

A Stateful Web Augmentation Toolkit

by

Matthew J. Webber
S.B. Course VI, XVIII MIT 2005

Submitted to the Department of Electrical Engineering and Computer Science

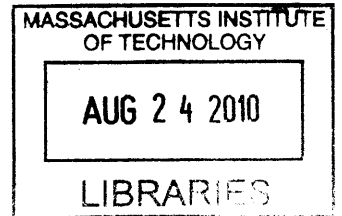
January 28, 2010
[February 2010]

in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

Copyright 2009 Matthew J. Webber.
All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to
distribute publicly paper and electronic copies of this thesis document
in whole and in part in any medium now known or hereafter created.



ARCHIVES

Author _____

Department of Electrical Engineering and Computer Science
January 28, 2010

Certified by _____

Robert C. Miller
Associate Professor
Thesis Supervisor

Accepted by _____

Dr. Christopher J. Terman
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

A Stateful Web Augmentation Toolkit

by

Matthew J. Webber

Submitted to the Department of Electrical Engineering and Computer Science

January 28, 2010

in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis introduces the Stateful Web Augmentation Toolkit (SWAT), a toolkit that gives users control over the presentation and functionality of web content.

SWAT extends Chickenfoot, a Firefox browser scripting environment that offers a variety of automation and manipulation capabilities. SWAT allows programmers to identify data records in database-backed web sites. Records are nodes of data corresponding to rows in the database backend. Programmers can append additional functionality to those nodes, and the resulting code can be bundled up and installed by users without technical expertise.

SWAT consists of three modules: a Site Profile module that identifies data records, a Tweak module that defines the look and behavior of an interactive widget, and a Storage module that persists the widget state across pages and browser sessions. Default implementations are provided for each module, and these implementations adhere to an API that encompasses all communication between modules. A programmer can extend or replace any module to improve a system built with SWAT.

With SWAT, end users can customize sites far beyond where their content providers stopped, and can add functionality that logically connects different data sources, changes how and where data is stored, and redefines how they interact with the web.

Thesis Supervisor: Robert C. Miller

Title: Associate Professor

Table of Contents

- Table of Contents 3
- Table of Figures 6
- Introduction 7
- Related Work 10
 - Structured Data on the Web..... 10
 - Mashups..... 11
 - Existing Mashup Tools 12
 - Limitations of Mashups..... 12
 - Other Existing Technologies..... 12
 - