Hopscotch: Towards User Interface Composition

Vassili Bykov
Cadence Design Systems, 2655 Seely Ave, San Jose, CA 95134

Abstract

Hopscotch is the application framework and development environment of Newspeak, a new programming language and platform inspired by Smalltalk, Self and Beta. Hopscotch avoids a number of design limitations and shortcomings of traditional UIs and UI frameworks by favoring an interaction model and implementing a framework architecture that enable composition of interfaces. This paper discusses the deficiencies of the traditional approach, provides an overview of the Hopscotch alternative and analyses how it improves upon the status quo.

1. Introduction

Newspeak [1] is a programming language and platform being developed by a team led by Gilad Bracha at Cadence Design Systems, Inc. The system is currently hosted inside Squeak [2], a descendent of Smalltalk-80 [3]. Smalltalk, along with Self [4] and Beta [5], is also among the most important influences on the design of the Newspeak language.

This paper focuses on Hopscotch, the application framework of Newspeak and the Newspeak IDE implemented concurrently with the framework. The framework and the IDE allow easy navigation through the complex information space of Newspeak, achieve usability improvements over the traditional Smalltalk tools as found in Squeak, and support easy creation of new tools by composing existing ones.

In the development of interactive systems, solid technical foundations and attention to usability issues are equally important contributors to final success. Developers that neglect one of these areas are likely to find that problems in the neglected area adversely affect not only the overall result, but also the progress in their area of focus. We believe that the positive results we have had building and using Hopscotch are attributable to its revised interaction and navigation model as much as to the underlying framework architecture. For this reason, this paper pays equal attention to the compositional framework architecture and to the changes in the interaction model of Hopscotch compared to more traditional development environments.

2. Traditional Interface Construction

> The art of devising forms to be filled in depends on three elements: obscurity, lack of space, and the heaviest penalties for failure.
>
> C. Northcote Parkinson

The traditional approach to user interface construction is based on the metaphor of a *form*[1]. A window is subdivided into areas, each occupied by a *widget* (called a *control* or a *view* by some frameworks). A widget is a reusable component displaying a simple generic data object like a string or a list of strings in a certain visual form such as a static label, an editable text area or a list box. A widget can also recognize user input gestures and translate them into actions on the data such as text editing. While a widget can become quite specialized (for ex-

Email address: vassili@cadence.com (Vassili Bykov).

[1] The commonly referred to "desktop metaphor" concerns user interaction rather than UI construction.

ample, Microsoft Windows provides a built-in control for IP addresses), interfaces constructed is this manner share the following important common characteristics.

- A widget is generic, in that it presents data of basic commonly used types such as strings or numbers.
- Complex objects are presented by breaking the information into pieces simple enough to be handled by generic widgets. An application UI is constructed by combining a number of such widgets inside a window.
- The layout of a window is often defined with the aid of a GUI builder and stored as a generated method or as an XML document. Some toolkits such as Apple Cocoa store layouts as binary data in a proprietary format.
- Different perspectives of looking at information space are usually provided by different tools with different layouts.

An interesting analogy is to compare widgets of an application framework to symbols of an alphabet, and application UIs to words over the alphabet composed by developers to express the interaction needs of their application. This analogy is quite illuminating in pointing out the weak point of this approach. The "expressive power" of an application, that is the ability to present information in a variety of ways ideally suited to the needs of the user, is limited within this approach to a rather small set of pre-composed "interaction words."

For example, the classic Smalltalk-80 environment [6] included a SystemBrowser for browsing the source of the entire system, a Class Hierarchy Browser for browsing the source in the hierarchy of one class, an Inspector, a Debugger, and a handful of ancillary tools. Each implements a particular perspective of viewing the information space, with navigation relying on opening a new window for each change of perspective. Any perspective not included in this scheme, for example viewing all example methods in the system grouped by class, would require building a specialized tool.

