DylanWorks Browsers Design

Scott McKay & Chris Fry

1. Introduction

This paper discusses a set of design goals and principles for browsers, describes an abstract model that can be used to escribe a broad class of browsers, and lists a set of browsing "tools" provided by DylanWorks. It also describes how the various browsers can interact with each other in a tightly integrated way. Finally, some specific architectures for browsing substrates are described in the appendices.

1.1 Browsing "Tools"

In DylanWorks, there are several important browsing "tools" that will be implemented on top of the browsing substrate, which provides data generation and filtering functionality, as well as basic data display functionality.

- Class browser.
- Generic function and method browser.
- Caller/callee browser.
- Module, including "apropos" facilities.
- Project/library/source browser.
- Thread and process browser.
- An "inspector".

Other tools may also use the browsing substrate, such as:

- Stack backtraces in the debugger.
- One or more "profiling" browsers.

1.2 Basic Strategy

Note that there are is a spectrum of basic strategies for building all these tools. At one end, we can build a uniform browser that can accommodate numerous kinds of "views" that default to different settings based on the type of data being browsed. At the other end, we can create a series of specialized browsers, each of which is tailored to a particular task. We plan to use a hybrid strategy, in which we build a set of "embeddable" tools which can be combined into a uniform browser, but which can also be used in a standalone frame.

2. Goals and Principles

2.1 Design Goals for a Browser

Here are the goals that our architecture must meet:

1. It needs to provide the ability to manipulate, that is, inspect and modify, a (potentially very large) body of heterogeneous data. It should be possible to manipulate this data in multiple ways using multiple views.
2. It needs to be able to reduce large amounts of *data* into a manageable amount of useful *information*. The user should be able to see a lot of data at once, and in what context the data is used. The user should have a good deal of control over this process.
3. It needs to assist users in developing a coherent and accurate model of the data.
4. It needs to assist users in developing a coherent and accurate model of how the browser itself operates.
5. The computer should do most of the mechanical work.

2.2 Overall Design Principles

1. Anything "of interest" that is displayed should be manipulable. (This contributes mainly to Goal 1 above.)
2. "Less interesting" data should be hidden or otherwise deemphasized, and "more interesting" data should draw the users attention. (This contributes mainly to Goal 2 above.)
3. The UI of the browser needs to assist people in walking through datasets. "Hot spots" should be obvious. Things should not appear to happen by magic. (This contributes mainly to Goal 3 above.)

2.3 UI Design Principles

1. It needs to conform to platform-specific UI guidelines wherever possible. Any contravention of these guidelines needs to have obvious benefit, or it must be an obvious extension of the existing guidelines. (This contributes to Goal 4 above.)
2. The usual other UI design principles apply: self-consistent, self-documenting, context-sensitive help, accessible documentation, and so forth. (This contributes to Goal 4 above.)
3. As far as possible, one "mental operation" should correspond to one UI operation. This suggests that there should be a reasonably fine granularity for the initial set of UI operations, and there should be a way to group common UI idioms in macros (and the most important of these should be pre-packaged). (This contributes to Goal 5 above.)

2.4 Additional Claims

1. Overloading too many operations in a single gesture might be convenient, but it impedes the ability of a user to model how the browser actually works. It's better to have slightly simpler gestures that make "important" operations explicit, because having to explicitly perform an operation will "stick" in the user's head. Of course, you can't choose the granularity too finely, or else the UI is a pain to use.
2. Grouping unrelated data together (by not providing good visual separation, for example) breeds confusion. Failure to group related data can also breed confusion.

3. An Abstract Model for a Browser

The functionality of any browser can be described in terms of three major (and orthogonal) axes of functionality: data generation, data display, and real estate management. There are some minor axes as well, which we will also describe. The separation of these three axes is meant to allow multiple views and interfaces to be built easily.

Along the way, we will give examples of various things that can be called "browsers". Note that the term browser is not intended to connote a tool that is capable only of examining data; we use the word to mean something that can modify data as well.

3.1 Data Generation

The first major axis is *data generation*. For any piece of data, there is a protocol that describes how to get to related data. This protocol also includes filtering and sorting.

