

Networking Squeak

Bijan Parsia

May 16, 2000

1 Background & Rationale (Squeaking Along?)

1.1 Why use Squeak for networking?

You're looking around for a web server. Or maybe a new email client. Or perhaps you want to write a web crawler. Why use Squeak? After all, Squeak networking apps tend to be lacking in maturity—no surprise in so young a system. For example, Scamper, the web browser, is neither compliant with the latest specs, nor particularly polished, nor dramatically quick (though it is quite snappy for many purposes). If what you wanted was the best or neatest or most powerful web browser as such, Scamper wouldn't fit the bill.

But Scamper (and every other Squeak networking app) derive some compelling advantages *simply* from being written in Squeak:

- They share in Squeak's hyperportability. Squeak images run seamlessly wherever there is a Squeak VM—no converting, recompiling, or tweaking needed. [Note about look and feel and platform specifics] Even better, Squeak's VM is very portable, and, indeed, has been ported nearly everywhere [Note about 1 personness of port, compare to Java VM, point out prims needed].
- They share in Squeak's malleability. Given the power and flexibility of Squeak's language and integrated development tools, it's very easy to program clients or servers, and even easier to tweak or extend existing ones. A few data points: The Pluggable Web Server, and the

various Swikis built on it, were put together by Mark Guzdial when he was still new to Squeak, while Scamper and the IRC client were written by Lex Spoon during a Summer Internship at Squeak Central. I, as a Squeak, Smalltalk, and programming novice was able to usefully modify all of these with very little effort.

Combined with its portability, Squeak's malleability makes it almost uniquely ideal for making custom, compact clients.

- Finally, they share in Squeak's seamless systematicity. Every application—web browser, web server, mail client, what have you—is simply a collection of classes which can be combined and reused in myriad and sometimes stunning ways. [Cyberdog lovers will much to like about Squeak.] A personal note: I've learned quite a bit about http, sockets, parsing, POP, and IRC just from playing around in Squeak. It's a great place to explore, experiment, and learn.

In Squeak, you can be a programming dabbler and still have control over your tools and environment. Right now, you give up a certain level of maturity and functionality that you find in current commercial programs and even many (or most) long standing free/open source software. But using Squeak now is like having been in on the ground floor of Linux—it's usable, it's only getting better, and it's getting better fast.

1.2 Why use networking in Squeak?

You're a Smalltalk veteran. You love hacking the compiler and the VM. You mess with meta-objects and pounce on primitives. Why muck with this mundane networking stuff?

Four reasons immediately rise up:

1. Squeak aims to be a comfy computing home. One should be able to do just about anything one'd care to do, and, of course, one thing most of us want to do is play on the Internet. So, we should be able to browse the web, send and read email, chat, publish a website, and so on all from within Squeak, and for it to range from pleasant to delightful to do so. Alas, while often pleasant and sometimes delightful, Squeak is not quite paradise. But it certainly has the potential, and that's a good reason to use Squeak for networking: to help

realize that potential. So, there's the public service motive, and, in Squeaky parlance, there's a lot of "blue plane" work to be done.

2. Squeak also aims to be a cutting edge research environment. Though some might say that "multimedia" is Squeak's focus, I think that's a little narrow to describe it. Multimedia is important to Squeak, at least in part, because the various media are crucial to *communication* and, more generally, to *interaction* (not just "with the machine", but with ideas, other people, tools, and so on). Squeak is a good place to do serious and seriously fun exploration of various forms of collaboration, both with standard and with novel tools. Plus, if you don't know much about http, or web serving, or HTML parsing, I can testify that it's very pleasant to learn about them by Squeaking.
3. Of the commercial projects using Squeak (at the time of this writing!) most seem to be focused on Squeaking the web. In general, there's a lot of projects for which using Squeak to build a website (for example) is an easy sell. Web apps are high profile and (with Squeak!) a lot of fun.
4. There are more folks who know something about HTML, HTTP, FTP, and so on than know something about Squeak. I've found both as a learner-of and an teacher-of Squeak that having that solid ground to work from is very helpful. More than just providing some self-confidence and orientation, there's typically a lot of immediate gratification in using Squeak for networking tasks (and in general, of course). And, typically, people have a lot of little niggling network jobs that they'd love to have automated (or simply done). It's quite pleasant to write little classes or even just workspace code to do the job that someone might have written a Python script for, and your Python wielding friends get a very nice taste of Squeak.