In more modern environments, embellishments such as movable view splitters and dockable panels may camouflage the underlying rigidity of such form-based UIs, and a good tools framework may ease the burden of creating a new tool. However, they still suffer from the following inherent problems of the form metaphor:

- Domain structure replication.
- Information fragmentation.
- Arbitrary display constraints.
- Monolithic tools.

Let us consider these in detail.

2.1. *Domain Structure Replication*

Complex domains have a hierarchical or directed graph-like structure. Presenting it in a truly graphical form typically involves considerable development effort and places high demand on screen real estate. In practice, the structure of a complex domain is often presented using list boxes. With this approach, a list displays the arcs originating in a graph node, and the application is programmed to respond to selection in the list by updating the UI to visualize the arc's target. Multiple list boxes can be arranged one after another to descend down multiple levels of a hierarchy, as was first done in the Smalltalk-76 browser [7]. In its classic form originating in Smalltalk-80 and repeated in nearly every Smalltalk implementation since, the browser has four list views side-by-side across the top of the window. Together they visualize the four levels of arcs leading to the four layers of code objects in the system: class categories, classes, method categories ("protocols") and methods.

This approach is appealing because of is its low implementation cost, as list box is an easy-to-use widget readily available in all widget frameworks. Its implication, however, is that the structure of the domain is replicated and hard-coded in the structure of the tool window. Such UI assumptions place gratuitous constraints on the domain object model. For example, VisualWorks 5 introduced class shared variables [8]. Because, like methods, they belong to classes, the user is required to group them in categories in order to fit their presentation in the browser. In practice categorization of variables turned out to be superfluous, with many classes having at most one or two of them filed under a non-descript category named "shareds" or "data".

Two solutions appeared over the years to relax this constraint: tree views and columnar lists. Tree views are used in Windows Explorer and the current VisualWorks browser. The most popular example of a columnar list is the column view of OS X Finder. In the IDE space, the TrailBlazer browser [9] of IBM Smalltalk uses this device.

These solutions, however, do not come without their own costs. Tree views have long been known to present cognitive difficulties for nonprogrammer users [10]. While programmers often consider them

natural [11], studies have shown that tree view user efficiency drops as the hierarchy deepens, falling behind the "drill-down" navigation style after 3 hierarchy levels [12]. Columnar list, while free from these deficiencies, complicates application logic with heterogeneous list management. If a hierarchy level presented by a list includes arcs leading to objects of different nature (for example, the VisualWorks SystemBrowser lists classes and namespaces together), application logic needs to account for all of these possibilities as it responds to user selections in the list. Depending on the nature of the object selected, different responses are appropriate. This problem can be held under control with proper object-oriented design, but the very need to control it illustrates the implementation and maintenance overhead introduced by heterogeneous lists.

2.2. *Information Fragmentation*

Directly related to domain structure replication is the problem of fragmented information. A tool such as the Smalltalk browser designed to present information in a highly structured form easily induces "tunnel vision", so that logically related entities such as closely related methods cannot be seen simultaneously. Thus, information structure replicated by the tool, while perfectly logical, gets in the way of user productivity.

A related problem is modality. An attempt to edit a method places the Smalltalk browser in editing mode, with navigation disabled in order to keep the browser focused on the method. This compounds the problem of fragmentation, making it impossible while editing a method to glance at a related one without using a separate browser. Such modality is encouraged by the apparent logic of keeping the tool "focused" on the change in progress. Modeless solutions, such as the author's DontModeMeIn add-on to the VisualWorks browser [13], are possible but require additional implementation effort. Thus, UIs presenting a hierarchical domain through reusable widgets in a form-like arrangement appear to encourage modes by their very nature.

This tunnel vision and modality are likely to be among the real reasons, besides the obvious inertia, why many users prefer plain source file view to structured editors that are literally based on the Smalltalk browser model.

