At Harlequin created on September 23, 1994 and last modified at Harlequin on November 11, 1994.

DylanWorks Editor

Scott McKay

1. Introduction

This paper describes the requirements for the editor to be used for DylanWorks. This editor might be implemented in-house, or it might be gotten from another source, but it needs to satisfy the requirements outlined in this paper.

2. Editor Requirements

Any editor we buy or build must satisfy at least the following requirements:

- It must be able to integrate smoothly with the other tools in the environment, for instance, the compiler, the debugger, browsers, and the source database. Secondarily, it must integrate with the "outside world" (that is, it should have a reasonably standard look and feel, and potentially integrate with embedded document protocols such as OpenDoc or Ole).
- It must be possible to tailor the display so that we can display more than just raw text. For example, we want to be able to embed icons in a buffer's display, and we want to be able to implement contracting/expanding regions in the buffer.
- It must be able to compose a "buffer" out of a set of first-class section nodes. It is not sufficient to have a buffer filled with raw text gotten by read the contents of a file. Rather, we need to be able to fill in a buffer's section nodes by calling a generating function.
- It needs "mode hooks" that allow the contents of a buffer to be processed in interesting ways, so that we can implement intelligent language support.
- It needs good support for undo and redo.
- It needs good support for the use of the mouse. It is not adequate to use the mouse only as a way of positioning a cursor within the buffer. Instead, the mouse should be usable as a device to accelerate common interactions, for instance, being able to use drag-and-drop techniques for composing programs, having the contents of buffers be mouse-sensitive in a context-dependent fashion, and so on.
- It should be possible to embed an editor "pane" in other tools.
- It should be customizable and reasonably extensible.

3. "If we buy the editor"

All of the pre-existing editors we might buy fall into two major categories: Gnu Emacs and not Gnu Emacs. Gnu Emacs (and its descendent, Xemacs — also known as Lucid Emacs) are unique

in that they are truly open, since their source code is freely available. All other non-Gnu Emacs editors are substantially less open.

The issue of openness is important because it is a good way of achieving the goal of integration. Another way of achieving this is via a scripting architecture (such as Apple Events, OSA, or Ole Automation); a pure scripting approach is somewhat more work because the "pipe" between tools is narrower.

The goal of tailoring display in a non-Gnu Emacs editor might be achieved via a pure "part" architecture (in the OpenDoc or Ole) sense of the word, although doing this properly is by no means simple. Xemacs already has a reasonable level support for this, which was done at great expense to Lucid. Gnu Emacs does not have any reasonable support for this, at least not for the foreseeable future. Unfortunately, FSF is not adopting Xemacs, so although it is being actively maintained now, the ultimate fate of Xemacs is not entirely clear.

Composing buffers from first-class sections is something that has to be built in from the outset, since there are subtle policy decisions with respect to many things (such as, what does mean to save something that might be modified in multiple buffers). The Gnu Emacs editors could be extended in this way. It is not clear yet what could be done with non-Gnu Emacs editors.

Gnu Emacs editors have clearly demonstrated their ability to be tailored to support particular languages. It is not clear how well off-the-shelf non-Gnu Emacs editors will do.

The support for undo/redo in both Gnu Emacs and non-Gnu Emacs editors alike is not as good as it could be. Many non-Gnu Emacs editors support only very short undo histories. Most editors provide no way to browse the undo history. Gnu Emacs is adequate, but it could benefit from some work in this area.

Gnu Emacs editors, on the whole, use the mouse only in the simplest ways. Xemacs is somewhat better than straight Gnu Emacs in this regard. Most non-Gnu Emacs editors are also quite poor in this area, supporting mostly conventional GUI mouse operations.

4. "If we build the editor"

The only reason to build our own editor is for the express purpose of meeting all the goals above We therefore assume that we will succeed in meeting the goals, and concern ourselves with *how* we can meet the goals.

Integrating with the other tools in the environment requires establishing an "editing protocol" and obeying the protocols of the other tools. This is no different from what we would have to do to integrate with an off-the-shelf editor. The difference is that, with our own editor, we can communicate at the object level rather than by using a more primitive scripting architecture. Past experience seems to show that communicating at the object level is simpler to implement, and results in a higher level of integration in which serendipitous occurrences are more likely.

We believe that, for the initial release, integrating our editor with the outside world is of lower priority than integrating it with our own tools. We anticipate that our own editor would be able to integrate with the outside world via OpenDoc or Ole.

The goals of tailoring the display, piecewise composition of buffers, and intelligent processing of the buffer contents are all straightforward. Some of the details are outlined below.

