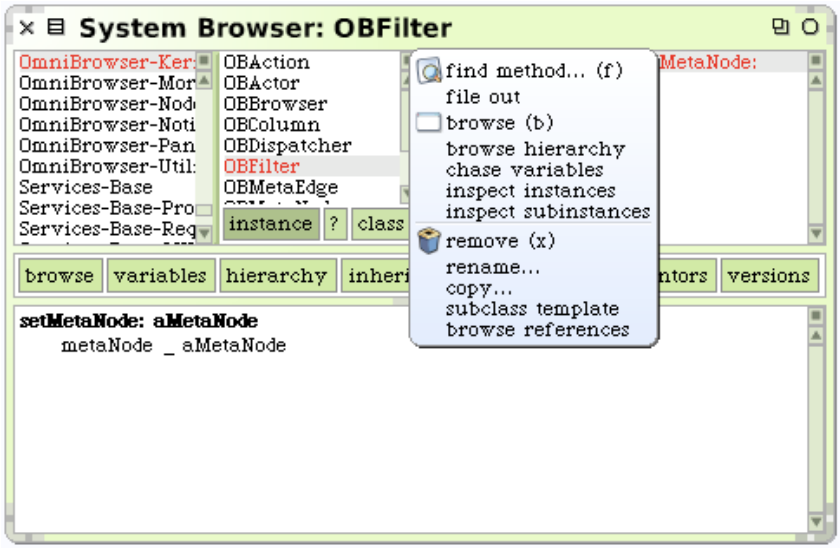


**Fig. 6.** Example of menu in the OmniBrowser framework system browser.

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.

