Chapter 1

Glamour

Browsers are a crucial instrument to understand complex systems or models. A browser is a tool to navigate and interact with a particular domain. Each problem domain is accompanied by an abundance of browsers that are created to help analyze and interpret the underlying elements. The issue with these browsers is that they are frequently (re)written from scratch, making them expensive to create and burdensome to maintain. While many frameworks exist to ease the development of user interfaces in general, they provide only limited support to simplifying the creation of browsers.

Glamour is a dedicated framework to describe the navigation flow of browsers. Thanks to its declarative language, Glamour allows one to quickly define new browsers for their data.

In this chapter we will first detail the creation of some example browsers to have an overview of the Glamour framework. In a second part, we will jump into details.

1.1 Installation and first browser

To install Glamour on your Pharo image execute the following code:

Jannik ►*i cannot load this code in PharoCore 2.0. We need a stable version*◄ Jannik ►*I am not sure that we must have loadDefault. We need a fixed version, like that, the source code will work in 3 years.*◄ Andrew ►*We will check this before publishing the book*◄

```
Gofer new
  squeaksource: 'Glamour';
  package: 'ConfigurationOfGlamour';
  load.
(Smalltalk at: #ConfigurationOfGlamour) perform: #loadDefault.
```

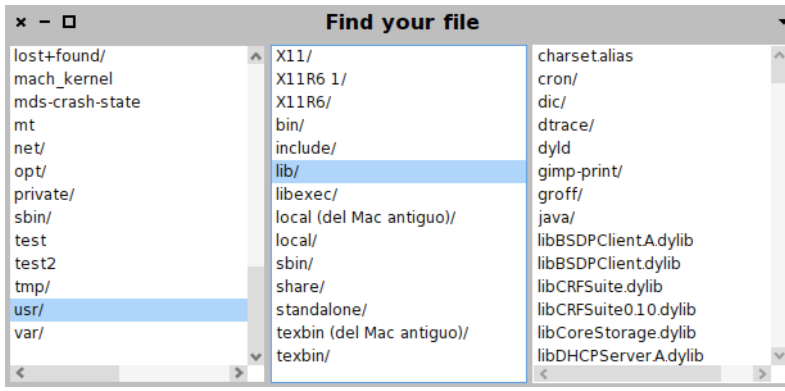

Figure 1.1: File finder as a Glamour implementation.

Now that Glamour is installed, we are ready to build our first browser by using Glamour's declarative language. What about building an Apple's Finder-like file browser? This browser is built using the Miller Columns browsing technique, displaying hierarchical elements in a series of columns. The principle of this browser is that a column always reflects the content of the element selected in the previous column, the first column-content being chosen on opening.

In our case of navigating through the file systems, the browser displays a list of a particular directory's entries (each files and directories) in the first column and then, depending on the user selection, appending another column (see Figure 1.1):

- if the user selects a directory, the next column will display the entries of that particular directory;
- if the user selects a file, the next column will display the content of the file.

This may look complex at first because of the recursion. However, Glamour provides an intuitive way of describing Miller Columns-based browsers. According to the Glamour's terminology this particular browser is called *finder*, referring to the Apple's Finder found on Mac OS X. Glamour offers this behavior with the class GLMFinder. This class has to be instantiated and initialized to properly list our domain of interest, the files:

```
| browser |
browser := GLMFinder new.
browser show: [:a |
  a list
```

```
    display: #children ].
browser openOn: FileSystem disk root.
```

Note that at that stage selecting a plain file raises an error. We will understand why and how to fix that situation soon.

From this small piece of code you get a list of all entries (either files or directories) found at the root of your file system, each line representing either a file or a directory. If you click on a directory, you can see the entries of this directory in the next column. The filesystem navigation facilities are provided by the Filesystem framework, thoroughly discussed in Chapter **??**.

This code has some problems however. Each line displays the full print string of the entry and this is probably not what you want. A typical user would expect only names of each entry. This can easily be done by customizing the list:

```
browser show: [:a |
  a list
    display: #children;
    format: #basename ].
```

This way, the message basename will be sent to each entry to get its name. This makes the files and directores much easier to read by showing the file name instead of its fullname.