As an example, let's look at a browser for "static" data. Here are some types of data and generation, sorting, and filtering operations that apply to the data:

Object	Attributes	Filters
Project	Libraries	
	Owner	
	Source "containers"	
	Compiler warnings	Filter based on fixed/unfixed
		Sort based on definition, type
Library	Uses	
	Clients	
	Modules (internal and exported)	Filter based on substring matching
	Source code	
Module	Uses	
	Clients	
	Variables (internal and exported)	Filter based on matching substring
	Source code	

Object	Attributes	Filters
Variable	Definitions	
	References (caller)	Filter based on library/module
		Sort based on caller types
Class	Subclass	Filter based on library/module
	Superclass	Filter based on library/module
	Slots (direct and indirect)	
	Methods (direct and indirect)	Filter based on library/module
	Source code	
Slot	Getter and setter	
	Type and allocation	
	Initarg, init value or init function	
Generic	Methods	
	Parameters and values	
	Source code	
Method	Parameters and values	
	Specializers	
	Callees	
	Source code	
Object	Class	
	Slot name and value	

The data generation for a debugger would include methods that generate stack traces, and so on.

The data generation for a traditional editor is something that, given a file name, reads the source code into an editor buffer. In a more “hyper” editor, the data generation function would be more complex, for example, generating callers of a function and filling in source sections in the editor based on the set of callers.

Related to data generation are two minor axes: storing of modified data (which we will not describe in detail here) and query modelling. *Query modelling* is a layer which data generation goes through that remembers the exact query that was used to generate the data, and it remembers the client who asked for the data. The idea of this layer is to provide multiple-view update, that is, if some other client modifies some data which is active in a query, the query modelling layer notifies all the other clients to update their views.

3.2 Data Display

The second major axis is *data display*. Roughly speaking, this is the display of a set of related information in a visually related grouping. “Related information” is slippery to define, but one simple model of this is that each piece of information should be conceptually of the same type. For instance, in a class browser, a class and all its subclasses (direct and indirect) are in one data display group, but the slots of one or more of these classes is in another visually distinct group.

Another example might be in an “inspector”, which displays a handle to a long list; selecting the list might enumerate each item in the list, which may of course all have different *data* types, but in the mind of the user, they are grouped because they were generated by a single action in the interface. (So maybe a less slippery way to define “related information” is “information that gets generated by a single call to a data generation function.)

The basic display protocol is, given a data set and a “canvas”, display the data on the canvas. Note that data display is actually broken into two bits: the display of each item in the display, and the layout of all of those items.

The display of an item is typically driven by what CLIM would call its presentation type. That is, the item displayer is a property of each item being displayed. In a directory editor, pathnames would be presentation as a pathname object, and the visual represention might be a name string or a file folder icon. In an inspector, the same pathname might simply be presented as a Lisp or Dylan object.

The layout of the displayed items is a property of each canvas. The sorts of layouts used during data display include:

- Tree graph. For hierarchical data (for example, a library and all the libraries that use it). Clicking on an “expansion target” adds a new node(s) to the tree.
- Directed graph. For heterarchical data (for example, a set of root classes and all the classes that use it). Clicking on an “expansion target” adds a new node(s) to the tree.
- Outline. For roughly hierarchical data, where a tree graph does not convey any extra useful information. Each item is on its own line, and levels of indentation represent depth in the hierarchy. For example, a directory editor.
- Table. A 2-column table where the left column displays an item name and the right column displays an item value. (Note that this sort of layout could also be done by the item displayer. This is the sort of arrangement that is found in an traditional Inspector tools.)
- “List.” A simple list of items, where there is no detail below any item. For example, a scrollable list containing a stack backtrace.
- “Detail list”. Like a list of items, except that each item has an additional control that might open a level of additional detail of a different sort. For example, a list of names of source definitions whose additional detail is the source code for the definition.
- “Editor”. A display that allows editing of text, etc.