2.3. *Arbitrary Display Constraints*

The smallest box of a form is sometimes the one in which the most information need be provided. As an electronic embodiment of the same idea, UI forms do not escape this paradox. In the author's image, the class category list of the Squeak browser open at its default size includes 476 items, only 14 of which (2.9%) are visible at a time. Expanding the window to full screen makes the bottom text view occupy half the screen to show a method that's typically only a few lines long. Even for large methods (relatively rare in Smalltalk), the proportions of this extremely wide but not very tall text view are exactly the opposite of the optimal.

The cause of this problem is simple. The primary factor that determines the *ideal* size of a widget is the content it displays at any given moment. The primary factors that dictate the *actual* size of a widget in a form are the size of the window and its overall layout including the sizes of other widgets it contains. Clearly, the two are not at all related. The ideal size is dictated from the inside out, the actual—from the outside in. At best, the UI designer can *assume* that a given widget *typically* needs more room than another and allocate the space accordingly.

Again, various mechanisms appeared over the years to mitigate the problem. Common examples are movable view splitters, tabs and and collapsible and expandable dockable panels. They still do not address the cause of the problem—the fact that window layout is determined and fixed in advance without regard to the display needs of the actual content.

2.4. *Monolithic Tools*

Easier reuse is an important promise of object-oriented programming. UI logic often makes up the bulk of an application, and ease of UI reuse is an attractive proposition. After model-view-controller and its refinement as pluggable views in ObjectWorks 4, perhaps the most important step along this path was the *model-view-presenter* pattern first created by Taligent [14] and reintroduced into the Smalltalk world by Dolphin Smalltalk [15].

That architecture indeed makes it easier to reuse the logic behind the UI. However, the layout of the UI is harder to reuse as a whole for many of the same reasons as discussed above. While it is easy from the programmatic point of view to incorporate

a portion of the UI serviced by an application inside another, at the interface design level such embedding often doesn't work well. The available space may be too small, or the embedded application may require menu bar items that should now be massaged into the menu bar of the host application, or the visual style of the embedded application may conflict with the guidelines of the host.

3. The Hopscotch Approach

The motivation for trying a different approach in the application framework and the IDE for Newspeak was two-fold. Obviously, we wanted a framework that would make platform users more productive. More immediately, there was a practical need for a Newspeak-aware browser. The Newspeak language introduces nested classes, and the classic Smalltalk browser with its expectations of a four-level hierarchy cannot be successfully adapted to Newspeak.

3.1. *Goals, Some Influences and Parallels*

To summarize our goal as an antithesis to the problems just discussed, we were seeking an interaction model and a framework architecture to allow:

- Easy navigation in a complex information space.
- Ability to integrate diverse tools into a seamless user experience.
- Integrated and modeless presentation of information.
- Easy creation of new tools by composing them fully or partially from already existing tools.

Tools and interaction models that support the first two goals have long existed—they are web browsers. It is important to realize what it is in particular that makes a web browser successful as the vehicle of the world wide web.

A browser is a reusable tool that can be an encyclopedia one minute and a mail client the next precisely because it is not a tool at all. It is a tool holder that takes care of switching the tools it contains on demand following a simple and predictable navigation model.

Not less importantly, the tools, or pages, can be so diverse because they typically follow the metaphor of *document* rather than form. Unlike a form that effectively *is* the window that displays it, a document is contained inside a window but as a rule is not fully constrained by it. Given that the form metaphor is at the root of problems ranging from suboptimal screen real estate allocation to obstacles to interface reuse and composition, the document metaphor is an attractive alternative.

Reassuringly, the document metaphor has already once been used for structured code presentation in the browser of Strongtalk [16]. Despite its lack of consistent navigation model, the tendency to open too many windows and lack of simple tool composition capabilities, the Strongtalk browser shows that the document metaphor is indeed a context where the problems of information fragmentation and gratuitous modes can be solved.

It is important to emphasize that the web browser and the document metaphor were chosen as an example of an attractive interaction model, but not as the technological foundation. Ultimately, the goal was to build a Newspeak-native framework that would include as its core players composable interactive tool objects whose navigation scheme would resemble the universally familiar web browser [2].

