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 the 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, a browser is described by a domain model and a metagraph that specifies how the domain space may be navigated. 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.

We will look at how to build new browsers from predefined parts and how to easily describe new tools. We will illustrate how to define sophisticated browsers for various domains with the help of three examples: a file browser, a remake of the ubiquitous Smalltalk system browser, and a coverage browser.

1.1 Building a simple browser step by step

To illustrate how to instantiate the OmniBrowser framework, we describe the implementation of a simple file browser supporting navigation of directories and files.

Figure 1.1 shows the file browser in action. 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,

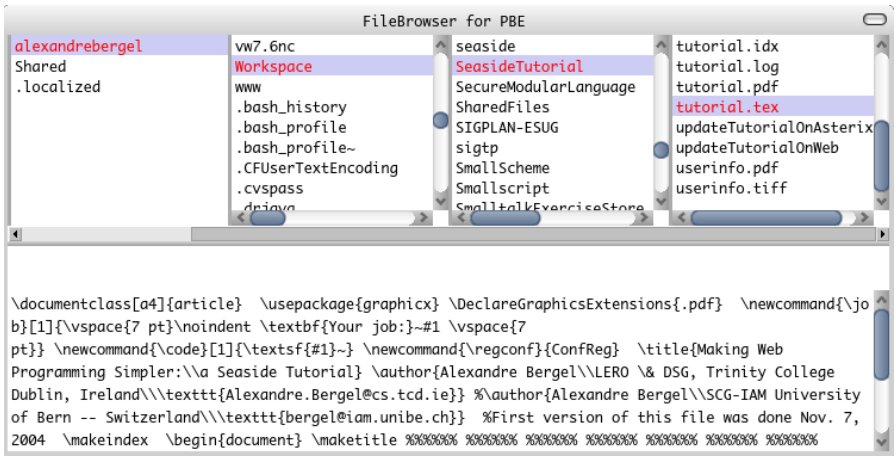

Figure 1.1: A minimal file browser based on OmniBrowser.

similar to the *Column View* of the Finder in the Mac OS-X operating system. Note that we can have an arbitrary number 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.

The class describing a browser must inherit from the class OBBrowser. The class FileBrowser may be defined as follows:

```
OBBrowser subclass: #FileBrowser
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'PBE−Omnibrowser'
```

The default browser title may be overridden by redefining the class method title:

```
FileBrowser class»title
   ↑ 'FileBrowser for PBE'
```

OmniBrowser uses a *metagraph* to model different navigation paths that a user may follow by clicking on browser items. To keep this example simple, we will assume that a file system contains only two kind of entities, files and directories.

```
FileBrowser 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
```

A metagraph is composed of metanodes, each of which represents one particular set of selected items. To model the navigation of a file system we thus need two metanodes in the metagraph, Directory and File. Within any directory of a file system, we can navigate to further files and other directories, hence we need two kinds of transitions out of a directory metanode, which will be labelled files and directories in the metagraph.

The next step is to create class nodes that will model browser items. For that purpose, we define two subclasses of OBNode:

```
OBNode subclass: #FileNode
  instanceVariableNames: 'path'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE−Omnibrowser'

FileNode subclass: #DirectoryNode
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE−Omnibrowser'
```

Instances of FileNode and DirectoryNode will represent files and directories of the domain being navigated. Nodes wrap objects of the browsed domain. First the class FileNode, 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.

The path variable needs accessors:

```
FileNode»path
  ↑ path

FileNode»path: aString
  path := aString
```

A node needs to answer to the message name and return the title used in the browser:

```
FileNode»name
    "return the name of a file"
    ↑ (self path subStrings: '/') last
```

When selected, a node may provide a character string used to fill the lower text pane. In our situation, clicking on a file node should display the content of the selected file. The method text has to be defined.

```
FileNode»text
    "return the first 1000 characters"
    ↑ ((FileStream readOnlyFileNamed: path) converter: Latin1TextConverter new;
            next: 1000) asString
```

Note that the method text provides a purely read-only view of the node. If we want to be able to edit the contents of a file, then we need to take a different approach, as we will see later on.

