Notes on the Vitality of Hypertext Systems

Mark Bernstein

Eastgate Systems, Inc.
E-mail: bernstein@eastgate.com

ABSTRACT

These notes address some thoughts on the implementation of spatial hypertext systems, particularly with respect to designing platforms that remain both malleable and stable through periods of experimentation and radical change. The concepts described here may well be familiar to my colleagues, but would be difficult to find in the literature.

WHERE ARE THE HYPERTEXT SYSTEMS?

At the first hypertext conference, in November 1987, one could stand in the demonstration area and survey a wide range of *new* hypertext systems. A few of these systems (NLS, Notecards, TIES, Intermedia) had been deplayed for a few years. Others (including Symbolics Document Examiner, HyperCard, Xanadu, Guide, gIBIS, Storyspace, HyperCard, SuperBook, Ntgeraid, Hypergate) were much younger.

With the exception of NLS/AUGMENT, the *old, established* systems were all younger than VIKI [5] is today. This is strange.

Hypertext systems are much easier to write today than they were in 1987. Our computers are much faster, our languages and libraries are greatly enhanced, our understanding of software architecture, design, and development have all improved dramatically. The docuverse, then considered a pipe-dream by even its most supportive sympathizers, now is real.

In part, it may be natural that we write fewer new systems today than we once did. In the 1980's, if your research plans required a hypertext system, you most likely had to build your own. Today, existing tools are ubiquitous, and a number of research laboratories possess tools and frameworks developed over the course of many years.

Nevertheless, I find the dearth of new systems startling. In part, perhaps this scarcity reflects a mid-90's research consensus that the interesting part of the hypertext system was in the back-end or middleware layer (cf. [7], [6], [8]). Perhaps veterans of the early development efforts remember too well the financial and human costs of the early tools and have not yet noticed that the economy has changed[1]. A variety of societal, economic, and ethnographic factors doubtless play a role. But the chief obstacle, I suspect, is a fear that has haunted the hypertext research community from its earliest days to the present: the fear that a system will grow rigid with age, fade, and ultimately disappear.

VITALITY

Designers of new hypertext systems have always been deeply concerned about their systems' demise. At first, the collapse of the system's platform was an insurmountable threat; when the last computer that could run your system was decommissioned, surely your research was doomed. Fortunately, developments in software emulation have rendered this anxiety largely obsolete; the entire Macintosh community spent half a decade on emulation life-support without particularly noticing. Nor does such longevity require a substantial user community. The Final FRESS demo has been a star attraction at *several* previous Hypertext conferences, and yet Durand reports that he has, once again, found a way to revive FRESS on new hardware.

The chief hazard to the survival of a hypertext platform is not loss of its platform but rather the gradual loss of its *vitality*, its ability to support change and experimentation.

A vital system is open to innovation, supporting strange ideas and unexpected affordances. Young systems are filled with vitality. In time, most research platforms become encrusted with obsolete code, inconvenient assumptions, obscure optimizations. Code grows; as it grows, it becomes harder to understand. Early architectural decisions may plague the design, and yet may be embedded so deeply that changing them seems impossible. The scars of old experiments, hasty patches, clever hacks whose meaning is now arcane, all gradually accumulate. Eventually, it seems easiest to start over, or to change fields.

TECHNIQUES FOR SUSTAINED VITALITY

In recent months, I have been engaged in designing *Hampshire*, a foundation intended to support two distinct lines of hypertext platform development:

- **Storyspace**: a reimplementation of a classic system for narrative, spatial hypertext
- **Ceres**: a spatial hypertext tool for making, analyzing, and sharing notes

Storyspace is now 12 years old; we'd like Hampshire to last a long time before it needs to be rebuilt. Ceres addresses an important problem domain that remains poorly understood;

[1] In my Hypertext 2001 paper, "Card Shark and Thespis: exotic tools for hypertext narrative", I describe two new hypertext systems, each of which was developed in the course of a few weeks.

it is expected to require unusually rapid change and enhancement. Both systems must be stable and polished from inception.