As will be seen in the architecture overview section, the objects that implement individual Hopscotch tools are called presenters. The architecture is indeed an extension of the MVP idea.

3.2. *Interaction Model*

Because of space constraints only one illustration is provided in this section. The tools and interaction will be demonstrated as part of the workshop presentation.

To the user, a Hopscotch window [3] looks very much like that of a web browser. There is a pair of buttons with back and forward arrows in the top left corner, a history button, a home button and a few others. An interaction session begins at the home page. In the current Hopscotch incarnation, the home page includes a number of predefined navigational links, the two most important of them to "System Source" and "Source Control".

Like in a web browser, clicking hyperlinks is the primary means of navigation. Auxiliary navigation mechanisms are the forward/back buttons and the

[2] The Hopscotch design would, in fact, allow running it in a web browser. This at the moment is an unused feature of the design and not a goal in itself.

[3] Because they were created simultaneously, "Hopscotch" refers to both the application framework and the Newspeak IDE built using it. The intended meaning is usually clear from the context.

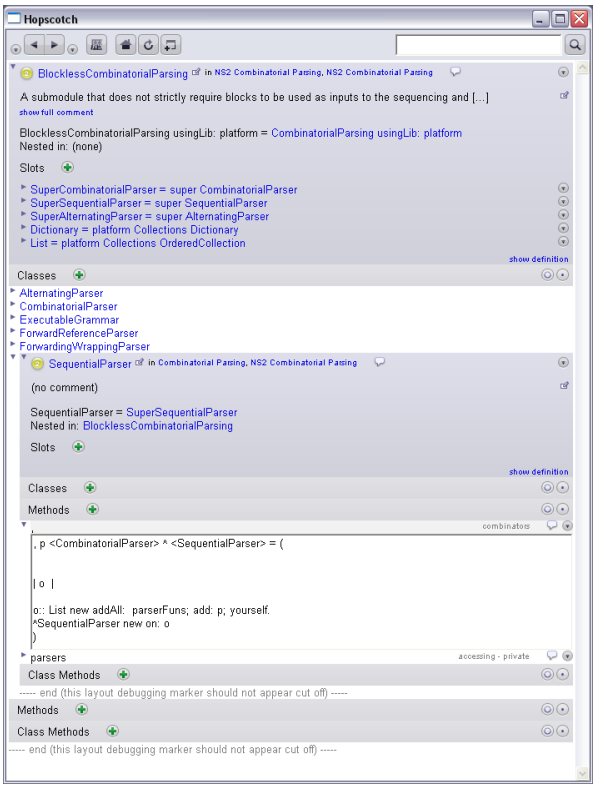

Fig. 1. Hopscotch view of a Newspeak class.

history page that lists all places visited in the same browser.

A class is presented as a header summarizing important class information such as the name, the containing category and package (both are links that can be clicked to go "out" into the enclosing context), and the comment. The header is followed by the view of nested classes and methods. Each method is an interactive element that switches between a collapsed state showing only the method name and an expanded state displaying full method source. Initially all methods are collapsed. Above the lists are buttons that allow expanding or collapsing all elements at once. Expanding all methods turns the view into a flat source view similar to that of a flat file editor (but aware of the displayed structure).

In a similar manner, class categories are presented as lists of class names with additional class statistics such as the number of methods. Each name is a link leading to the full class presentation. A class can also be expanded in-place into its full presentation. In that embedded presentation methods can also be expanded as described above, which makes it possible, if desired, to produce a flat view of full source of all classes of the entire class category.

Typing inside a method view marks the view as containing unsaved changes, with a bright orange bar showing "accept" and "cancel" buttons and making unsaved changes easy to spot. *Editing a method is not a mode.* It is possible at any moment to navigate away to look at another class or senders or implementors of a message and then use the "back" button or the history page to come back and continue editing. It is also possible to edit multiple methods at once, be they in the same class or in different ones.