Each node is displayed in a column, and when selected, it provides the list of nodes used to fill the next column. Clicking on a directory must cause the list of contained files and folders to be displayed in the column located to the right. The transitions directories and files have to be defined as methods:

```
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 name directories and files are also the names of the transitions between the Directory and File meta nodes. When one of the two #directories and #files metaedges is traversed, the name of this metaedge is used as a message name sent to the metanode's node. The two methods are invoked when a directory node is selected.

The implementation of FileNode and DirectoryNode 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.

The file browser is opened for a given directory, *e.g.*, the root directory of the file system. Thus the metagraph's root metanode represents a directory. The default root node is given by the class method defaultRootNode:

```
FileBrowser class»defaultRootNode
    ↑ DirectoryNode new path: '/'
```

Our file browser may now be opened by executing FileBrowser open. Currently, no much can be done. Only navigating through directories and displaying file contents. In the subsequent sections, we will see how commands and icons may be added.

1.2 Enhancing the file browser with actions, definitions and icons

Spawning browsers

A command defines how the user can interact with browsed items. All interactions triggered from a menu have to be defined as commands. We will define a browser command that opens a second file browser on a selected node. Figure 1.2 depicts the activation of the contextual menu.

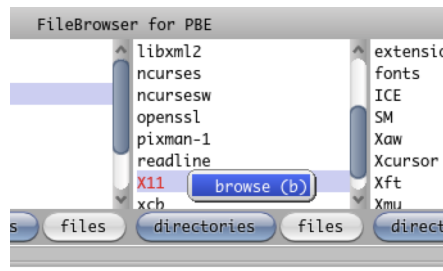

Figure 1.2: The browse command in the filer browser.

BrowseCommand has to be a subclass of OBCommand and must override at least 4 methods in order to be effective. Each command must have a menu item label.

```
BrowseCommand»label
    ↑ 'browse'
```

A command may or may not be active (*i.e.*, listed in the contextual menu) depending on some selection or the particular state of the selected item. In

our case, we just need to redefine isActive to answer whether the node for which the command is triggered is selected. This implementation of isActive is very common and you will probably need it for most of your commands.

```
BrowseCommand»isActive
  ↑ (requestor isSelected: target)
```

When these methods are evaluated, 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).

A command may be also invoked using a particular keystroke. The apple key (on Mac) or alt key (on MS Windows and Linux) has to be combined with the character returned by keystroke to activate the command.

```
BrowseCommand»keystroke
  ↑ $b
```

The shift key may be added to the keystroke by returning a capital letter instead of a minuscule one. Note that by defining a keystroke, the label is automatically appended with "(b)" in menus.

The method execute is evaluated when the command is active and triggered by either an appropriate keystroke or a menu selection.

```
BrowseCommand»execute
  FileBrowser openOn: target.

FileBrowser class»openOn: aNode
  ↑ (self root: aNode selection: aNode) open
```

The method openOn: needs to be added to the class side of FileBrowser to open a browser on a given node.

The very last step is to link FileBrowser to BrowseCommand. This is done by defining an instance side method that returns the class command (and not an instance of it). This name of the method needs to begin with "cmd", this is a naming convention similar to testing method names beginning with "test". The instance side method needs to be defined:

```
FileBrowser»cmdBrowse
  ↑ BrowseCommand
```

The command is now fully defined. Open a new file browser and right click on an item.

Creating new files

Our browse command is very simple: there is no list to refresh; no interaction with the user; no automatic node selection. We will cover these important topics by adding a command for creating files. The browser column will have to be updated with a new entry (the file recently created), and the user will have to enter the name of the file to be created.

The NewFileCommand is define as follows:

```
OBCommand subclass: #NewFileCommand
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'PBE−OmniBrowser'

NewFileCommand»keystroke
   ↑ $n

NewFileCommand»label
   ↑ 'new file'
```

The command should be activated when we right click either on a directory (to create the file in it), or on a column. This suggests a slightly more elaborate isActive method:

```
NewFileCommand»isActive
   ↑ ((requestor isSelected: target) and: [target isKindOf: DirectoryNode])
         or: [requestor selectedNode isNil]
```

The OmniBrowser framework provides a number of utility classes to ask for user input. These class are in the package OmniBrowser−Notifications. We will use OBTextRequest to ask for user input.

```
NewFileCommand»execute
   | filename fullFilename stream nodeToSelect |
   filename := OBTextRequest
               prompt: 'Please type the name of the file to create'
               template: ''.
   fullFilename := target path, FileDirectory pathNameDelimiter asString, filename.
   stream := FileStream newFileNamed: fullFilename.
   stream close.

   nodeToSelect := target files detect: [:fileNode | fileNode name = filename ].
   requestor announce: (OBNodeCreated node: nodeToSelect).

   self select: nodeToSelect with: requestor announcer.
```

When a new file is created, it is automatically selected. The last line of execute selects noteToSelect.

Modifying files

So far, we've only seen how to create empty files. To modify the content of a file, the browser needs to accept user input from the lower text pane and perform an action on it.

Let's subclass OBDefinition:

```
OBDefinition subclass: #FileDefinition
   instanceVariableNames: 'fileNode'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'PBE−OmniBrowser'
```

The variable fileNode is intended to refer to the selected file node. This variable will have to be initialized when the definition is instantiated. We need accessors:

```
FileDefinition»fileNode
   ↑ fileNode

FileDefinition»fileNode: aFileNode
   fileNode := aFileNode
```

As we saw previously, the content of the lower text pane is given by the text methods defined on FileNode. As we now use a definition, FileNode»text is now useless. You can safely remove it and define it on FileDefinition instead:

```
FileDefinition»text
   "return the first 1000 characters"
   ↑ ((FileStream readOnlyFileNamed: fileNode path) converter: Latin1TextConverter
      new;
         next: 1000) asString
```

The message text: is sent to the OB definition when the user presses apple-s on Mac or alt-s on other platforms. When the content is "accepted", we need to open the file, write the content provided by the user into it, and close the file:

```
FileDefinition»text: aText
   "return the first 1000 characters"
    (FileStream fileNamed: fileNode path)
            nextPutAll: aText asString;
            close.
   ↑ true
```

FileDefinition is currently not linked to the browser. To do so, define the method definition in FileNode:

```
FileNode»definition
  ↑ FileDefinition new fileNode: self
```

Each selected item can have its own definition.

Distinguishing folders from plain files

To visually distinguish files from directories when browsing a directory with our file browser, we will denote directories with a small icon. This section shows how to import new icons into Pharo, then how to associate icons to browsed items.

The first step is to integrate the icon itself into a Pharo image. In the class OBMorphicIcons you can see some pre-defined icons stored in methods such as package. 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, the package icon is defined as:

```
OBMorphicIcons»package
  ↑(Form
  extent: 14@14
  depth: 32
  fromArray: #( 4294960327 4289901233 .....  4294960327)
  offset: 0@0)
```

The list of numbers corresponds to the image bit encoding of the icon.

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 DirectoryNode:

```
DirectoryNode »icon
  ↑ #folder
```

If you do not have a .gif under hand, you may replace #folder by #package , since a package icon is available. The method icon is evaluated for every

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

Another way to distinguish files from directories is to keep them separated in different tabs. Figure 1.2 illustrates this. This is done by simply adding a filter to the directory metanode when defining the metagraph:

```
FileBrowser class»defaultMetaNode
  ...
  directory
      childAt: #directories put: directory;
      childAt: #files put: file;          "Change the period for a semi−column"
  addFilter: OBModalFilter new.    "Line to add"
  ...
```

The effect of adding a filter to a metanode is to let the user select the metaedge to follow. One button is created for each metaedge. Without any filter, all edges are used to compute of the next list as we previously saw.

Coloring and font selection

Item names in a menu may be colored and use font modifier such as italic and bold effect. This feature is useful for distinguishing some items from others or for emphasizing some items. Font particularization is achieved by returning instance of Text instead of a simple string when overriding OBNode >>name.

