Case Study: Pluggable Web Server (PWS)

Squeak: Object-oriented design with multimedia applications
Building Web Applications and Interfaces

- Grabbing information off the Network
  - Sockets and protocols
  - HTML and GIF/JPEG images

- Putting information on the Web
  - Starting the Pluggable Webserver (PWS)

- Creating user interfaces for the Web
  - Forms, Post, and “Servlets” in PWS

- Case study: Swiki/CoWeb
Accessing the Network

- Squeak provides access to **Sockets**
  - Note: One place where platform incompatibilities arise
    - (Macs are whimpier network machines than others)
    - `Socket initializeNetwork “To start”`
- Sockets support connection between two TCP/IP addresses
  - Each side gets to treat the other basically (but not exactly) as a stream
- Many uses don’t require Socket-level programming
Using Sockets to Access the Network

- Subclass of **Socket** provide high-level access to other protocols
  - **SimpleClientSocket** is a general mechanism for building *clients* (more later…)
  - **POPSocket/SMTPSocket** enable email access
  - **FTPSocket** allows you to store and retrieve from FTP servers
  - **HTTPSocket** allows you to store and retrieve from Web servers
FTPSocket Example

ftp server: 'cleon.cc.gatech.edu'. "host"
ftp user: 'guzdial'.
ftp password: 'fredflintstone'.
ftp directory: '/net/faculty/guzdial'.
ftp openFTP.
ftp putFile: (FileStream fileNamed: 'myfile') named: 'remotefile'.
ftp getFileNamed: 'remotefile' into:
    (FileStream fileNamed: 'myfile-downloaded') open.
ftp quit.
HTTPSocket Access

To get a page:

- (HTTPSocket httpGet: 'www.cc.gatech.edu/index.html') contents
  - httpGet: returns a **RWBinaryStream**
  - contents returns the text

To get an image:

- (HTTPSocket httpGif: 'www.cc.gatech.edu/newhome_images/CoC_logo.gif') display "Returns a Form"
- HTTPSocket httpJpeg: 'blah.jpg'
How Web Servers Work

- *Client* sends a request to a server as URL
  - URL specifies a server and a *protocol* for access
- *Server* finds the file based on the path in the URL
- Server returns the file specified
  - HTML, GIF, whatever

```
http://www.cc.gatech.edu/index.html
server directory
index.html
Computer: www.cc.gatech.edu
```
Hyperlinks, Images are Same

- Embedded images, links, etc. inside the HTML page generate new server references
  - `<image src="images/home.gif">`
  - Means, “Now ask the (same) server for the home.gif file under the images directory under the server directory”
  - Works the same way to references to other servers
MIME Types

How does the client (browser) figure out what’s in this stream of bits coming in?

MIME Types: Standardized text string that identifies the type of the multimedia material

- text/html is good ole HTML
- image/gif is GIF

Check out your browser’s “Helpers” to see other examples
Interface (CGI): A Different Approach

- Same basic structure, but instead of just sending the file back, execute the file and send the result (HTML, GIF, whatever)
- How does the server know when to execute vs. send the file?
  - Typically, the server has a certain directory identified (e.g., in a configuration file) as a CGI directory
  - For example, http://www.mycompany.com/cgi-bin/fred.pl
- How input gets to file, how file gets executed, how output gets collected is all platform dependent
Servlets: “Internal CGIs”

- A *servlet* is a piece of executable code that lives inside the server (e.g., a module)
- Reference via URL is server-specific
  - Can be completely invisible to client if response is from file or servlet
- Works the same as a CGI
  - Servlet returns some element back to the client
Setting up the Squeak PWS

For any server, tell it where the server directory is

- `ServerAction` class message `serverDirectory`
- Simply return the path

Specific to PWS: Tell it to serve normal files

- "Make the default server action be serving a file."
- PWS link: 'default' to: `ServerAction` new.
- "Start serving"
- PWS `serveOnPort: 8080 loggingTo: 'log.txt'.
- "Stop it with PWS `stopServer"
Ports, URLs, and Names

- That 8080 is a port
  - Typically, Web servers run on port 80
  - On Windows and UNIX, Squeak users can't typically create a server at port 80, so do it elsewhere

- Finding out your machine's name
  - NetNameResolver nameForAddress: (NetNameResolver localHostAddress) timeout: 30).