Importantly, Hopscotch provides no menu bar with actions that would operate on methods, classes or any other objects it presents. A method presentation itself includes elements for interacting with the method. There is a menu of cross-references (associated with the "speech bubble" icon in method and class headings), allowing to see all senders and implementors of the messages sent or implemented by the method. There is also a menu of actions (associated with the "down triangle" icon) supporting various manipulations such as deleting the method. Each method (or any other object supporting actions) has its own action menu button. The menu operates on the object whose presentation contains the menu.

In contrast to the common model of a menu bar providing all possible actions, this approach makes it intuitively obvious in accordance with the Gestalt principle which object an action affects. It also avoids another vestige of modes in traditional UIs: menu items mysteriously disabled because they are not applicable to the currently selected object. (Which illustrates that in traditional UIs the concept of current selection or other UI state often introduce modes). In the Hopscotch model, the action menu of each object only includes the actions the object supports.

In fact, with the adoption of the document metaphor and the dismissal of the menu bar, the concept of selection at the level higher than text editing becomes less prominent and the UI less stateful. A class view, for example, has no currently selected element of any kind. It is simply a collection of visual artifacts mapped onto the collection of program objects they represent. Operating on a visual artifact operates on the program object—direct manipulation at its purest.

The information space available for navigation in Hopscotch is not limited to classes. The Source Control tool is implemented as part of the same frame-

work, appearing as a page listing Monticello [17] packages categorized by their status (modified, updated on the server, etc) and expandable to reveal details such as recent version history.

The framework also includes an inspector. In fact, the class browser and the inspector are intertwined. The action menu of a class includes an "Inspect Class" action that navigates to the inspector on the class object, and each inspector includes a link navigating to a browser on the class of the inspected object, all within the same Hopscotch window. This shows seamless navigational integration of what were considered to be distinct tools. Integration can be even more direct. In an inspector, each slot of an object can be expanded in place into a full inspector on the value of the slot. In a similar fashion, a class nested inside another can be expanded in place in the Newspeak class browser view as shown in the screenshot on the previous page. Getting a little ahead of the presentation, it is also entirely possible to embed a browser into an inspector view, or an inspector into a browser with a trivial one-line code change.

3.3. *Architecture Overview*

Hopscotch may appear similar to a web application. This resemblance is natural considering the deliberate choice of a web browser-like navigation model, however it is also skin-deep. Hopscotch is not an HTML renderer implemented in Newspeak, it is an application framework supporting a wide array of interaction models other than the favored hypertext-like one.

Hopscotch tools ("pages") are displayed inside an outer navigator window. The navigator window is a generic document holder. The navigator is usually unaware of the purpose and function of the tools it contains, however for a particular application it can be customized with domain-specific tools. For example, the window used for browsing Newspeak and Smalltalk code has a Search box in the tool bar to search for packages, classes and methods.

The basic unit of the framework is a presenter. A presenter is responsible for displaying and manipulating a domain object in a particular way. A presenter may contain other presenters (child presenters), and may or may not be contained inside a higher-level presenter (parent presenter). A navigator window is considered to be the parent of the topmost ("page-level") presenter.

A presenter creates and controls widgets such as text labels, editors, buttons, hyperlinks, or graphical images required to present its model. The widget system used by Hopscotch is Brazil, a cross-platform native widget framework developed for and in Newspeak. As of this writing, Brazil works with the Morphic and native Windows APIs. Future support is planned for some Unix-based UI systems.

Unlike the traditional MVP architecture, a Hopscotch presenter does not interact directly with model objects. Instead, it does that through a *subject*. A subject plays several roles in the framework.

First of all, it is a *location marker* identifying the domain object being presented. Continuing the web browser analogy, we could say that by representing a "place" in the information space, a subject is similar to a URL. Indeed, subjects are used by the framework as addresses: when the user clicks a link or initiates navigation in some other way, the request the navigator receives holds an instance of the subject identifying the destination. The subject is then used by the navigator to manufacture the appropriate presenter.