How may this be done?

PLUGGABLE VIEWS

Storyspace 1.x offered multiple views of a single hypertext. Map, charts, outlines, and treemaps could each be instantiated, once or many times, focusing on different parts of a document. Views were constructed by a Factory function, and each view instance acted as an observer on a shared Hypertext structure[2]. Each view was thus independent, and new views could be added without substantial changes to other views or to the Hypertext structure.

The views themselves, however, tended to be formidable chunks of code, since each view manages its own layout, display, and interaction. A good deal of logic, notably mouse handling, was replicated among views, making it difficult to enforce consistency. Adding a new feature – status rollovers, say, or contextual menus — required changes to each kind of view. Making these changes was tedious and error-prone.

In Hampshire, pluggable views delegate many of their responsibilities to Strategy objects:

LayoutPolicy: responsible for deciding which items are visible in the view, and where they appear on the screen.

DrawingPolicy: responsible for drawing an item in a specified area of the screen

LinkPolicy: responsible for drawing any links that may appear in the view

These strategy objects are interchangeable, permitting facile experimentation: nothing prevents you from taking a policy designed for a map and plugging it into a chart or a list. Policies can be freely mixed and matched. This freedom makes it easy to provide extra services in a wider range of contexts; for example, where Storyspace 1.x provided a modal, alphabetical list to help users locate an item by name, Hampshire lets us share presentation style, contextual menus, and modeless observer logic, all inherited from HypertextView or acquired from a plug-in policy and therefore nearly free.

To ensure that policies remain pluggable, we must restrict hidden dependencies between different kinds of policies. If policies use each other's services, or rely on services from the Views they serve, then some policies will make assumptions about their partners and we will lose orthogonality. To prevent this, policies do not communicate with each other, nor do they contain back-pointers to the View that uses them.

This restriction is sometimes inconvenient: for example, the LayoutPolicy doesn't know anything about the drawing policy you plan to use. The DrawingPolicy, in turn, must cope with whatever screen space it's given and cannot (as in OpenDoc [1]) initiate a negotiation for additional space. But, in compensation, this impoverished communication path also offers developmental independence: changes to one kind of policy never change the others. Because dependencies are limited, we avoid massive recompilations even when undertaking radical changes to a policy. In particular, there is no disincentive for making changes high in the policy hierarchy or for undertaking radical experiments that might need to be backed out[3]. The difference between an experiment that takes seconds to recompile and one that requires minutes is substantial, especially when trying interesting experiments that might not work. Fast edit-compile-test cycles keep the system young.

A further consequence of this strategic architecture is that it enforces a separation between the abstract hypertext Node and the graphic object that represents it in a view. Nodes cannot draw themselves; the DrawingPolicy doesn't simply call

```
node->Draw(…)
```

because the node doesn't know what policy to follow. If there are several classes of Nodes, each of which might need to be handled separately, the layout and drawing policies can become complex. Conditional logic may suffice if only a few Node classes are needed;

```
if (node->IsSpecial())
      DrawSpecial(node,…);
```

Overloading, or a Visitor arrangement[9], may help if the conditional logic becomes burdensome.

GIANT CLASSES AND OTHER CODE SMELLS

Pluggable policies embedded in pluggable views helps to keep the view hierarchy vital, and the view hierarchy, after all, is where spatial hypertext happens.

Core classes like Hypertext, Link, and Node are equally hard to keep vital. First, these classes have broad responsibilities, and so they tend to acquire large interfaces. Worse, nearly everything in a hypertext system is likely to need to understand what a Hypertext is and how a Node or a Link works. Changes to core classes tend to be traumatic: at best, everything needs to be recompiled; at worst, everything breaks. Abstractions inherent in the core classes become implicit assertions throughout the system: if we assume that a Link is a pair of references to Nodes, then client classes

[2] Storyspace 1,x for Macintosh was implemented in Pascal, and then reimplemented for Windows in C. For convenience, I use modern terms to describe structures and functions in these pre-object systems. Names of patterns conform to [3] unless otherwise specified,

[3] If narrow communication channels are too confining, a blackboard might provide a useful alternative

will internalize the assumption and introducing multiheaded links later may prove infeasible [4].

