Chapter 1

Creating Browsers with OmniBrowser

In this chapter, we present OmniBrowser, a browser framework that supports the definition of browsers based on explicit metamodels. In OmniBrowser framework, a browser is a graphical list-oriented tool to navigate and edit any 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, a browser is described by a domain model and a metagraph which specifies how the domain space may be 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 by the framework, OmniBrowser allows for interaction with the end user.

In the following, we show how to build new browsers from predefined parts and how to easily describe new tools. Three exemplary browsers, a file browser, a remake of the ubiquitous Smalltalk system browser, and a coverage browser, will illustrate how to define sophisticated browsers for various domains.

1.1 Representing State of a User Interface

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 may be 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.

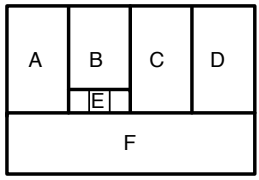

Figure 1.1: The traditional Smalltalk System Browser roughly depicted.

Figure 1.1 depicts the different widgets of a traditional Smalltalk class system browser (see Figure 1.7 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 D with pane C 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). 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 open up (being able to select a method name, for example). 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 to a 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 and often error-prone logic. In addition it makes tool elements difficult to extend and reuse in different context.

The main problem when building a browser is representing the mapping from the intended navigation model to the domain model and widgets. In the next section, we describe OmniBrowser, a framework to design browsers where the domain model is distinct from the navigation space **Andrew** ►*isn't that always the case?*◄ . The latter is being described by a metagraph. The state of a browser is defined by a path in this metagraph.

1.2 Graph and Metagraph of a Browser

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.

Overview of the OmniBrowser framework

The major classes that make up the OmniBrowser framework are presented in Figure 1.2, 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 cyclic because for example file systems or structural meta models of programming language (*i.e.*, packages, classes, methods...) contain cycles,

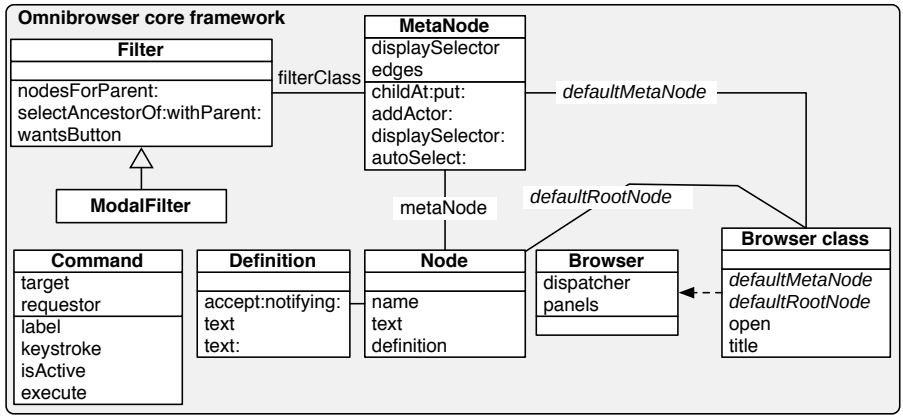

Figure 1.2: Core of the OmniBrowser framework.

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 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 a user traverses the graph of domain objects. A metagraph is composed of metanodes and metaedges. A metanode identifies a state in which the browser may be. A metanode may reference a filter (described below) 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), sibling nodes are created from a given node by invoking a method that has the name of the metaedge.

A *metanode* has the ability to be auto selected with the method MetaNode »autoSelect: aMetaNode. When a particular child for auto selection is designated, the first node produced by following its metaedge will be selected.

Command. A *Command* enables interaction and manipulation of the domain graph. Commands may be available through menus and buttons in the browser. They therefore have the ability to render themselves in a user interfaces and are responsible for handling exceptions that may occur when triggered.

Commands are defined in a non-invasive way: adding and removing commands is done without any method redefinition of the core framework. This enables a smooth gathering of commands independently realized.

A command is defined by subclassing `OBCommand`, then redefining its four main methods with the desired behavior and finally defining a method on the browser class whose name begins with `cmd`. This method has to return a command class. An example is provided in the following subsection.

Filter. The metagraph describes a state machine. When the browser is in a state in which more than one transition are available, the user decides which transition to follow. To allow that to happen OmniBrowser displays the possible transitions 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 made yet, and the definition pane is still displaying the current definition. Once a click is made, the transition actually happens, the definition pane is updated (and perhaps other panes such as button bars), and OmniBrowser 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 may be needed to filter out all but one edge 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 of the text is expected by the browser user, then a definition is necessary to handle modification and commitment (*i.e.*, an *accept* in the Smalltalk terminology). A definition is produced by a node.