```
FileNode»name
  "return the name of a file"

  ↑ Text string: ((self path subStrings: '/') last) attribute: TextColor blue

DirectoryNode»name
  "return the name of a directory"
  ↑ Text string: ((self path subStrings: '/') last) attribute: TextColor gray
```

Plain files are now listed in blue and directories in gray.

A number of emphasizes are available: bold, italic, underlined, and barred. The attribute corresponding to the emphasis is obtained by sending a message to the TextEmphasis class.

Attributes may be combined by being added to a text using addAttribute:. For example, the following name definition makes directory appears in gray and underlined:

```
DirectoryNode»name
  ↑ (Text fromString: ((self path subStrings: '/') last))
```

```
addAttribute: TextEmphasis underlined;
addAttribute: TextColor gray
```

1.3 Graph and Metagraph of a Browser

The remainder of this chapter reviews the notions introduced so far, but in a more formal fashion. We describe the internals of the framework, which should help the reader to understand when OmniBrowser is appropriate for a given task and what are its strengths and limitations.

The OmniBrowser framework is structured around (i) an explicit domain model and (ii) a metagraph, a state machine, that specifies the navigation and interaction with the domain model. The user interface is constructed by the framework, and uses a layout similar to the code browser, with two horizontal parts. The top part is a column-based section which supports navigation. 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 1.3, and explained briefly in the rest of this section.

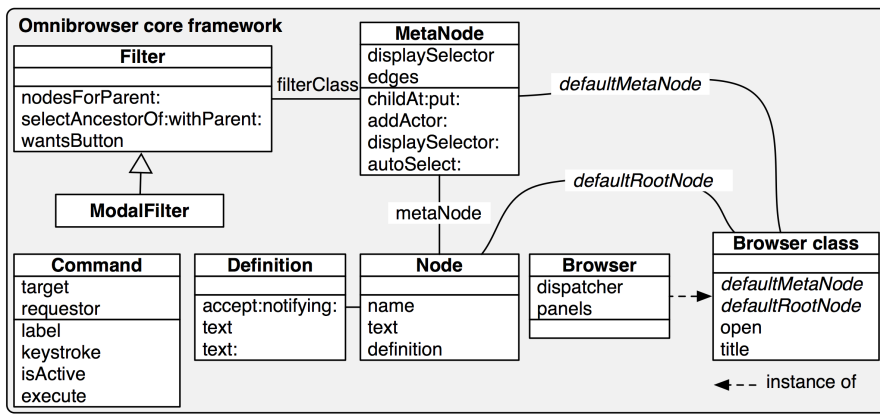

Figure 1.3: Core of the OmniBrowser framework.

Browser. A *browser* is a graphical tool to navigate and edit a domain space. This domain has to be described in terms of a directed cyclic graph (DCG). It is cyclic because, for example, file systems or structural meta models of

