Scripting Browsers with Glamour

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Philipp Bunge

April 2009

Leiter der Arbeit
Prof. Dr. Oscar Nierstrasz
Dr. Tudor Gîrba
Lukas Renggli

Institut für Informatik und angewandte Mathematik

glam·our /'glæmər/ *noun*
the attractive and exciting quality that makes a person, a job or a place seem special, often beacause of wealth or status. [Hornby, 2000]

Abstract

Browsers are a crucial instrument to understand complex systems or models. 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 rewritten 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.

In this thesis we present a dedicated model to describe browsers that equally emphasizes the control of navigation flow within the browser. Our approach is designed to support arbitrary domain models allowing researchers to quickly define new browsers for their data. To validate our model we have implemented the framework *Glamour* which additionally offers a declarative language to simplify the definition of browsers. We have used Glamour to re-implement several existing browsers and to explore the creation of new browsers.

Acknowledgements

First and foremost I thank Tudor Gîrba without whom this thesis would not have been possible. His unalterable confidence, resolute support, and seemingly persistent aspiration have managed what they tried to achieve—that I am proud of the work I present in this thesis. Doru, you did much more than "just your job." In equal measure, I wish to thank Lukas Renggli for his pervading support, both technical and general. Your door was always open to me when I had a question and I have learned a great amount from you.

My sincerest gratitude goes to Prof. Oscar Nierstrasz. I must thank him not only—like many have done before me—for permitting me to write my thesis at the Software Composition Group, but much more for the dedication and care he committed to me and my work, to the lectures he holds and to all of his students in general. It is this disposition that nurtured my motivation and interest for the field of computer science from the first lecture I had with him.

My appreciation extends to the entire SCG (regardlessly of whether you supported me or encouraged me by questioning my work) and to my professors, tutors and colleagues from whom I have learned so much in the past years.

I am deeply indebted to my friends. You know who you are and I would do you no justice by attempting to enumerate all of you. Without you I would not have studied, would not have finished, and—most importantly—would not have had such an incredibly enjoyable time doing either.

To my parents, my sister, Rahel and my entire family, both here and abroad—I have no words that could express my appreciation for the love and unconditional support you have given me. Not only in my studies, but as long as I have known you.

Thank you.

Contents

Chapter 1

Introduction

Browsers are important tools to understanding complex systems or models. Browsers allow us to interact with a system, to inspect its elements and to learn its structure. What distinguishes browsers from other user interfaces is the structure of the underlying data and how it is mapped to the visual representation shown to the user.

The models behind browsers can be seen as graphs. The nodes represent the entities that are apparent in the browser and the edges the relationships between these. In the case of a filesystem manager for example, the files and folders are the nodes of the browser and their hierarchical relationship—which can be queried using a set of messages on the objects—gives us the edges.

What is interesting is that there is a variety of methods that can be used to represent a model in a user interface. Some browsers may decide to show a tree like structure, where nodes can be expanded and collapsed. Other browsers may show just one folder at a time, allowing the user to descend into subfolders and return to parent folders. Yet other browsers may use other methods to allow the user to interact with the objects. We define methods by the *navigation flow* that they impose in a browser. Which navigation flow is most appropriate for a browser is specific to the application—certain browsers may favor specific types of interaction and different use-cases for the same model may require distinct navigational flows.

1.1. A Historical Introduction to Browsers

When object-oriented systems were first conceived, objects were thought of as self-contained processes, encapsulating all their required state as well as their behavior in single entity [Dahl and Nygaard, 1966]. In Smalltalk-76, this concept not only applied to the behavior of objects in solitude and their interaction in a larger system, but also in their responsibility to represent themselves to the user. In 1978, Daniel H. H. Ingalls described this as "the reactive principal," stating that

> the salient feature of Smalltalk is that all objects are active, ready to perform in full capacity at any time. Nothing of this aliveness should be lost at the interface to the human user of the system. In other words, all components of the system must be able to present themselves to the user in an effective way, and must moreover present a set of simple tools for their meaningful alteration. [Ingalls, 1978]

This concept related to the apparent disposition of "objects" in nature. An animal, plant or inert item simply represents itself, visually or otherwise, and does not require the interpretation of its inner processes by an outer entity to do so.

This restriction however, proved too limiting for visually presenting the same objects within different scenarios or contexts. While working as a visiting scientist at the Xerox Palo Alto Research Center between 1978 and 1979, Trygve Reenskaug wished to use Smalltalk-76 as the basis for a system for production control in shipbuilding [Reenskaug, 1996a]. He needed to represent the production schedule in multiple ways—depending on its intended usage—and found the generic representation of the objects to be insufficient. As a result, he tore each of the original objects apart, separating the original into three objects: an object responsible for the information, one for presenting the information, and one for capturing input from the user viewing the information in order to manipulate it accordingly. This work would later be integrated into Smalltalk-80 [Goldberg and Robson, 1989] and become widely known as the *Model-View-Controller* (MVC) pattern [Reenskaug, 1979].

Reenskaug, however, never intended for the breaking of encapsulation that the pattern promotes. He wrote later that the "top level goal was to support the user's mental model of the relevant information space and to enable the user to inspect and edit this information." [Reenskaug, 2003] and noted his exposure to Douglas Engelbart's work on *computer augmentation*—an approach to improving the intellectual effectiveness of the individual human being through the aid of computers [Reenskaug, 1996b; Engelbart, 1962]. In fact, the attempt to entitle the user to view and manipulate the domain objects directly is hampered by the model-view-controller when the view and controller become an instrument to the developer to abstract the model as much as possible before presenting it to the user.

Furthermore, the separation of responsibility also results in tighter coupling between the view and controller and the model. In their discussion of the model-view-controller pattern in *Pattern-Oriented Software Architecture — A System of Patterns*, Buschmann *et al.* note that

> both view and controller components make direct calls to the model. This implies that changes to the model's interface are likely to break the code of both view and controller. This problem is magnified if the system uses a multitude of views and controllers. [Buschmann *et al.*, 1996, pg. 142]

One possibility to mitigate this issue is to automatically generate a view and controller from a given model. An example of a framework that provides such a self-generative approach is the Naked Objects framework by Richard Pawson. The framework presents objects in

a consistent way to the user and presents standardized input methods to manipulate these objects [Pawson, 2004]. In this sense, Naked Objects promotes the presentation of the domain objects in an unadorned fashion—or, strictly naked.

This approach not only resolves the coupling issue but also returns control to the end user who can now understand and manipulate the underlying objects directly. Pawson also argues that this leads to improved development cycles as developers and end-users can share a common language. The framework furthermore promotes the behavioral completeness of objects and although Daniel Ingall's "reactive principal" is not followed in the strictest of senses, we can extend the above analogy on the relation to "objects" in nature by comparing the generic view to the processing of light by our retina that gives us a visual presentation of an entity's form. The mechanism with which we visually interpret objects using the reflective properties of light is generic and cares not about the specific entity in question.

A downside of this approach is that it sacrifices the use-case specific presentation for the sake of generality and simplicity. In fact, it is this generality that motivated Trygve Reenskaug to separate objects using the Model-View-Controller pattern.[1] Richard Pawson and Robert Mathews suggest that domain objects can be wrapped with a facade to adapt them to the needs of specific use-cases and further suggest that "capable progammers can extend the framework themselves to add new kinds of generic views" [Pawson and Matthews, 2002, pg. 66]. As a consequence, the generality imposes an expensive additional cost on the developer who wishes to optimize the presentation of the model to the end-user. Not only must the developer be an expert of the domain model, but to extend the presentation he must become an expert of the framework as well.

The importance of presenting the subject model unaltered to the user while providing sufficient flexibility for a wide range of use-cases becomes particularly important in the domain of browsers. With each emerging problem domain, a plethora of browsers are created to analyze and interpret the underlying elements. Alone for the purpose of navigating source code in Smalltalk based systems several new browsers have been developed over the years such as the *Refactoring Browser* [Roberts *et al.*, 1996; Roberts *et al.*, 1997; Fowler *et al.*, 1999], the *StarBrowser* [Wuyts and Ducasse, 2004], and *Whisker* [Way, 2005] that complement or displace the existing browsers built into the respective environment.

The issue with these browsers is that they are inflexible, hard to extend, often tied to a particular use-case and tightly coupled with their model. The navigation flow is intertwined with the rendering—we cannot use the rendering mechanism in the context of a different browser. This is a real advantage of the Naked Objects framework in that the rendering mechanism can be reused—it is entirely independent of the context. Furthermore, the navigation and interaction are frequently hardcoded, which may result in the browsers breaking when the underlying model changes, and makes them hard to maintain. This leads to browsers being rewritten from scratch—a time intensive and costly endeavor due to the

[1] It should be noted that Reenskaug was an external expert to Pawson's Ph.D. defense and speaks favorably of the "user-empowerment" that the Naked Objects framework provides.

lack of frameworks for building such browsers.

In our research, we focus strictly on browsers rather than on user interfaces in general as is the case with the Model-View-Controller or the Naked Objects framework. The reasons for this are twofold—first, there is no indication whether a generic system that tackles the aforementioned problems can even exist for all problem domains and secondly, there is a lot of opportunity to be gained from being able to easily construct browsers.

For example in the field of engineering—be it forward, reverse or re-engineering—the understanding of complex software systems is an important task. This has promoted some of the browsers mentioned above and tools such as the Moose reengineering environment [Ducasse *et al.*, 2005], as well as a multitude of metric measurement and visualization tools such as CodeCrawler [Lanza, 2004] and Mondrian [Meyer *et al.*, 2006]. But static analysis is often insufficient. Answering the question whether metrics and visualizations provide a sufficiently profound understanding for reengineering, Demeyer *et al.* write in *Object-Oriented Reengineering Patterns* that

> measurements alone cannot determine whether a entity is truly problematic: some human assessment is always necessary. Metrics are a great aid in quickly identifying entities that are potential problems but code browsing is necessary for confirmation. [Demeyer *et al.*, 2008]

Much can be gained from a framework that simplifies the construction of browsers. Researchers can create browsers to gain a better understanding of their models and end users can be permittedx direct access to the underlying objects. We propose a solution that provides these benefits while remaining flexible enough to adapt the presentation in a generic fashion for specific use-cases.

1.2. Challenges

With this introduction in mind we can present a requirements catalog of what we believe a modern browser framework should fulfill:

The browser should accommodate arbitrary domain models. To allow users and researchers to use the browsers to understand and inspect the underlying models, the browser should support arbitrary models and should not impose any requirements on a specialized model other than that the data can be queried by a simple meta-model.

The navigation flow needs to be completely controllable. Different use-cases require a different flow of information within the browser—independently of the domain model. For this the developer needs to be able customize and control the flow of navigation within the browser.

The presentation should be flexible. Implementors of the browsers need the flexibility to display the domain model in a use-case specific way. The browser model should therefore allow for both flow-control and rendering-control at the discretion of the developer, while providing reasonable defaults.

1.3. Our Approach

The challenge in our work lies in the difficulty to find a model that can comprehensively and flexibly transform a given domain model into a dedicated browser user interface.

We propose to map the entities of the domain models onto *panes*, which have a fixed position within a browser and take arbitrary *presentations* which can be changed on the fly. The navigation within the domain model is complemented by specifying the flow of data between the panes. Since the flow is triggered by actions performed upon a pane, the connections between panes are called *transmissions*. In this sense, our model is again a directed and possibly cyclic graph—albeit of a different abstraction than the domain model. The actual rendering then requires only a model consisting of these components and is entirely independent of the underlying domain model.

The framework puts the user in charge by using and presenting the underlying domain objects directly. When creating the browser model, the developer simply states with which instruments the objects should be displayed and how to get from one object to another. Thus, the browser model is simply a use-case specific representation of the data. More importantly, this approach enforces a strict separation between the *navigational flow* of a browser and its *representation*. The same navigational flow can be displayed using a variety of representations. The same representation can be applied to arbitrary navigation flows.

While we discuss related work in detail in chapter 5, it is worthwhile to note here that our framework has a resemblance to that of Vicki de Mey which she uses for visual composition of software applications [de Mey, 1995]. Despite the similarity between the two frameworks, the motivation and intent differ. De Mey's work serves the purpose of composing smaller software components into larger applications by ensuring plug-compatable interfaces between the components and facilitates this with a visual composition system. Our framework, on the other hand, aims to provide an infrastructure with which browsers for domain specific models can be easily created.

Another important contribution to simplifying the construction of browsers is the OmniBrowser framework, which mediates the definition of browsers using an explicit metamodel [Bergel *et al.*, 2007]. This browser framework is strongly influenced by the functionality of the Smalltalk style source code browsers and aspires to be a framework that supports the construction of such user interfaces. A disadvantage of the OmniBrowser framework this is that the flow is hardcoded—the framework mainly supports a style of navigation

that follows a left-to-right list display pattern (the selection of an item in one pane creates a new pane to the right). Furthermore, the meta-model works at the type-level of the model, making it difficult to follow a different information flow or a different presentation depending on the actual object in the model, rather than its type.

It may not be immediately apparent why the operation at the type-level can be a restraint. In practice however, models frequently have types which do not match the structure of the browser one wants to define or might even be inherently broken. An actual example is a file library where various types of files are exposed such as directories, symbolic links, device files, named pipes, and regular files. The model may not necessarily expose these different file types as different object types but only make the differences apparent through a set of query methods. Regardless of how the objects are implemented in the model, the browser must be able to accommodate these file types and be able to integrate new types without modifying the meta-model. Models which require a browser model which operates on the instance-level to adequately map them to a workflow occur frequently.

This is sufficient motivation to add a fourth item to the list of requirements:

The browser framework should work at the instance level of the model. Flow decisions need to occur at the instance level rather than at the type level as models might provide only limited reasoning from only their types.

1.4. Contributions

From the description and discussion we provide in the following chapters we can summarize these contributions which are at the core of our work:

1. Based on our analysis of existing browsers and browsing frameworks we identify a number of factors that are essential for a flexible browser model.

2. We implement a full-fledged reference implementation including a declarative language and multiple renderers that display the browsers using different graphical or web user-interface toolkits.

3. We show how existing browsers can be expressed with our approach and how our approach may be used to explore new paradigms for navigation.

We provide a tutorial for the declarative language in chapter 2, delving into the technical discussion of the model in chapter 3. We also show how our framework can be used to easily implement existing browsers in chapter 4. We compare our model with related work in chapter 5 and finally revisit the challenges in chapter 6 to assess whether Glamour fulfills these.

Chapter 2

Tutorial on Glamour

This chapter serves as a motivation for specifying custom browsers and as a hands-on introduction to our reference implementation. The screen images in this tutorial represent the appearance of the Adobe Flex [Adobe, 2006] graphical user interface for which a Glamour renderer is available in the VisualWorks implementation.

The chapter provides working examples. See appendix A on how to install the reference implementation in order to evaluate them.

2.1. 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 2.1.

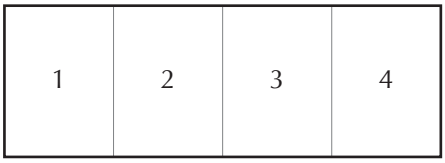


Figure 2.1. Wireframe representation of a Smalltalk class navigator.

The class navigator functions as follows: Pane 1 shows a list or a tree of *packages* (containing classes) and *bundles* (containing other bundles or packages) which make up the organizational structure of the environment and are collectively known as *pundles*. When a package is selected, pane 2 shows a list of all classes in the selected package. When a class is selected, pane 3 shows all *method categories* (a construct to group methods) and all methods of the class are shown on pane 4. When a method category is selected in pane 3, only the subset of methods that belong to that category are displayed on pane 4.

2.2. 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 this list of packages. The listing below shows how to create a new, simple browser and to open it on a given object. Since this object is a collection, Glamour uses a list for the presentation if nothing else is specified. Figure 2.2 shows the browser that is displayed by evaluating the script.

```
browser := TableLayoutBrowser new.
browser openOn: self model allPundles
```

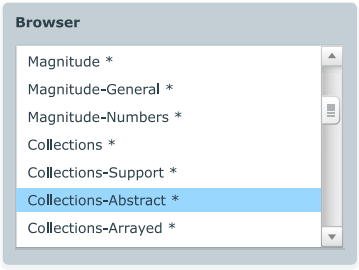


Figure 2.2. Basic browser construct, displaying a list of packages and bundles

2.3. Using Transmissions

In Glamour browsers are composed in terms of *panes* and the *flow of data* between them. The flow is specified by means of *transmissions*. These are triggered when certain changes occur, such as the change of the selection in a list.

To exemplify this, we add the second pane to display a list of classes for the currently selected package. Pane 1 can contain both *packages* and *bundles*, but only packages contain classes. Therefore, we should only attempt to display the list of classes on pane 2 when the selected item is actually a package.

The browser created by the following listing is displayed in figure 2.3. The lines marked in bold show the additions to the previous listing.

```
browser := TableLayoutBrowser new.
browser
    column: #pundles;
    column: #classes.
```

```
browser showOn: #classes; from: #pundles; using: [
    browser list
        display: [ :pundle | pundle containedClasses ];
        when: [ :pundle | pundle isPackage ]
].

browser openOn: self model allPundles
```

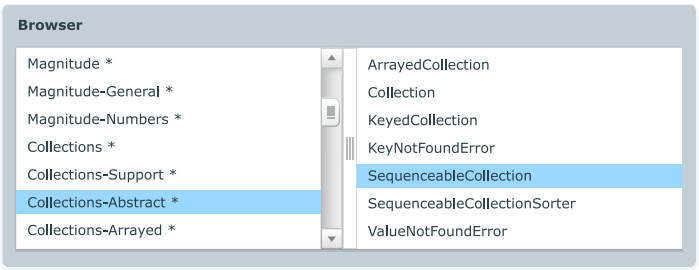


Figure 2.3. Two pane browser. When a package is selected in the left pane, the containing classes are shown on the right pane.

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 "pundles" and "classes" and arrange them in columns using the `column:` keyword. Similarly, a `row:` keyword exists with which panes can be organized in rows.

The `showOn:` 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 *pundles* pane to the *classes* pane. The `from:` signifies the *origin* of the transmission and `showOn:` 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 `using:` 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 condition in `when:` ensures that we only attempt to show this list when the selected item is a package and not a bundle.

Both `when:` and `display:` simply store 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 will attempt to display the object in some generic way. In the case of list presentations, this means that the `displayString` message will be sent to the object to retrieve a standard string representation.

2.4. Another Presentation

Up to now, we have been displaying the pundles as a list. The pundles in Smalltalk, however, are actually organized in a hierarchy and we have only been looking at the first level of this structure. To mend this, we specify that a tree presentation should be used instead:

```
browser := TableLayoutBrowser new.
browser
    column: #pundles;
    column: #classes.

browser showOn: #pundles; using: [
    browser tree
        children: #childPundles
].

browser showOn: #classes; from: #pundles; using: [
    browser list
        display: [ :pundle | pundle containedClasses ];
        when: [ :pundle | pundle isPackage ]
].

browser openOn: self model pundleRoot ~~allPundles~~.
```

Since the registry of pundles is specified in `openOn:` and not on a explicit pane, we use the keyword `showOn:` —without any `from:` —to create a transmission in which we can specify what to display on the *pundles* pane. The tree presentation uses a `children:` argument rather than `display:` that takes a selector or a block which specifies how to retrieve the children of a given item in the tree. In the example, the message `childPundles` will be sent to each pundle expanded in the view to retrieve its children. Since the children of each pundle are now selected by our tree presentation, we have to replace the selector for all pundles with just the root of the pundle hierarchy in the `openOn:` argument on the last line.

At this point, we can also add pane 3 that shows the method categories as shown in figure 2.1. The listing below introduces no new elements that we have not already discussed:

```
browser := TableLayoutBrowser new.
browser
    column: #pundles;
    column: #classes;
    column: #categories.
```

```
browser showOn: #pundles; using: [
    browser tree
        children: #childPundles
].

browser showOn: #classes; from: #pundles; using: [
    browser list
        display: [ :pundle | pundle containedClasses ];
        when: [ :pundle | pundle isPackage ]
].

browser showOn: #categories; from: #classes; using: [
    browser list
        display: [ :class | class categoryNames ]
].

browser openOn: self model pundleRoot.
```

Notice that we have not specified a `when:` condition for the list of methods. By default, the only condition is that an item is in fact selected, i.e. that the display variable argument is not `nil`.

The browser resulting from the above changes is shown in figure 2.4.

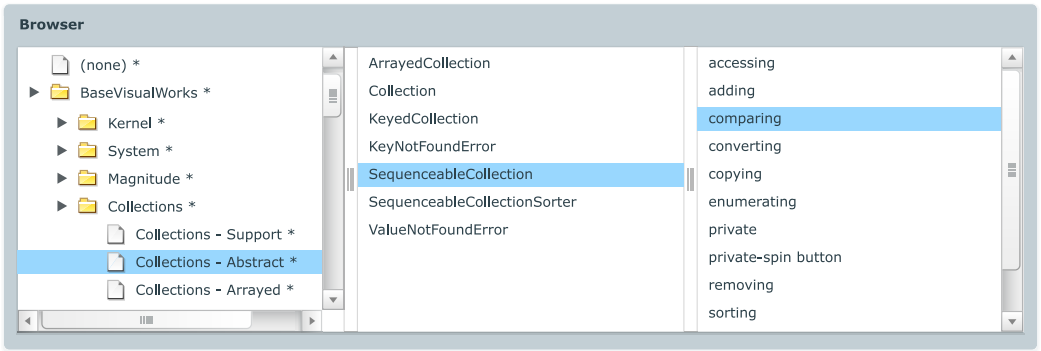


Figure 2.4. Improved class navigator including a tree to display the pundles and a list of method categories for the selected class.

2.5. Multiple Origins

The mechanism to show the methods is slightly more complicated. When a method category is selected we want to show *only* the methods that belong to that category. If no category is selected, we want to show *all* methods that belong to the current class.

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

```
browser := TableLayoutBrowser new.
browser
    column: #pundles;
    column: #classes;
    column: #categories;
    column: #methods.

browser showOn: #pundles; using: [
    browser tree
        children: #childPundles
].

browser showOn: #classes; from: #pundles; using: [
    browser list
        display: [ :pundle | pundle containedClasses ];
        when: [ :pundle | pundle isPackage ]
].

browser showOn: #categories; from: #classes; using: [
    browser list
        display: [ :class | class categoryNames ]
].

browser showOn: #methods; from: #classes; from: #categories; using: [
    browser list
        display: [ :class :category | class methodNamesIn: category ].
    browser list
        display: [ :class | class allMethodNames ];
        when: [ :class :category | category isNil ]
].

browser openOn: self model pundleRoot.
```

The listing shows a couple of properties we have not seen before. 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, we do not explicitly specify a condition so Glamour uses the default condition that all arguments must not be nil. The second condition matches only when the category is undefined but the class is not. We can therefore omit the category from the display block.

The completed class navigator is displayed in figure 2.5.

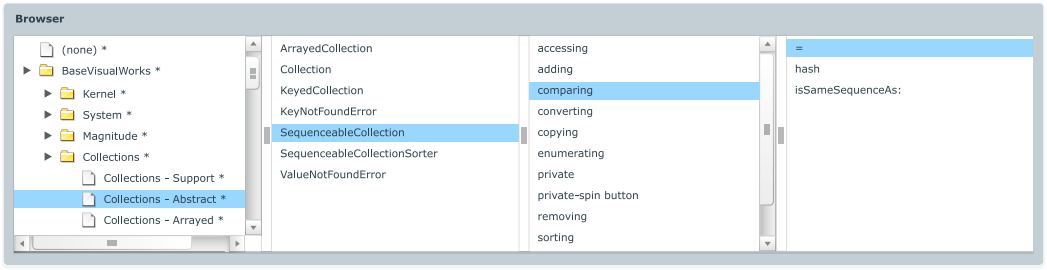


Figure 2.5. Complete code navigator. If no method category is selected, all methods of the class are displayed. Otherwise, only the methods that belong to that category are shown.

2.6. 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 showOn: #categories; from: #classes; using: [ ... ]
browser showOn: #categories; from: #classes -> #selection; using: [ ... ]
```

Other ports exist and may be used depending on the presentation. For example, the list presentation also populates the *hover* port when the user hovers over an item over a list and a text presentation updates the *text* port to reflect its contents as a user types within it. For a full reference, see the documentation of the presentations being used.

2.7. Reusing Browsers

On of the strengths of Glamour lies in the ability to use browsers in place of primitive presentations such as lists and trees. This allows us to reuse browsers and nest them within each other.

In the next example we want to create a class *editor* as shown in figure 2.6. Panes 1 through 4 are equivalent with those described in the class *navigator* in section 2.1. Pane 5 shows the source code of the method that is currently selected in pane 4.

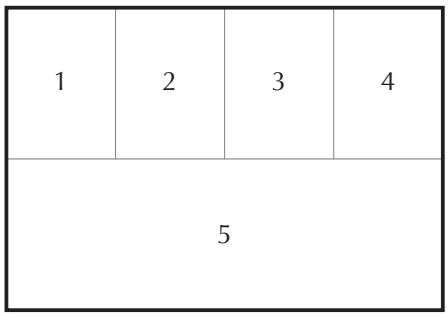


Figure 2.6. Wireframe representation of a Smalltalk class editor.

As panes 1 through 4 are the same as in the class navigator it would be convenient to reuse the class navigator rather than duplicating its code. To achieve this, we make the class navigator available through a method named `codeNavigator` which returns the browser, rather than opening it. We can then use the browser as shown in the listing for the class editor below.

```
browser := TableLayoutBrowser new.
browser
    row: #navigator;
    row: #source.

browser showOn: #navigator; using: [
    browser show: self codeNavigator
].

browser openOn: self model pundleRoot.
```

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.

When evaluating the code, a new browser is opened that shows the navigator embedded in the top pane and an empty pane at the bottom. No source code will be displayed because we have not yet created any connections between the panes. To get to the source, 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 the `sendToOutside: from:` message. For this we append the following lines to `codeNavigator:`

```
browser
    sendToOutside: #selectedClass from: #classes -> #selection;
    sendToOutside: #selectedMethod from: #methods -> #selection.
```

This will send the selection within classes and methods to the *selectedClass* and *selectedMethod* ports of the containing pane. Alternatively, we could have added these lines to the `self codeNavigator` instruction in the code editor—it makes no difference to Glamour at what point these are added. 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.

Note that a message for achieving the reverse—importing a port from the outside pane and storing its value on one of the browser's panes also exists and is known as `sendTo: fromOutside: .`

We extend our code editor example as follows:

```
browser := TableLayoutBrowser new.
browser
    row: #navigator;
    row: #source.

browser showOn: #navigator; using: [
    browser show: self codeNavigator
].

browser
    showOn: #source;
    from: #navigator -> #selectedClass;
    from: #navigator -> #selectedMethod; using: [
    browser text
        display: [ :class :method | class sourceCodeAt: method ]
].

browser openOn: self model pundleRoot.
```

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 2.7.

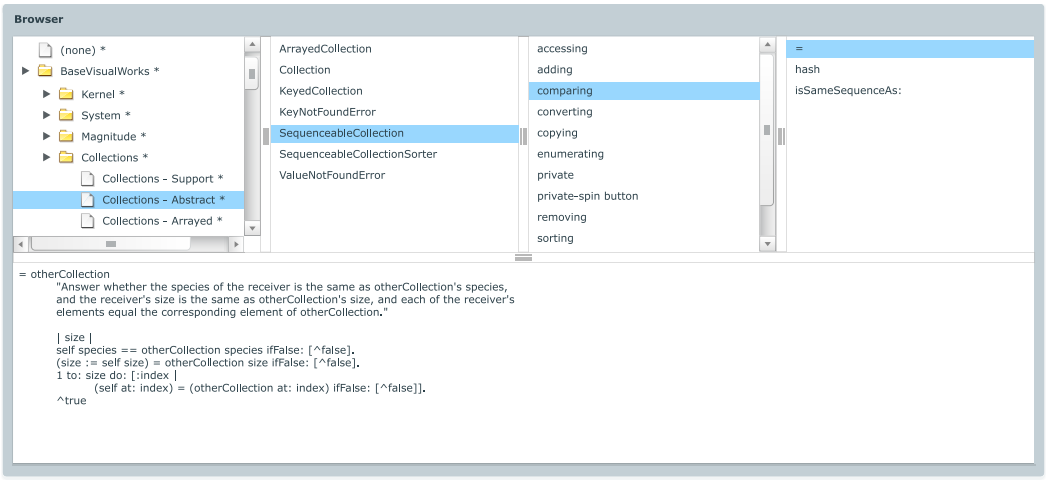


Figure 2.7. Composed browser that reuses the previously described class navigator to show the source of a selected method.

2.8. Actions

Browsers generally rely on *actions*—first-class behavioral objects that are executed when a keyboard shortcut is pressed or when an entry in a context menu is clicked. Glamour supports such actions through the `act: on:` message sent to a presentation:

```
browser := TableLayoutBrowser new.
browser
    row: #navigator;
    row: #source.

browser showOn: #navigator; using: [
    browser custom: self codeNavigator
].

browser
    showOn: #source;
    from: #navigator -> #selectedClass;
    from: #navigator -> #selectedMethod; using: [
    browser text
```

```
        display: [ :class :method | class sourceCodeAt: method ];
        act: [ :presentation :class :method |
            class installCode: presentation text at: method
        ] on: $s
].

browser openOn: self model pundleRoot.
```

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 need not 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 arguments to the block 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 installCode: presentation text at: method
    ] on: $s entitled: 'Save'
```

2.9. 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 visualization of their *system complexity* as well. Glamour includes support to display and interact with visualizations created using the *Mondrian visualization engine* [Meyer, 2006]. To add a second presentation, we simply define it in the `using:` block as well:

```
browser showOn: #classes; from: #pundles; using: [
    browser list
        display: [ :pundle | pundle containedClasses ];
        when: [ :pundle | pundle isPackage ].
    browser mondrian painting: [ :view :pundle |
```

```
        view nodes: pundle item allDefinedClasses.
        view edgesFrom: #superclass.
        view treeLayout.
    ]
]
```

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 2.8.