Another problem is that the code does not distinguish between files and directories. If you click on a file, you will get an error because the browser will send it the message children that it does not understand. To fix that, we just have to avoid displaying a list of contained entries if the selected element is a file:

```
browser show: [:a |
  a list
    when: #isDirectory;
    display: #children;
    format: #basename ].
```

This works well but the user can not distinguish between a line representing a file or a directory. This can be fixed by, for example, adding a slash at the end of the file name if it is a directory:

```
browser show: [:a |
  a list
    when: #isDirectory;
    display: #children;
    format: #basenameWithIndicator ].
```

The last thing we might want to do is to display the contents of the entry if it is a file. The following gives the final version of the file browser:

```
| browser |
browser := GLMFinder new
  variableSizePanes;
  title: 'Find your file';
  yourself.

browser show: [:a |
    a list
        when: #isDirectory;
        display: [:each | [each children sorted]
                        on: Exception
                        do: [Array new]];
        format: #basenameWithIndicator].

browser show: [:a |
    a text
        when: #isFile;
        display: [:entry | [entry readStream contents]
                        on: Exception
                        do:['Can''t display the content of this file']]].

browser openOn: FileSystem disk root.
```

This code extends the previous one with variable-sized panes, a title as well as directory entry sorting **Veronica** ►*which sorting?*◄, access permission handling and file content reading. The resulting browser is presented in Figure 1.1.

This short introduction has just presented how to install Glamour and how to use it to create a simple file browser.

1.2 Presentation, Transmission and Ports

This section gives a realistic example and details the Glamour framework.

Running example

In the following tutorial we will be creating a simple Smalltalk class navigator. Such navigators are used in many Smalltalk browsers and usually consist of four panes, which are abstractly depicted in figure Figure 1.2.

The class navigator functions as follows: Pane 1 shows a list or a tree of *packages*, each package containing classes, which make up the organizational structure of the environment. When a package is selected, pane 2 shows a list of all classes in the selected package. When a class is selected, pane 3

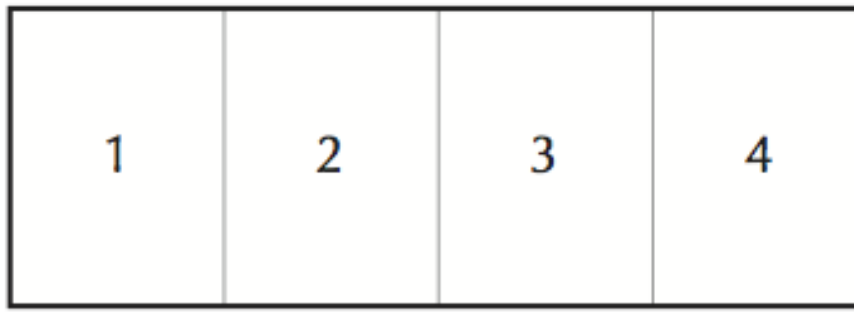

Figure 1.2: Wireframe representation of a Smalltalk class navigator.

shows all *protocols* (a construct to group methods also known as method categories) and all methods of the class are shown on pane 4. When a protocol is selected in pane 3, only the subset of methods that belong to that protocol are displayed on pane 4.

Starting the Browser

We build the browser iteratively and gradually introduce new constructs of Glamour. To start with, we simply want to open a new browser on the list of packages. Because the example is going to involve more code than the previous file browser, we are going to implement the code browser in a dedicated class.

The first step is then to create the class with some initial methods:

```
Object subclass: #PBE2CodeNavigator
  instanceVariableNames: 'browser'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'PBE2-CodeBrowser'

PBE2CodeNavigator class>>open
  ^ self new open

PBE2CodeNavigator>>open
  self buildBrowser.
  browser openOn: self organizer.

PBE2CodeNavigator>>organizer
  ^ RPackageOrganizer default

PBE2CodeNavigator>>buildBrowser
  browser := GLMTabulator new.
```

Executing PBE2CodeNavigator open opens a new browser with the text "a RPackageOrganizer" and nothing else. Note that we now use the GLMTabulator class to create our browser. A GLMTabulator is an explicit browser that allows us to place panes in columns and rows.