programming languages (*i.e.*, packages, classes, methods...) contain cycles, and we need to be able to model these. 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.

Metagraph. A browser's *metagraph* specifies the way a user traverses the graph of domain nodes. 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 with 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 interface and are responsible for handling exceptions that may be raised when they are executed.

Commands are defined in a non-invasive way: commands are added and removed without redefining any method of the core framework. This enables a smooth gathering of independently realized commands.

Filter. The metagraph describes a state machine. When the browser is in a state in which more than one transition is 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, the 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 an item selected, the transition actually occurs, 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 the text is expected to be edited by the browser user, then a *definition* is needed to handle modification and commitment (*i.e.*, an *accept* in the Smalltalk terminology). A definition is produced by a node.

Core Behavior of the Framework

The core of the OmniBrowser framework is composed of 8 classes (Figure 1.3).

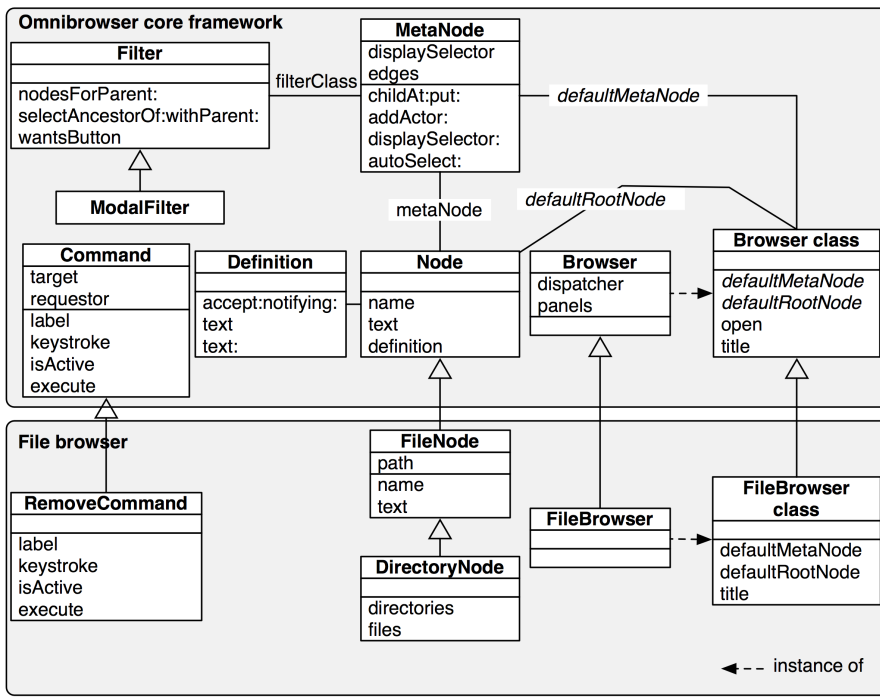

Figure 1.4: Core of the OmniBrowser framework and its extension for the file browser.

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. As we already saw, 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.

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 to be commands. Each of this method should return the *class* of the command (and not an instance of it).

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, the 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 occurs, 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 1.4). 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 meta-