- End URL
PWS URL Parsing

- PWS keeps an *actions* table
  - Dictionary object with string keys and *action* object values
- When a URL enters the server, first word in path is used as lookup in the table
  - If the word is found, then the *request* is sent to the action object as argument to *process*: message
  - If word is not found, request is sent to action at ‘default’ key
- The *request* is an instance of PWS
  - It contains the URL, the socket, and other information about the request
PWS Serving Files

“ServerAction” process: request

| pieces |
self checkAuthorization: request.
pieces := self parse: request.
self log: pieces to: request.
self replyTo: pieces from: request.
Actual File Serving

“ServerAction” replyTo: pieces from: request
   (StandardFileStream isAFileNamed: pieces)
      ifTrue: [ “Return HTTP Protocol pieces first”
         request reply: PWS success;
            reply: PWS contentHTML, PWS crlf.
         request reply:
            (FileStream fileNamed: pieces)
               contentsOfEntireFile]
      ifFalse: [ request error: PWS notFound].
HTTP Protocol Response Format

- First: Success, redirect, or error
  - PWS success, redirectTo:, notFound

- Second: MIME type
  - PWS contentHTML, content:
  - MIMEDocument guessTypeFromName:, guessTypeFromExtension:

- Third slot, but typically, just CR-LF
  - PWS crlf

- Finally, content
PWS “Servlets”

- The action object doesn’t have to reference a file!
  - Action can subclass ServerAction, but as long as it implements process: doesn’t matter

“SillyAction” process: request
  - request reply: PWS success; reply: PWS contentHTML; reply: PWS crlf.
  - request reply: ‘<html>
    <head><title>Silly Page</title></head>
    <body><p>This is a silly page!</p></body></html>’

- PWS link: 'silly' to: SillyAction new.

- http://mymachine:8080/silly
Creating User Interfaces on the Web

- HTML defines a set of tags for *forms* and for user interface elements within those forms
- A form references an “action” URL
  - Typically, a CGI script or servlet
  - Something that can *do* something with the UI input
- When the form is *submitted*, the information from the UI elements are packed into the request to the webserver
- The input is made available to the script/servlet
  - *How* available is platform/language dependent
The Challenge of Web UI

- There is no state!
  - Server doesn’t “remember” you
  - Everything carried from page-to-page must be embedded in the page, or recorded somewhere on the server

- You can’t do dynamic updates (without using applets or JavaScript)
  - Server sends a page, user responds with a form
  - User selections can’t immediately change the page
Specifying Forms in HTML

- `<form></form>` to define an input form
- `<input>` to define an input field
- `<select></select>` to define a selection list
- `<option>` to define a list item
- `<textarea></textarea>` to define a text input field
Sample Form

<html>
<head><title>Sample Input Form</title></head>
<body>
<h1>Sample Input</h1>
<form method=post action="http://mymachine/input.pl">
<p>Input examples:
<p>10-character text input:<input name="firstinput" type=text size="10">
<p>Number:<input name="secondinput" type=int>
<p>Check it off:<input name="thirdinput" type=checkbox>
</form>
</body>
</html>
Sample Form, Part 2

<p>Select from:
<select name="fourthinput">
<option>Us
<option>Them
</select>
<p>Write comments:
<textarea name="fifthinput"></textarea>
<p><input type=submit><input type=reset>
</form>
</body>
</html>
Result of Sample Form

Sample Input

Input examples:

10-character text input:

Number:

Check it off: 

Select from: Us

Write comments:

Submit Query  Reset
Can also Select Scrolling List

<select name="fourthinput" multiple=true>
  <option>Us</option>
  <option>Them</option>
</select>
Other Important Variations

- Can provide a *name* on the Submit button which appears on the button label
  - Default is “Submit Query”