We now extend our browser with a new pane to display a list of packages.

Jannik ►*does it work with RPackage or Categories or both ?*◄ **Alex** ►*It works with RPAckage only, there is a ref to RPackageOrganizer*◄ **Jannik** ►*this is a problem, because RPackage is not in 1.3 by default*◄ **Alex** ►*I think PBE2 should be based on Pharo 1.4*◄

```
PBE2CodeNavigator>>buildBrowser
 browser := GLMTabulator new.
 browser
  column: #packages.

 browser transmit to: #packages; andShow: [:a | self packagesIn: a].

PBE2CodeNavigator>>packagesIn: constructor
 constructor list
  display: [:organizer | organizer packageNames sorted];
  format: #asString
```

Glamour browsers are composed in terms of *panes* and the *flow of data* between them. In our browser we currently have only one pane displaying packages. The flow of data is specified by means of *transmissions*. These are triggered when certain changes in the browser graphical user interface occur, such as an item selection in a list. We make our browser more interesting by displaying classes contained in the selected package (see Figure 1.3).

```
PBE2CodeNavigator>>buildBrowser
 browser := GLMTabulator new.
 browser
  column: #packages;
  column: #classes.

 browser transmit to: #packages; andShow: [:a | self packagesIn: a].
 browser transmit from: #packages; to: #classes; andShow: [:a | self classesIn: a].

PBE2CodeNavigator>>classesIn: constructor
 constructor list
  display: [:packageName | (self organizer packageNamed: packageName)
     definedClasses]
```

The listing above shows almost all of the core language constructs of Glamour. Since we want to be able to reference the panes later, we give them the distinct names "packages" and "classes" and arrange them in columns using the column: keyword. Similarly, a row: keyword exists with which panes can be organized in rows.

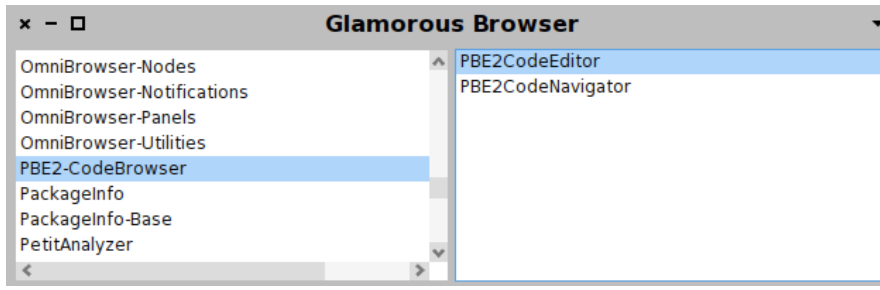

Figure 1.3: Two-pane browser. When a package is selected in the left pane, the contained classes are shown on the right pane.

The transmit:, to: and from: keywords create a *transmission*—a directed connection that defines the flow of information from one pane to another. In this case, we create a link from the *packages* pane to the *classes* pane. The from: keyword signifies the *origin* of the transmission and to: the *destination*. If nothing more specific is stated, Glamour assumes that the origin refers to the *selection* of the specified pane. We show how to specify other aspects of the origin pane and how to use multiple origins below.

Finally, the andShow: specifies what to display on the destination pane when the connection is activated or *transmitted*. In our example, we want to show a list of the classes that are contained in the selected package.

The display: keyword simply stores the supplied block within the presentation. The blocks will only be evaluated later, when the presentation should be displayed on-screen. If no explicit display block is specified, Glamour attempts to display the object in some generic way. In the case of list presentations, this means that the displayString message is sent to the object to retrieve a standard string representation. As we have previously seen, format: is used to change this default behavior.

Along with display:, it is possible to specify a when: condition to limit the applicability of the connection. By default, the only condition is that an item is in fact selected, *i.e.*, that the display variable argument is not null.