Building 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.

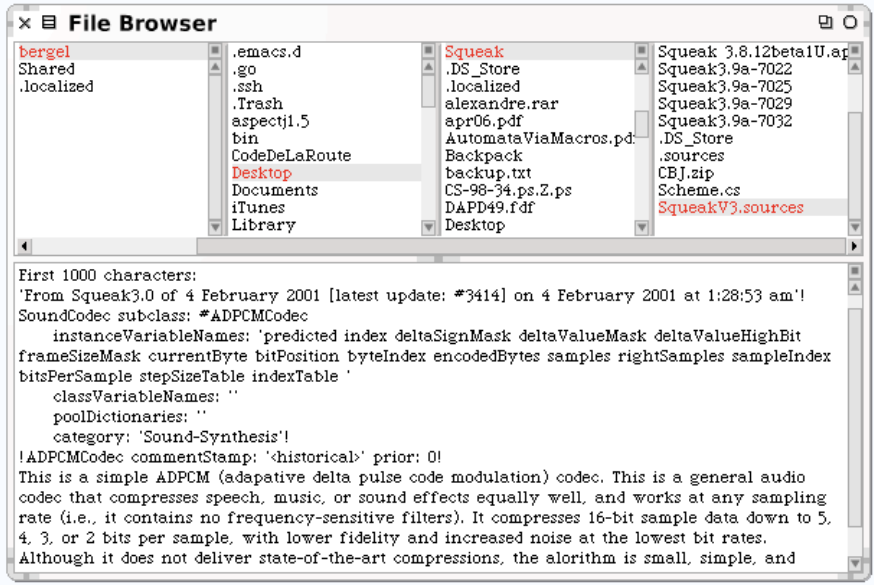

Figure 1.3: A minimal file browser based on OmniBrowser.

Figure 1.3 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 panes 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.

Metagraph Definition. A filesystem encompasses basically two kind of entities, files and directories. To model the navigation of a filesystem we thus need two metanodes in the metagraph, Directory and File. Within any directory of a filesystem, we can again find files and other directories, hence there are two kind of transitions outgoing from a directory metanode, files and directories. When opening the filesystem browser, we launch it for a given directory, *e.g.*, the root directory of the filesystem. Thus the metagraph's root metanode represents a directory. Figure 1.4, right, shows this metagraph describing a filesystem.

To concretely implement this filesystem metagraph we define a class OBFileBrowser as a subclass of OBBrowser and write the method defaultMetaNode on the class side. This method first defines the two metanodes Directory and File and specifies second the two transitions leaving directory and going to the metanodes Directory and File, respectively. These transitions are implemented as children of the metanode Directory and are called directories and files, respectively. defaultMetaNode finally answers the root metanode, in our case Directory.

```
OBFileBrowser class>>defaultMetaNode
    "returns the directory metanode that acts as the root metanode"

    | directory file |
    directory := OBMetaNode named: 'Directory'.
    file := OBMetaNode named: 'File'.
    directory
        childAt: #directories put: directory;
        childAt: #files put: file.
    ↑ directory
```

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.

As soon as we have defined the metagraph, we can model the domain with node classes. For every metanode in the metagraph we also need a concrete node class in our model, in this case we need two node classes, one representing a directory, the other a file. As the root metanode in the graph represents a directory, the concrete node in the model has to be a concrete directory node, eg. representing the root directory of the filesystem. This default root node is answered by the class-side method defaultRootNode of OBFileBrowser:

```
OBFileBrowser class>>defaultRootNode
    ↑OBDirectoryNode new path: '/'
```

The next step consists of modeling the domain objects, *i.e.*, nodes.

Node definitions. Nodes wrap objects of the browsed domain. First the class OBFileNode, a subclass of OBNode, has to be defined. Instances of this class will represent concrete files. A file node is identified by a full path name, stored in a variable. A directory is another entity in our model that contains directories and files. A directory can be simply modeled as a special kind of file. The only difference between a file and a directory node is that for a directory the path variable points to a directory, not to a file.

```
OBNode subclass: #OBFileNode
    instanceVariableNames: 'path'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'OBExample-FileBrowser'

OBFileNode subclass: #OBDirectoryNode
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'OBExample-FileBrowser'
```

The name of the node is simply the name of the file selected:

```
OBFileNode>>name
    ↑ (self path subStrings: '/') last
```

The variable path has to be accessed:

```
OBFileNode>>path
    ↑ path

OBFileNode>>path: aString
    path := aString
```

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

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

The methods files and directories are defined on the class OBDirectoryNode.