- Can make *hidden* input fields
  - Can use these hidden input fields to carry state (e.g., decisions/input from past pages)
How to Deal with the Fields in PWS

The parsing of the HTTP request is done in PWS
- In other languages, like Perl, this can be left to the CGI/servlet

The PWS request carries a *fields* Dictionary
- Key is name from form, value is input
- request fields at: ‘firstinput’
- Value is a string
Typical Input-Handling Action Process

Figure out whatever you need from the input forms
  - via request fields

Compose HTML (or GIF or whatever)
  - Can compose raw strings, or use methods in `HTMLformatter`

Compose the HTTP response fields

Return the composed content
...But There’s An Alternative in PWS

- EmbeddedServerAction is just like ServerAction in terms of serving files
- But if the served HTML contains Squeak inside of special tags, the Squeak gets evaluated and the result is returned in the tag’s place
  - `<? ‘abc’, ‘def’ ?>`
- PWS link: ‘embedded’ to:
  EmbeddedServerAction new
- Now, HTML files in embedded directory under serverDirectory has its tags evaluated
Example of Embedded Squeak

<html>
<title>Sample Embedded Page</title>
<body>
<h2>Welcome</h2>
<p>Today is <?Date today printString?>
<p>Now is <?Time now printString?>
</body> </html>
Single File CGI using PWS

<html><title>Factorial Calculator</title>
<body>
<form method="POST" action="factorial.html">
<p><b>Number to compute:</b><input type=text name="number"
    value="<?request fields notNil
        ifTrue: [request fields at: 'number' ifAbsent: ['0']]
        ifFalse: ['0']?>>" size=10 maxlength=10></p>
<p><input type=submit name="action" value="Compute Factorial"></p>
<hr>
<p><b>Factorial</b><p>
<?request fields notNil
    ifTrue: [(request fields at: 'number' ifAbsent: ['0']) asNumber factorial]
    ifFalse: ['nothing yet']?/>
</form></body></html>
MVC on the Web

- Web interfaces can easily be structured like MVC
  - Controllers go away: That role is played by browsers and HTTP
  - View objects can be responsible for formatting data for the Web and interpreting user input
  - Models are still the same
- But views predominate the interaction, and dependencies are less important
  - Who cares if a Model signals a change, if the View is handing the Model the data and formatting the result?
- Then why do MVC? Separating the Model and the View is good engineering
Case study: Swiki/CoWeb

- Original WikiWikiWeb by Ward Cunningham
  - Also invented CRC Cards & Extreme Programming
  - WikiWiki is Hawaiian Creole for “Quick”
    - WikiWiki tour bus around Honolulu Airport
    - Q: What’s the quickest way to make a website?
    - A: Invite everyone on the Web to edit your pages and create new ones
What is a Swiki?

- (Also called “CoWeb” for Collaborative Website, to avoid the 10 minute story about Hawaiian creole…)
- A normal website, except that every page has an edit button
  - Click it, and you get a textarea containing the source of the page. Save it to update the page with changes
  - *AnyPageName* creates a link to a page of that title, creating the page if it doesn’t already exist
- Other features: Recent Changes, attachments
- Recent enhancements in Comanche Swiki (not PWS-based)
  - History, more non-HTML enhancements, easier HTML entry, admin tools
History of Swiki

* PWS was created as a tool for exploring collaborative spaces (by Mark Guzdial)
  - Based on a Squeak Webserver by Georg Gollman

* Swiki was created as an example application for PWS
  - Tried the original Wiki, but it disallows HTML and it’s in Perl, which disappointed many of our potential users
  - In use by almost 1000 Georgia Tech students in the very first term

* Swiki and PWS became part of the Squeak release in early ‘98
  - Since then, an Open Source effort with use and fixes from all over the world
Setting up a Swiki