Historically, Squeak networking has had a rich community of novice programmers, new Squeakers, gurus, and plain old end users. The people involved are a joy to be around. They also are an exhaustingly productive bunch. If "Net time" is quick, and "Squeak time" is quicker, I supposed that I shouldn't be surprised that "Squeak net time" is blindingly fast!¹

Here are four ways to get acquainted with networking in Squeak:

¹Draft only: Of course, these are all dwarfed by "Deadline approaching time" :))

- One can use the multifarious classes available ad hocly to aid's one SqueakWorking.
- One can use, and perhaps enhance, the collection of useful end user network apps.
- One can write new an interesting network aware programs.
- One can explore making the hodge podge of protocols, message formats, file types, name schemes, and so forth that make up our wired world as simple, natural, and transparent as possible to the programmer, user, and programmer/user alike.

2 Some simple SqueakWorking

How does SqueakWork? Very well, thanks.

With the obligatory *terrible* joke out of the way, I want to tell how to impress your friends.² Most of the sexy demos of Squeak involve doing crazy things with animation, 3D, sound, and so on, but I've found that for a lot of folks certain simple tricks have the biggest impact.³

All this demo needs is a Workspace and a net connection.

The way I start off is to type and inspect the following in a Workspace:

```
'http://www.squeak.org/' asUrl.
```

²Studies show that the above joke is not the way to go about it

³Mark Guzdial likes to tell this story (<http://minnow.cc.gatech.edu/squeak/52>):

[Code for a very small drawing program.] When you demonstrate this, be sure to draw over window boundaries, through scroll bars, and over title bars.

In my classes, when I do this for undergraduate and graduate students who have grown up with Microsoft Windows and the Apple Macintosh, their jaws drop. "But, you can't do that!" Suddenly, windows become software: You can draw on them, change them, do something different with them. That's scary. They tell me how inherently unreliable this is, to have windows that you can change. Yeah. Freedom is scary. :-)

The beauty of this demonstration is in how it completely breaks the habits of thinking his students were comfortable with *and* helps orient them in the worldview of Squeak *and* shows them how to go about exploring (in a workspace window, and with panache). And best of all, it only takes seven lines of code.

[Display: HttpUrlInspector.eps]

Cool! An URL is an object (hey, this is Smalltalk), and scanning the instance variables reveals a lot about the structure of URLs. When I first discovered this, I was grabbing URLs left and right, just to see what Squeak would make of them. (And it works for ftp, file, and mailto urls as well!)

When the charms of URL parsing wear thin, grab one of these URL inspectors, then enter and inspect following line in its “code pane”⁴:

```
self retrieveContents.
```

Whoa! Not only did Squeak go out and grab the web page at the end of the URL, but it doesn’t just pop out the text—it represents the page as a MIMEDocument! What? Huh? Html files delivered over the web are MIMEDocuments? Hmm. I remember that now. In fact, inspecting

```
'http://www.squeak.org/SQ100x100.gif' asUrl retrieveContents.
```

will give me another MIMEDocument, only this time it’s of type ‘image/gif’. That sure beats guessing the type of the content from the filename extension.

[Display: MimeDocs.eps]

But, when we get down to it, the MIMEDocument version of the HTML is just a pretty wrapper around some text (the main type *is* text, after all). Can’t Squeak do better than that?

Of course it can. In the usual way, enter:

```
HtmlParser parseString: (self content).5
```

and Explore (cmd/alt-I) it. (I find using the ObjectExplorer is more fun than inspectors for parse trees; I showed the ObjectExplorer to a friend and he shouted out, “Python should have this!!!”)

After poking around the parse tree, and examining some of the parse nodes, “do”:

⁴I hope it’s obvious that this will work a *lot* better if you’re occurrntly connected to the Internet.

⁵Bijan added method

self openAsMorph.⁶

[Display: HtmlParseTree.eps]

And if *that* hasn't knocked their socks off, click on some blue text to open Scamper onto a new page. Yes, folks, those links are *live*.