Editing is taken to be an extension of the display protocols. For example, deleting a line from an outline in a directory editor might delete the corresponding file, or there might be a gesture in an inspector that allows a value to be modified, and so forth.

Note that, in this model, data display is responsible for all “in-line expansion”. For example, trees and DAGs know how to incrementally add nodes to their display, and are responsible for displaying targets that indicate where the expansion takes place. Outlines know how to add new lines to the outline, and are again responsible for displaying targets to indicate where the expansion takes place.

3.3 Real-Estate Management

The third major axis is *real estate management*, that is, the arrangement of "display groups" into a larger scale layout. This might be provided by careful layout of display groups within a single window, or by multiple panes in a frame (tiled or not), or by multiple top-level windows. Real estate management includes such things as placement and size of the display groups. This real estate management might be done explicitly under user control, or the browser itself might have a real estate management policy.

Traditional inspectors typically have only a single display canvas. The browsers in SWM's CLIM-based environment usually have a DAG display, which is coupled with one or two list displays; the relationship between these displays is not made clear. Fry's Outliner has almost all of the displays, and the real estate management is done by a "best guess" (with hints provided by the user based on choosing slightly different targets in each display).

A good implementation of a browser should permit some user control over layout without requiring the user to carefully arrange windows and panes in order to get useful views for typical operations.

4. Adding a UI to a "Abstract Browser"

Of course, given the three major components (data generation, data display, real estate management), they need to be combined under a user interface that allows the functionality provided by the components to be called. The display component needs to provide "handles" on pieces of data that allow further data to be generated and displayed. The display component might also provide for filtering and sorting existing data sets. The real estate manager might provide for "hot links" between display groups.

We believe that each of the three components of a browser should be separately and explicitly manipulable. This is based on a model of a user that says that, if any UI gesture does "too many" things, then it is hard to build a model of what the UI is actually doing.

SWM's CLIM-based browsers do not currently have any manipulable real estate management; the multi-pane browsers have hard-wired real-estate management. The most a gesture will do is generate data and display it. This is not a sufficiently flexible architectures. There is also no explicit connection between the various panes of a browser.

What is called "in-line expansion" in Fry's Outliner falls mainly under the data generation and data display componenets. "Out of line" expansion has aspects of all of data generation, data display, and real estate management. What this means is that single gestures can simultaneously generate data, pick a new piece of real estate, and do display; we believe that is too much functionality to wrap into a single, and it makes modelling the Outliner itself more difficult. From a pure UI point of view, the Outliner does not adequately visually distinguish the displays of different groups of data; perhaps the use of embedded bordered regions will help here.

Apple's "Binder" requires explicit real-estate management via pane-splitting, and a "hot link" is created between the original pane and its new sibling (that is, there is a data generation operation

done at the same time). Binder then groups data generation with data display. We think that of the three reasonable browser implementations we can point to, Apple's Binder has the best overall architecture, although there is significant room for improvement. It is also reasonable to question whether splitter panes are the best way to distinguish between different groups of data. The creation of hot links is very simple, and they are really useful; there is some visual indication of the connection between the output of one pane and input of another, but it is not very good.

Fry's Outliner provides for the mixing of all sorts of different displays. SWM's browsers and Apple's Binder have a more restrictive model, where each group of data is fairly homogeneous and has a single appearance. I would make the argument that the flexibility provided by the Outliner tends to make it more visually confusing, and the more restrictive display model provided by SWM's browsers and Apple's Binder actually make them simpler to understand (that is, a user has an easier time modelling what these browsers are actually doing).

Fry's Outliner also allows recursive nesting of in-line and out of line display, whereas Apple's Binder does not do this -- it allows real estate management only at a single, top level. The recursive structure of the Outliner could be ameliorated by providing more borders, colored background, etc., but it's not clear that these measures will make it sufficiently less confusing to justify the effort. It's also possible that the fact that we don't all have enormous screens makes deeply recursive browsers impractical even if we can solve the problem of visual confusion.

The splitter panes in Apple's Binder take up a lot of room on the screen. This could be a genuine problem.