Besides identifying the location, a subject can also identify the *viewpoint* used. For example, a ClassSubject represents the full view of a class with its methods and other details. In contrast, a ClassInheritanceSubject represents a view of a class as part of the hierarchy that includes its superclasses and subclasses. Together, the location and the viewpoint determine "the subject of the presentation," hence the term.

Indirection achieved by using subjects rather than presenters as navigation targets is important in separating the specification of navigation from a particular policy of displaying domain objects. In different usage contexts with different display constraints, different presenters could be used for a particular subject. For example, a ClassSubject can have two presenters, one of them optimized for smaller screens to include less information. At the same time, tools can request navigation by specifying only the target subject without regard to how the subject would be presented in the current system context.

The subject can also play the role of a convenient stand-in for a domain object in situations when there isn't one in the domain. For example, in the traditional Smalltalk design there is no object that corresponds to a particular class category. Instead, category members are retrieved from a singleton categorizer object using a key. In this situation, a ClassCategorySubject would encapsulate that key and

maintain the illusion of a real domain object for the rest of the Hopscotch framework.

The final role a subject can play is to serve as a utility kit providing methods that help presenters extract and process domain data.

A presenter defines the particular presentation of its subject. It does that, however, not by directly instantiating and configuring visuals (widgets), but in abstract form as a definition method composed of combinators. Because of limited space, we will consider only one simple example of a presenter for a subject that represents a "dual" view of a class, both as a class metaobject and as a class browser. The following definition will present it as a heading with the name of the class followed by an inspector and a browser [4]:

```
definition = (
    ^column: {
        majorHeading: (label: subject model name).
        include: (Inspector on: subject model).
        include: (Browser on: subject model).
    }
)
```

The `label:` message takes care of creating a label with the specified text. The `majorHeading:` message then wraps it with additional components to format it to prominently stand out (more on that later). The `column:` combinator finally arranges the header and the presentations of the two subjects, one representing a view of the subject's model (the class metaobject) as a plain object (displaying slots and their values), the other—as a class (displaying methods and other class details). `Inspector` and `Browser`, in Newspeak semantics, are also message sends to the receiver—they are not the `Inspector` and `Browser` classes defined in Smalltalk. We assume that arrangements have been made in the module containing this definition to bind these names to the appropriate subject classes.

Such indirect specification of the interface—in what is essentially a domain-specific language—by sending messages to the (implicit) receiver leaves significant freedom in its interpretation. For example, the specific visual appearance of the major heading in the above definition is left up to the receiver to define. The language mechanisms of Newspeak provide many interesting possibilities.

The receiver may receive a definition of the `majorHeading:` method from a mixin together with other similar "formatting combinators". Thus, mixins can play the role of style sheets imported into presenter classes to define their appearance.

Alternatively, `majorHeading:` may be a method defined by an outer class, such as the module. In this case the message send in the presenter definition is a Newspeak outer message send. With this scheme, a module can centrally control the appearance of all presenters it contains.

As yet another possibility, the handling of `majorHeading:` may be implemented so that the receiver walks up the chain of its parents searching for a handler. With this "dynamic" policy, a presenter handling this message would impose a particular formatting style on all of its children. In our review of obstacles to interface composition within the traditional UI approach, we mentioned as one such obstacle the fact that the appearance of a child is hard-wired in its UI and may clash with that of the parent. In contrast, this example shows how in our architecture we are able to late-bind the determination of the particulars of a child layout, enabling the parent to impose a common visual style on its children.