We plan for the user interface substrate to support undo and redo, and semantically-based mouse handling. Our own editor will be built on top of this.

4.1 Details of the Quattro Editor

This section describes a possible editor implementation. For historical reasons, we refer to this editor as “Quattro”.

A *buffer* is used to group some set of data in Quattro. A buffer may be displayed in zero or more *windows*. A buffer is filled in by some sort of a *generating function* that creates a linked list of *nodes*; each node contains some data. Some nodes contain a *section*, which is represented as a set of *lines*. For example, reading a file into Quattro creates a buffer that contains nodes, each of which contains one section full of lines from the file.

4.1.1 Lines

The line is the basic unit of storage in Quattro. The **<basic-line>** class has slots to store the following state: next line, previous line, modification tick, its owning section, properties, etc.

The **<line>** subclass of **<basic-line>** is used to store ordinary textual lines. It supports high-speed textual display in a single, default font. A simple specialization of **<line>** supports textual display in multiple fonts, perhaps having different pitch and height.

The **<diagram-line>** subclass of **<basic-line>** supports the display of diagrams, icons, or other embedded structures (such as dialogs or even other embedded applications in the OpenDoc or Ole sense of the word). Lines that include both text and diagrams are stored using an instance of **<diagram-line>**.

Note that this organization of lines does not support true “line flow”. That is, a line extends horizontally the entire width of a buffer. We do not believe that this restriction will overly limit the usefulness of the editor (in fact, almost all editors, including many in commercial text formatting systems, have this restriction).

4.1.2 Buffer pointers

A buffer pointer (a “BP”) is an object that represents a location in a buffer. It stores a line and an index within the line. There are two classes of buffer pointers: **<simple-bp>**’s (which do not move when text is inserted or deleted in their vicinity), and slightly heavier-weight **<bp>**’s (which do move when text is inserted or deleted in their vicinity).

4.1.3 Sections

A section is simply a set of lines. The line linkage in a section does not extend outside of the section (that is, the previous-line slot in the first line of a section is empty, and the next-line slot in the last line of a section is empty). Each section remembers what nodes it is being used in. Sections are represented by the **<section>** class.

We use this design so that sections can be stored in multiple nodes, each of which might be in a distinct buffer. This allows us to compose buffers from arbitrary section, and allow changes made to a section in one node or buffer to be reflected in other nodes and buffers.

4.1.4 Nodes

An *interval* is a range in a buffer, and is represented by a start BP and an end BP. Simple intervals are represented by the **<interval>** class.

The **<node>** class is a subclass of **<interval>** that maintains additional state: next node, previous node, parent node, children nodes, modification tick, etc. A buffer is built from a sequence of nodes, and most nodes contain a section (which itself consists of lines).

There are a number of subclasses of **<node>** that are used to represent additional information. For example, source code buffers might contain “definition nodes”, and directory editing buffer might contain “pathname nodes”. A “definition node” might contain information that describes what module or package the definition came from, and so on.

Each node also has a slot containing the “displayer” for the node. Usually, the node’s displayer simply calls the normal line-based display function on each line in the node, requesting it to display itself. However, the node’s displayer might be something different, such as a function that displays only a marker for the node’s contents; this can be used to implement “opening” and “closing” of nodes in a buffer. Sometimes the displayer for a section might add some output, for example, a little “tool bar” of interesting operations on the section.

As we pointed out above, the line linkage of a section in a node does not extend outside the section. Thus, to map over all the lines in a buffer, a program must map over all the nodes in the buffer, and map over the lines in each node’s section.

4.1.5 Buffers

A buffer is composed of a sequence of nodes. Buffers can be filled in any number of different ways:

- Fill the buffer with the contents of a source file (which might be a database source file, or a traditional flat file).
- Find a single definition (meta-point).
- Create nodes containing all the callers of some function.
- Create nodes containing a class and all its subclasses,

- Create nodes containing all the direct methods of a class.
- Create nodes containing all the methods of a class having "-setter" in their name.
- Fill the buffer with a directory listing.

There are a number of buffer classes. Two common ones are the class **<file-buffer>** (which contains the contents of a source file) and the class **<generated-buffer>** (which contains a set of nodes created by some sort of "generating function", such as "all callers" or "direct subclasses").

Buffers maintain some additional state, such as a current text style, the default module (or default package for Lisp buffers), and so on.

4.1.6 Major Modes and Minor Modes

The *major mode* of a buffer controls the overall behavior of the buffer, including how the data in the buffer is interpreted. For example, the Dylan major mode is responsible for parsing the contents of the buffer as Dylan source code.