Another Presentation

So far, packages are visually represented as a flat list. However, packages are naturally structured with the corresponding class category. To exploit this structure, we replace the list by a tree presentation for packages:

```
PBE2CodeNavigator>>packagesIn: constructor
 constructor tree
  display: [ :organizer | (self rootPackagesOn: organizer) asSet sorted ];
```

```
  children: [ :rootPackage :organizer | (self childrenOf: rootPackage on: organizer)
      sorted ];
  format: #asString

PBE2CodeNavigator>>classesIn: constructor
 constructor list
  when: [:packageName | self organizer includesPackageNamed: packageName ];
  display: [:packageName | (self organizer packageNamed: packageName)
      definedClasses]

PBE2CodeNavigator>>childrenOf: rootPackage on: organizer
 ^ organizer packageNames select: [ :name | name beginsWith: rootPackage , '-' ]

PBE2CodeNavigator>>rootPackagesOn: organizer
 ^ organizer packageNames collect: [ :string | string readStream upTo: $- ]
```

The tree presentation uses a children: argument that takes a selector or a block to specify how to retrieve the children of a given item in the tree. Since the children of each package are now selected by our tree presentation, we have to pass only the roots of the package hierarchy to the display: argument.

At this point, we can also add Pane 3 to list the method categories (Figure 1.4). The listing below introduces no new elements that we have not already discussed:

```
PBE2CodeNavigator>>buildBrowser
 browser := GLMTabulator new.
 browser
  column: #packages;
  column: #classes;
  column: #categories.

 browser transmit to: #packages; andShow: [:a | self packagesIn: a].
 browser transmit from: #packages; to: #classes; andShow: [:a | self classesIn: a].
 browser transmit from: #classes; to: #categories; andShow: [:a | self categoriesIn: a].

PBE2CodeNavigator>>categoriesIn: constructor
 constructor list
  display:  [:class | class organization categories]
```

The browser resulting from the above changes is shown in figure Figure 1.4.

Multiple Origins

Adding the list of methods as Pane 4 involves slightly more machinery. When a method category is selected we want to show *only* the methods that

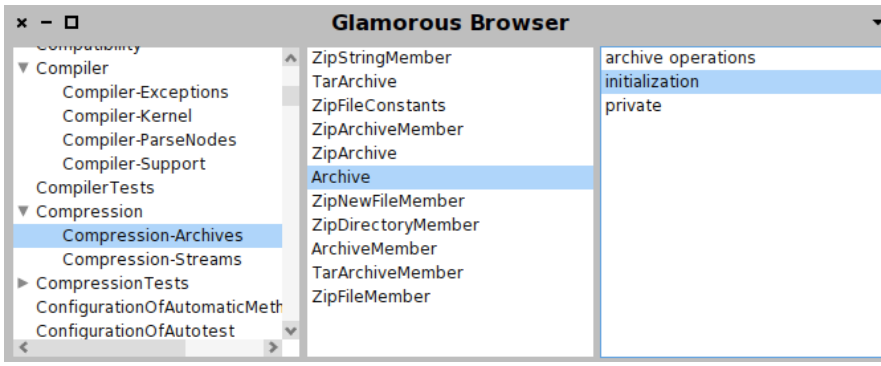

Figure 1.4: Improved class navigator including a tree to display the packages and a list of method categories for the selected class.

belong to that category. If no category is selected, *all* methods that belong to the current class are shown.

This leads to our methods pane depending on the selection of two other panes, the class pane and the category pane. Multiple origins can be defined using multiple from: keywords as shown below.

```
PBE2CodeNavigator>>buildBrowser
 browser := GLMTabulator new.
 browser
  column: #packages;
  column: #classes;
  column: #categories;
  column: #methods.

 browser transmit to: #packages; andShow: [:a | self packagesIn: a].
 browser transmit from: #packages; to: #classes; andShow: [:a | self classesIn: a].
 browser transmit from: #classes; to: #categories; andShow: [:a | self categoriesIn: a].
 browser transmit from: #classes; from: #categories; to: #methods;
      andShow: [:a | self methodsIn: a].

PBE2CodeNavigator>>methodsIn: constructor
 constructor list
  display: [:class :category |
           (class organization listAtCategoryNamed: category) sorted].
 constructor list
  when: [:class :category | class notNil and: [category isNil]];
  display: [:class | class selectors sorted];
  allowNil
```

The listing shows a couple of new properties. First, the multiple origins are reflected in the number of arguments of the blocks that are used

in the display: and when: clauses. Secondly, we are using more than one presentation—Glamour shows all presentations whose conditions match in the order that they were defined when the corresponding transmission is fired.