[Display: RenderedHtml.eps]

This kind of demonstration shows a number of important points, even, I hope, to the most skeptical:

- First, and always, Squeak is dramatically cool—and not just for multimedia whiz-bangatry.
- Squeak is remarkably net-savvy, with a sensible understanding of URLs, MIME types, HTML, and so on.
- Squeak's net-savvy is accessible through a variety of powerful and interesting tools. There's something compelling about browsing through an HTML parse tree at one moment and viewing the rendered HTML in another. The line between “programming tool” and “networking app” is rather thin, and this is an *advantage*, not a failing.
- This advantage isn't just productive, it's also didactic. What you know about networking guides you through the tools and the code. And one can learn about network “objects” (e.g., the structure of URLs) by using the Squeak IDE. Squeak provides an *uniform perspective* of your computing world. Opening an inspector on an URL is very much like opening an inspector on a SmallInteger. This systematicity, of course, is a driving design principle behind Squeak: *everything*, from the top to the bottom, from the interface to the VM should be accessible, or perhaps better put: *transparent*, to the same set of tools and techniques.
- The final point is that some of those most skeptical types only yield to whiz-bang multimedia extravaganzas. Fortunately, Squeak can oblige them with a networkish twist.

[Display: FlyingScamper.eps]

⁶Bijan added method

3 Service with a Squeak

I can confidently say that if it had not been for Swikis and the Pluggable Web Server, I would have taken a lot longer to learn how to program Squeak. When the PWS was first incorporated into the Squeak main distribution, I had been experimenting with various ways to integrate an interactive website with the classes I was teaching. There were a few facts about my University that made this endeavor far more difficult than needs be:

- My students were, on average, far behind the technological curve. The first time I tried to introduce a little tech to make things “easier”, the majority of my students couldn’t use email. By the time the PWS rolled around most could use email, but only had a rudimentary notion of how to browse the web and *no* knowledge of HTML. So, whatever I was going to use, I had to be able to train my students up to within the confines of my class time. (Frankly, a Bioethics syllabus looks *decidedly* odd when the first two weeks read, “Classes 1 & 2: Introduction to email. Classes 3 & 4: Getting around the website without getting hurt. Classes 5 & 6: Submitting work.”)
- There was little to no support for custom web applications. The IT department wouldn’t let individual instructors install cgis on the web server, and there was no provision for evaluating and adopting new ones. There were a few canned one’s available, but a hit counter wasn’t *exactly* what I had in mind for instructional tech.
- My department did have two net-connected machines available for graduate student use in our computer lab, but we had 40 some students and neither machine was particularly beefy. Neither machine was configured as a server, and they were Windows boxes, of which I knew little.

The PWS—with bundled Swiki implementation—came to the rescue on all fronts. The Swiki⁷ was simple enough that I could get my students up and running in a single class period. Since the Swiki came integrated with a web server I didn’t have to deal with the IT department at all, and

⁷More on this below

I was able to run my class Swikis *in the background* of a heavily used lab machine (it was a pokey 486 with 20 megs of ram, running Windows 95). Netscape was often in the foreground. This worked adequately for about two years, in which time I was able to tweak the Swiki in numerous ways and tweak my use of it even more (e.g., for one course, all of the non-exam written work consisted in Swiki pages). Compared to hunting down a Windows web server, configuring it, hunting down some collaboration cgis or what have you, configuring them, and so on, this was a breeze. I could evolve and debug my customizations on my Mac at home, and then just pop them into the server with nary a twitch.

Of course, those early days were, shall we say, a touch rocky. My Swiki had this distressing tendency to lock up a few times a week, necessitating forty minute drives at odd hours to reboot. However Mark Guzdial and I bonded over two months of remote, email-mediated bug chasing wherein Mark tore apart the PWS and the Swiki in several different ways and I spent late nights designing stress tests to force the bug.⁸