To show some of the additional capabilities of our scheme, here is how we can change the definition above to make the browser view initially hidden and expandable upon request.

```
definition = (
    ^column: {
        majorHeading: (label: subject model name).
        include: (Inspector on: subject model).
        heading: (label: 'Browser') details:
            [include: (Browser on: subject model)].
    }
)
```

This definition can be further refactored for readability, capitalizing on the fact that it is written in plain Newspeak and not in any specialized markup language.

```
definition = (
    ^column: {
        majorHeading: (label: subject model name).
        expandable: Browser named: 'Browser'.
        expandable: Inspector named: 'Inspector'.
    }
```

[4] The Hopscotch combinator language is far from being final. This listing only serves to illustrate the general idea, while the particular combinators and their names are subject to change.

```
)

expandable: subjectClass named: title = (
    ^heading: (label: title)
     details: [subjectClass on: subject model]
)
```

By now, the approach may begin to resemble text typesetting in TEX. This resemblance is more than a mere coincidence.

The tree structure of presenters and their containers is very important in establishing the context for any nested tool. There is a facility for observer pattern-like "indirect sends" over this tree. For example, this facility enables a presenter to request its containers to scroll or expand to ensure that the requestor is visible.

Another important use of this facility is the mechanism behind hyperlink navigation. A presenter with responds to a link click by sending a navigation request up the presenter hierarchy as an indirect send. Thus, interpretation of a hyperlink response lies within a particular presenter and is not hard-wired into the framework.

The presenters we have considered so far have the form of a document composed vertically from the top down—essentially, a column. This is so because the resulting document-like shape of presenters easily lends itself to composition, and has so far been enough to serve our needs. The framework itself does not require a presenter to have any particular shape. A presenter can be defined in the shape of a diagram, or a document with an attached outline-style view for easy scrolling, or even one that looks like the classic Smalltalk browser.

To summarize, the particular Hopscotch variation of MVP could be expressed as mSP(v). The subject and the presenter are much more important players, as far as programmer effort is concerned, than the others. The model is at least partially shielded from application code by the subject. The views (widgets) are also somewhat decoupled from application code by the combinator language used to write presenter definitions and play a less important role than they do in traditional MVP.

4. Conclusions: UI Composition Instead of UI Building

As already mentioned, Hopscotch-the-IDE was the immediate motivation for creating Hopscotch-the-framework. Work on both began in early August 2007. By mid-September 2007 the IDE had enough features for the author to abandon the traditional browser and use Hopscotch for all further development work. By the end of 2007 Hopscotch was used by the entire Newspeak team and included browsers for Smalltalk and Newspeak classes, package and category presenters, a search facility, a selector senders/implementors presenter, a message inheritance presenter, an inspector, a Brazil visual hierarchy explorer, and a prototype debugger.

This progress in such a short time, in parallel with active framework development that so far included two complete redesigns, demonstrates the high productivity of compositional UI construction approach as compared to the traditional one. In this section we will review the problems of the traditional approach and how Hopscotch avoids them.

Domain structure replication. Hopscotch presenters do not have rigid structure created in a GUI builder or otherwise, with a fixed number of widgets constrained to fit a rectangle of a particular size. Instead of making assumptions about the structure of domain objects, a Hopscotch presenter can adapt to whatever structure the model has at the moment.

Information fragmentation. Because presenters are easily composable, it is easy to define new views to bring together information that previously was not available in one place. The approach also makes it easy to link related views for quick navigation.

Arbitrary display constraints. The document-like style of presenters and their composition ensures that presenter UI is not constrained by the window. Thus, a presenter can indeed size itself "from the inside out" to best accommodate its current content.

Monolithic tools. Within the Hopscotch approach, the only way to create an interface is to create a presenter. A presenter is guaranteed to be composable with others by virtue of being a presenter. Creating a presenter in Hopscotch feels less like building a tool and more like defining a picture of an object, expecting that it can later be part of a larger picture still.

Last but not least, presenters are defined and composed within a powerful general-purpose language. Even the simple example above is enough to show how Newspeak, thanks largely to its late-bound message send semantics and lack of the boilerplate of explicit self sends, can be a flexible and controllable user interface definition language obviating the need to work with specialized structural markup and style sheet languages at the framework user level.

5. Future Work

Our experience with this model so far has been enough to appreciate both the attractiveness and the vast scope of the subject of interface composition. Two areas in particular appear to be very interesting to explore in future framework development.