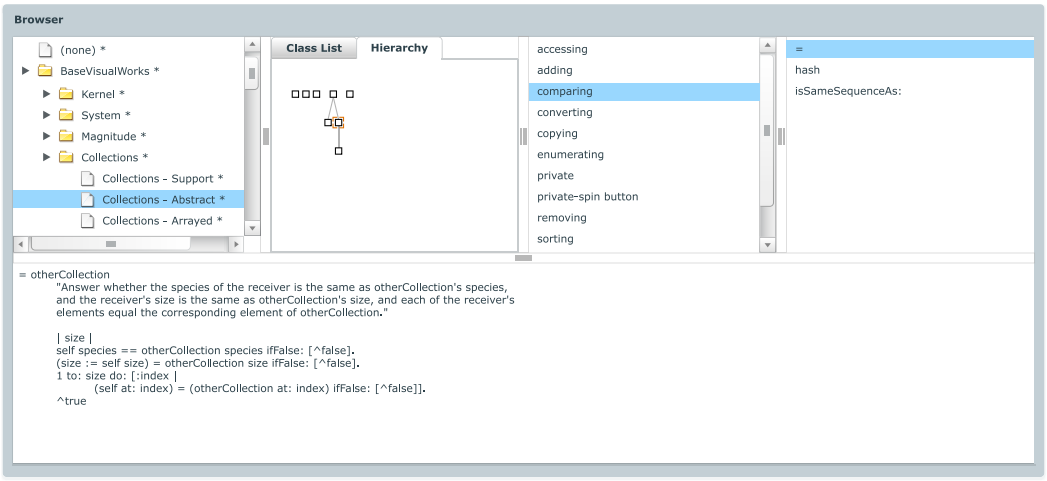


Figure 2.8. Code editor sporting a Mondrian presentation in addition to a simple class list.

2.10. Other Browsers

Up to now in the tutorial, we have only used the `TableLayoutBrowser` which is named after its ability to generate custom layouts using the aforementioned `row:` and `column:` keywords. Additional browsers are provided in the reference implementation 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*.

The `TableLayoutBrowser` 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 implements a *Miller Columns* style browsing, and is named after Mac OS X's *finder*, which

also employs such a mode of operation. Since the panes are created for us, we need not use the `from: to:` keywords but can simply specify our presentations:

```
browser := Finder new.

browser list
    display: [ :class | class subclasses ];
    whenKindOf: Class.

browser openOn: Collection
```

The listing above creates a browser (shown in figure 2.9) 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.

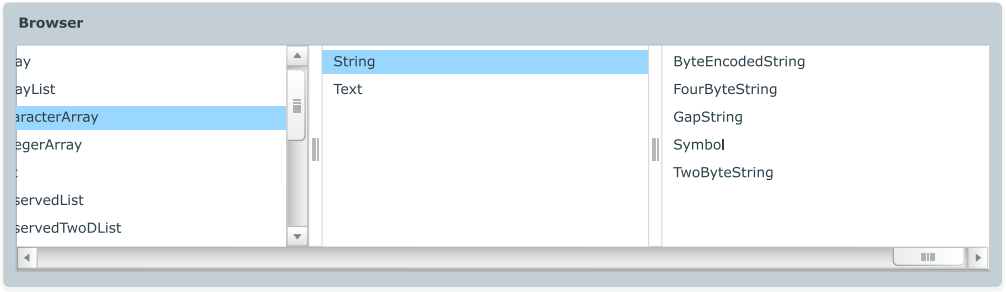


Figure 2.9. Subclasss navigator using Miller Columns style browsing.

2.11. Tutorial Conclusion

This concludes our tutorial of Glamour. Please note that this tutorial 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. Additionally, we have only focussed on the language in this chapter. In the next chapter *Inside Glamour*, we describe our model and the concepts behind Glamour in more detail.

Chapter 3

Inside Glamour

In this chapter we delve into the model of Glamour, the core contribution of our work. We cover Glamour's structure and the motivations behind our design. We also introduce our Smalltalk based reference implementation from a technical perspective.

A coarse overview of Glamour's structure can be seen in the UML class diagram depicted in figure 3.1 [Object Management Group, 2007].

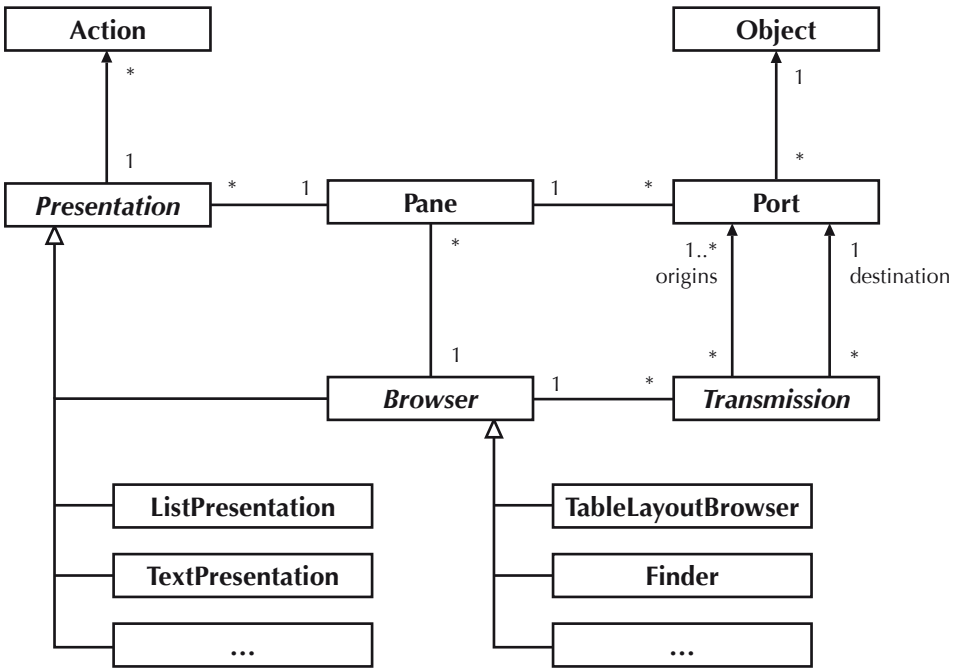


Figure 3.1. An overview of Glamour as a UML class diagram.

Pane defines the basic building block for browsers. A pane consists of a number of named

ports, which can store arbitrary data. Panes may also have one or more *presentations*.

Presentations declare a display strategy for a panes. With presentations, panes can change their visual display on the fly. A presentation may have units of behavior associated with them called *actions*.

Transmissions transfer information between panes or—more accurately—their ports. When triggered, transmissions take data from one or more origin ports and deposit the data at a destination port. Two concrete subclasses of transmissions exist: `SimpleTransmissions` which connect one origin to one destination and `BundleTransmissions` which may have multiple origins and can set presentations on the destination pane when they are triggered.

Browsers manage panes and transmissions. They are responsible for triggering transmissions. Browsers are themselves presentations, thus allowing them to be reused in other browsers in place of primitive presentations.

Not shown is the *renderer* which implements a visitor that transforms a composition of panes, presentations and browsers into user interface elements that are specific to the platform and that can be rendered on-screen or on a different medium as desired by the user. The medium–specific transformations are specified by a concrete subclass of `Renderer` such as a `WidgetryRenderer` for on–screen GUI elements or a `GlareRenderer` to stream user interfaces to an Adobe Flex client over a network.

While class diagrams are useful for showing the relationship between types, there is a necessity for describing *instances* of browsers and their current state. Rather than using standard UML object diagrams, we have developed our own graphical notation as a consequence of our work. Our abstract notation—of which an example can be seen in figure 3.2—simplifies the description and reasoning about browsers created using the Glamour meta-model. An extensive reference to the notation can be found in appendix B.

Figure 3.2 shows the abstract representation of the class editor which we created in the tutorial chapter (cf. figure 2.8). The browser is used for navigating through classes and their methods and consists of an outer browser (1), that contains one pane showing the source of the currently selected class and another pane that contains a second browser (2) which provides panes for the packages, classes, method categories and methods of a system. The panes (or actually their ports) are connected using a set of directed transmissions that describe the flow of information from one pane to another. We can see that the basic structure of the notation resembles that of the actual rendering of the browser.

In the following section we describe the components of a browser in detail with the aid of our abstract browser notation. In 3.1 we show how browsers are constructed and how their components are interconnected; in 3.2 we describe how presentations are employed as a strategy on how to represent an underlying entity; in 3.5 we discuss various browser variants and the style of workflow they represent and in 3.6 we show how these browsers are rendered. Finally, we discuss some specifics of the our reference implementation in 3.7.

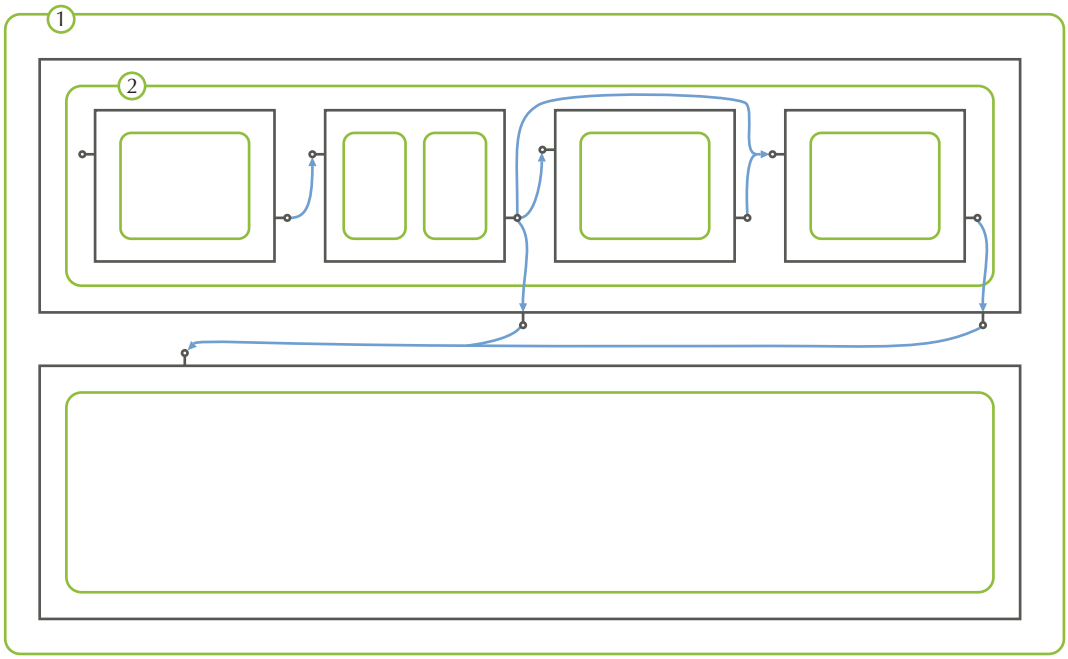


Figure 3.2. The tutorial browser from figure 2.8 in abstract notation.

3.1. Browsers, Panes and Transmissions

At the core, Glamour's model is a directed, possibly cyclic graph, consisting of *panes* that are connected using *transmissions*. This graph is encapsulated by a *browser*.

The main responsibility of a pane is to store arbitrary values at named locations—its *ports*. Ports have no enforced polarity or type—their interpretation depends entirely on the current *presentations* of the pane which access and manipulate the pane's ports. The names usually reflect their intended use, however, such as *selection* being the current selection of a list, *text* holding the content being inserted into a text input, *etc.*

The *transmissions* move data from one or more origin ports to one destination port. The browser acts as a broker, determining when and under which conditions transmissions should be triggered. In general, this occurs when a pane notifies its browser that one of its ports has changed its value. The browser then determines all the transmissions that originate at the port and triggers them sequentially in the order they were added to the model.

Two concrete transmission classes exist in our reference implementation. The first, `SimpleTransmission`, has exactly one origin port and one destination. Simple transmissions are

used whenever there is a requirement of simply copying a value from one port to another. An example of such a case is when we want to update the highlighted entity in a pane when the selection of another pane changes. When the selection of the first pane is modified, the pane notifies its containing browser which triggers the simple transmission originating at the *selection* port of the first pane, which then copies the value to the *highlight* port of the destination pane, effectively updating the highlighted item.

Further use cases for simple transmissions include sending port values to the outside of browsers by *forwarding* them and sending values to the inside of browsers by *capturing* thems (described in section 3.4) as well as the setting of port values by presentations contained within the pane (described in section 3.2).

The second type of transmission is a `BundleTransmission` which may have multiple origins but still only one destination. Bundle transmissions are the standard transmission type used between panes within a browser. They may carry a payload of a set of presentations which are inserted into the destination pane when the transmission is triggered. The use of bundle transmissions permits the modification of the representation of the pane on the fly. By making it the transmissions's responsibility to set the presentations, we maintain locality between the port values being transmitted and the presentations that will display them.

The origins of bundle transmissions are distinguished between *active origins* and *passive origins*. Both are specified by the developer creating the transmission. The browser will trigger the transmission whenever one of its active origins changes but not when only the port value of a passive origin changes. With passive origins, bundle transmissions are able to "pull-in" additional values that are relevant to displaying information on the destination pane.

Figure 3.3 focuses on a subset of the abstract notation diagram that highlights these components. In the following sections we add to this sub-diagram.

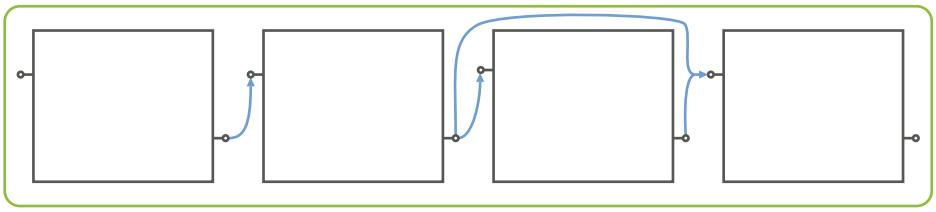


Figure 3.3. Browsers contain panes and transmissions, which connect panes via their ports.

3.2. Presentations

Presentations provide visual semantics to the state of panes. They read and interpret the values of selected ports—known as *input ports*—and, in turn, may choose to populate a set of other ports with values—the *output ports*.

A pane can have no presentation, a single presentation or multiple presentations at any given moment. Multiple presentations are usually displayed with the use of a tab panel. As the state is encapsulated by the pane, multiple presentations on the same pane will share that same state. In our abstract notation, presentations are displayed within their panes as depicted in figure 3.4.

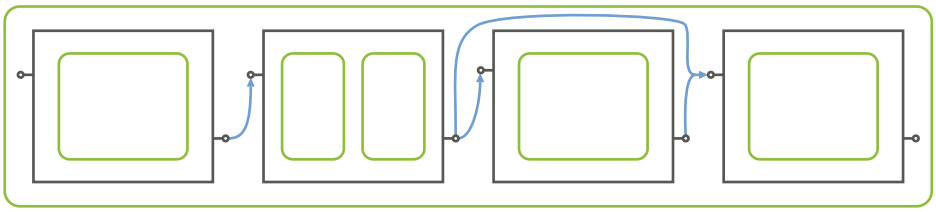


Figure 3.4. Presentations interpret and modify the state of their pane by reading from and writing to its ports.

Various concrete subclasses of presentation exist and are usually named after their recommended visual representation. For example, `ListPresentation` is rendered as a list user interface widget, `TextPresentation` as a text input. The concrete representation is not encoded within the presentation class due to the intentional separation of widget-toolkit specific behavior from our model. However, the renderers which create the user interface elements heed the suggested representation and render the presentation accordingly. In return, presentations provide a well-defined and extensive interface to the rendering client to avoid that renderers directly access or manipulate the pane and its ports to reflect changes of the user interface. This prevents coupling as the renderers do not have to access the state of the presentations directly but rely solely on a higher-level and well defined interface.