Web servers have been a prominent, perhaps the most prominent to date, Squeak application. Since [version? date?], bundled with a web server (PWS) and several server apps (most notably, the Swiki), and many people have used it just for that. Furthermore, the PWS has been ported to several other Smalltalk variants (e.g., Dolphin Smalltalk and Smalltalk/X), and served as the inspiration for a server (WikiWorks) written in yet another dialect (VisualWorks). Mark Guzdial's Swiki is, I believe,⁹ the first Wiki implemented in Smalltalk, a situation which, post Swiki, has been rectified with (at least) three Wiki implementations in Squeak alone! The creation of a Squeak based Wiki and its adoption into the core Squeak release was critical, I believe, in spurring the explosion of Swiki use in the past three or so years, and indeed, was instrumental in introducing Wikis and Wiki-like values to a *lot* of people. There are now *at least* thirty public Smalltalk based Wikis (and growing!) not counting swiki.net (which is a professional Swiki hosting organization), plus countless private ones (including a slew run by yours truly, but I've also heard of Swiki's used "in house" at various corporations, in classrooms, and, of course, for single

⁸Of course, it turned out to be the failure of a method to close its file after editing. We couldn't see it because it was only ever called in a template, which weren't stored in the image. Grr! Of course, I learned a heck of a lot about processes, and filestreams, and semaphores, and sockets, and so on. It was no loss, indeed.

⁹But do I believe *correctly*?

person use).

Servers—and, in particular, http servers—are a surprisingly common, and even more surprisingly essential component of modern *everyday* computing.

3.1 The Pluggable Web Server

There are three¹⁰ aspects of the PWS’s history that I find particularly striking:

1. It evolved in a “feedback triangle”: it, at once, served as a *tool* of community building (the main Squeak information site is a Swiki and the PWS itself has attracted a fair number of people to Squeak), a *focus* of a community (one loosely organized by the PWS mailing list), and as a *bridge* between communities (most obviously, between the various Smalltalk implementations’ authors and users).
2. It served as an important research and development platform both for web servers and web applications, and as the inspiration for new development.
3. It was thrown together in a hurry by a very bright guy¹¹ without a lot of Smalltalk experience.

The PWS has its roots in two earlier Squeak web servers which, while having some interesting properties of their own, never achieved the popularity of the PWS.

[Need to talk about Georg’s and Tim’s stuff. Plus get the genealogy chart done.]

[Indeed, I want to sing Georg’s praises.]

3.1.1 The Architecture

A slightly naive, but useful, view of a web server is that it takes in a HTTP request and pops out a page of HTML. What’s *interesting* for the web pro-

¹⁰Draft Copy: Lists, lists, I loooooove lists! Okay, okay, obviously some of these will get transitions and turn into normal prose :)

¹¹Namely, Mark Guzdial, who’s Smalltalk style in PWS I’ve struggled with and made fun of over the years :)

grammer, of course, is what happens in between the request and the response. Ideally, the application author should be isolated from the details of the request and response protocols, either simply so that one might focus on the *functionality* of one’s program, or to make it more or less *independent* of any particular protocol. This isolation is a primary design goal of both the PWS and its successor, Comanche.

The PWS *system* consists of two major components:

1. The webserver, which is implemented by the PWS class.
2. The web applications, which are all descendants of ServerAction.

PWS has four key class variables ¹² which correspond to four key actions of the server:

Class Var	Stores	Facilitates
ServerPort	aConnectionQueue	listening for requests
ServerProcess	aProcess running loopOnPort:loggingTo:	general server behavior
ServerLog	aFileStream	logging
ActionTable	aDictionary	dispatching to ServerActions

ServerPort, ServerProcess, and ServerLog are all initialized with PWS class::serveOnPort:loggingTo: and destroyed with PWS class::stopServer. One consequence of the current structure is that you can’t have PWS listening on more than one port¹³ Tidbit: if you’re trying to understand listening in depth, it helps to have a clear understanding of Squeak’s threading model. That clear understanding also will help if you want to try adding multi-threading to PWS.¹⁴

The ServerProcess polls the ServerPort for a connected socket. If there is one, then ServerProcess creates an instance of PWS on that socket (in PWS class::serve:), which then reads and parses the HTTP request from the socket connection. The instance of PWS (a “request”) now dispatches itself (in PWS::getReply) to a ServerAction for processing. The ServerAction reads the data out of the request and sends whatever response it has

¹²I neglect discussion of the class variables BackupJobs, ClientNameCache, and ServerStatus because these serve peripheral functions.

¹³Though it’s pretty clear how to extend ServerPort and ServerProcess to handle that, an earlier version in fact did exactly that; the gain in simplicity seems worth the loss of functionality as you can always run another image for each additional port.