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

OBDirectoryNode>>files
    | dir |
    dir := FileDirectory on:  path.
    ↑ dir fileNames collect: [:each |
                    OBFileNode 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). As there is no further navigation leaving a file node, such a node does not have to define navigation methods such as directories or files.

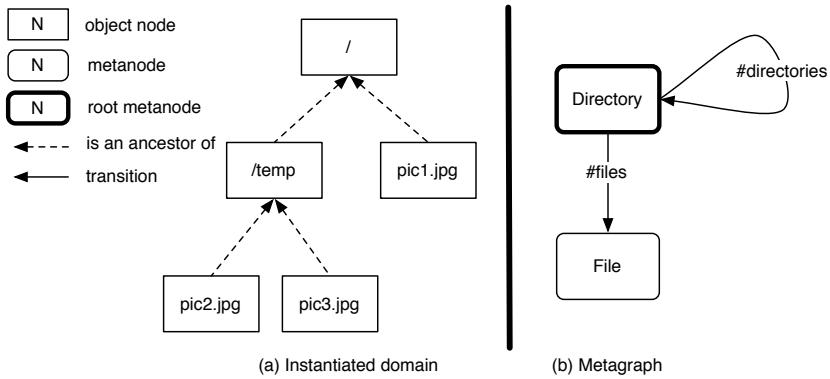

Figure 1.4: A filesystem as a graph (a) and its corresponding metagraph (b).

To visually distinguish files from directories when browsing a directory with our file browser, we can add an icon to each element in the list. To illustrate this, we will denote directories with a small folder icon.

The first step is to integrate the icon itself into a Squeak image. In the class OBMorphicIcons you see some pre-defined icons stored in methods such as arrowUp. To import an icon stored as an image (*e.g.*, as a GIF file), you can use this code:

```
| image stream |
image := ColorForm fromFileNamed: '/path/to/icon.gif'.
stream := WriteStream with: String new.
image storeOn: stream.
stream contents.
```

Inspect this whole code listing. In the inspector you see the definition of the color form for the icon. You can now install the content of this ByteString as a method in the method protocol icons of OBMorphicIcons in a method called folder. Make sure that you do not return the string, but the code within the string, so that if the method gets invoked a color form for the folder icon is returned. For example, a flag icon is defined as:

```
OBMorphicIcons>>flag
```

```
↑ ((ColorForm
   extent: 12@12
   depth: 8
   fromArray: #( 437918234 437918234 437918234 436470535 101584139
    387389210 436404481 17105924 303634202 436666638 ...
```

In the second step you can take this icon and display it in the columns for every directory. To achieve this, simply add a method icon to the class OBDirectoryNode:

```
OBDirectoryNode >>icon
  ↑#folder
```

The method icon gets executed for every element that is added to a column. If it answers a symbol, then the method of OBMorphicIcons with the same name is executed, answering the icon as a color form to be added on the left of the list element, *i.e.*, the directory name.

At this stage, we can open a file browser by evaluating OBFileBrowser open in a workspace. To allow users to perform actions on a selected file, we add commands to the browser. Note that you will need to open a new browser to see these command in effect. They are implemented with subclasses of OBCommand:

```
OBCommand subclass: #OBRemoveFileCommand
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'OBExample-FileBrowser'
```

The functionality of this command is basically implemented in four methods:

- isActive - test condition to determine if this command is active in the current column for the currently selected node
- keystroke - a letter used to trigger this command with the keyboard
- label - the string denoting this command in the command menu
- execute - holds the functionality to be triggered if the user executes this command

When these methods get executed, the command already knows the column from which it gets triggered (stored in the instance variable requestor) and the target node for which the action has to be exectuted (stored in the instance variable target). With this information available we can implement these four methods as follows:

```
OBRemoveFileCommand>>isActive
    "only active for files"
    ↑ (target isKindOf: OBFileNode) and: [requestor isSelected: target]

OBRemoveFileCommand>>keystroke
    ↑ $d

OBRemoveFileCommand>>label
    ↑ 'remove file'

OBRemoveFileCommand>>execute
    FileDirectory deleteFilePath: target path
```

To integrate this command the class OBFileBrowser has to be extended with a method whose name needs to start with 'cmd':

```
OBFileBrowser>>cmdRemoveFile
   ↑OBRemoveFileCommand
```

Open a new browser, then right click on a selected file and you will get a menu that contains this command. Currently, the list of files is not refreshed when files are removed. Refreshing can for instance be done by announcing a nodeDeleted announcement in the execute method. This can be achieved by inserting the expression target announce: (OBNodeDeleted node: self). Since this is a common operation, an helper is provided for that purpose: simply send the signalDeletion message to target.