To keep core classes vital, they need to be small. This means we need to delegate responsibility systematically and thoroughly. We might begin, for example, by factoring Content out of Node, so Nodes can delegate responsibility for representing, serializing, and changing their content to a separate Content object[2]. This means, however, that pervasive dependency on a few big classes is replaced by widespread dependency on myriad small classes:

```
TheNode->TextLength()
```

becomes

```
TheNode->GetContent()->TextLength()
```

Delegation chains themselves add complexity, and some find them unsightly; common chains can be migrated back to Node, at the cost of increasing Node's interface.

Core classes also ossify because adding new data to a class requires an entire family of changes to

- serialize the revised objects
- adapt existing files
- provide error handling for illegal input (easily overlooked but especially important since the new serialization is bound to be error-prone)
- provide accessors and mutators for the client of the new data

Hampshire addresses this problem by keeping almost all its data in attribute-value lists. The interface to Value, moreover, assures that any value can be coerced to, or constructed from, a string; in consequence, serializing the attribute/value list is straightforward and extending it is almost free. The cost — maintaining an associative array of attribute-value pairs for each Node, performing a lookup for each data access, manipulating strings for every i/o operation — turns out to be affordable on current systems even though it would have been insupportable in the recent past[4].

For example, suppose one afternoon we decide each Node should have an AccentColor, which some views might use when displaying the node. Conventionally, we'd add a new field to Node

```
RGBColor accentColor
```

and then set about initializing the color in each constructor, saving and restoring the field in files, deciding what to do when reading old-format files, and so forth. Because Node's interface has changed, everything that depends on Node (essentially the entire system) may need to be recompiled and relinked. All this represents a substantial deterrent; unless we *know* we *must* have accent colors, it's tempting to defer the effort. In Hampshire, though, the drawing code simply asks for the accent color:

```
SetColor(node-
>Get("AccentColor")->AsColor());
DrawSomething(…)
```

In old documents, we'll automatically get the default accent color. When we save, the accent color gets saved automatically. The cost at runtime is chiefly an associative lookup, but inexpensive memory makes the cost of a small map or hash table quite bearable.

NANARD/SCHWABE TEMPLATES

HTML or XML export has essentially supplanted the printer as the natural way for information to leave a hypertext system. Each writer will have different requirements and expectations for the kind of HTML or XML document they wish to export, and these requirements will change frequently.

I came to Hypertext '98 with plans for a sophisticated, rule-based system that would let writers declaratively specify the HTML export process, defining precisely how the hypertext should be transformed on export. Jocelyn Nanard heard me out with great patience, and then, having summoned Marc Nanard and Daniel Schwabe to consult, improvised a method that is at once far more flexible to use and incomparably simpler to implement.

We begin, as most large Web design projects begin, by creating one or more "wireframe" pages — pages in which the design elements are all present, but the contents are represented by placeholders such as greeked text.

Next, we replace the placeholders with symbolic references to information from the hypertext. For example, we write

^title()

to say "insert the title of the current node here". Similarly, ^text(parent) stands for the text contained in the node's parent. Indeed, we can refer to any attribute of the node in this way, and may use logical expressions:

^if(exists(next)) ... provide a "next" link ...^endif

This HTML (or XML) file becomes a template into which exported nodes are poured. The identity of a node's template is itself an attribute like any other. Templates may ^include the contents of other nodes; the included nodes are themselves processed against their own export templates.

---

[4] Processor speed and pipelining help, but the real key here the availability of large memory spaces on widely-used machines. The availability of good associative arrays in the underlying libraries is also a comparatively recent phenomenon.