¹⁴Which has been done before [pointer to Mark’s code], but is a good medium level learning project.

back to the request (via `PWS::reply()`), which writes it back to the socket, and returns control to the listening thread, `ServerProcess`. The `ServerPort` can, in the default setting, hold a backlog of 8 connections, so assuming that your traffic is reasonably light, and your processing isn't too terrible lengthy, all your incoming connections should be handled, and in good time.

Aside: Alas, for most of the PWS's life, this wasn't quite true. First, there was a bug in `ConnectionQueue::listenLoop` (a fascinating place to peek, by the way), but more significantly, the original socket primitives were built with the rather archaic `MacTCP` system in mind, and, for reasons that a cursory search of the Squeak Mailing list will reveal, there is a gap between a socket accepting a connection, and the system's ability to accept the next one. On light duty servers one *might* think this occurrence rare, but 1) it's more common than you think, especially for a class `Swiki` during finals, and 2) if you have multiple images, or other separate content (other media, frames, etc.) then most browsers will try to open two or three (or four!) connections with the server to download those other pieces in parallel. (Certain browsers let one set this figure, and setting it to zero is quite dramatic.) The result is that a single page request can blossom in to three, four, five, or more sets of closely spaced requests over a reasonably long period. (And note, one navigation bar, made up of six gifs, is enough to do the job.) Not only may one client end up blocking others, but the more common problem is unloaded images on pages. The current workaround is to have another, conventional, server handle the static files, say, on another machine or port. While this works, it defeats one of the compelling aspects of a PWS-like solution—everything in one place. Fortunately, I am quite hopeful that by the time this is published, the socket primitives will be up to snuff across all platforms.

`ServerAction` dispatching is mediated through the `ActionTable`, which is a dictionary of strings onto instances of various `ServerAction` classes. One registers an Action with PWS with `PWS class::link:to:`, and there can be an arbitrary number of actions (or names for a single action). (This is one way in which the WS is P) While for anything but the simplest web apps, it's probably a good idea to subclass `ServerAction`, but it's quite possible to experiment without creating a new class at all.

So, first initialize PWS, which puts an empty dictionary in `ActionTable`.¹⁵

¹⁵`PWS class::initializeAll` is worth studying as it sets up a slew of sample Actions,

Now, PWS has the notion of a “catchall” action to handle all the requests which aren’t directed to any other specific action. This catchall action is named “default”, so, since we won’t have any other actions in play, we’ll let all request be handled by our little test action.

We now have a name for our action, we need the action to go with it. We’re going to use a `SinglePlugServerAction`¹⁶ (this is the second way that the WS is P). `SinglePlugServerAction` take a block which processes any requests it receives and yields a response. For this example, we’re going to return what request’s “URL”.

```
PWS initialize.  
PWS link: 'default'  
to: (SinglePlugServerAction new  
    processBlock: [:request |  
        request reply: PWS success;  
        reply: PWS contentHTML , PWS crlf.  
        request reply:  
            '<html><head>  
            <title>A Really Simple App</title></head>  
            <body><h2>', request url, '</h2></body>  
            </html>'] ).  
PWS serveOnPort: 80 loggingTo: 'log.txt'.
```

[Display: SimpleAppMontage.eps]

As is clear in this screen shot, a PWS instance contains only part of the original URL¹⁷, but passes the rest to a particular `ServerAction` to parse and to process as the author sees fit.

The “default” action is a special action: it’s dispatched to whenever a request isn’t handled by some other action, and it thereby receives the entire “path” of the URL (i.e., everything except the host). Normally, the first part of the URL’s path is the name, as it appears in the `ActionTable`, of the desired action.

Changing the block slightly:

including Swikis.

¹⁶Not very euphonic, I know, but *I* didn’t name it!

¹⁷Sadly, the PWS antedates the truly file URL classes... but that doesn’t mean one can’t use them today!

```

PWS link: 'default'
  to: (SinglePlugServerAction new
      processBlock: [:request |
          request reply: PWS success;
          reply: PWS contentHTML , PWS crlf.
          request reply:
            '<html><head>
             <title>A Really Simple App</title></head>
             <body><h2>', request url, '</h2></body>
            </html>' ]).

```

and evaluating the result gives us two actions in the ActionTable, with slight, but distinguishable, behavior.