In the first presentation, the condition matches when all arguments are defined (not null), this is the default for all presentations. The second condition matches only when the category is undefined and the class defined. When a presentation must be displayed even in the presence of an undefined origin, it is necessary to use allowNil as shown. We can therefore omit the category from the display block.

The completed class navigator is displayed in Figure 1.5.

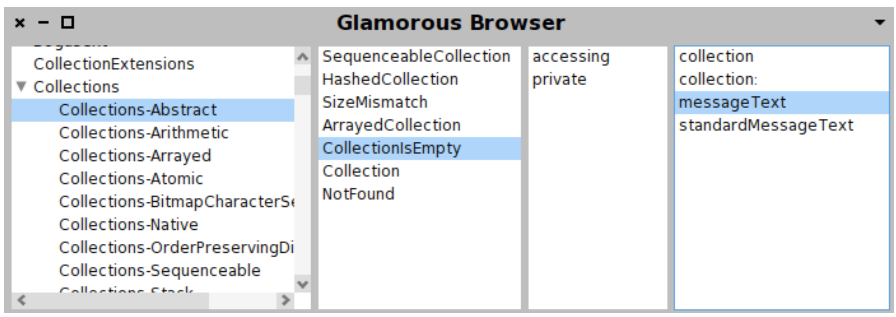

Figure 1.5: Complete code navigator. If no method category is selected, all methods of the class are displayed. Otherwise, only the methods that belong to that category are shown.

Ports

When we stated that transmissions connect panes this was not entirely correct. More precisely, transmissions are connected to properties of panes called *ports*. Such ports consist of a name and a value which accommodates a particular aspect of state of the pane or its contained presentations. If the port is not explicitly specified by the user, Glamour uses the *selection* port by default. As a result, the following two statements are equivalent:

```
browser transmit from: #packages; to: #classes; andShow: [:a | ...].
browser transmit from: #packages port: #selection; to: #classes; andShow: [:a | ...].
```

1.3 Composing and Interaction

Reusing Browsers

One of Glamour strengths is to use browsers in place of primitive presentations such as lists and trees. This conveys formidable possibilities to compose and nest browsers.

The subsequent example defines a class *editor* as shown in figure 1.6. Panes 1 through 4 are equivalent to those described previously. Pane 5 shows the source code of the method that is currently selected in pane 4.

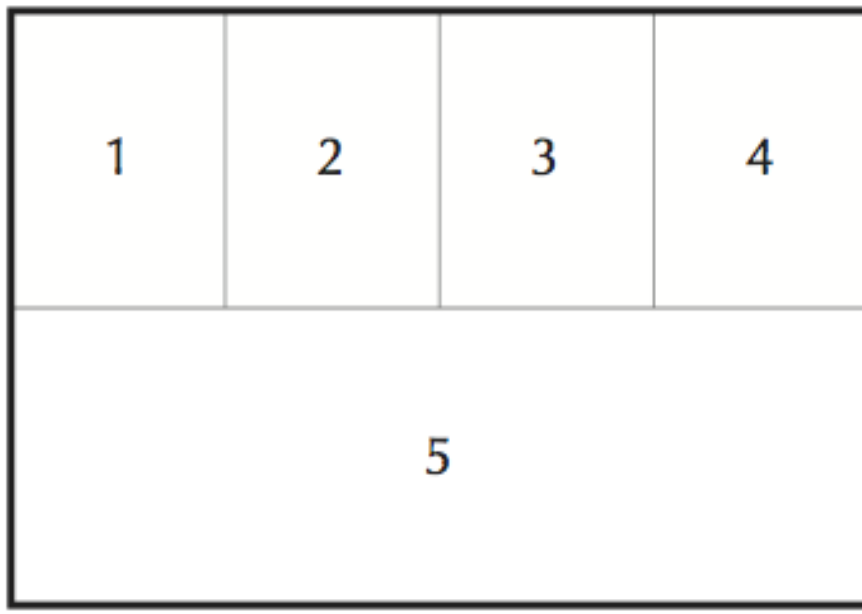

Figure 1.6: Wireframe representation of a Smalltalk class editor.