graph 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 (as it will see later on in Figure 1.7) is 4, therefore the system browser has 4 lists (Figure 1.5). Because the metagraph of a file system contains a cycle (remember `directory childAt: #directories put: directory` in the file browser metagraph in Section 1.1), 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 1.5.

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

1.4 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 Pharo programming environment. It enables code navigation and code editing. Figure 1.5 shows the graphical user interface of this browser, and how it appears to the Smalltalk programmer.

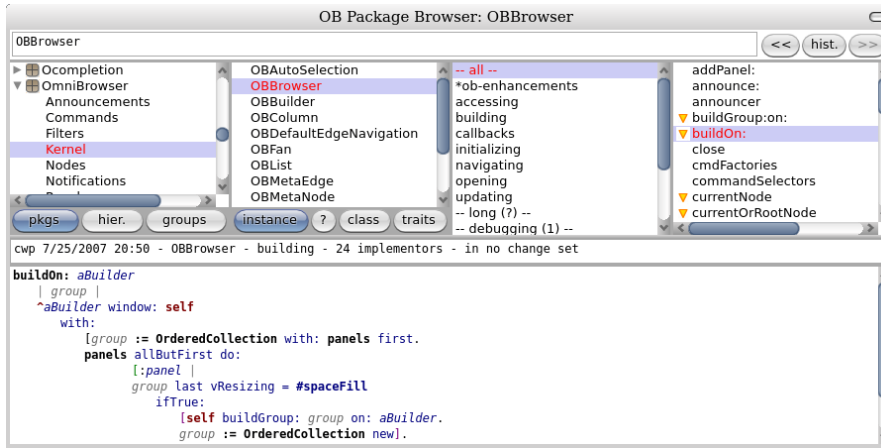

Figure 1.5: OmniBrowser based Smalltalk system browser.

The system browser is mainly composed of four lists (upper part) and a panel (lower part). From left to right, the lists represent (i) categories, (ii) classes contained in the selected 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 1.5, the class named Class, which belongs to the 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 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 Pharo 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 1.6 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.

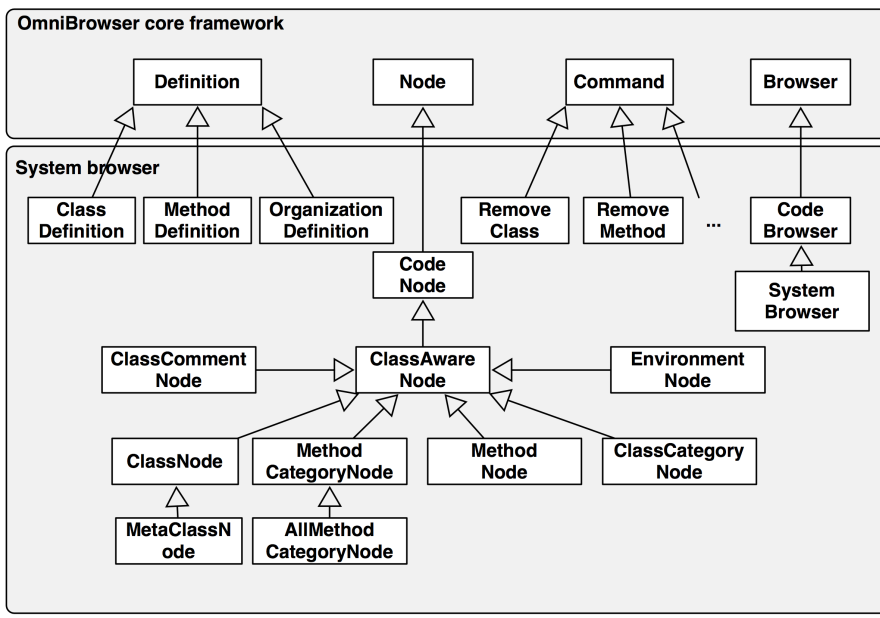

Figure 1.6: Extension of the OmniBrowser framework to define the system browser.

Compared to the default implementation of the Pharo System Browser this is less code and better factored. In addition other code-browsers can freely reuse these parts.

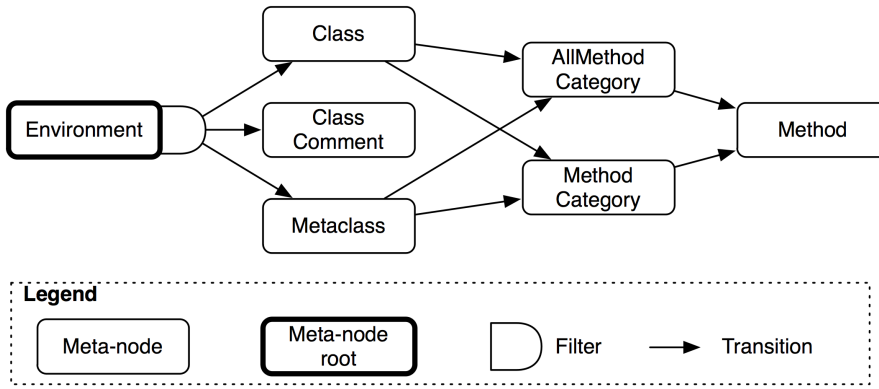

Figure 1.7: Metagraph of the system browser.

Figure 1.7 depicts the metagraph of the system browser. The metanode environment contains information about 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
```