- Download the Server files
  - http://guzdial.cc.gatech.edu/st/Server.tar
- Point PWS to the new serverDirectory
- **PWS initializeAll** “Installs everything”
- For each Swiki
  - Create a directory for the Swiki you want to create
  - **SwikiAction setUp**: ‘NewSwikiName’.
- At startup time
  - **SwikiAction new restore**: ‘NewSwikiName’.
  - **PWS serveOnPort**: 8080 loggingTo: ‘log.txt’
## Swiki Class Structure

### SwikiAction
- name
- source
- formatters
- process:
- browse:
- edit:
- inputFrom:
- log:
- pageUrl:

### HTMLFormatter
- specialCharacter
- swikify:linkhandler:

### URLmap
- linkFor:
- from:
- storingTo:
- recent
- searchFor:
- newpage:

### SwikiPage
- name
- address
- file
- date
- time
- coreID
- user
- editReqDate
- editReqTime
- pageStatus
- text
- text:
Class Responsibilities: SwikiAction

- Handles the incoming request
  - Basically, URL structure defines the request
  - SwikiName.[PageRef].[edit|versions]
  - SwikiName.[searchResult|recent]
- Provides the HTML formatter and the template formatters (more later)
- Knows its directory where pages are stored
Class Responsibilities:

URLmap

- Provides functionality over the set of pages
  - Searching
  - Recent Changes
  - Creating new pages
  - Finding all the links on a page
Class Responsibilities:
SwikiPage

- Knows how to read and write self from disk
  - Via the text/text: accessors
- Knows all the information about a SwikiPage
  - time, date, name, IP address of last poster, edit times to avoid conflicts, status of page
Class Responsibilities:

HTMLformatter

- Handles the translation of Swiki-format into HTML
  - Including getting/creating new pages (via URLmap)

```plaintext
formatter swikify: pageRef text
  linkhandler: [:link | urlmap
    linkFor: link
    from: request peerName
    storingTo: OrderedCollection new
    page: formattedPage]
```

- Also handles interpreting Squeak in embedded HTML pages which is how pages are formatted and served
How Pages are Actually Served

- There are templates in the Server/swiki folder
- Templates define look and feel of Swiki pages
- Templates contain references to “request” (which in this case is the SwikiPage) for creating URLs, filling in text, etc.
<font size=-1>
<a href="<?request url,'.edit'?>">Edit this Page</a>
<br/>
<a href="<?request url,'.searchresult'?>">Search for References to this Page</a>.

<form action="<?request map action name,'.'?>searchresult" method=POST>
Search for text in all pages:
<input type="text" name="searchFor" />
<input type="submit" value="Start Search" />
</form>

<a href="<?request url,'.all'?>">Display this Page and all its References</a>
</font>
</body>
</html>
Critiquing Swiki’s Design

- Too much in class methods
  - PWS, HTMLformatter
- Wrong placement of some responsibilities
  - URLmap creates SwikiPages!
  - Has made it hard to do variations of SwikiPages: protected, caching, rendering, etc.
- Not very flexible formatting
- Too much in-image
  - Only bad for our audience: Lots of non-Squeakers doing Swiki!
- Comanche Swiki fixes many of these
Evaluating Swiki in Actual Use

- In general, usability good
  - On several questionnaires across many departments, people claim that they can use within five minutes

- Breadth of usability is so-so
  - Spring 2000: A ChemEng and Arch class started using CoWebs
  - Week 10: Arch had 1500 student-created pages. ChemEng had zero.
Why people don’t Swiki

- Interviews help us understand why users might not use it
- Is the teacher there? If not, little benefit
- Why use it? Need activities that are worthwhile and encourage use
- Finally and Most Common: “Oh? It’s editable?”
  - If the activities (user task) doesn’t require editing, users won’t learn the feature.
Summary

- Socket and its subclasses in Squeak allow for access to the network across several protocols
- PWS allows pages to be served from Squeak
- Web user interfaces require use of `<form>` tags and dealing with the stateless client-server model of HTTP
  - But MVC is still useful as a paradigm here
- Swiki is a case-study of using Squeak for Web UI
  - Useful pieces: Actions, templates, HTMLformatter
  - Usability is good (by questionnaire), but goals for software interact with users’ tasks (discovered by interview)