A new class PBE2CodeEditor will implement this editor. An editor will delegate the presentation of panes 1 through 4 to the previously implemented PBE2CodeNavigator. To achieve this, we first have to make the existing navigator return the constructed browser.

```
PBE2CodeNavigator>>buildBrowser
  ...
  "new line"
  ^ browser
```

We can then reuse the navigator in the new editor browser as shown

below.

```
Object subclass: #PBE2CodeEditor
   instanceVariableNames: 'browser'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'PBE2−CodeBrowser'.

PBE2CodeEditor class>>open
 ^ self new open

PBE2CodeEditor>>open
 self buildBrowser.
 browser openOn: self organizer

PBE2CodeEditor>>organizer
 ^ RPackageOrganizer default

PBE2CodeEditor>>buildBrowser
 browser := GLMTabulator new.
 browser
   row: #navigator;
   row: #source.

 browser transmit to: #navigator; andShow: [:a | self navigatorIn: a ].

PBE2CodeEditor>>navigatorIn: constructor
 constructor custom: (PBE2CodeNavigator new buildBrowser)
```

The listing shows how the browser is used exactly like we would use a list or other type of presentation. In fact, browsers are a type of presentation.

Evaluating `PBE2CodeEditor open` opens a browser that embeds the navigator in the upper part and has an empty pane at the lower part. Source code is not displayed yet because no connection has been made between the panes so far. The source code is obtained by wiring the navigator with the text pane: we need both the name of the selected method as well as the class in which it is defined. Since this information is defined only within the navigator browser, we must first export it to the outside world by using `sendToOutside:from:`. For this we append the following lines to `codeNavigator:`

```
PBE2CodeNavigator>>buildBrowser
 ...
 browser transmit from: #classes; toOutsidePort: #selectedClass.
 browser transmit from: #methods; toOutsidePort: #selectedMethod.

 ^ browser
```

This will send the selection within classes and methods to the *selected-*

Class and *selectedMethod* ports of the containing pane. Alternatively, we could have added these lines to the navigatorIn: method in the code editor—it makes no difference to Glamour as follows:

```
PBE2CodeEditor>>navigatorIn: constructor
  "Alternative way of adding outside ports. There is no need to use this
   code and the previous one simultaneously."

  | navigator |
  navigator := PBE2CodeNavigator new buildBrowser
        sendToOutside: #selectedClass from: #classes -> #selection;
        sendToOutside: #selectedMethod from: #methods -> #selection;
        yourself.

  constructor custom: navigator
```

However, we consider it sensible to clearly define the interface on the side of the code *navigator* rather than within the code editor in order to promote the reuse of this interface as well.

We extend our code editor example as follows:

```
PBE2CodeEditor>>buildBrowser
   browser := GLMTabulator new.
   browser
      row: #navigator;
      row: #source.

   browser transmit to: #navigator; andShow:  [:a | self navigatorIn: a].
   browser transmit
      from:  #navigator port: #selectedClass;
      from: #navigator port: #selectedMethod;
      to: #source;
      andShow:  [:a | self sourceIn: a].

PBE2CodeEditor>>sourceIn: constructor
   constructor text
      display: [:class :method | class sourceCodeAt: method]
```

We can now view the source code of any selected method and have created a modular browser by reusing the class navigator that we had already written earlier. The composed browser described by the listing is shown in figure 1.7.

Actions

Navigating through the domain is essential to find interesting elements. However, having a proper set of available actions is essential to let one to