Appendix A. SWM's Proposed Browser Design

This is an attempt to capture the key characteristics of SWM's ideas for the implementation of a browser that meets the abstract architecture described above. It is meant to be descriptive rather than provide rationale or comparison to the Outliner, though we will insert rationale or comparison in places purely to aid in making the description clear.

A.1 Overall Architecture

The Browser starts out with single large window that will be used to contain multiple panes. The panes get created on the fly (see below). It's not clear if the "single large window" is scrollable; this will have to be decided based on some prototyping (note that the Binder's "single large window" isn't scrollable).

There can be multiple Browsers; a new one can be created from an existing Browser, but on the whole there isn't much communication between Browsers.

The "single large window" has a bunch of tiled panes, each of which potentially has scroll bars. The user can create new panes by "splitting" an existing pane. This splitting might be accomplished the way the Binder does it (via a splitter control in each pane), or by some other other means. The important point here is that all of the panes within the main window are kept in a simple, tiled layout.

A user can enlarge any one pane at the expense of an adjacent pane by dragging the border between the two panes.

Each pane has a “title bar”, whose contents will include:

- The name of the description, or generator function for the pane.
- The name of the “parent” node for the pane, that is, the piece of data that will be passed to the pane’s generating function.
- A way to choose a new generator for the pane.
- A way to choose filtering and sorting options.
- A way to “reparent” the pane, and redirect the output of the pane (where the output is given by the set of selected objects in the pane). This will be accomplish by having “In” and “Out” targets in the title bar. A user reparents a pane by dragging the “out” icon from one pane into the “in” icon of the pane you want to reparent. The “in” and “out” icons of linked panes will be designed such that it will be clear from their visual appearance what their corresponding out or in icon is in the parent or child pane of the pane in question. [Using unique colors and shapes will work here.]

A.2 Display Within a Pane

A pane has only one generator associated with it at any point in time. This generator is used both to determine the initial contents of the pane [in conjunction with the selected item in the parent pane] as well as any “sub-items” shown in the pane. When browsing heterogeneous data, the user will need to use a new pane for each “new kind” of data, since that data will probably not be able to use the same generator as the generator its parent was created from. For example, if you’re descending a class-and-subclasses structure (displayed as a DAG) and want to see the slots of one of the classes, then another pane will be required to see the slots. Or you are browsing a directory structure and you run into a file, the generator of “Directory Contents” won’t be applicable to a file, so you might create a new pane and assign to it a generator of “File Contents” or “Definitions in File” or whatever.

The pane will also have the ability to filter and sort the output of the generator to determine exactly which items to display. These filters and sorters are chosen by the user, probably through icons/menus accessable from the pane’s title bar.

The display of the contents of a pane has two aspects.

The first aspect is the “item view”, that is the way each individual data item is shown. Usually all items in the same pane will have the same item view. The most common examples of an item view might just be the “name” of an item or some other short handle, for instance, the name of a class or a file name with an associated folder icon. For source code items, something more complex is called for: a “handle” consisting of the definition name with a target that calls up an in-line editing area which displays the source code for the definition.

In general, this Browser doesn’t mix different item views in the same pane. For example, a directory editing pane will not typically allow a user to in-line expand the list of source definition in a

source file. To do this, you need to hot-link in another pane that is prepare to display source code or source code handles.

The other aspect of data display a “layout”, which determines how the items are positioned w.r.t. each other. There is only one layout per pane. Typical examples are an “outline” layout, letting you expand each item into a place immediately beneath and indented from the item. Another layout is a “graph” layout allowing you to show DAG’s and interactively expand each item in both directions. Still another might be a “tabular” layout, although it may be better to implement this in terms of “item view”.

A.3 Creating New Panes

The steps to create a new pane are:

- Create the space for the new pane. For example, this can be done by dragging a splitter bar on an existing pane.
- Drag the output icon of the desired parent pane into the input icon of the new pane. By default, the new pane will be linked to the pane it was split from.
- Optionally, choose a generator to use to expand the parent into the new pane. The Browser will try to pick a reasonable generator, so this step isn’t always necessary.