The presentations implement a strategy pattern [Gamma *et al.*, 1995] that uses a pane as its context object as is shown in figure 3.5. It allows us to change the behavior of the pane on the fly and dynamically apply a new filter. This has the advantage that the communication structure determined by the panes and components can be statically defined for a particular browser, but the representation can be dynamically changed depending on the particular instance in the domain model that is currently being displayed.

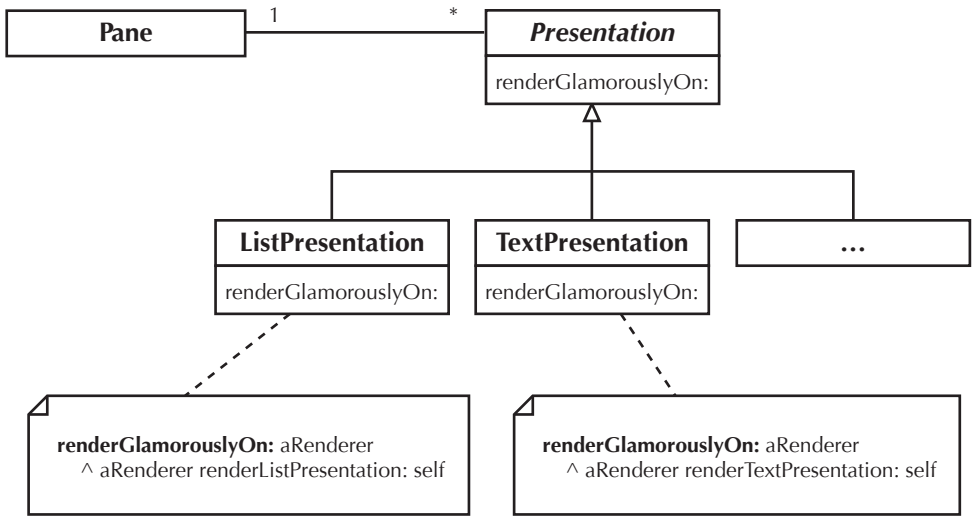


Figure 3.5. UML displaying how presentation strategies are employed by panes.

3.3. Actions

All presentations can be configured with a number of *actions*. Actions encapsulate units of behavior that can be executed upon the presentation or its corresponding panes. They may also be associated with a string name or a keyboard shortcut. The concrete representation of the actions lies in the responsibility of the renderer which may choose to simply trigger them when a key-combination is pressed on the keyboard or display them as a context menu to the widget that the presentation is associated with.

A frequent application for actions is to create a menu item or a keyboard shortcut that triggers a navigation, much like clicking on an item in a list may results in a navigation. The solution is to create an action that populates a port which is then connected to a destination pane using a normal transmission. When the action is executed, the port value changes, thus causing a navigation.

3.4. Composition

Browsers in Glamour can be composed by treating presentations and browsers equally. In fact, browser *are* presentations as one can see from our class diagram in figure 3.1. This means that, anywhere a list presentation or other type of primitive presentation may be used, a browser can be substituted instead. This simplifies the declaration of browsers in Glamour and promotes their reuse.

One major difference to a typical composition pattern rests in the use of the strategy pattern for presentations as discussed above. Since panes contain presentations and browsers contain panes (which are themselves presentations), the composition of browsers results in an indirect nesting as exemplified in the alternating chain shown in figure 3.6.

Figure 3.6. UML object diagram showing how components are composed.

With composed browsers, it is often a requirement to access port values of panes that are within browsers from outside that browser. The motivation for such behavior in our class editor was discussed in the tutorial chapter (see figure 2.7). To meet this requirement, the value of a port can be forwarded to another port. The usage of such forwarding to export values to the outside of a browser is shown in figure 3.7

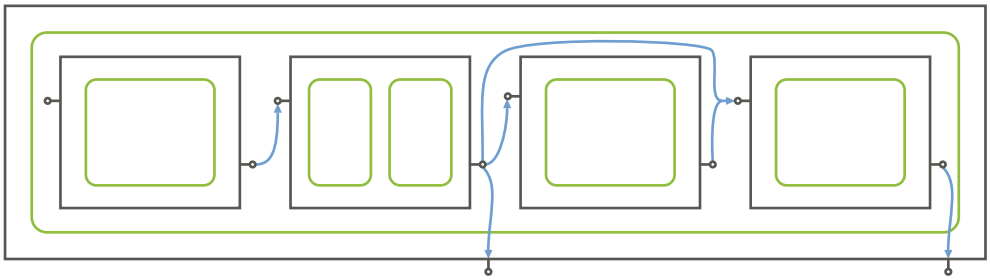


Figure 3.7. A browser forwards a port of one of its panes to its containing pane so that its value can be accessed from outside.

It is noteworthy that this forwarding of information is implemented as a standard *simple transmission*, managed by the inner browser. Whenever the origin port changes, the pane will inform the browser of this, triggering the transmissions originating at the port. As the browser's outer pane—and with it the destination port of the forwarding transmissions—may vary at runtime as the browser is deployed and removed from various panes, the destination port of a forwarding transmission is a *lazily evaluated* port. A lazily evaluated port is resolved to an actual port at the point where the transmission is fired.

Just as port values can be forwarded to an outside pane, browsers can also capture port value changes on the encapsulating pane and forward these to an inner port. This would essentially be the reversal of the direction of the arrows to the outside shown in figure 3.7. The purpose of this type of capturing would be to allow certain properties of a browser—such as the selection or the highlighted item of a specific pane—to be modified from the outside.

3.5. Browser Implementations

The exact handling of panes and transmissions depends on a concrete subclass of browser. Several types of browsers exist but we can differentiate between two general categories: browsers with *explicit* pane configurations and browser with *implicit* pane configurations. In the first case, panes must be declared and configured by the user and usually remain static at runtime. In the second case, the browser creates and destroys panes as needed. We provide an example for each.

The `TableLayoutBrowser` requires its panes to be explicitly declared and is named after the ability to customize the layout of its panes. The browser is configured with a number of rows or columns which then may be subdivided in columns or rows respectively. Each cell that is defined in this way represents a pane and is given a unique name.

An example of a browser which uses implicit pane configurations is the *finder* which implements a *Miller Columns* style browsing. The browser starts by displaying only one pane. When the selection changes within this pane, the browser will create a new pane to the right, connect the two panes using a transmission and fire the transmission to populate the new pane. This process is always repeated for the rightmost pane. When the selection changes for a pane that is not the last one in the list, all panes to the right will first be destroyed before creating a new pane.

3.6. Rendering

In our reference implementation we have created two concrete renderers: `WidgetryRenderer` builds a representation of browsers using the Widgetry GUI library in VisualWorks and `GlareRenderer` registers a Glare application [Bunge *et al.*, 2008] that streams the user interface to an Adobe Flex client over a network connection.

A renderer acts as a *visitor* [Gamma *et al.*, 1995] that traverses the component tree of browsers, panes and presentations, creating appropriate user interface elements for the components it encounters. As these elements are created, the renderer is also responsible for connecting the appropriate library-dependent callbacks to populate panes and their ports with the appropriate values when actions are performed or the state of the user interface components changes.

In `GlareRenderer` for example, selecting an item within a list widget triggers a callback that sets the value of the *selection* port of the corresponding pane. Double clicking populates the *execution* port and hovering over an item sets the value of the *hover* port. Different widgets might set other port values and some renderer implementations may choose not to support hovering or other operatives at all. Nevertheless, renderers do not communicate with the pane directly and use only the interface given to them by the presentation. This helps

renderers to work in a consistent and predictable way. A user writing a browser would not expect that in one rendition of a model the selection of an item in the list would result in the setting of *selection* and in another in the setting of *choice*.

Developers wishing to create user interface elements for other environments need only create a new subclass of `Renderer` and implement the appropriate methods. An incomplete list of these is shown in figure 3.8. Due to their similarity, the existing renderers may serve as a guide in determining the expected behavior of new implementations.

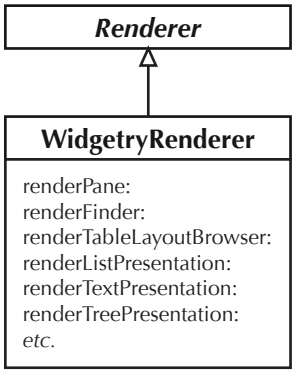


Figure 3.8. A subclass of renderer and some of the implemented methods.

3.7. Smalltalk Implementation

While our core contribution is the meta-model, we have also developed a declarative language in our reference implementation to construct specific browser models. The declarative language of Glamour is implemented as an internal domain specific language [Fowler, 2007] (also known as an embedded DSL [Hudak, 1998]) that uses Smalltalk as its host language.

In our language implementation, we make heavy use of Smalltalk's features such as block closures and cascades. Take for example the following code snippet:

```
browser showOn: #classes; from: #pundles; using: [
    browser list
        display: [ :pundle | pundle containedClasses ];
        when: [ :pundle | pundle isPackage ].
    browser mondrian painting: [ ... ]
].
```

The block closure argument passed to `using:` ensures that the contained list of presentations belong to the same transmission. We use cascades as with `display:` and `when:` to pass multiple options to the same object—a presentation in this example. Block closures are also used to define anonymous callbacks for these arguments, such as the filter condition in when or the display block.

Rather than building an intermediate representation our declarative language works directly on the Glamour model. When we call `browser list` for example, our script tells the browser to add a new list presentation to the latest transmission and returns that presentation so that `display:` and `when:` can be sent to it. To facilitate this, our scripting language is implemented as a set of *class extensions* to the core model. We use extensions rather than implementing the methods in the classes directly to provide some separation between the programmatic and the scripting interfaces, and thus to improve maintainability. Figure 3.9 shows a selection of the class extensions introduced by the scripting implementation.

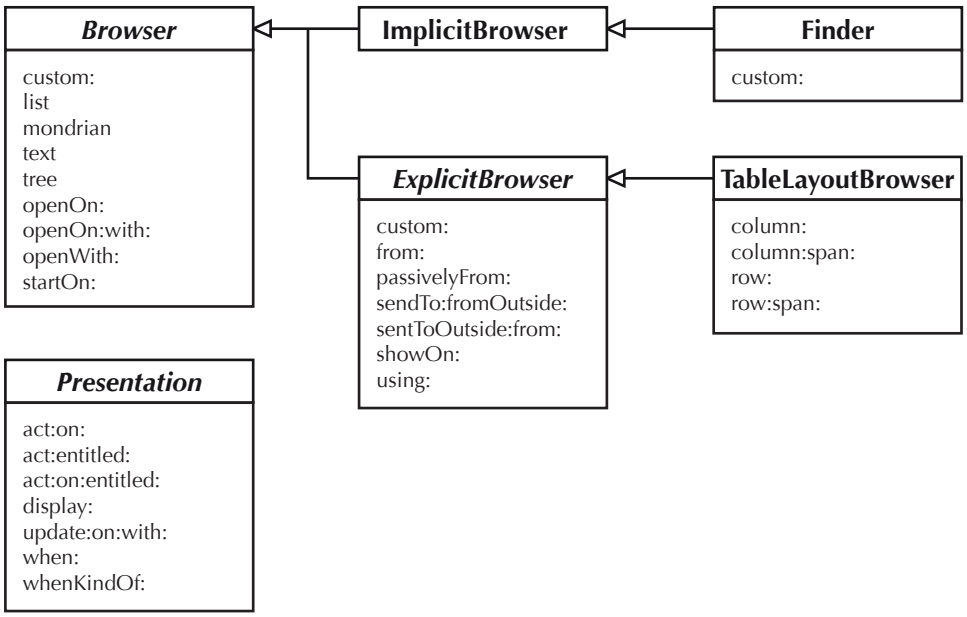


Figure 3.9. Class extensions made by the scripting language implementation.

The most interesting method shown above is the `custom:` message. Sending this message adds a *custom presentation* to the current context of the browser. The `list`, `text`, `tree`, and `mondrian` messages are just convenience messages that call `custom:` with an appropriate presentation instance. For example, the `list` message is implemented as follows:

```
list
    ^ self custom: ListPresentation new.
```

Most of the extensions are quite straightforward, simply calling their programatic equivalent on the model. To add an action to a presentation for example, the `act: on:` message is implemented as:

```
act: aBlock on: aCharacter
    self addAction: (Action new
        action: aBlock;
        shortcut: aCharacter;
        yourself)
```

The list of messages is evolving and some are added simply for convenience to the developer. An example is the navigate-upon-action mechanism described in 3.3 where the pressing of a keyboard shortcut or the clicking of a menu item triggers a navigation within the browser. Since this is such a common requirement, our reference implementation provides the dedicated message:

```
update: aPortSymbol on: aCharacter with: aBlock
```

Whenever the shortcut key defined by `aCharacter` is pressed, the message causes the port named by `aPortSymbol` to be updated with the result of evaluating `aBlock`, whose arguments are described in 2.8. Similar messages with `entitled:` are available for defining menu entries.

The following example shows how this mechanism is used to navigate to either the subclasses or the superclass of a selected class when a particular shortcut is pressed:

```
browser showOn: #classes; using: [
    browser list
        update: #relatedClass on: $b with: [ :list |
            list selection subclasses ].
        update: #relatedClass on: $p with: [ :list |
            Array with: list selection superclass ]
].

browser showOn: #relatedClasses;
    from: #classes -> #relatedClass; using: [
    browser list
].
```

3.8. Model Implementations

In addition to our VisualWorks reference implementation, Glamour has attracted the interest of other researchers who work on other platforms. Tudor Gîrba, Lukas Renggli and David Röthlisberger have successfully ported Glamour to the Pharo Smalltalk dialect. The code base is taken largely from the reference implementation but other renderers have been implemented to fit the environment. As Pharo primarily uses the direct-manipulation user interface *Morphic* to represent its widgets, building a corresponding visitor for Glamour was required. Additionally, a renderer for the Seaside web application framework has been written which renders Glamour models using a combination of basic HTML components and asynchronous Javascript (AJAX).

Probably the most important result of this work for us is that the researchers found it easy to write additional renderers. It provides additional justification for separating the rendering from our model and shows that the added complexity arising from this separation is reasonable.

Chapter 4

Constructing Common Browsers

There is a necessity to validate the applicability and effectiveness of our core contribution: the meta-model. In order to achieve this, we have created the reference implementation *Glamour* and used it to implement various browsers.

As an initial validation of Glamour's expressiveness, we have used our framework to re-implement several browsers which are valuable in their domain. Although these browsers are generally hardcoded, they prove to be valuable as they provide a workflow that is tested and effective for the navigation and interaction with their domain model. This makes them an interesting case-study for the validation of Glamour as we can investigate if these workflows can be adequately replicated using our model.

4.1. Filesystem Navigation

Almost every desktop environment provides a tool to navigate and manipulate the filesystem. Common functions of such file managers or file browsers are the creation, editing, viewing and opening of files and folders. While many different methods for displaying the filesystem exist, we have focussed on three navigational models that are used in many graphical file managers. Furthermore, while the prototypes displayed here focus more on the actual navigation of the filesystem, it is easy to add functions such as the creation and editing of files and folders with the use of Glamour's actions.