[Display: SimpleAppComparison.eps]

3.1.2 Two Simple Examples

WebChat

SqueakTop Cam

3.2 Comanche—the Next Generation

Comanche: charging the next generation web revolution

After PWS comes Comanche, a boiler room of innovation. More a framework for creativity than a web server, Comanche changes the way you approach web content creation, with a Squeaky clean internals.

Comanche is not a rigid monolithic behemoth, but rather a string of components that deal with various aspects of the web serving. The portion that deals with the HTTP and network is probably of no interest to the application designer, whereas the plugs that are closer to the functionality of the program are of utmost importance.

[Display example]

3.2.1 Problems with PWS

The simplicity of PWS makes it an attractive platform for rapid prototyping of web applications.

However, this simplicity comes at a cost: it limits the scalability in terms of performance and sophistication of applications. Code duplication is inevitable, since there is little between the socket and the PWS servlet.

Comanche grew as a project to address weaknesses of PWS. It aimed at providing a cleaner framework with numerous reusable components. Most ugly HTTP and network level aspects were swept under the carpet. In fact, many applications can be built without reliance on any specific protocol. Such applications can have two or more user interfaces: web based and native Squeak UI.

3.2.2 How an HTTP request is processed?

(for the following discussion, we'll assume Comanche listens to port 8080).

When a client (web browser) opens a connection to Comanche's port, say 8080, Comanche forks a handler. For HTTP requests, the default protocol adapter is `HttpAdaptor`. `HttpAdaptor` does exactly what its name implies: adapts HTTP for use by others. It creates an `HttpRequest`, which contains items such as HTTP header fields, GET and POST form fields, cookies, etc. If we stopped here, Comanche wouldn't be any different from other web servers. What makes it unique (TODO: check this claim) is ability to define custom request types.

Lets take Swiki as an application example. After `HttpAdaptor` reads the request and returns an `HttpRequest`, Swiki takes control and attempts interpreting the request as a `SwikiRequest`. If successful, it processes the request.

Once the application is done processing, it returns it result (e.g., a `SwikiPage`), which is then converted into `HttpRequest` by the `HttpAdaptor`. However, the application doesn't need to know that such conversion will take place. For debugging purposes or under a different environment, HTTP may not be present at all. Thus, the important thing is to return the right object, which should know how it can be converted into an `HttpRequest`.

In some cases, it may be possible to define "filters", thereby eliminating the need to make each class of objects explicitly convert to `HttpRequest`. For example, many objects are printable, so their printed form can be used for conversion to `HttpRequest`

- 3.2.3 A new architecture**
- 3.2.4 Comanche Modules**
- 3.2.5 The Making of a Module**
- 3.2.6 Requests and Responses**
- 3.2.7 Peers and Children**
- 3.2.8 The Two Examples**

3.3 Swikis

3.3.1 A WikiWiki History

The classic Smalltalk system and IDE is essential a very powerful hyper-text/media tool. It provided for easy input and browsing of small chunks of text (methods), grouped into "texts" (classes) which were related hierarchically (by inheritance) and by keywords (class and method categories). This hypermedia system is focused on describing and manipulating a fairly elaborate set of computational processes (the "virtual image"). Though invented with a slightly different vision [ref], the World Wide Web evolved into a rather rudimentary and static system. Writing, reading, and browsing (the latter two together—with an emphasis on the browsing—constituting the activity of "surfing") were radically separated activities, as the classic edit-browse-tweak cycle demonstrated.

Ward Cunningham's original WikiWiki site was a brilliant realizing of Smalltalk-like hypertext system in a standard HTML/HTTP/CGI setup. WikiWiki writing bears a lot of similarities, both superficial and deep to standard Smalltalk programming:

- Smalltalk is dynamic typed so any variable can hold an object of any type. Typically, there are no barriers between a user and the system: anyone can alter just about any part of the system. Disaster is generally averted by social conventions and powerful, though simple, tools. Similarly, a standard WikiWiki, unlike most web "discussion groups" or "bulletin boards", doesn't enforce a structure on the discussion, nor does it protect any bit of text from overwriting. Anyone can edit any page at any time. This freedom has proved to be a great boon to building community based *collaborative* websites.