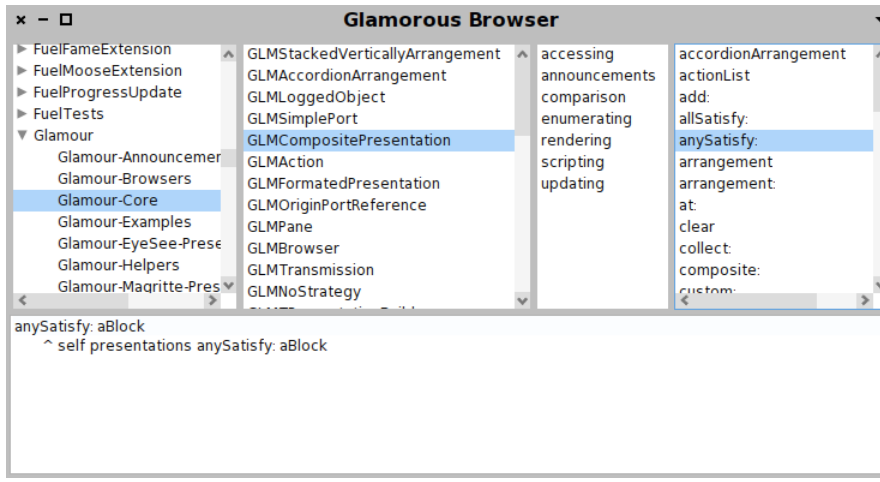

Figure 1.7: Composed browser that reuses the previously described class navigator to show the source of a selected method.

interact with the domain. Actions may be defined and associated to a presentation. An action is a block that is evaluated when a keyboard shortcut is pressed or when an entry in a context menu is clicked. An action is defined via act:on: sent to a presentation:

```
PBE2CodeEditor>>sourceIn: constructor
 constructor text
   display: [:class :method | class sourceCodeAt: method ];
   act: [:presentation :class :method | class compile: presentation text] on: $s.
```

The argument passed to on: is a character that specifies the keyboard shortcut that should be used to trigger the action when the corresponding presentation has the focus. Whether the character needs to be combined with a meta-key—such as command, control or alt—is platform specific and does not need to be specified. The act: block provides the corresponding presentation as its first argument which can be used to poll its various properties such as the contained text or the current selection. The other block arguments are the incoming origins as defined by from: and are equivalent to the arguments of display: and when:.

Actions can also be displayed as context menus. For this purpose, Glamour provides the messages act:on:entitled: and act:entitled: where the last argument is a string that should be displayed as the entry in the menu. For example, the following snippet extends the above example to provide a context menu entry to "save" the current method back to the class:

```
...
```

```
act: [:presentation :class :method | class compile: presentation text]
on: $s.
entitled: 'Save'
```

The contextual menu is accessible via the triangle downward-oriented above the text pane, located on the left hand side.

Multiple Presentations

Frequently, developers wish to provide more than one presentation of a specific object. In our code browser for example, we may wish to show the classes not only as a list but as a graphical representation as well. Glamour includes support to display and interact with visualizations created using the *Mondrian visualization engine* (presented in Chapter **??**). To add a second presentation, we simply define it in the using: block as well:

```
PBE2CodeNavigator>>classesIn: constructor
 constructor list
   when: [:packageName | self organizer includesPackageNamed: packageName ];
   display: [:packageName | (self organizer packageNamed: packageName)
                definedClasses];
   title: 'Class list'.

 constructor mondrian
   when: [:packageName | self organizer includesPackageNamed: packageName];
   painting: [ :view :packageName |
        view nodes: (self organizer packageNamed:  packageName)
                    definedClasses.
        view edgesFrom:  #superclass.
        view treeLayout];
   title: 'Hierarchy'
```

Glamour distinguishes multiple presentations on the same pane with the help of a tab layout. The appearance of the Mondrian presentation as embedded in the code editor is shown in figure 1.8. The clause title: sets the name of the tab used to render the presentation.

Other Browsers

We have essentially used the GLMTabulator which is named after its ability to generate custom layouts using the aforementioned row: and column: keywords. Additional browsers are provided or can be written by the user. Browser implementations can be subdivided into two categories: browsers that have *explicit panes*, *i.e.*,, they are declared explicitly by the user—and browsers that have *implicit panes*.