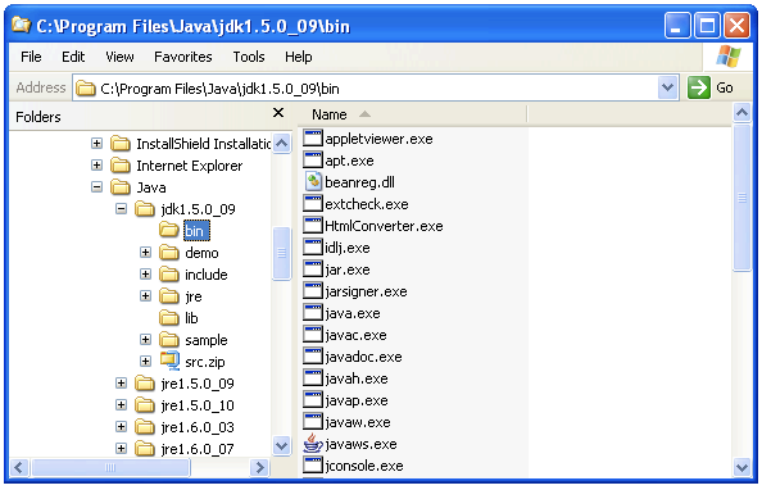
Tree navigation. A prominent style of a navigational manager is the use of two panes, one displaying the directories of the system in a tree layout and another displaying the content of the currently selected directory or just the contained files. This is the style used also by the *Explorer* on Microsoft's *Windows* operating system. Figure 4.1 shows an explorer window and an equivalent Glamour implementation. The Glamour implementation—albeit a simplification of the Windows Explorer—is shown below. The script contains two keywords that we have not yet encountered: the `span:` argument defines how much the column

should span in the table layout and the `format:` argument tells the tree presentation how to display each item (we only want the "tail", or the last part of the file path).

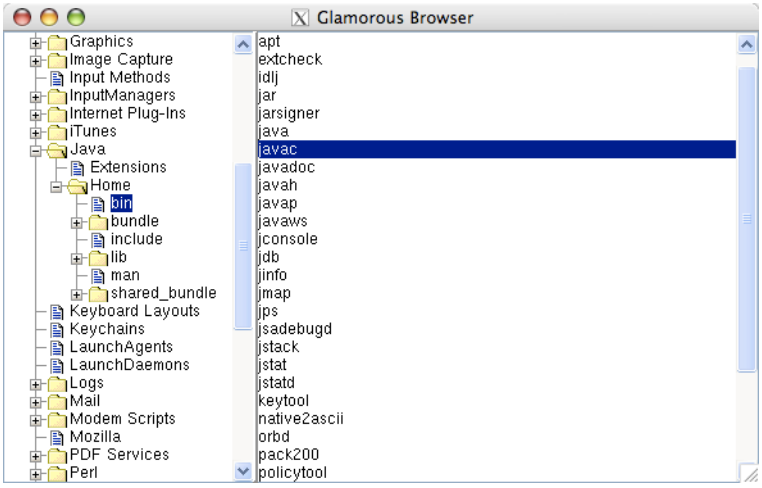
```
fileExplorer

    | browser |
    browser := TableLayoutBrowser new.
    browser
        column: #folders;
        column: #files span: 2.

    browser showOn: #folders; using: [
        browser tree
            title: 'Folders';
            children: [ :each | each files select: #isDirectory ];
            format: #tail.
    ].

    browser showOn: #files; from: #folders; using: [
        browser list
            format: #tail;
            display: [ :folder | folder files reject: #isDirectory ];
            when: [ :folder | folder asFilename isDirectory ].
    ].

    ^ browser
```

Finder navigation. Another well known example is the *Miller Columns* style navigation used by the *Finder* on Apple's *OS X* platform. Here, each column represents a folder or a file and shows the contents of that item. Folders display a list of their content and create a new column to the right if an item is selected. In Glamour, such left-to-right navigation is provided by the *Finder* browser implementation which—unlike its namesake—can be used to navigate over other types of objects as well in the same manner. The original finder and its Glamour correspondent are shown in figure 4.2, of which the latter is implemented in less than ten lines of code.

```
fileFinder

    | browser |
    browser := Finder new.
```

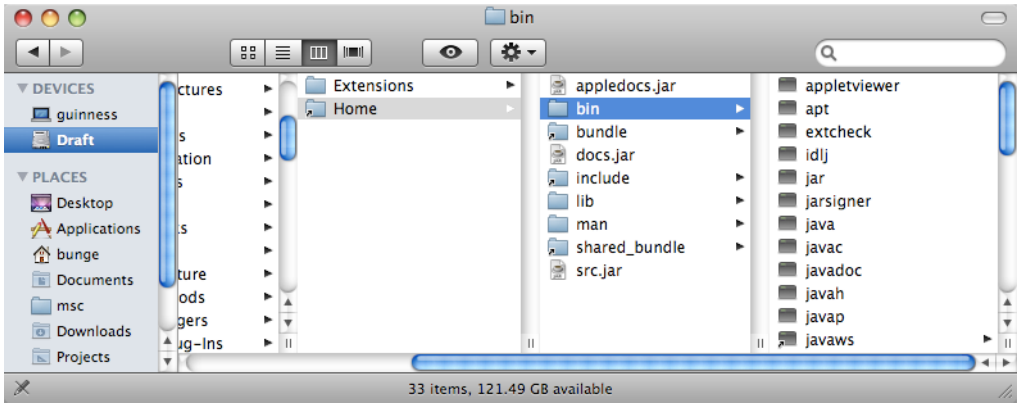


(a) Original

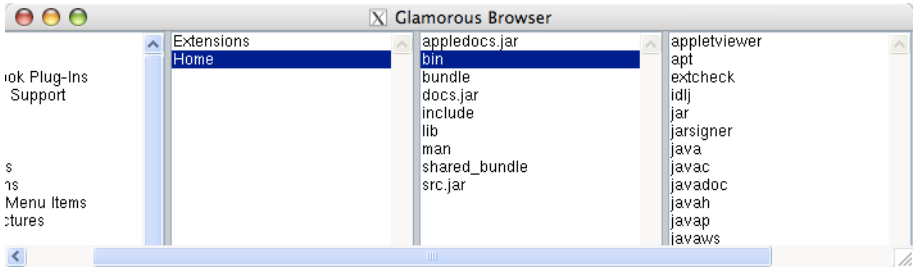


(b) Glamour

Figure 4.1. File Explorer in the original and as a Glamour implementation.



(a) Original

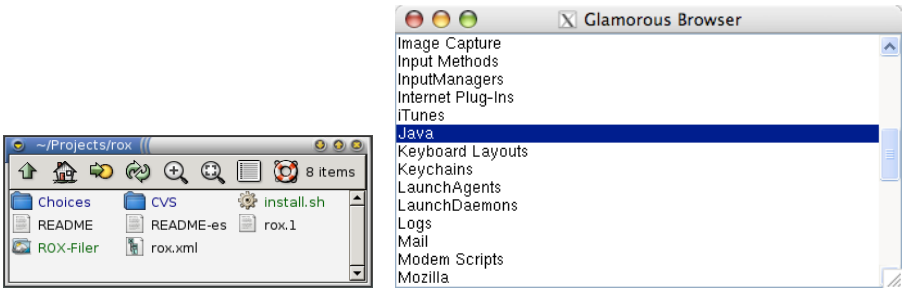


(b) Glamour

Figure 4.2. File finder in the original and as a Glamour implementation.

```
browser list
    display: #files;
    format: #tail;
    when: #isDirectory.
browser text
    display: #yourself;
    when: [ :file | file isDirectory not ].

^ browser
```

Spatial navigation. The final type of file system navigation we demonstrate is that of the *spatial navigation* in which folders are represented as windows, presenting their contents as a list or a grid of files and folders. This model was used in older versions of Windows and OS X and remains popular even today in file managers such as GNOME's *Nautilus*



(a) Original (b) Glamour

Figure 4.3. Spatial file browser in the original and as a Glamour implementation.

or the ROX File Manager. Proponents of this style of file manager stress that the objects on screen have a stronger apparent spatial relationship. The obvious disadvantage is that navigation quickly clutters the screen with a multitude of windows. A possible alternative is to replace the current window—or actually its contents—with the folder into which the user just descended. This is, in fact, a behavior supported by Nautilus as shown in figure 4.3. The corresponding Glamour implementation is also quite simple with around ten lines of code.

```
fileWindow

    | browser |
    browser := TableLayoutBrowser new.
    browser column: #files.

    browser showOn: #files; using: [
        browser list
            display: #files;
            format: #tail.
    ].

    browser showOn: #files; from: #files -> #executed; using: [
        browser list
            display: #files;
            format: #tail.
    ].

    ^ browser
```

The filesystem navigation browsers provide a full coverage of the aspects under which we wish to validate our model. Considering the challenges we enumerated in chapter 1, the filesystem navigation is a good example on how our browser can accommodate arbitrary domain models—there is no need to adapt the filesystem model, we simply define the message sends that should be performed on the model.

Furthermore, the example shows how both the navigation flow and the presentation can be independently customized. The three browsers each use a navigation flow which is quite distinct from one another. The folders are shown both as lists and as tree widgets.

Finally, the filesystem navigation shows how Glamour works at the instance-level of the domain model. We are provided not with distinct types for the folders and files but simply with messages such as `isDirectory` that allow us to infer the type of the object. The browser could also easily be extended with the navigation with other types of files, such as symbolic links and device descriptors in the same manner.

4.2. Source Code Navigation

A second usecase that is extensively shown in this thesis is the Smalltalk style class browser. Figure 4.4 shows the class browser in VisualWorks and a Glamour implementation. An important attribute of our implementation of the class browser is that we are re-using an existing browser component within another browser. The code navigation at the top is encapsulated in a separate browser, allowing the workflow to be reused within other browsers as well.

Additionally to how Glamour promotes the reuse of existing browsers, the source code navigation shows a further mechanism with which our model promotes the flexibility of the presentations. Simple usecases can be accommodated in parallel within one pane by simply defining multiple presentations. The multiple presentations can then show the same data in more than one way, such as—in the example—a list or a hierarchical representation.

The code editor combines the code navigator and a pane with the source code:

```
codeEditor

    | browser |
    browser := TableLayoutBrowser new.
    browser row: #navigation; row: #sourceCode.

    browser showOn: #navigation; using: [
        browser custom: self stCodeNavigator.
    ].
```

```
    browser
        showOn: #sourceCode;
        from: #navigation->#selectedClass;
        from: #navigation->#selectedMethod;
        from: #navigation->#selectedPundle;
        using: [
            browser text
                title: 'Method source';
                display: [ :class :method | class sourceCodeAt: method ];
                when: [ :class :method | class notNil & method notNil ].
                act: [ :presentation :class :method |
                    class installCode: presentation text at: method
                ] on: $s;
            browser text
                title: 'Class definition';
                display: [ :class | class definition ];
                when: [:class | class notNil].
            browser text
                title: 'Class comment';
                display: [ :class | class comment];
                when: [:class | class notNil].
            browser mondrian
                title: 'Pundle System Complexity';
                painting: [ :view :class :method :pundle |
                    view nodes: pundle containedClasses.
                    view edgesFrom: #superclass.
                    view treeLayout. ];
                when: [:class :method :pundle | pundle isPackage ].
        ].

    ^browser
```

The code navigator component used in the browser above:

```
codeNavigator

    | browser |
    browser := TableLayoutBrowser new.
    browser
        column: #pundles;
        column: #classes;
        column: #categories;
```

```
        column: #methods.

    browser showOn: #pundles; using: [
        browser tree
            children: #childPundles
    ].

    browser showOn: #classes; from: #pundles; using: [
        browser list
            display: [ :pundle | pundle containedClasses ];
            when: [ :pundle | pundle isPackage ]
    ].

    browser showOn: #categories; from: #classes; using: [
        browser list
            display: [ :class | class categoryNames ]
    ].

    browser showOn: #methods; from: #classes; from: #categories; using: [
        browser list
            display: [ :class :category | class methodNamesIn: category ].
        browser list
            display: [ :class | class allMethodNames ];
            when: [ :class :category | category isNil ]
    ].

    browser
        sendToOutside: #selectedClass from: #classes -> #selection;
        sendToOutside: #selectedMethod from: #methods -> #selection.

    ^ browser

    | browser |
    browser := TableLayoutBrowser new.
    browser
        column: #pundles;
        column: #classes;
        column: #methods span: 2.

    browser showOn: #pundles; using: [
        browser tree
            children: #childPundles
    ].
```

```
    browser showOn: #classes; from: #pundles; using: [
        browser list
            display: [ :pundle | pundle containedClasses ];
            when: [ :pundle | pundle isPackage ]
    ].

    browser showOn: #methods; from: #classes; using: [
        browser custom: (self stCodeMethods
            display: #yourself;
            title: 'Instance').
        browser custom: (   self stCodeMethods
            display: #class;
            title: 'Class').
    ].

    browser sendToOutside: #selectedPundle from: #pundles.
    browser sendToOutside: #selectedClass from: #classes.
    browser
        sendToOutside: #selectedMethod from: #methods -> #selectedMethod.

    ^browser
```

And finally the method sub-browser included above:

```
codeMessages

    | browser |
    browser := TableLayoutBrowser new.
    browser
        column: #categories;
        column: #methods.

    browser showOn: #categories; using: [
        browser list display: [ :class | class categoryNames ].
    ].

    browser
        showOn: #methods; from: #outer -> #entity; from: #categories;
        using: [
            browser list
                display: [  :class :category |
                    class methodNamesIn: category ].
            browser list
```

```
            display: [ :class :category | class allMethodNames ];
            when: [ :class :category | category isNil ].
].

browser sendToOutside: #selectedMethod from: #methods.

^browser
```

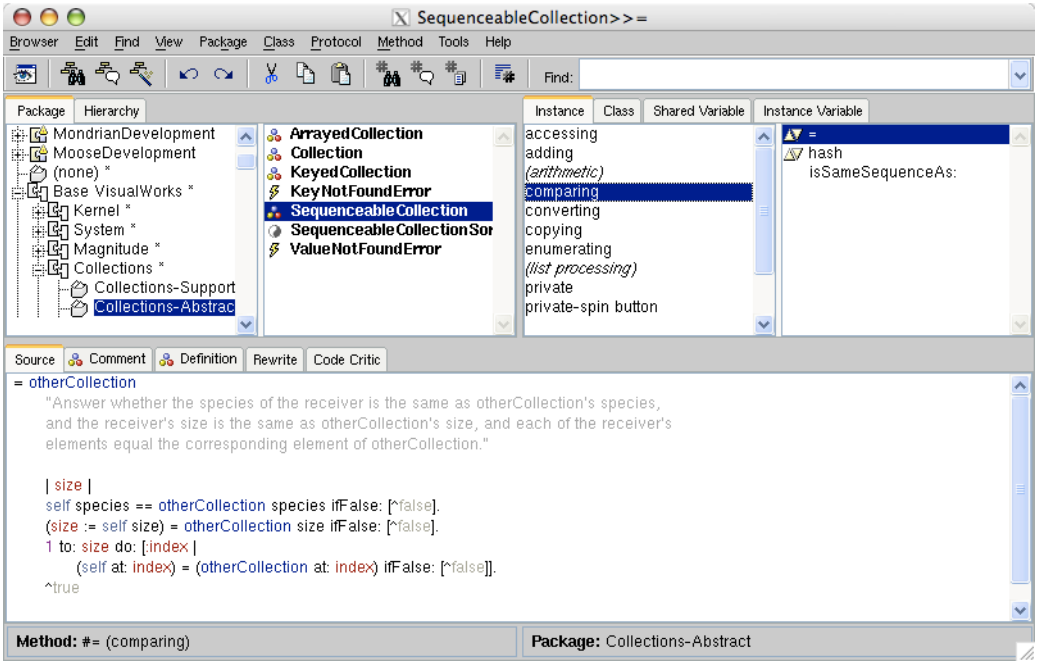
4.3. Software Dependency-Analysis

As a larger use-case we created an imitation of Softwarenaut, a tool used for the top-down exploration of large software systems [Lungu and Lanza, 2006]. Softwarenaut promotes the navigation of such systems by providing three views that interact with each other: the *exploration perspective* shows a graph-like representation of the current focus of interest in the system as nodes, with edges representing the interaction between those nodes; the *detail perspective* provides further information on the currently selected element and the *map perspective* provides a grand overview of the system and highlights the position of the current focus within that system.