Core Behavior of the Framework

The core of the OmniBrowser framework is composed of 8 classes (Figure 1.2). We denote the Smalltalk metaclass hierarchy by a dashed arrow.

The metaclass of the class OBBrowser is OBBrowser 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 return the root metanode and the root domain node, respectively. A browser is opened by sending the message open to an instance of the class OBBrowser.

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

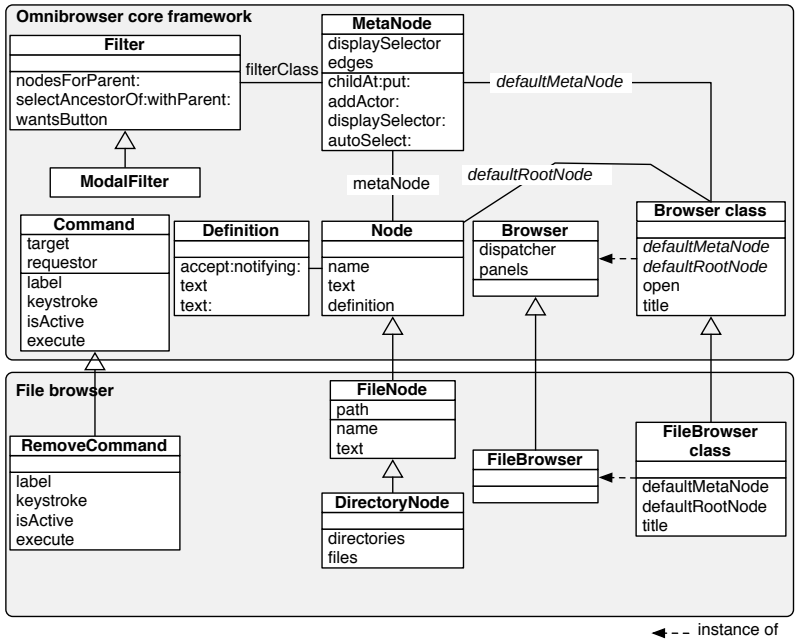

Figure 1.5: Core of the OmniBrowser framework and its extension for the file browser.

At runtime, the graph traversal is triggered by user actions (*e.g.*, pressing a button or selecting a list entry) which send the metaedge's name to the node that is currently selected. 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 OBCommand is instantiated by the framework and the set of commands for a browser is discovered (through the Smalltalk reflection API) when a browser is instantiated. All methods starting with the cmd prefix are considered as commands. Each of this method should return the *class* of the command (and not an instance of it).

The class OBNode 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,

the message definition is sent and it is expected to return an instance of a subclass of OBDefinition. 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 navigation 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 lists 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 1.3). Class OBFilter 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 OBFilter is subclassed into OBModalFilter, a handy filter that represents transitions in the metagraph that can be traversed by using a radio button in the GUI.

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 commands 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).

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 1.9) is 4, therefore the system browser

has 4 lists (Figure 1.7). Because the metagraph of a filesystem may contain cycles (*i.e.*, a directory may contain directories, as shown in Figure 1.4), 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 1.7.

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 the user. These actions enable interaction with the domain model, however they do not allow further navigation in the metagraph.

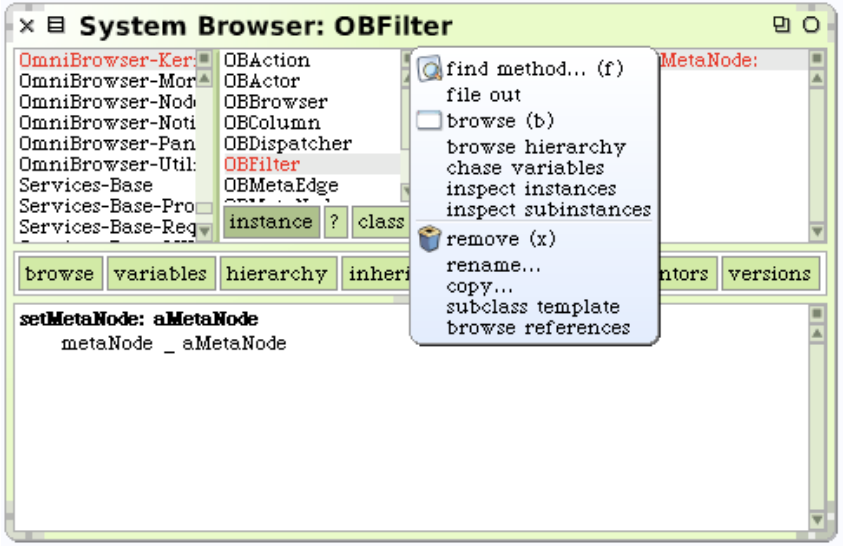

Figure 1.6: Example of menu in the OmniBrowser framework system browser.