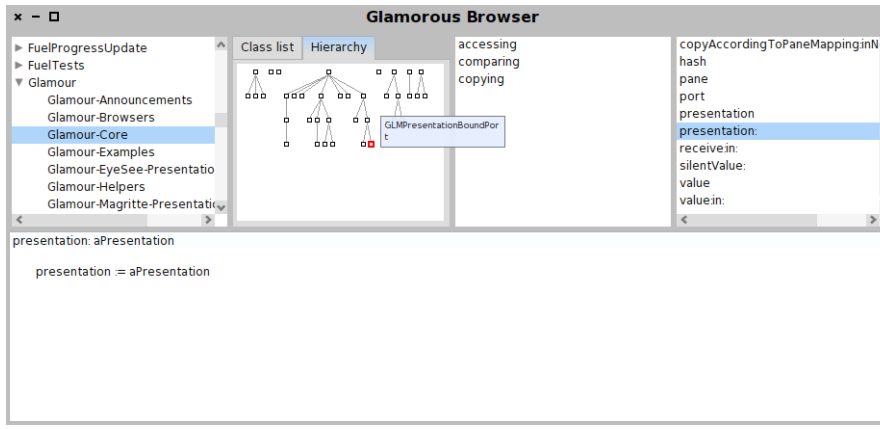

Figure 1.8: Code editor sporting a Mondrian Jannik ►*is it still available ?*◄ presentation in addition to a simple class list.

The GLMTabulator is an example of a browser that uses explicit panes. With implicit browsers, we do not declare the panes directly but the browser creates them and the connections between them internally. An example of such a browser is the Finder, which has been discussed in Section 1.1. Since the panes are created for us, we need not use the from:to: keywords but can simply specify our presentations:

```
browser := GLMFinder new.

browser list
  display: [:class | class subclasses].

browser openOn: Collection
```

The listing above creates a browser (shown in figure 1.9) and opens to show a list of subclasses of *Collection*. Upon selecting an item from the list, the browser expands to the right to show the subclasses of the selected item. This can continue indefinitely as long as something to select remains.

To discover other kinds of browsers, explore the hierarchy of the GLMBrowser class.

1.4 Chapter Summary

This chapter gave a short introduction to the Glamour browser framework. Glamour is essentially used to build to build tool that enable one to navigate and interact with an arbitrary domain, made of plain objects.

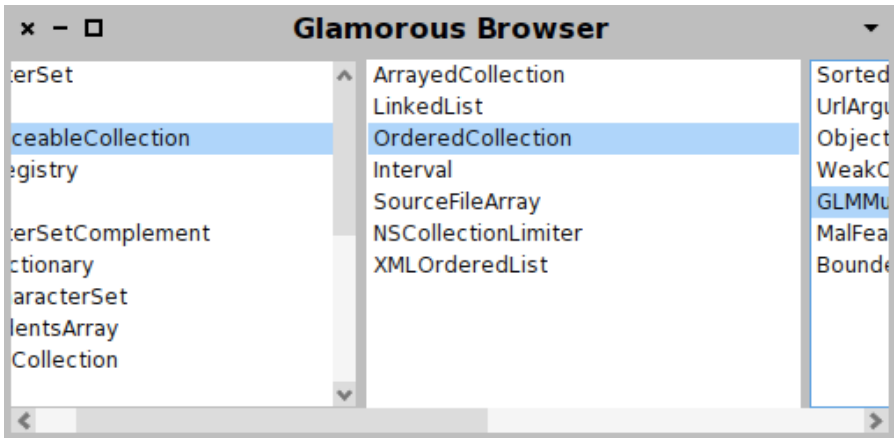

Figure 1.9: Subclass navigator using Miller Columns style browsing.

- GLMTabulator is a generic browser in which widget are ordered in columns and rows.
- Columns are defined by successively sending column: with a symbol name as argument. Rows are defined with row:.
- Data flows along transmissions set with transmit from: #source; to: #target.
- A transmission may have several source.
- List and text panes are obtained by sending list and text to a browser. Content is set with display: and items are formatted with format:.
- Ports define the component interface of a browser. This enables easy reuse and embedding.
- Interaction is defined in term of actions, defined by sending act: to a widget.
- Glamour support multiple presentations.
- Glamour is not made to build a general purpose graphical user interfaces.

Note that this chapter is not meant to give an exhaustive overview of Glamour, but is merely intended to introduce the reader to the usage and to our intent for our approach. For a more extensive view of Glamour, its concepts and implementation, the Moose book[1] has a dedicated chapter dedicated.

[1] http://www.themoosebook.org/book