One is describing and composing the semantics of nested presenters. We briefly mentioned the significance of the presenter nesting hierarchy and the existing "indirect send" facility. More experience programming in this environment may point out common patterns that can be captured in a composition mechanism that would make interface composition even easier.

The other is a facility to automatically manage changes in the structure of domain objects, making presenter definitions fully declarative. At the moment, a presenter definition is used as an initial executable specification evaluated at the time a presenter is created. Some presenters display objects whose structure may change (for example, methods may be added to a class), requiring changes in the structure of nested presenters. At the moment such presenters explicitly mutate their structure to match the new state. Thus, their structure is effectively specified twice, once in the definition, then again in the mutation logic. Such duplication is inelegant and invites problems, especially in its highly imperative mutating part. Re-evaluating the definition in a new context and automatically "massaging" existing presenters and views into the new shape is a more attractive proposition. It is complicated by the fact that the UI is inherently stateful, and useful state such as unsaved edits and the collapsed/expanded state of items needs to be preserved. However, we believe that it is a solvable problem, and solving it will further simplify the use of the framework.

On the application side of Hopscotch-the-IDE, the new freedom of composing tools and combining information opens many interesting opportunities. They are too early to speculate about before prototyping and testing.

As for a GUI builder—it is considered harmful.

6. Acknowledgments

The author would like to thank Peter Ahé, Gilad Bracha and Eliot Miranda for their interest, support and encouragement since the early days of the framework development.

References

[1] Gilad Bracha, *The Newspeak Programming Language*, talk at HPI Potsdam, 2008. Available at `http://www.tele-task.de/page50_lecture3490.html`

[2] Dan Ingalls *et al*, *Back to the future: the story of Squeak, a practical Smalltalk written in itself*, ACM SIGPLAN Notices, Volume 32, Issue 10, 1997.

[3] A Goldberg and D Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[4] David Ungar, Randall B Smith, *Self: The power of simplicity*, Conference on Object Oriented Programming Systems Languages and Applications, 1987.

[5] Ole Lehrmann Madsen, Birger Mller-Pedersen, Kristen Nygaard, *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993.

[6] A Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, 1983.

[7] Larry Tesler, *The Smalltalk Environment*, BYTE magazine, August 1981.

[8] *VisualWorks Application Developer's Guide*, p. 81, Cincom Systems, 2002.

[9] *IBM Smalltalk User's Guide*, ch. 12, IBM Corp., 2002. The TrailBlazer browser is created by Chris Gerken.

[10] Alan Cooper, Robert M Reimann *About Face 2.0: The Essentials of Interaction Design*, Wiley, 2003.

[11] Jiri Kopsa, Jakub Franc, *Usability Study Report: NetBeans Enterprise Pack 5.5*, ui.netbeans.org, 2006.

[12] Panayiotis Zaphiris, Ben Schneiderman, Kent L Norman, *Expandable Indexes Versus Sequential Menus for Searching Hierarchies on the World Wide Web*, International Journal of Human Computer Studies, 2002.

[13] Vassili Bykov, *Don't Mode Me In*, blog article, 2007. Available at `http://www.cincomsmalltalk.com/userblogs/vbykov/blogView?entry=3347470901`

[14] Mike Potel, *MVP: Model-View-Presenter, The Taligent Programming Model for C++ and Java*, 1996. Available at `http://www.wildcrest.com/Potel/Portfolio/mvp.pdf`

[15] Andy Bower and Blair McGlashan, *Twisting the Triad, The evolution of the Dolphin Smalltalk MVP application framework*, ESUG tutorial, 2000.

[16] Gilad Bracha, David Griswold, *Strongtalk: Typechecking Smalltalk in a Production Environment*, Proceedings of the OOPSLA'93 Conference on Object-oriented Programming Systems, Languages and Applications, 1993.

[17] Avi Bryant, Colin Putney, *Monticello User Manual*. Web publication. Available at `http://wiresong.ca/Monticello/UserManual`