The *minor mode* of a buffer controls other bits of behavior. For example, there might be a minor mode in a Dylan major mode buffer that displays all Dylan reserved words in bold and comments in italics.

4.1.7 Windows

A window is a view into a buffer. A window displays one buffer at a time, but a buffer can be displayed simultaneously in many windows. Windows maintain the state that drives Quattro's redisplay.

Note that much of Quattro's redisplay is driven by its own undo substrate. That is, the undo substrate tracks the changes made by a user command, and arranges for the appropriate redisplay to happen. Individual editor commands will only rarely have to advise Quattro what to redisplay.

4.1.8 Editors and Editor Frames

An *editor* is an object that maintains overall editor state. This state is not kept in the editor application itself, since we expect that Quattro will support simple editing gadgets and panes.

An *editor frame* is an application frame that has an editor object, plus a command loop and perhaps a thread of its own.

4.1.9 Integration with User Interface Substrate

Quattro's user interface will be built on the user interface substrate provided by "DUIM", although in certain critical areas, Quattro will use customized implementations of the DUIM protocol.

In particular, Quattro's output will not be implemented using ordinary DUIM output recording. Instead, Quattro will have a custom redisplay algorithm optimized for displaying a (relatively small number of) lines of data from a (potentially much) larger number of lines. Presentations will be generated on the fly by language-specific methods on the buffer's major mode.

Quattro will use CLIM presentation-based input model, so that the power of this model can be brought to bear on the contents of a buffer. This will allow a substantial amount of integration to be done with other DylanWorks tools, with useful operations being directly available on the things displayed in buffers.

4.1.10 Files and Streams

We expect that Quattro intervals will support the DylanWorks streams library protocol, at least to the extent that ordinary ASCII character streams can be read from and written to buffers.

We also intend to be able to read directly from the source database, creating "hard" sections in buffers that correspond to the section entities in the source database. When we read in "flat" files of program source, the sections will be generated on the fly by a language-specific sectionizer.

4.1.11 Other functionality

In addition to the basic Emacs-like functionality that users expect, Quattro will support the following editor functionality as well:

- Variable pitch, variable height fonts; embedded diagrams and gadgets.
- Full context-sensitive use of the mouse in editor buffers.
- Tags Tables, which are integrated with the DylanWorks system configuration substrate.
- Multiple "possibilities" buffers, for example, multiple Tags Searches or Edit Callers active simultaneously.
- String searching and simple regular-expression searching.
- Patching tools.
- Keyboard macros, and being able to customize key bindings for commands.

In addition, Quattro will import functionality from other DylanWorks tools, such as:

- Class browsing functionality.
- Generic function and method browsing functionality.
- Cross-reference browsing functionality.
- Interface to incremental compilation and dynamic linking.

4.2 Open Issues in the Quattro Design

Zmacs and Gnu Emacs maintain their undo history on a per-buffer basis. In Quattro, a single section can be in more than one buffer. This can confuse things — changes to a section in more than

one buffer will get recorded in different undo histories. Do we stick with per-buffer undo histories, or do we do per-section, or do we have (as Tucker Withington suggests) a global undo history that gets managed on a section-by-section basis?

Gnu Emacs is powerful in large part because of its extensibility. What can we do about this?

Zmacs has 10,000 parameters. What should be parameterizable in Quattro?

5. Open Issues

Is it an accurate conclusion to say the if we “buy” an editor, it should be either Gnu Emacs or Xemacs (probably the latter)?

We need to investigate the exact state of Xemacs, and figure out what its disposition is (especially with respect to Gnu Emacs). One thing to note here is that these editors are “copylefted”, which might possibly restrict just how effectively we can use and extend them.

We need to investigate other editors for use under Windows, and whether we realistically think they are powerful enough for use with a programming environment.

We need to see what other Windows programming environments use as their code editors, that is, what other editors are commonly used under Windows? How attached do people get to them?

We need to realistically estimate how much it would cost to build an editor. A good estimate is that it would take about 1 man-year to build something rather good, which could then be grown over time.

We need to realistically estimate how much it will cost us to integrate our environment with an existing editor. Lucid, for one, spent a very long time building Xemacs for their Energize product (2 full time people for nearly 2 years). Note that integrating with an exisiting editor is in many was less predictable than building our own — we don’t really know how good a job we can do, nor how long it will really take.

How does a user extend Quattro? That is, Quattro is written in Dylan and runs in DylanWorks. Presumably the user writes some new Dylan code that is intended to be loaded into DylanWorks itself. Can we make this work?

How much do we care about what the extension language is? If we build Quatro, the extension language will be Dylan; with Gnu Emacs, it is Elisp. Is this important?