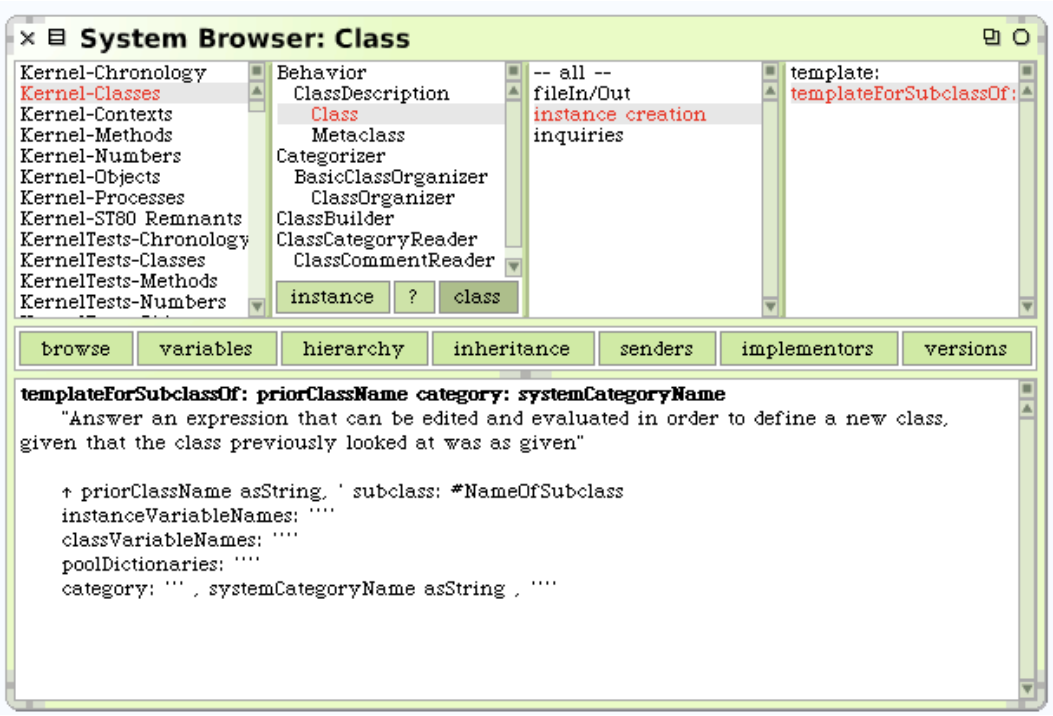


**Fig. 7.** OmniBrowser framework based Smalltalk system browser.

This browser just replicates the traditional four panes system browser discussed in Section 2. The system browser is mainly composed of four lists (upper part) and a panel (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 textual panel display 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 panel shows the definition of this class. If a method is selected, then the definition of this method is displayed. The text contained in the panel can be edited. The effect of this is to create a new class, a new methods, or changing the definition of a class (*e.g.*, adding a new variable, changing the superclass) or redefining 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 panel, 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). 220 methods are spread over these 19 classes. Figure 8 shows the classes in 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 this is less code and better factored. 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 panels 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 conduct events passing between widgets of a browser. The vocabulary of events is the following one:

- refresh is emitted when a complete refresh of the browser is necessary. For instance, if a change happens in the system, this event is triggered to order a complete redraw.
- nodeSelected: is emitted when a list entry is selected with a mouse click.