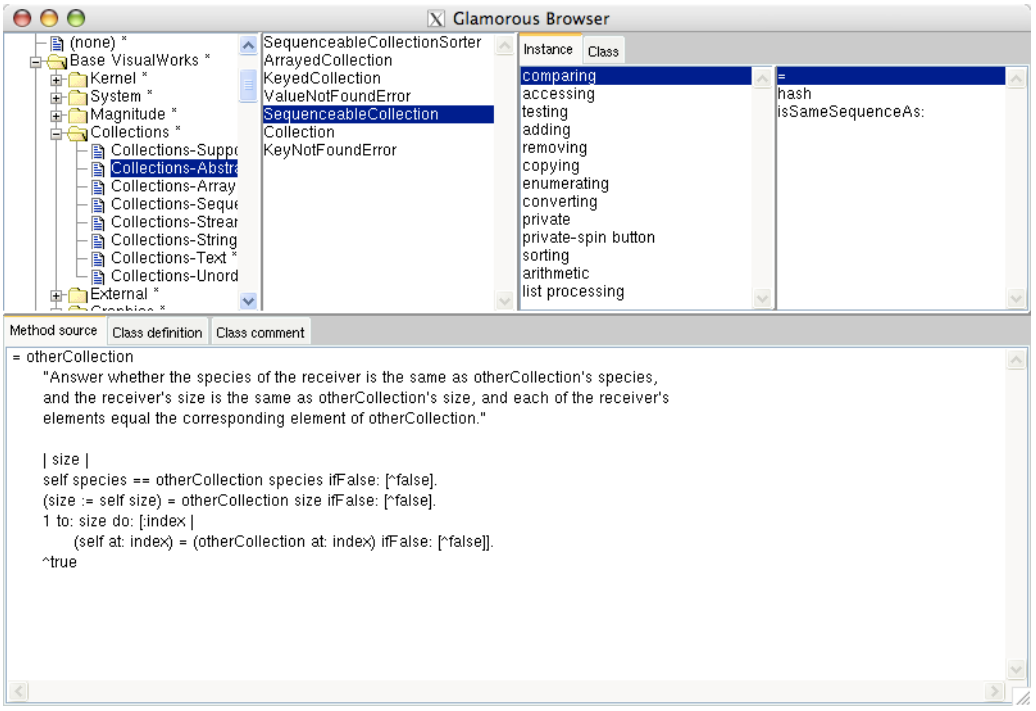
To complement this structure, Softwarenaut provides three "navigation primitives:" *expand* replaces a node with its children; *collapse* removes a node and all its siblings from the view and replaces them with their parent and finally *filter* subselects the nodes according to some arbitrary criterion.

Although the aforementioned navigation operations seem quite complex, this type of navigational flow can quite easily be implemented in Glamour using panes and transmissions. The expand and collapse operations are implemented using transmissions that use the same pane as origin and destination, although the exact port is a different one. The flow of information between the panes is implemented using normal transmissions just like in other browsers. Figure 4.5 shows the original Softwarenaut and the Glamour equivalent.

```
| browser |
browser := Glamour.TableLayoutBrowser new.
browser
    row: [:r | r column: #main span: 3; column: #details];
    row: #overview.
browser showOn: #overview; from: #outer->#entity; from: #main; using: [
    browser mondrian
        painting: [ :view :originalNamespace :inFocus |
            originalNamespace viewSubtreeOn: view
            withSelection: inFocus ].
```



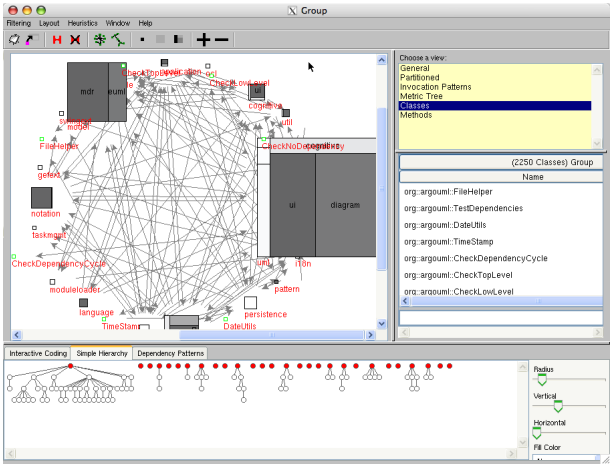
(a) VisualWorks



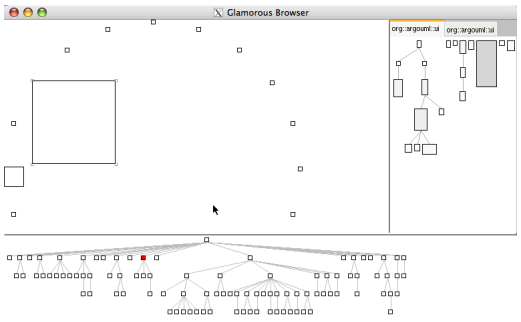
(b) Glamour

Figure 4.4. Smalltalk code browser in VisualWorks and using Glamour.

```
    ].
    browser showOn: #main; from: #outer->#entity; from: #overview; using: [
        browser mondrian
            painting: [ :view :originalNamespace :selectedNamespace |
                selectedNamespace isNil
                    ifTrue: [
                        originalNamespace viewInvocationCircleOn: view]
                    ifFalse: [
                        selectedNamespace viewInvocationCircleOn: view]
            ].
    ].
    browser showOn: #details; from: #main; using: [
        browser mondrian
            painting: [ :view :selectedNamespace |
                selectedNamespace classGroup viewSystemComplexityOn: view.
            ];
            when: [:selectedNamespace |
                selectedNamespace classGroup notEmpty].
        browser list
            display: #classes.
    ].
    ^ browser
```



(a) Original



(b) Glamour

Figure 4.5. Original Softwarenaut and Glamour implementation.

Chapter 5

Related Work

Our problem statement and the design decisions underlying or model have been influenced by a large amount of research which is directly or indirectly related to our work. In this chapter, we compare our approach with some of this related work, showing both differences and similarities that are either intentional or were discovered later.

We only list a small selection of technical research and publications that we have found to have a significant relation to our work. We have divided these into two distinct categories. The first category includes research that has the underlying attempt to *expose domain objects* in order to make them accessible and manipulable by the user as well as proposals that have another similarity with our meta-model in terms of their approaches in user-interface construction. The second category describes work that is concerned with *software composition*, an attribute of Glamour that we have not yet explicitly mentioned in this thesis.

5.1. Exposing Domain Objects

With Glamour, we provide a method to create user interfaces on the basis of domain models. The emphasis lies in providing the users of the browser with direct access to the underlying model, enabling them to navigate and understand as well as to manipulate the model objects.

That objects should be behaviorally complete and expose themselves directly to the user had been an integral concept to object-oriented systems when it was first envisioned. In fact, the design principles in Smalltalk-76 required that all objects be capable of presenting themselves to the user in an effective way [Ingalls, 1978]. This paradigm can be restrictive however, when wishing to display objects differently in individual use-cases. In Glamour we address this issue by providing a lightweight mechanism that allows a domain model to be mapped to a browser. Others have taken different approaches to abstracting user-interfaces from models of which we describe a few in the following sections.

5.1.1. Model-View-Controller

The Model-View-Controller (MVC) scheme was created by Trygve Reenskaug while working with Smalltalk-76 as a visiting scientist at the Xerox Palo Alto Research center between 1978 and 1979 [Reenskaug, 1996a; Reenskaug, 1979]. Reenskaug was attempting to use Smalltalk to create a system for production control in shipbuilding and was confronted with the dilemma that he wished to show objects specific to different contexts such as in an activity diagram or in an object editor. As a result, he started to pull the objects apart in a "model" capturing the attributes and behavior of the domain object, a "view" representing the object and a "controller" that propagated manipulations of the user to the model. This concept was later adopted and extended in Smalltalk-80 [Goldberg and Robson, 1989].

Although Reenskaug intended the structure to be used to augment the user's mental model and enable the user to inspect and manipulate the underlying domain objects, the pattern has become an instrument to shield the user from the model [Reenskaug, 2003]. This is additionally promoted by views being manually constructed rather than having them generated automatically or semi-automatically from the model—a factor that furthermore leads to strong coupling between the model and the view and controller [Buschmann *et al.*, 1996].

As in Reenskaug's original motivation, Glamour is able to present objects differently depending on the use-case. A single object or collection may be shown with different widgets such as lists, trees and other components. Furthermore, the navigational flow between panes can be flexibly defined by the developer of the browser. Since these refinements are declaratively specified, objects are still exposed to the user rather than being heavily shielded as can be the case with a programatically defined view and controller.

5.1.2. Naked Objects

The Naked Objects framework was developed by Richard Pawson to provide a mechanism that allows user interfaces to be automatically generated from a domain model, and directly presents the objects to the user—in other words, they are presented strictly naked [Pawson and Matthews, 2002; Pawson, 2004]. The framework therefore circumvents the coupling issues present with the Model-View-Controller pattern by forcing the view mechanism to be entirely independent of the actual model. Unfortunately, this generality comes at the cost of specifiability, hindering the ability to display objects in a use-case specific fashion as was the original motivation for Trygve Reenskaug to create the Model-View-Controller pattern.

In comparison, our objects can be considered to be "half-clothed." Glamour's presentation mechanism can be refined to allow for use-case specific representation of the objects. Excessive coupling is mitigated by relying only on the model responding to the messages that are defined in the browser scripts. As long as these remain the same, there is no necessity

to adapt the browsers when the model's implementation changes.

5.1.3. OmniBrowser

OmniBrowser is a framework that supports the definition of browsers based on an explicit meta-model [Bergel *et al.*, 2007]. OmniBrowser is implemented in Smalltalk and was conceived to create Smalltalk style browsers, such as the typical code browser, and has been readily adopted for many other such style browsers such as a coverage browser [Bergel *et al.*, 2008] or a browser to cope with scoped changes [Haldimann, 2005].

While Glamour provides a wide range of flow-control, OmniBrowser generally assumes an implicit flow through lists that are constructed in a left-to-right fashion. When an item in a list is selected, the meta-graph is traversed to display a new list to the right. Browsers which require more fine-grained flow control or a different style of navigation are difficult to create with the framework.

Furthermore, OmniBrowser's flow control occurs at the typelevel of the domain model. It is not possible to navigate at the instance level of the underlying objects. This can be an issue when we want more fine-grained control over how we navigate through the domain model as we have discussed in 1.3.

5.1.4. Hopscotch

Hopscotch [Bykov, 2008] is the application framework and development environment of Newspeak, a programming language inspired by Smalltalk and others [Bracha *et al.*, 2008]. As with a web browser, Hopscotch remembers the navigation path and provides a history and a back-button for the user to retrieve his steps. The framework provides a "tool holder" that consists mainly of a mechanism to navigate to a particular "page" or comparable object in a hypermedia fashion. Navigation is achieved by "subjects" that combine a sense of location of a domain object with a particular "viewpoint" reflecting the usecase for the object.

The Hopscotch framework alleviates the issues present in many statically designed browsers by providing a browser model that enforces a specific, dynamic navigational flow. Browsers are arranged and nested in a tree-like fashion. They are registered to a particular subject and display further subjects which, when expanded, open as new browsers. With this type of pattern, Hopscotch works directly on a meta-description (the "subject") which extends the underlying objects with a usecase description (the "viewpoint") and is therefore quite comparable with Glamour's approach of "half-clothed" objects.

The author of Hopscotch also emphasizes that the framework prevents "arbitrary display constraints" as the representation does not impose any size restrictions to the browsers be-

fore they are displayed—they can simply expand to the size they require. While some of Glamour's browsers provide such restrictions, this is a simple issue of presenting the browser in a different manner. The `Finder` already alleviates this partially and a custom browser that imitates the behavior of Hopscotch could be easily written.

Finally, Glamour promotes reuse of browsers by allowing them to be nested within others, very much like Hopscotch allows the nesting of browsers. This encourages developers to build complex browsers from smaller components, effectively discouraging the construction of *monolithic tools*.

It would be interesting to build a Hopscotch-like environment using Glamour, which could be accomplished using a custom `Browser` implementation. This would also provide us with an additional use-ase for the implementation of a history functionality in Glamour as we discuss in 6.5.

5.1.5. Interface Builder

Apple's *Interface Builder* is a graphical interface builder for the OS X operating system. It is the current instance of the *NeXT Interface Builder* [Webster, 1989], originally developed as the *SOS Interface* by Jean-Marie Hullot at INRIA [Hullot, 1986].

Interface Builder strongly encourages a Model-View-Controller separation of concerns. It allows the developer to drag and drop components to a user interface to compose the user interface and then link the components to one another and to the controller using *outlet* and *action* connections. For example, so that a controller may access the widgets of a view, it defines a set of named outlets. The interface designer can then interactively connect the two by dragging a connection between the widget and the controller's outlet. Out of the perspective of the controller, the field representing the outlet simply contains a reference to the widget. The same applies for actions.

Actions allow a widget to call a controller's method when it is acted upon by a user. For this, the controller designates a callback method as an action. The interface designer can then interactively hook up a widget by dragging a connection from the button or comparable widget to the action.

The Interface Builder has recently been extended for use with mobile devices. Additionally to the introduction of *event* connections to support multi-touch gestures, Apple also provides *view controllers* for implementing navigational patterns that are optimized for small screens. The view controllers usually show one pane at a time which may be switched either by a *radio toggle* of alternative views, by a *navigational* drill-down mechanism that provides a back button, or with a modal overlay. The patterns may be combined to provide more complex user interfaces. As with Glamour, the flow is defined by connecting the views with one another using connections within a particular view controller.

The advantage of such a visual application builder is that it allows rapid development of graphical interfaces and an immediate visual representation of what the user interface will look like at runtime. Aside from that however, it provides no real additional benefits to a programatic or scripting based approach. In comparison to Glamour, the framework provides no mechanism with which the view can be defined generically. Furthermore, the rendering of the user interface and the flow control are still entangled.

5.1.6. Mondrian

Mondrian is a domain independent, composable and declarative visualization framework created by Michael Meyer [Meyer *et al.*, 2006; Meyer, 2006]. Although more specific to the domain of *visualization* rather than browsers or user-interfaces, the experiences with Mondrian have inspired many of the design decisions in Glamour. Like our approach, Mondrian does not assume any specific data structure of the model being studied. The only requirement is that the data can be queried by a simple meta-model. The meta-model is defined through a simple declarative language implemented in Smalltalk and is later rendered to an on-screen interactive graphic or to a different medium.

Mondrian provides a comprehensive set of default shapes and behavior, easing the complication of constructing visualizations. Visualizations can therefore be iteratively refined to produce more specific presentations if so needed. This property has been shown to enable the rapid prototyping of research tools and to improve development [Lienhard *et al.*, 2007]. These results also motivated the provision of comprehensive default behavior in Glamour, in an attempt to make it *easy* for researchers and developers to create browsers to understand and communicate their domain models.