At this point, there is a new pane that is ready to be populated with new data. To fill the new pane:

- Select a parent item to expand in an existing pane.

Note that the creation of a new pane can be accelerated by providing more sophisticated commands on displayed objects that perform the splitting and linking automatically.

Of course, the layout and item view of the new pane will default to something reasonable like a standard view for each, or perhaps “inherit” them from the [semantic] parent or [syntactic] parent pane. The user can change these defaults through controls on the pane’s title bar.

Note the semantic parent pane is the pane containing the selected parent item to be expanded, whereas the syntactic parent is the pane that was split to make the new pane. The Outliner does not have a distinction between the syntactic and semantic parent panes since a pane’s parent pane [or “outlevel”] is always above or to the left of the pane. (There are exceptions to this for new window and for nested out-levels in some contexts.)

A.4 Generators

For each presentation type for an item there is a set of generators (that is, functions that produce a set of “kids” from a data item) that take that item’s data as input and produce the new output. Rather than using the current Outliner’s current “generic” generators, there are specifically named generators such as “subclasses” and “directory contents”, so that it is very clear what the user will get when they choose a generator. When you pop up the generators menu for a parent of type class, you will “subclasses”, “superclasses”, “slots”, etc., but not “file contents” as an option.

A.5 Reusing Panes

To avoid pane proliferation, you may want to reuse a pane. So rather than expanding an item into another pane, we'll have some way of replacing the contents of a pane with the expansion of one of its items. The old contents of the pane will be put on a history list which can be recalled later by picking it off of a menu accessable from the pane's title bar as in the Lispworks inspector.

Appendix B. Fry's Proposed Browser Design

B.1 The Design

The Fry/Plusch general browser design (referred to hereafter as *Workspace*) incorporates numerous features to fulfill the goals listed in the strategy section. Some of its key ideas follow.

Recognition of the need to formally characterize the layout between parents and children. This permits the architecture to support multiple ways of showing the relationship between a parent and its children. The two fundamental categories are "children near parent(s)" (typical of outline views and graph views), and "siblings together" (where the children appear in an area that is spatially separated from the area containing the parent and the parent's siblings). Horizontal layouts such as the classical Smalltalk browser capture some advantages of both.

Information hiding. This is essential when the user is walking very deep or very bushy data structures. A key concept here is to hide the "uncles" (the non-direct ancestors) of the data being focused on, making it easier to see the context of the focused data set. Hiding the uncles also requires less screen real-estate, meaning that more of the relevant data can be shown. Should a user wish to see uncles, grand-uncles, and so on, that can be achieved by "opening up" one of the ancestors, which remains visible on the display. This makes it easy to backtrack and go down other paths.

Each instance of a *Workspace*-style browser contains a number of "view frames", each of which contains one or more views into the data being display in that "view frame". The major views in one of these frames are the *description* of the data-set being shown (for example, "Subclasses" or "Variables in Module") and a view of the data-set itself (for example, the set of subclasses or the names of the module variables).

The simplest model of a description is that it is a function that takes one argument, typically a selected data item from another frame, and generates a collection of new data items. Typical generating functions include subclasses-of-class, slots-of-class, methods-of-class, methods of generic function, attributes-of-method, documentation-on-symbol, and so on. More complex descriptions may take additional user-specified arguments, all of which will have defaults; this extra information can be used to filter the generated data (for example, only the direct methods of a class, or only the module variables that have the string "-setter" in their name).

The resulting data is shown in a collection view. The exact appearance of the collection view can be tailored by the user, such as a simple scrollable list, a graph, or even an editor buffer containing sources for the selected functions. The view of the frame is also user changeable to show more or

less detail of the description set as well as the relationship of the description to the data set. The expanded view of a description might be a menu or dialog box that gives the user finer control over the items to be viewed. An example of such control would be something like the complex "Find Symbol" dialog in Emacs Menus, where the things displayed in the view can be filtered by their home package, the occurrence of a substring in the name, and so forth. Note that such detailed control should be permitted, but not typically required, since all such additional arguments will have generally useful defaults. As previously mentioned, the relationship between view frames (which boils down to the relationship between parent and children) is also user-changeable. Changing this view could have the effect of changing between an outline view and a "siblings together" view. A user might also toggle the direction of browser growth between horizontal and vertical by changing the parent-child view.