The template files are, in effect, declarative specifications of the appearance and page structure of a site or subsite. But the specifications are created in the designer's familiar language, with her customary tools and viewed with a Web browser. The destination need not be HTML or even XML, since the template mechanism is quite general[5].

Nanard/Schwabe templates contribute to the system's vitality by providing an extremely simple, declarative facility for moving information from the hypertext document to cooperating clients and processes. New templates can be designed by any user, without access to the source code. Acquiring new clients and capabilities is an important factor in extending a system's vitality.

ANOTHER LAYER OF INDIRECTION

The objection might be raised, not unreasonably, that as a stand-alone, single-user hypertext system Hampshire stands apart from modern hypertext systems research, and that it ought, in fact, to have been built on top of a modern, collaborative back-end or structure server, In the short term, this would pose difficulties for many Hampshire users, but its long-term attractions are clear. Could this be done?

First, it is worth noting that the XML streams Hampshire uses for configuration and file input need not be local files, and indeed need not be files at all. It should be fairly easy to wrap any sufficiently-similar hypertext model, providing document-level locking that might prove adequate for small-team collaborative tasks while permitting interoperability with open hypertext systems.

Hampshire's uniform attribute-value storage might also provide sufficient hooks for achieving node or facet-granularity integration with a synchronous collaborative server. In principle,

```
node->Get("Xpos")
```

could easily be a network service[6]. Difficulties arise in the presence of faults; for example, the implicit transactional rollback offered by the Undo stack can conflict with the transactional rollback provided by the remote server, and keeping both models in sync (and explaining them to the user) may prove quite difficult.

ACKNOWLEDGMENTS

I am grateful to Peter Nuernberg for many fruitful discussions, especially regarding integration with remote services. Frank Shipman and Cathy Marshall provided helpful advice on implementation experience with other systems. Marc and Jocelyn Nanard and Daniel Schwabe convinced me to follow the template approach. Eric Cohen read an early draft and offered many helpful correcttions. I've sometimes disregarded their advice, and errors are entirely my fault.

[5] Increasingly, even HTML export may not be destined primarily for human eyes. Just as a generation of IT tools depends on "scraping" logical 3270 screens, many applications today extract their data from HTML wrappers intended to provide mere style markup for browsers.

[6] Fully integrating asynchronous servers, on the other hand, appears to require an entirely different approach to client design. Another layer of indirection won't solve this.

REFERENCES

. 1. Apple Computer, *OpenDoc Programmer's Guide*. 1995, Reading, MA: Addison-Wesley.

2. Martin Fowler, *Refactoring*, ed. s. 1999, Reading, MA: Addison-Wesley.

3. Erich Gamma, *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software.* 1995, Reading, MA: Addison-Wesley.

4. Catherine C. Marshall, *et al.*, *Aquanet: A Hypertext Tool to Hold Your Knowledge in Place*, in *Hypertext '91*. 1991: San Antonio. pp. 261-275.

5. Catherine C. Marshall, Frank M. Shipman III, and James H. Coombs, *VIKI: Spatial Hypertext Supporting Emergent Structure*, in *ECHT '94*. 1994: Edinburgh. pp. 13-23.

6. Dave M. Millard, *et al.* "FOHM: A Fundamental Open Hypertext Model for Investigating Interoperability Between Hypertext Domains" in *Hypertext 2000* Proceedings. San Antonio, Texas: ACM pp. 93-102

7. P.J. Nuernberg, J.J. Leggett, and E.R. Schneider, "As We Should Have Thought" in *Proceeding of Hypertext '97*. 1997: Southampton, UK. pp. 98-101.

8. Siegried Reich, et al., "Addessing interoperability in open hypermedia: the design of the open hypermedia protocol" *New Review of Hypermedia and Multimedia*, 1999. **5**: p. 207-246.

9. John Vlissides, *Pattern Hatching: Design Patterns Applied*. 1998, Reading, MA: Addison Wesley.