5.1.7. ApplFLab

ApplFLab is a *reflective application builder* with an emphasis on promoting the reuse of components [Steyaert *et al.*, 1996]. The framework addresses the issue that, while most visual application builders are more flexible in comparison to domain-specific component-oriented development environments, the latter allow for more rapid application programming and are more end-user oriented. Steyaert *et al.* proposed a solution that makes application building itself the problem domain, leading to a reflective application builder—i.e. an application builder used to build application builders. Reuse of components is achieved by making them parameterizable, allowing the parameters to be bound to the domain model when the components are integrated in an application. ApplFLab was used to create several domain-specific user interfaces including code and pattern browsers for software development [Wuyts, 1996].

An important difference between ApplFLab and Glamour is that Glamour provides a more comprehensive set of default behaviors. While in ApplFLab creating a simple application is

quite costly with the construction of new components and the flow of information between these, several browser patterns are already provided by Glamour. Of course, ApplFLab is not specialized on browsers as is Glamour and therefore supports a broader range of user-interface applications.

5.1.8. Django Admin Site

Web development frameworks occasionally provide an automatically generated administrative interface for the creation and maintenance of the web application's model. A well known example is the the admin site of *Django*, a Python based rapid web development framework. Django's administrative interface inspects the application's declared persistent fields and creates appropriate user interface components for each. Since the types of the fields are declared, Django can display custom widgets for each and provide input validation. Model fields that define database-style multiplicities between relations provide the interconnection of the instances on the admin site.

Customizability is achieved by linking each domain model with a custom `ModelAdmin` class that allows the developer to declaratively customize its visible fields, appearance properties and navigation to other objects. Additional widgets can be defined and integrated into the administrative interface. Django's admin site is a specialized and simple solution for web applications. It does not support flow control as Glamour does and the decision on how to present the objects is defined at the type-level.

5.1.9. ThingLab II

ThingLab II is a re-implementation of ThingLab [Borning, 1981], an object-oriented constraint-system built on top of Smalltalk [Freeman-Benson, 1989]. ThingLab promotes the direct manipulation of objects by exposing *Things*, user-manipulable objects that are equivalent to the operations and data structures provided by high-level languages such as numerical operations, points, strings, bitmaps, conversions, etc. Constraints can be defined between Things, and the composition of constraints and Things again constitute higher level Things. Browser navigation can be defined by employing constraints between the individual components that constitute the browser. The use of constraints is a declarative description of the flow of information in the browser but in ThingLab there is no explicit distinction between the domain and its navigation.

Glamour's browser composition strongly resembles that of ThingLab, but ThingLabs's component connections employ a very different operation than those of Glamour. While ThingLab describes generic constraints that exist between components and the constraint mechanism assures that these are fulfilled, Glamour describes the flow between panes explicitly.

5.1.10. HyperCard

HyperCard is a hypermedia application which conceptually resembles a stack of cards [Goodman, 1998]. A card contains texts and graphics and links to other cards through buttons, which typically carry an icon representing the destination card. Only one card is displayed at a time—clicking on the button navigates to the linked card. HyperCard provides a set of simple components that can be manipulated interactively and, alternatively, cards may be manipulated through the dedicated scripting language HyperTalk. HyperCard does not provide means to represent and interact with an existing model, the domain is mixed in with the navigation.

A browser that imitates the behavior of HyperCard could be implemented with Glamour as well. Implicit browsers such as the `Finder` already implement a comparable behavior, except that they show the complete history of visited "cards" rather than just the current. The rendering could be adapted to display only the latest pane at a particular time. In contrast to HyperCard, Glamour provides no mechanism to interactively manipulate the cards. Presentations are usually modified via the scripting language at design time.

5.1.11. Curry

Hanus and Kluß have proposed a declarative description for user-interfaces that builds on the functional logic language *Curry* [Hanus and Kluß, 2009; Hanus, 1997]. Their approach is based on the separation of the structural, functional and layout aspects of the a user interface. Like Glamour, their approach allows for user interfaces to be described independently of the representation. The authors have implemented a renderer for desktop user interfaces as well as for web user interfaces using a combination of standard HTML components and asynchronous Javascript for communication with the server. Unlike Glamour, their approach serves to define user-interfaces in general and does not include default navigation flow mechanisms.

5.2. Software Composition

An interesting parallel can be drawn between the *classification model* and our work [Wuyts and Ducasse, 2004]. The classification model is a lightweight mechanism to combine tools that were not meant to be integrated with one another. The model was used to create *StarBrowser*—a browser to integrate various tools in Smalltalk.

StarBrowser manages to consolidate tools by wrapping each as a *service*, which provides an interface to process any type of *item* that it is passed. An *item* is anything that has the notion of an *object* from a software development perspective such as a class, a method, an image, *etc.* Items can further be statically or dynamically collected within *classifications*

which are themselves items. When a service receives a request to display a certain *item*, it translates that request in a manner that is understandable to the underlying tool and delegates the request to it. If the service has no specialized behavior for the particular item, it simply performs a default action. In Glamour, the presentations have a similar role to that of the *services* in the classification model—they wrap the underlying components to provide a consistent interface to the composition model. There is no restriction in Glamour's presentations that prevent them to wrap more complex tools in addition to the generic components that are currently implemented.

In contrast to Glamour, the StarBrowser has a static configuration of the flow of information between the integrated components. StarBrowser displays two panes where the left pane shows a tree of available classifications and the right pane provides space for the tools opened on these classifications. New classifications can be added by dragging items out of the displayed tools.

The integration of components with one another to promote reuse is of particular focus of *component-oriented* architectures and composition frameworks. Component oriented software architectures allow for systems to evolve by viewing applications as compositions of reusable and configurable components [Nierstrasz and Dami, 1995]. To be able to integrate a component in a composition system, the component must be designed to be able to collaborate together with others. This means, that a component is usually not designed in isolation but as part of a framework of collaborating components. Schneider and Nierstrasz state that in order for components to be plugged together successfully

> it is necessary that i) the interface of each component matches the expectations of the other components and ii) that the "contracts" between the components are well-defined. [Schneider and Nierstrasz, 1999, pg. 3]

In cases like in the StarBrowser, where incompatible tools should be integrated with one another, we need to adapt the interfaces to receive components with compatible interfaces. StarBrowser's *service* wrappers are typical example of such *glue* code. Similarly, Glamour accommodates components using *presentations* which can be interacted with using the primitive port-based interface.

Glamour uses a declarative scripting language to construct browsers from individual parts. It has been argued that scripting languages are richly suited for assembling applications from individual components [Ousterhout, 1998]. *Piccola* is a language based on a formal calculus [Milner, 1991] that has taken this concept a step further by providing an entire composition system based on components and frameworks, software architectures, glue, and scripting [Lumpe, 1999; Achermann and Nierstrasz, 2001]. In contrast, Glamour is by no means designed to be a comprehensive composition language. Our port-based communication only allows for very loose contracts to be defined between components. Further yet, our presentations—which form the *glue* to adapt existing components—need to be programatically defined.

Chapter 6

Conclusions

To conclude or work, we revisit our original goals and assess how our model was developed to achieve these. We offer a short discussion on three aspects of our work that have not yet been discussed: we show how our model defines a new paradigm for constructing browsers, we discuss the advantages and disadvantages of using an internal scripting language, and we discuss our motivation for the abstract browser notation. Finally, we introduce some prospective future work for Glamour.

6.1. Our Goals Revisited

The goal of this thesis was to develop a model with which dedicated browsers that accommodate arbitrary data can be easily expressed. In the introduction we described four requirements that our framework needs to fulfill. We revisit them here and determine in which ways our model meets these goals.

The browser should accommodate arbitrary domain models. Our model imposes no restrictions on the underlying model that should be displayed in a browser apart from that it may be queried by simple message sends. No specific classes need to be implemented and which messages are sent are specified through the use of block closures by the browser's developer, who has sufficient knowledge of the domain model.

The navigation flow needs to be completely controllable. The flow of information is completely determined by the transmissions between panes, which can be freely specified by the developer. Some browsers provide an implicit flow—such as the finder—but the type of browser and therefore the flow is also specified by the developer.

The presentation should be flexible. As with the flow of information, the developer needs to be able to define how a particular domain object should be displayed. With Glamour, presentations can be defined that show an object as a list, a tree, a graphical visualization or a wide range of other representations. In addition, objects can be displayed using nested browsers that provide their own flow.

The browser framework should work at the instance level of the domain model. Developers define closures that are evaluated with the current domain instance in order to make flow and presentation decisions. This allows the the flow and presentation decisions to be customized for a particular instance instead of relying on the type of the object.

6.2. Flow based Browsers vs. Side-Effect based Browsers

Glamour was not designed to be able to declaratively describe *all* possible browsers. Much more, Glamour provides an alternative paradigm for describing browsers on the basis of *navigational flow* rather than programatically defined behavior.

This is apparent when trying to imitate the behavior of certain existing browsers. For example, the integrated development environment *Eclipse* presents the structure of the source code in a software project using a tree widget, showing the structure of projects, packages, files, classes and methods. When the user double clicks on an item, a new tab is shown on the right hand side with the file in which the item is contained. Tabs always represent on distinct file and no file appears twice. Also, tabs are only removed when the user explicitly closes them.

The opening of files in an Eclipse-style browser could still be implemented using normal transmissions in Glamour but transmissions *replace* the value at the designated port rather than appending to it. Using the default behavior, such user interface side effects could therefore not be implemented using Glamour. A solution would be to implement a custom browser in Glamour that supports this behavior, much in the way that the *finder* provides a custom browser.

We believe that our paradigm is not in itself a restriction but that it requires developers to think differently about how browsers are constructed. It is still open to further research to investigate which paradigm more closely follows the user's mental model of the domain being investigated and whether Glamour promotes browsers that are more intuitive to use than browsers that have implicit side-effects.

6.3. Declarative Scripting Language

The current declarative language that we provide with Glamour is introduced in the tutorial (chapter 2) and further described in section 3.7. The language is an internal declarative language that builds on top of Smalltalk. Rather than having the declarative language construct an intermediate representation which is then transformed into a browser model, the language directly manipulates the model. The language is implemented as a set of class extensions to the model and we make heavy use of Smalltalk specific features such as block closures and cascades.

While most of the extensions simply call the programatic interfaces of the model, there are some that require state. This is particularly the case when we construct transmissions—the `showOn:` message constructs a transmission that is then populated in the subsequent `using:` block. This transmission needs to be temporarily stored somewhere for which we need an instance variable. This—however—breaks the encapsulation of the model as we are adding additional state to it.

A solution would be to provide a *builder* that contains the script state and constructs the browser. This would also prevent the scripting methods from cluttering the model interface. Such an external builder could also be combined with an *external scripting language*. An advantage of developing an external scripting language would be that we could create a language that intently matches the developer's mental model of the browser he or she is creating. A dedicated declarative script would also have the advantage that it is not strictly bound to the grammar and semantics of the underlying language and therefore could be ported to other platforms as well.

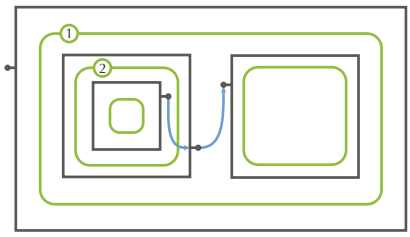
A clear disadvantage of such an external language is the added complexity by requiring an explicit grammar and a dedicated parser and by introducing a new language that the developer must learn, just to be able to define the browsers.

6.4. Browser Notation

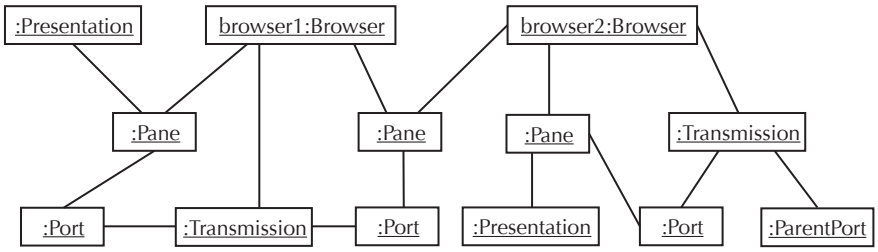
As introduced in chapter 3, we have developed an abstract notation in the course of our work in order to describe and understand more complex browsers—especially ones that consist of multiple, nested browsers. While the notation evolved as a natural way to describe our browsers on a whiteboard or one paper, the question arises if there even is a need for a new type of notation or if a representation such as with UML object diagrams would not suffice [Object Management Group, 2007].

In order to justify our notation, consider the browser constructed in the tutorial which is presented in abstract notation in figure 3.2. We have represented a simplified subset of this browser in our abstract notation and as a UML object diagram in 6.1. The browsers consists

of an outer browser (1) with two panes of which the first contains yet another browser (2) and the second just a simple presentation. The inner browser contains just a single pane with a simple presentation. It also exports a port on this pane to a port on its parent pane, which the outer browser connects to its second pane using a simple transmission.



(a) Abstract Notation



(b) UML Object diagram

Figure 6.1. A nested browser in our abstract notation and as a UML object diagram.

The browser described above is quite simple, but we can immediately recognize how complex describing browsers in UML may become. Although the object diagram purveys a better notion of the types involved in the browser model, the structure of the browser is immediately recognizable from our notation in a manner that cannot be displayed by UML, even if we had found a better layout. This property confirms the necessity of our notation, which facilitates the discussion and reasoning about our browser models.

A reference of our notation can be found in appendix B.

6.5. Future Work

One aspect which certainly should be followed and would provide an interesting research topic is the capturing of the history of the state of the browsers. This would permit the implementation of a "back button" allowing users to retrieve their steps, but possibly also a

simpler method to enter the browser at a specified state. A possibility to implement such history capturing would be to record to the triggering of the transmissions (see section 3.1) and then replaying them or iterating them in reverse direction.

The question remains how sideeffects incited by actions can be captured with such a model.

Further related work includes the exploration of different scripting languages to define browsers and the extension of supported presentations and browser models.

6.6. Concluding Remarks

The success of Glamour will be determined by its adoption and advancement. Already, developers have started using and enhancing our model for purposes outside of this thesis. Their creativity and feedback will provide further insight on the capabilities and possible issues with Glamour and will further the development of our approach.

Appendix A

Installation

At the point of writing, two implementations of Glamour exist—one for VisualWorks Smalltalk and another for Pharo, created by Tudor Gîrba, Lukas Renggli and Daniel Röthlisberger. This appendix provides a brief introduction on how Glamour can be acquired and installed.

A.1. Glamour for VisualWorks Smalltalk

VisualWorks is a commercial Smalltalk product produced by Cincom and can be obtained as a non-commercial version for private use from the company's website.[1] Currently, Glamour is developed against VisualWorks 7.6 and will probably work most reliably with that version.

After downloading and installing the environment according to the accompanying instructions, we need to connect to the *store* of the Software Composition Group at the University of Bern. To do so, select "Store" then "Connect to Repository" (see figure A.1) from the VisualWorks main window and enter the following information:

Interface: PostgreSQLEXDIConnection
Environment: db.iam.unibe.ch:5432_scgStore
User: storeguest
Password: storeguest

You can save the store as "SCG" or under any name that you like.

Once you have successfully connected to the repository, select "Store" then "Published Items" and search for Glamour by typing in the word under Bundles and Packages.