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. For the 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. The modal filter was introduced earlier in the chapter.

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 category according to their position in a class hierarchy. If a 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 1.8.

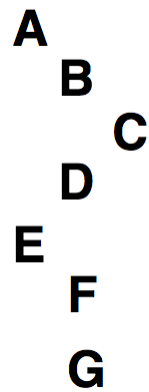

Figure 1.8: How OBClassSortFilter sorts and indents two distinct class hierarchies in one category.

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 category node and aCollection holds all class nodes this 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:.

The other filter defined for a category metanode, OBModalFilter, has a different task: It selects one edge of the three outgoing edges from the category metanode, *i.e.*, instance, comment or class. The user of the system browser can select using the switch in the class column 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 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.

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.

Widget 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.

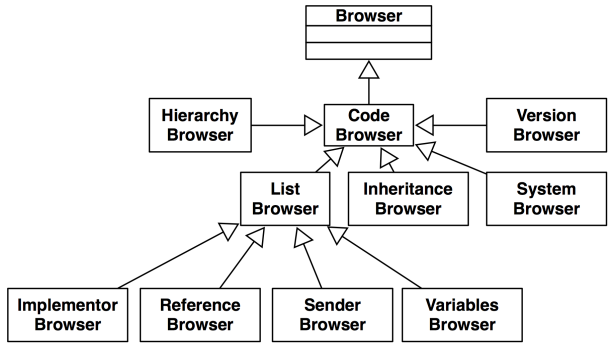

Figure 1.9: Some code browsers developed using OmniBrowser framework.

- 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 Pharo 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 in this path is associated to a domain node. This preserves the synchronization between different graphical widgets of a browser.

1.5 Evaluation and Discussions

Several other browsers supporting new language constructs such as Traits have been developed using OmniBrowser, which demonstrate that the framework is mature and extensible. Figure 1.9 shows some browsers that are based on OmniBrowser. We now discuss the strengths and limitations of the framework.

Strengths

Ease of use. The fact that the browser navigation is explicitly defined in one place lets the programmer easily understand and control 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. 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 Pharo 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 guarantee that a class has been 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 Pharo 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, the OmniBrowser framework constrains the space of its own extension. The OmniBrowser framework does not support well the definition of navigation that does not follow the strict 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 Chapter Summary

This chapter presents the construction of a very simple browser based on the OmniBrowser framework. It also presents the design and implementation of a complex tool, the Smalltalk system browser.

- A browser is implemented by subclassing the class OBBrowser.
- All the navigation permitted by the browser for a given domain is defined with a metagraph.
- A browser's metagraph is returned by the defaultMetaNode class-side method.
- A metagraph is made of instances of OBMetaNode linked each other by sending the message childAt: aName put: aMetaNode. The symbol aName is then used as a message sent to a node to obtain the nodes to populate the following node.
- Domain nodes are implemented by subclassing OBNode.
- The root domain node is returning from a browser by sending the defaultRootNode class-side message.
- An action the user can perform is implemented with a command.
- A command must subclass OBCommand and the methods label, isActive, keystroke and execute are usually overridden.
- To enable a user to edit the content of the lower text pane, a definition needs to be defined by subclassing OBDefinition.
- A definition answers to text to returns a textual content and text: to accept a new content.

- A node needs to answer to the definition message to enables the user to modify its textual content.
- An icon may be added to a node by overriding OBNode»icon.
- Filters may be added to introduce panes in order to split the list of elements.

Acknowledgment. Colin Putney developed the original Omnibrowser. A number of contributions were made by the Pharo community. In particular, we gratefully acknowledge David Röthlisberger for his devotion and valued work on the Pharo tools.