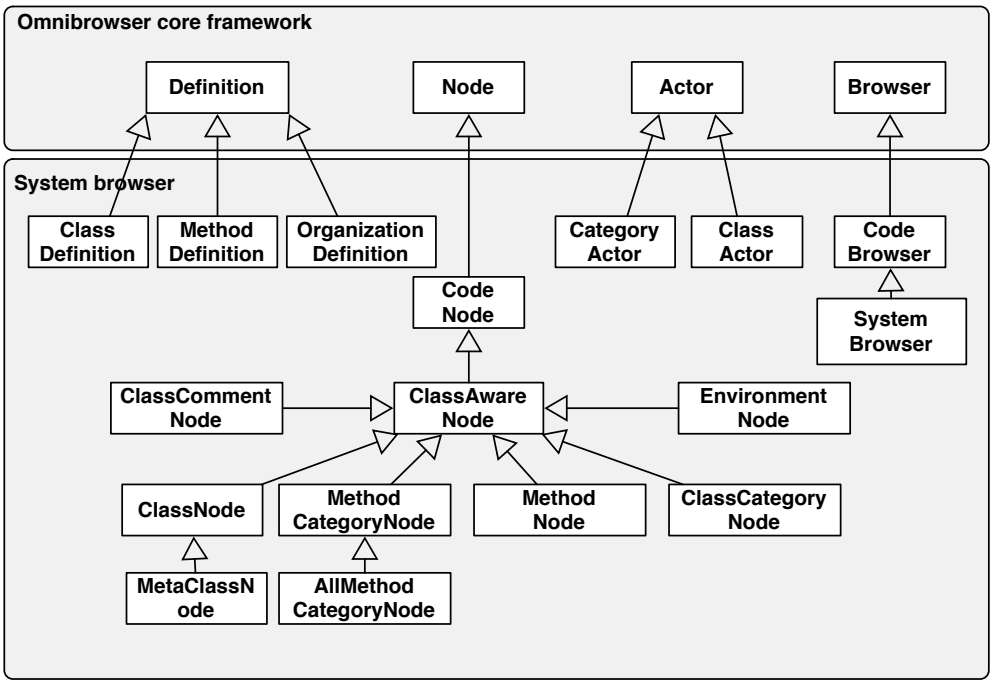


**Fig. 8.** Extension of OmniBrowser framework to define the system browser.

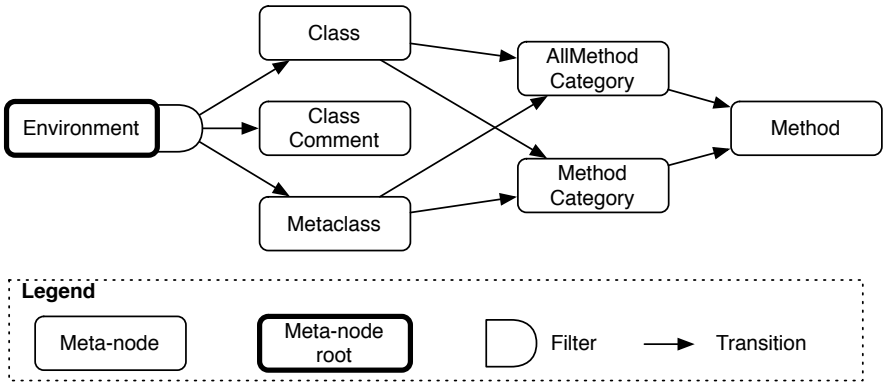


**Fig. 9.** Metagraph of the system browser.

– 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 loose some text edition why changing the content of a text panel if this was modified without being validated. This happens when first a user writes the definition of a method, without accepting (*i.e.*, compiling) it, and then another method is selected.

Each graphical widget composing a browser are listeners and can emit events. Creation and registration of widgets as listeners and event emitters 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 taking part of 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 to 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.

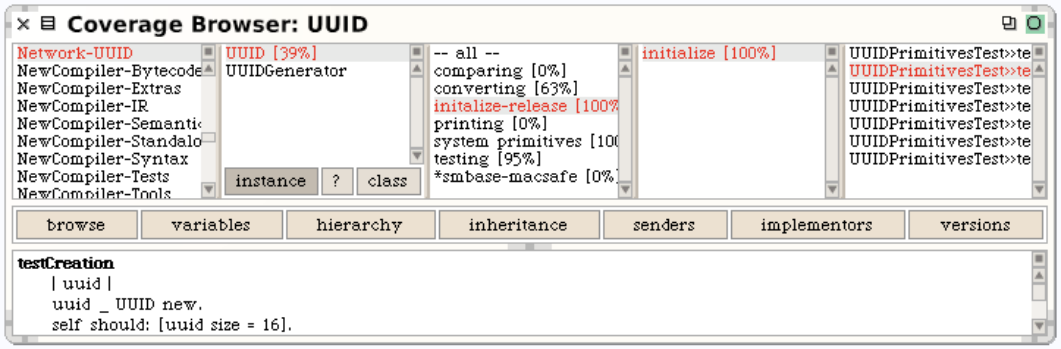


**Fig. 10.** Screenshot of the coverage browser.

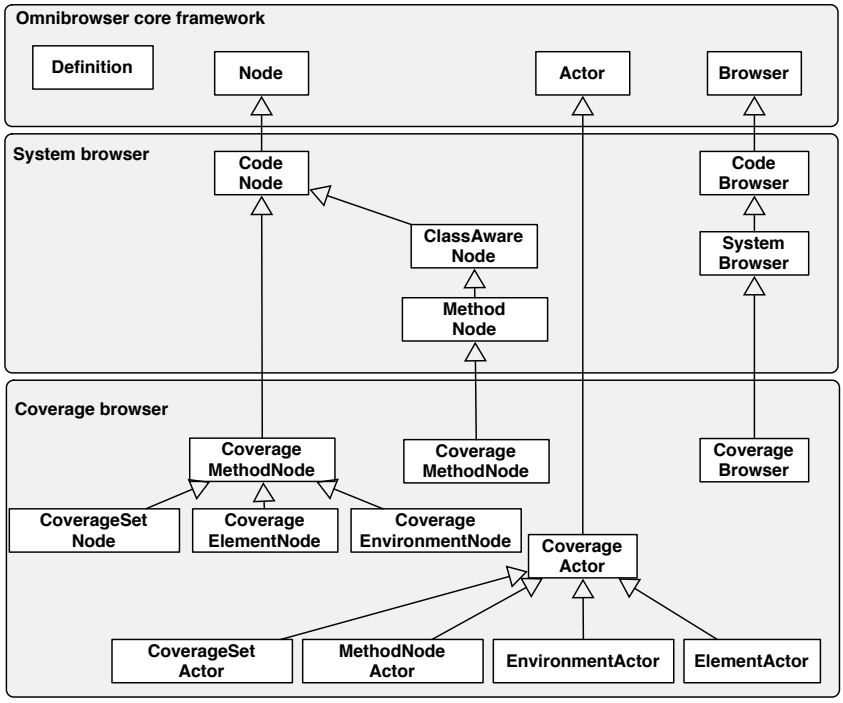


**Fig. 11.** Extension of Omnibrowser and system browser to define the coverage browser.

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.