Figure 1.6 shows an example of a menu offering actions related to

a class. These correspond to the list of commands defined in the class OBCodeBrowser.

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

1.3 The OmniBrowser-based System Browser

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

The Smalltalk System Browser

The system browser is probably the most important tool offered by the Squeak programming environment. It enables code navigation and code editing. Figure 1.7 shows the graphical user interface of this browser, and how it appears to the Smalltalk programmer.

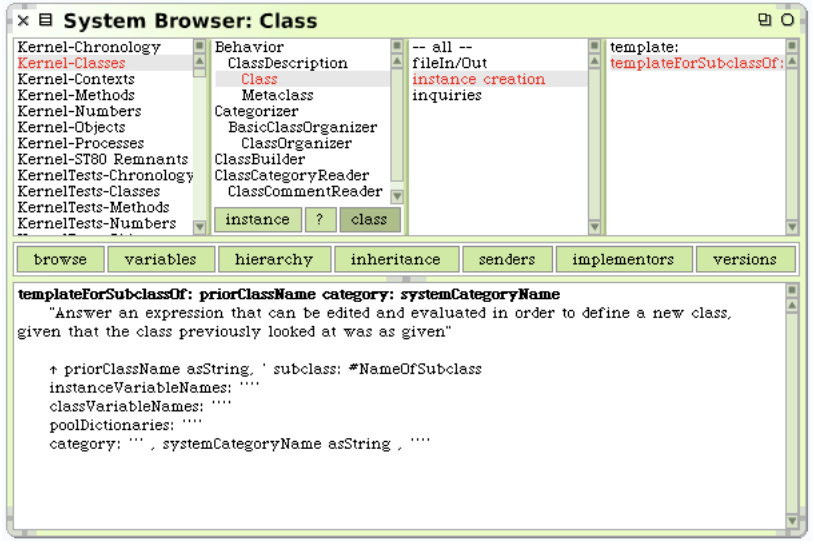

Figure 1.7: OmniBrowser based Smalltalk system browser.

This browser just replicates the traditional four panes system browser discussed in Section 1.1. 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 1.7, 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 displaying information about the current selection in the lists. Selecting a class category triggers the display of 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 definition panel, there is a toolbar intended to open more specific browsers like a hierarchy browser or a variable access browser.

The -- all -- method category gets automatically selected when no other method category is selected. This is specified in the OBMetagraphBuilder» populateClassNode method by invoking autoSelect: aMetanode.

System Browser Internals

The OmniBrowser-based implementation of the Squeak system browser is composed of 17 classes (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). Figure 1.8 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 1.9 depicts the metagraph of the system browser. The metanode

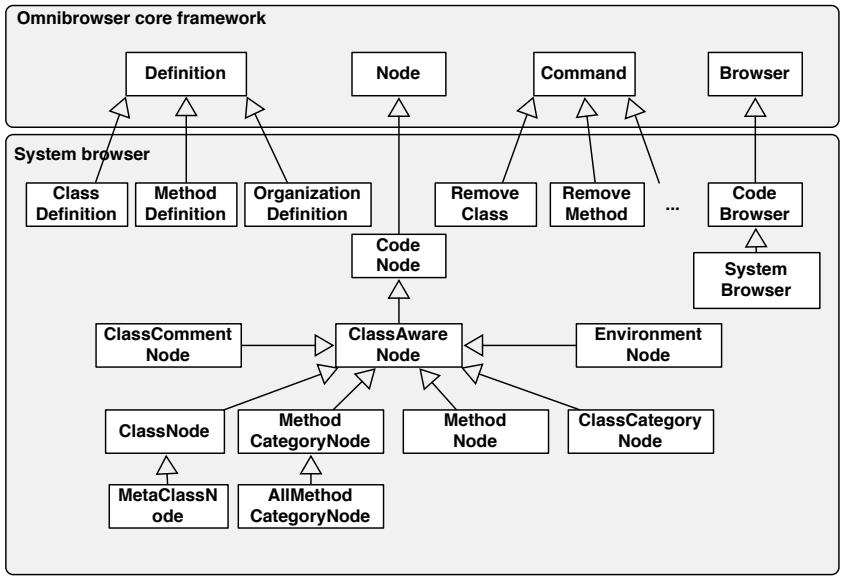

Figure 1.8: Extension of the OmniBrowser framework to define the system browser.