Select the Glamour bundle in the list and then right click and select "load" on the top most version on the right hand side as shown in figure A.2. The necessary dependencies will be

[1] `http://www.cincomsmalltalk.com/`

loaded for you.

Finally, you will also need to load a *renderer implementation* so you can actually display the browsers on-screen. We recommend "Glamour-Widgetry" which uses the Widgetry user-interface components. Simply load the corresponding package as you did for the core Glamour bundle above.

A.2. Glamour for Pharo

Pharo is a fork of the Squeak open-source Smalltalk platform and is a port of our VisualWorks reference implementation. The source base is mainly the same, with the exceptions that Pharo currently has no notion of namespaces so all class names contain a prefix with the string "GLM", and that the implementation contains a different set of renderers.

To get started with Glamour for Pharo, start by downloading Pharo from the project's website.[2]

To load Glamour, start a Pharo image and click on the background to select the "Monticello Browser" from the world menu as shown in figure A.3.

Click the "+Repository" button and then on "HTTP" to add a new HTTP-based repository and then add the following information:

```
MCHttpRepository
    location: 'http://www.squeaksource.com/Glamour'
    user: ''
    password: ''
```

Click "OK" to save the repository and return to the main Monticello screen where you may click on "Open" to open the repository. Load Glamour by selecting "Glamour-All" and then the newest version and finally clicking the "Load" button as shown in figure A.4.

Finally, you will need to load a renderer to be able to display your browsers. From the same repository, load either "Glamour-Morphic" or "Glamour-Seaside." The latter requires a full installation of Seaside 2.9.

[2] `http://pharo-project.org/download`

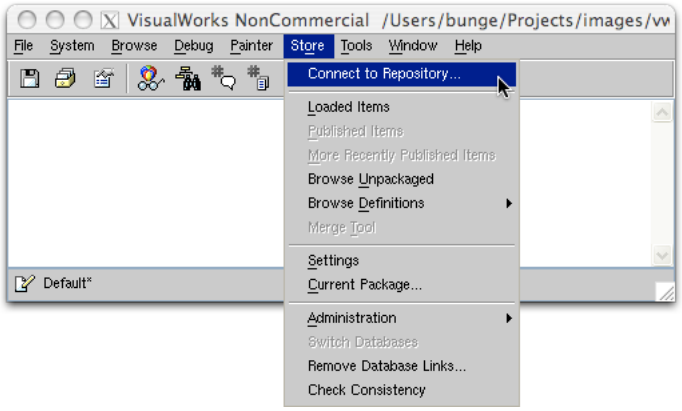


Figure A.1. Adding a store account in VisualWorks.

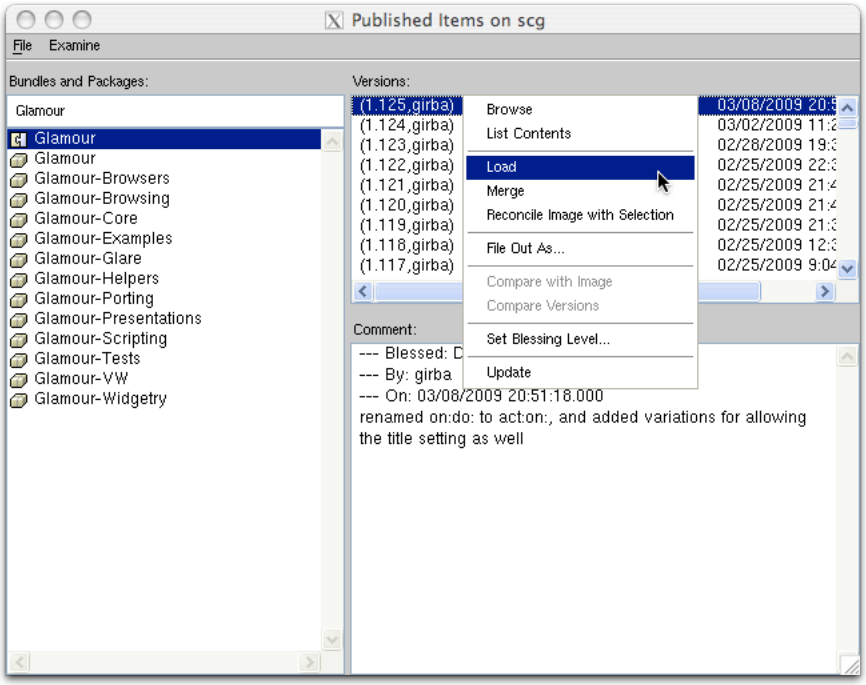


Figure A.2. Loading a package in VisualWorks.

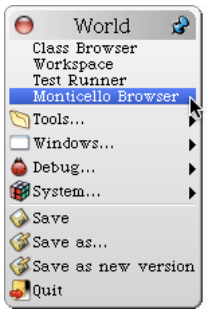


Figure A.3. Starting the Monticello Browser.

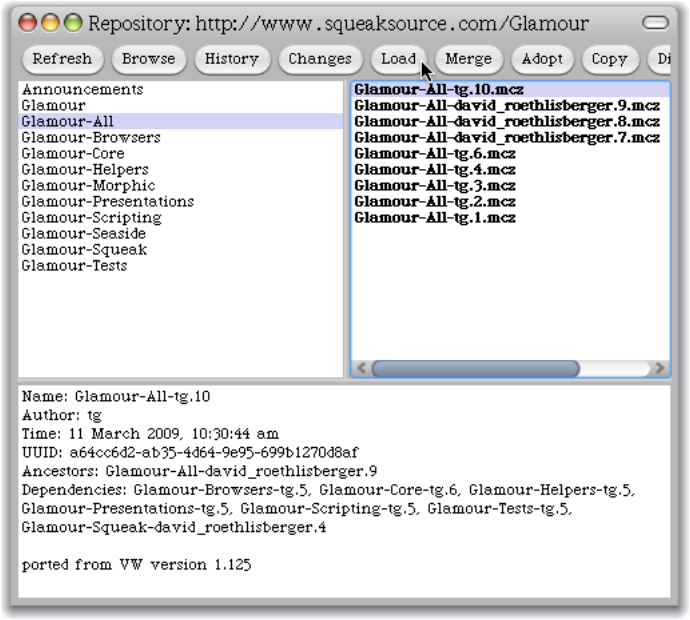


Figure A.4. Loading Glamour in Pharo.

Appendix B

Browser Notation

As an alternative to UML object diagrams we have developed an abstract notation to describe the current state of a browser. The abstract notation improves the understanding of the browser by providing a spatial notion on the positioning of the components of the browser as it would be rendered. This chapter provides a quick reference to our notation.

Panes. Panes are represented as gray rectangles with optional ports marked as "lollipops" on the outside of the panes.

Figure B.1. Graphical representation of a pane in our abstract notation.

Presentations. Presentations are represented as green rounded rectangles and are displayed within the pane to which they are applied. A presentation's pane can be omitted if it is the outermost pane and if it adds no value to the diagram.

Figure B.2. Graphical representation of a presentation in our abstract notation.

Browsers. Browsers are presentations and are represented as such. Contained panes are drawn within the presentation's figure. Panes need only be drawn if they are relevant to the information the diagram is trying to purvey. This means that there is no way to distinguish between a browser and a primitive presentation in a diagram if no panes are drawn inside the browser.

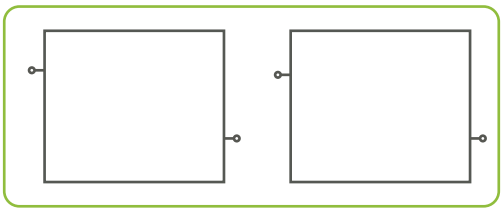


Figure B.3. A browser containing two panes in abstract notation.

Transmissions. The transmissions between panes are drawn as blue directed arrows connecting the ports to which they are connected. Bundle transmissions which have more than one origin have more than one tail but only one arrow head. Transmission are generally contained within the browser that manages them. The only transmissions that cross the browser boundary are port accesses to the browser's outer pane as in the case of port forwarding and capturing. In the case of transmissions which are not bound to a pane, the origin of the transmission should attempt to best convey the spatial location of the origin. In the case where a presentation sets a port value due to interaction with the user, the transmission should originate at the corresponding presentation.

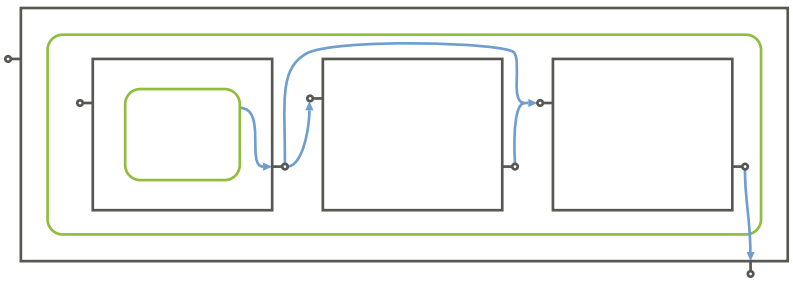


Figure B.4. A browser with various types of transmissions in abstract notation.

Comments. Comments are displayed using the standard UML style note symbol and dotted line.

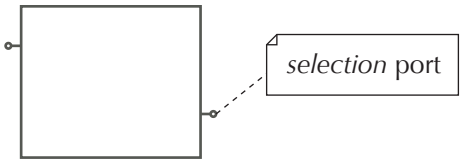


Figure B.5. A comment in abstract notation.

List of Figures

Bibliography

[Achermann and Nierstrasz, 2001] Franz Achermann and Oscar Nierstrasz. Applications = components + scripts — a tour of Piccola. In Mehmet Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[Adobe, 2006] Adobe. Flex 2 technical overview. Technical report, Adobe, 2006. `http://www.adobe.com/products/flex/whitepapers/pdfs/flex2wp_technicaloverview.pdf`.

[Bergel *et al.*, 2007] Alexandre Bergel, Stéphane Ducasse, Colin Putney, and Roel Wuyts. Meta-driven browsers. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, volume 4406 of *LNCS*, pages 134–156. Springer, August 2007.

[Bergel *et al.*, 2008] Alexandre Bergel, Stéphane Ducasse, Colin Putney, and Roel Wuyts. Creating sophisticated development tools with OmniBrowser. *Journal of Computer Languages, Systems and Structures*, 34(2-3):109–129, 2008.

[Borning, 1981] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM TOPLAS*, 3(4):353–387, October 1981.

[Bracha *et al.*, 2008] Gilad Bracha, Peter Ahe, Vassili Bykov, Yaron Kashai, and Eliot Miranda. The Newspeak programming platform. `http://bracha.org/newspeak.pdf`, May 2008.

[Bunge *et al.*, 2008] Philipp Bunge, Tudor Gîrba, and Lukas Renggli. Glare UI — flashing user-interface with Smalltalk. European Smalltalk User Group Innovation Technology Award, August 2008.

[Buschmann *et al.*, 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley & Sons Ltd., West Sussex PO19 1UD, England, 1996.

[Bykov, 2008] Vassili Bykov. Hopscotch: Towards user interface composition. In *International Workshop on Advanced Software Development Tools and Techniques (WasDeTT)*, July 2008.

[Dahl and Nygaard, 1966] Ole-Johan Dahl and Kristen Nygaard. SIMULA: an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[de Mey, 1995] Vicki de Mey. Visual composition of software applications. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 10, pages 275–303. Prentice-Hall, 1995.

[Demeyer *et al.*, 2008] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2008.

[Ducasse *et al.*, 2005] Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Moose: an agile reengineering environment. In *Proceedings of ESEC/FSE 2005*, pages 99–102, September 2005. Tool demo.

[Engelbart, 1962] Douglas C. Engelbart. Augmenting human intellect: A conceptual framework. Technical report, Stanford Research Institute, Menlo Park, CA 94025, October 1962.

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

[Fowler, 2007] Martin Fowler. Domain specific languages, November 2007. `http://martinfowler.com/dslwip/`, Work In Progress.

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

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[Goldberg and Robson, 1989] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

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

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

[Hanus and Kluß, 2009] Michael Hanus and Christof Kluß. Declarative programming of user interfaces. In Andy Gill and Terrance Swift, editors, *Practical Aspects of Declarative Languages*, volume 5418 of *LNCS*, pages 16–30, Berlin Heidelberg, January 2009. Springer-Verlag.

[Hanus, 1997] Michael Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997.

[Hornby, 2000] Albert Sydney Hornby. *Oxford Advanced Learner's Dictionary*. Oxford University Press, sixth edition, 2000.

[Hudak, 1998] Paul Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[Hullot, 1986] Jean-Marie Hullot. SOS Interface: un generateur d'interfaces homme-machine. In *Actes des journees AFCET sur les Langages Orientes Objets*, 1986.

[Ingalls, 1978] Daniel H.H. Ingalls. The Smalltalk-76 programming system design and implementation. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 9–16, New York, NY, USA, 1978. ACM.

[Lanza, 2004] Michele Lanza. Codecrawler — polymetric views in action. In *Proceedings of ASE 2004 (19th IEEE International Conference on Automated Software Engineering)*, pages 394–395. IEEE CS Press, 2004.

[Lienhard *et al.*, 2007] Adrian Lienhard, Adrian Kuhn, and Orla Greevy. Rapid prototyping of visualizations using mondrian. In *Proceedings IEEE International Workshop on Visualizing Software for Understanding (Vissoft'07)*, pages 67–70, Los Alamitos, CA, USA, June 2007. IEEE Computer Society.

[Lumpe, 1999] Markus Lumpe. *A Pi-Calculus Based Approach to Software Composition*. Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, January 1999.

[Lungu and Lanza, 2006] Mircea Lungu and Michele Lanza. Softwarenaut: Exploring hierarchical system decompositions. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages 351–354, Los Alamitos CA, 2006. IEEE Computer Society Press.

[Meyer *et al.*, 2006] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[Meyer, 2006] Michael Meyer. Scripting interactive visualizations. Master's thesis, University of Bern, November 2006.

[Milner, 1991] Robin Milner. The polyadic π-calculus: a tutorial. ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh, October 1991.

[Nierstrasz and Dami, 1995] Oscar Nierstrasz and Laurent Dami. Component-oriented software technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice-Hall, 1995.

[Object Management Group, 2007] Object Management Group. Unified modeling language superstructure v2.1.2. Technical report, Object Management Group, November 2007.

[Ousterhout, 1998] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[Pawson and Matthews, 2002] Richard Pawson and Robert Matthews. *Naked Objects*. Wiley and Sons, 2002.

[Pawson, 2004] Richard Pawson. *Naked Objects*. Ph.D. thesis, Trinity College, Dublin, 2004.

[Reenskaug, 1979] Trygve M. H. Reenskaug. Models - views - controllers, December 1979.

[Reenskaug, 1996a] Trygve Reenskaug. *Working with Objects — The OOram Software Engineering Method*. Manning Publications, 1996.

[Reenskaug, 1996b] Trygve M. H. Reenskaug. Working with objects in the user interfaces. *ObjectEXPERT*, July 1996.

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

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

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

[Schneider and Nierstrasz, 1999] Jean-Guy Schneider and Oscar Nierstrasz. Components, scripts and glue. In Leonor Barroca, Jon Hall, and Patrick Hall, editors, *Software Architectures — Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

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

[Way, 2005] Doug Way. Whisker: The O-O stacking browser, December 2005. `http://www.mindspring.com/~dway/smalltalk/whisker.html`.

[Webster, 1989] Bruce F. Webster. *The NeXT book*. Addison-Wesley, 1989.

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

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

0x4c