## 6 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 13 shows some browsers that are based on OmniBrowser framework. We now discuss the strengths and limitations of the OmniBrowser framework.

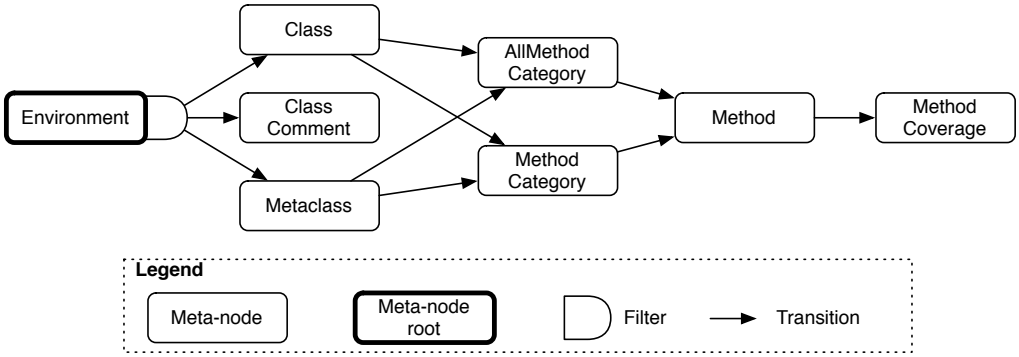


**Fig. 12.** Metagraph for the coverage browser.

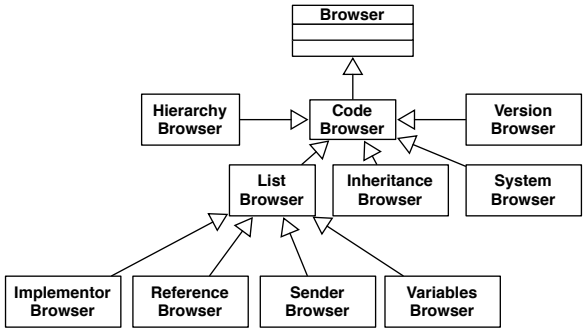


**Fig. 13.** Some code browsers developed using OmniBrowser framework.

### 6.1 Strengths

**Ease of use.** As any good framework, extending it following the framework intention make it easy to specify advanced browsers. The fact that the browser navigation is explicitly defined in one place lets the programmer understanding and controlling the tool navigation and user interaction. The programmer does not have the burden to explicitly create and glue together the UI widgets and their specific layout. Extra decorating widgets such as extra-menu is possible and 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. The code that describes some GUI present in the JHotDraw [JHo] contains the pattern checking for a nil value of variables that may reference graphical widgets.

Such as situation does not happen in OmniBrowser framework, as meta-graphs are declaratively defined and each metaedge describes an action the user can perform on a browser, states a browser can be in are explicit 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.

### 6.2 Limitations

**Hardcoded flow.** As any framework, OmniBrowser framework constraints the space of its own extension. OmniBrowser framework does not support well 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 the one of the VisualWorks standard browser. In VisualWorks, it is possible to select a package, then select one class of this package and as third step 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 passing among the widgets to implement

## 7 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. In HotDraw, however, the events to go from one state to another are taken from a limited set of possible actions such as mouse over.
**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. Still there is not as such a metagraph describing the navigation of a domain graph.
**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 being notified. 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 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 by constraints between the different elements that composed a browser. But there is no explicit distinction between the domain and its navigation.

## 8 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 Niklaus Haldimann and Stefan Reichnart for their use of the OmniBrowser framework.

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.

## References

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

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.

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*, 30:79–95, 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.