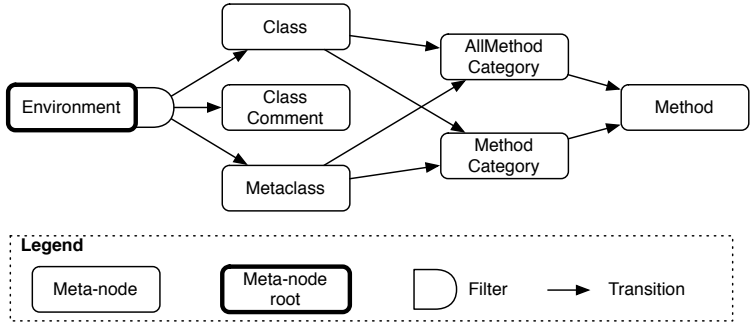

Figure 1.9: Metagraph of the system browser.

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 all methods.

As in the file browser example, we implement a method defaultMetaNode on the class side of the browser class, *i.e.*, OBSystemBrowser, returning the root metanode of the metagraph. This method reads:

```
OBSystemBrowser class>>defaultMetaNode
  | env classCategory |
  env := OBMetaNode named: 'Environment'.
  classCategory := OBMetaNode named: 'ClassCategory'.
  env childAt: #categories put: classCategory.
  classCategory ancestrySelector: #isDescendantOfClassCat:.
  self buildMetagraphOn: classCategory.
  ↑env
```

There is a dedicated utility class called OBMetagraphBuilder to create the complex metagraph of the system browser. The method defaultMetaNode outsources most parts of the metagraph building to this class. OBMetagraphBuilder implements its functionality in several small methods, *i.e.*, for every metanode of the metagraph there is a method holding all code to create this metanode and the outgoing edges, hence it is easily possible to adapt the metagraph by providing a dedicated subclass overriding the appropriate methods to change the right metanodes.

The root node of the domain graph is answered by the method defaultRootNode. For the system browser, the root node is the environment node:

```
OBSystemBrowser class>>defaultRootNode
  ↑OBEnvironmentNode forImage
```

Ancestry mechanism. As shown in Figure 1.8 there is a number of different nodes that are required to implement the system browser, such as class node, metaclass node, method node, method protocol node, class comment node, etc. We do not want to cover all these nodes in detail. Instead we report on an important feature of OmniBrowser framework to locate specific nodes in a large domain graph: the ancestry mechanism.

When a target node has to be selected, we start from the root node and traverse the tree down to the target node, remembering all nodes we pass during the traversal. Starting from the root node, we test for all children whether a child is an ancestry of the target node or not. If so, we go one level deeper and test the same for all children of this child, and so on, until we reach the target node. Every metanode, which basically models one level in the domain graph or tree, knows the ancestry selector to be used on this level. For a class node, the ancestrySelector is called isDescendantOfClass:. If

we search for a class node in the domain tree, we test for every class node if the class to be found is a descendant of that class, *i.e.*, if it is the same class as we search for. On the class category level, the ancestry selector is called descendantOfClassCat:, expecting a class category as a parameter. For every class category, we test whether the target node is a descendant of the passed class category or not.

This method descendantOfClassCat: is implemented as follows for a node having a class associated (*e.g.*, a class node or a method node):

```
OBClassAwareNode>>isDescendantOfClassCat: aClassCategoryNode
  ↑(self theNonMetaClass environment organization
    listAtCategoryNamed: aClassCategoryNode name)
      includes: self theNonMetaClassName
```

To define which metanode, *i.e.*, which level in the tree, uses which ancestry selector, we just pass this selector when building the metagraph, using the method ancestrySelector: aSymbol of OBMetaNode. With these kind of methods, it is possible to locate any node in the domain tree to *e.g.*, jump to it. This is for instance used when opening a browser for a certain node, *e.g.*, by using the OBSystemBrowser class-side method openOn: aClass selector: aSymbol.

Filtering of nodes. In the metagraph we can also define several filters for a metanode, used to filter and otherwise manipulate the nodes represented by this metanode before they get displayed in columns **Andrew** ►*Is this the same kind of filter we have previously seen?*◄ . For the class category metanode, for instance, there are two filters defined: a class sort filter and the modal filter used to select one of the three outgoing metaedges instance, comment or class.

Let's have a look at these two filters, starting with the class sort filter implemented in class OBClassSortFilter. Its responsibility is to sort and indent all classes of a class category according to their position in a class hierarchy. If a class category for instance contains two distinct class hierarchies, *e.g.*, class C inherits from B, and B and D inherit from A, and E has two subclasses F and G, then the class sort filter sorts and indents these classes as shown in Figure 1.10.