Most Wiki's lack even an informal moderator and yet have proved remarkably resistant to vandalism and tend to have rather high signal to noise ratios. Furthermore, like Smalltalk images, Wikis have turned out to be rather effective in building large scale, long term, *evolving* systems.

- Both Smalltalk and standard Wikis use a very simple, yet expressive, syntax coupled with a similarly simple, yet expressive, semantic model.

For example, class names in Smalltalk begin with a capital letter and generally are InterCapped. This fits in nicely with their being AbstractNouns. In many Wiki's, including the original, you create a link—and, perforce, a new page or page reference—by InterCapping. Note that SomePeople (and TheyKnowWhoTheyAre) find this sort of WikiStyle QuiteAnnoying. Indeed, MarkGuzdial's Swiki eliminated this WikiFeature in favor of having ArbitraryPhrasesOf-Text which have been EnclosedWithAsterisks serve as a PageNames and PageReferences. In MyOwnExperience the two style produce some radical differences in conceptualization, and indeed, in SimpleWritingFlow. But ComparedWithTraditionalHTML, either style—InterCapping or the use of SimpleDelimiters—is a VastSimplification.¹⁸

Furthermore, such common formatting as lists, emphasis, and even tables have gotten a variety of simplifying support.

- Both Smalltalk and Wikis provide simple and flexible tools which tend to abstract away from the details of the “supporting” OS and provide incentive to experiment.

For example, in most Smalltalk systems, instead of a gazillion little class and method definition files, you typically have three: the image file, the sources file, and the changes file. The image is what you “see” (or maybe you *feel* it), and you see it as an active system of live objects (i.e., it doesn't matter if the image is organized as a single file

¹⁸I've used both styles in a teaching context, and think both have much to recommend them. Both can get stilted, and both require some careful management. In newer S/Wikis, there have been a some interesting innovations such as “aliases” (well, the ability to have the hypered text and the the linking text be distinct) and the ability to rename a page.

or memory region, or as a series of loadable files, or as a database, or...) In the original WikiWikiWeb, the various “pages” were stored in a database and rendered on demand, whereas in Swikis, each standard page has it’s own file. But there are also “composite” or “virtual” pages—the *Recent Changes* page comes to mind—which are constructed (typically dynamically) from smaller nodes. The point is, of course, that just as someone browsing a site doesn’t typically have to (or *shouldn’t* have to) know about the directory structure of the web pages she encounters, so too the *authors* and *editors* of a S/Wiki site shouldn’t have to know either.

The Recent Changes page, itself, is an amazingly handy feature. It requires no work to maintain, and yet makes it very easy to keep track of the pulse of the S/Wiki. I tend to bookmark the Recent Changes page of the S/Wiki’s I visit.

When learning to use a typical Smalltalk IDE, there is usually a big “aha” experience when one figures out the “Find Senders” (i.e., display all the methods which *use* a certain selector and “Find Implementers” (i.e., display all the methods which *implement* a certain message). Swikis have had a pair of similar features, to wit, “Display references to this page” and “Display all references in this page”. The ability to find all pages which mention the current page one is browsing is surprisingly useful for finding one’s way about the site. And the ability to display all the pages referenced *within* the currently browsed page led to, in my experience, all sorts of useful and interesting page factorings. In essence, “Display all references in this page” lets one build composite pages by placing a simple table of contents at its head. I’ve had students organize fairly large slices of information in two or three orthogonal structures which were then very easy to print out and use as a study guide.

Finally, no S/Wiki would be complete without an integrated and reasonably quick search facility with *at least* full text search. An absolutely essential feature!¹⁹

- Finally, just as the changes files keeps a record of every edit one makes to a method or class, so too the S/Wiki keeps a history of

¹⁹While this is the last feature I discuss, it is by no means the last “tools” S/Wiki authors have implemented. Hot lists, browser trails, and appending are just three examples.

every edit one makes to a page. Typically, S/Wikis let you get at past versions so that one can restore trashed or garbled pages. Recently, the hot thing has been to provide colorful diff-displays of page's history.

4 Squeaky Clients

4.1 NewsAgent

4.2 Scamper

4.3 Celeste

4.4 IRC shenanigans

5 Client/Server Squeakergy

5.1 Comanche Chat revisited

5.2 SwikiBrowser

5.3 MailMouse

6 In the Pink