B.2 Problems in Window Management

The Smalltalk Agents browsers and the MCL inspector (originally Fry's design) typify one of the common problems witnessed in the Programing Environments Review, namely, screen clutter due to a profusion of little windows. Some of the tools on the Lisp machine resulted in the opposite problem: one window takes over the whole screen and doesn't let the user see additional contextual information. Solutions that involve several large, resizable windows (about 20 in the case of LispWorks) permit the user to see more than one thing at once, but they tend to hide important contextual history, which forces the user to continually position and resize the windows to see necessary information. Ultimately, this is an unsolvable problem since there are only so many pixels. Even if the user is wearing infinite resolution eyephones, the eyes and brain can only focus on so much information at a time. We can maximize screen real-estate and the user's abilities better than existing environments by clever dynamic layout with quick UI gestures for allowing the user to adjust views. This was one of the overall concerns throughout the process of developing the *Workspace* architecture.

B.3 Overwhelming Quantity of Choices and Options

The challenge of presenting an easy-of-use interface in the presence of a large number of options must be considered carefully. One approach is simply to limit the number of options available at a time. Specialize-purpose browsers take advantage of this. This is the easy solution and has been proven to work passably well. Its disadvantages include lack of flexibility and the necessity to implement (as well as learn how to use) a variety of different browsers. With good, uniform UI design the learning curve can be minimized, but the fact remains that switching from one browser to another can require a "context switch" on the part of the user. Sometimes this is the desired behavior — using multiple browsers to represent different activities — but sometimes the frame-of-mind change in undesirable because it causes the user to lose track of the task at hand. Multiple browsers may also limit the amount of useful information visible at any one time due to screen clutter of a bunch of different overlapping windows.

The other basic strategy of one very general browser "workspace" necessitates having lots of options available at a time to accommodate the users wishes. We need more than just good graphic design to minimize the potential confusion here. Making common choices easy and obvious while preserving a smooth ramp of increasing difficulty for invoking the less common choices

is one strategy. Another is to have smart, dynamically changing defaults. For example, if a user wants to walk down the subclass tree as opposed to the slot tree for a set of classes, he can choose the “subclass description” for the top object, then just click on title of the descriptions menu to get the inherited default of subclass in each successive frame. In addition to the problem of streamlining this “navigation” process, there is also the problem that, in a heterogeneous browser, it is not always obvious what relationship a parent object has to its children.

B.4 Too Much To Implement

We run the danger of attempting to include so many features that we won’t be able to make any of them reliable by a reasonable code-freeze date for Release 1. The separate browsers strategy has modularity going for it: we can simply not bother implementing certain browsers. It has the disadvantage of redundancy, i.e., some of the same features need to be implemented for each browser albeit in a slightly different way for each. Using a common browsing substrate should ameliorate this latter risk.

The one general browser approach has economy of code on its side as well as economy of pixels. This may help out when running DylanWorks on smaller machines. Unfortunately it requires that the underlying architecture be “just right” to accommodate the wide range of possibilities. The one general browser may also prove less than optimal for certain highly specialized tasks. The underlying principles of design (such as parent-child relationships, description-dataset relationship, and so on) aim for generality so that unforeseen holes will be serendipitously (to use SWM’s term) covered anyway. The *Workspace* design is also modular but on a different axis. First, each of the view/layout categories (parent-child, frame, description, and collection) are designed to have a number of possibilities, where selection of the used one is made dynamically by the user on a case-by-case basis (with inheriting defaults). The initial release could have a small number of views, say 1 to 3 possibilities, for each category. Additional view types can be plug-in add-ons to be implemented in subsequent releases. Also the descriptions is an extensible list where we can provide a basic set in Release 1 and extend them as we see fit later on. We may even choose to publish the specs for generators, filters, and views, as well as provide documented base classes so that a user can create new subclasses should they so desire.