When a metanode is asked for its children nodes (in method childrenForNode: aNode) it asks its associated filters to answer the nodes by invoking their nodesFrom: aCollection forNode: aNode method. In the case of the class sort filter, aNode refers to the class category node and aCollection holds all class nodes this class category node returns when the message classes is sent to it. The class sort filter can now sort the passed class nodes and indent them appropriately in the method OBClassSortFilter >> nodesFrom:forNode:.

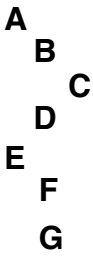

Figure 1.10: How OBClassSortFilter sorts and indents two distinct class hierarchies in one class category.

The other filter defined for a class category metanode, OBModalFilter, has a different task: It selects one edge of the three outgoing edges from the class category metanode, *i.e.*, instance, comment or class. The user of the system browser can select using the switch in the class column (widget E in Figure 1.1) whether he wants to see the instance-, the class-side or the comment of the selected class. OBModalFilter remembers the selection of the user. Dependent on this selection, it answers the corresponding metaedge to be traversed, *e.g.*, the comment metaedge. This is done in the method edgesFrom: aCollection forNode: aNode. The metanode, *i.e.*, the class category metanode, passes all available metaedges to this method, along with the currently selected class node, and the modal filter answers just the metaedge selected by the user. Other filters than a modal filter, such as the class sort filter, typically just return all edges passed to them.

There are two other important tasks performed by filters besides filtering edges and nodes: Manipulating the name of a node to be displayed and defining an icon shown along with a node in the column. The former is handled in the method displayString: aString forParent: pNode child:, the latter in icon: aSymbol forNode: aNode. Before a node's name gets displayed, all defined filters can manipulate the display of its name, *e.g.*, emphasize it in bold. Note that the filter also has access to the parent of a node to be displayed, not the current node alone. There are also filters enriching a node with an icon before display, the OBInheritanceFilter for instance adds arrow up, down icons to methods, if a method overrides a method with the same name from a super class or is overridden in subclasses. **Andrew** ►*how do yo do this?*◄

A metanode can have arbitrarily many filters, resulting in a chain of filters. However, if several filters do the same kind of task, *e.g.*, adding an icon to a node, the last added filter providing this functionality will finally be responsible to define the icon which the node gets. Hence the order in

which the filters get added to the metanode is relevant.

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 trigger a complete redraw.
- nodeSelected is emitted when a list entry is selected with a mouse click.
- nodeDeleted is emitted when a list entry has been removed, *e.g.*, by executing a remove command.
- nodeChanged is emitted when the node that is currently displayed changes. This typically occurs when a filter button 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 losing some text edition while changing the content of a text panel if this was modified without being validated. This happens when a user writes the definition of a method, without accepting (*i.e.*, compiling) it, and then selects another method.

Each graphical widget composing a browser is a listener 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.

1.4 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 1.11. It shows us that 39% of the class UUID is covered by tests, and that the method initialize is covered a 100% by the tests shown in the right-most pane. One of these tests is testCreation.

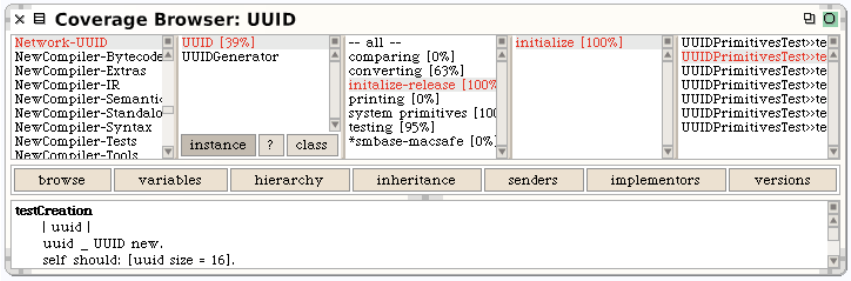

Figure 1.11: Screenshot of the coverage browser.

The coverage browser is composed of 11 classes (one class for the browser, five commands and five nodes). Figure 1.12 illustrates how classes in OmniBrowser and in the system browser are extended to define this new browser. The metagraph is depicted in Figure 1.13 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.

1.5 Evaluation and Discussions

Several other browsers such as a browser specifically supporting new language constructs such as Traits have been developed using OmniBrowser framework demonstrating that the framework is mature and extensible. Figure 1.14 shows some browsers that are based on OmniBrowser framework. We now discuss the strengths and limitations of the OmniBrowser framework.

Strengths

Ease of use. As any good framework, extending it following the framework intention makes it easy to specify advanced browsers. The fact that

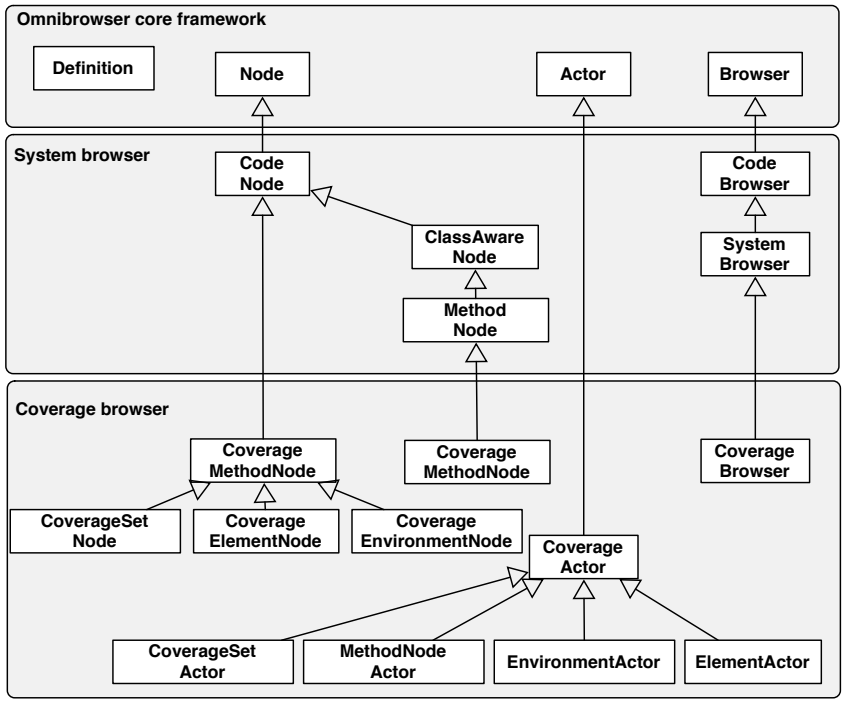

Figure 1.12: Extension of Omnibrowser and system browser to define the coverage browser.

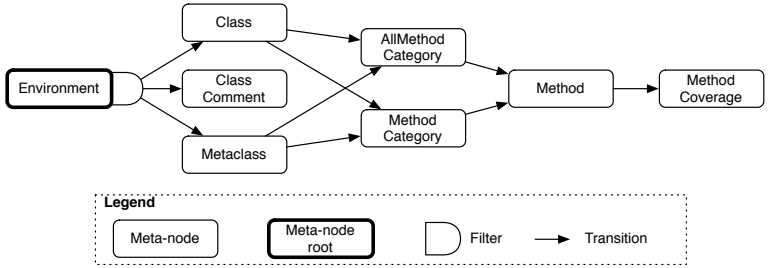

Figure 1.13: Metagraph for the coverage browser.

the browser navigation is explicitly defined in one place lets the programmer easily understand and control the tool navigation and user interaction.

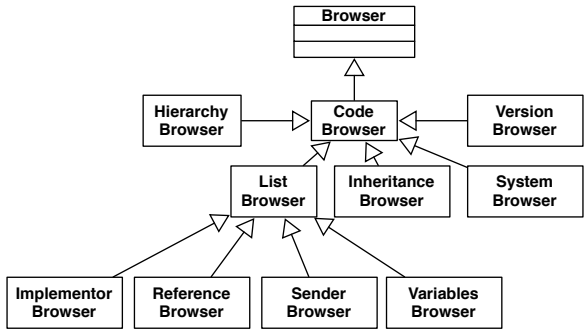

Figure 1.14: Some code browsers developed using OmniBrowser framework.

The programmer does not have the burden to explicitly create and glue together the UI widgets and their specific layout. To add additional custom widgets in a concrete browser, the developer can simply define a class implementing this widget and add an object of this class to the list of widgets used during the creation of the browser. This list is defined on the class-side of OBBrowser in the method panels. 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:

```
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 are 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 whether a list is selected or not and 62 of them send the message ifNotNil:. Those are not isolated Smalltalk examples. The

code that describes some GUI present in the JHotDraw framework also contains the pattern checking for a nil value of variables that may reference graphical widgets.

Such a situation does not occur in OmniBrowser framework, as metagraphs 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.

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.

1.6 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, OmniBrowser is 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 *meta-graph* and describing the interaction between the browser and the domain through *commands*. 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 window is used to visualize and edit nodes selected in the top half.

The framework is implemented in Squeak Smalltalk and called OmniBrowser. This Chapter shows 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. Of course, there are more instantiations available than we have not discussed in this chapter. The validation shows that the goals of the frameworks are met. Building the System Browser with the OmniBrowser framework shows that the code is much simpler. The Code Coverage browser shows that it is easy to extend an existing browser.

Bibliography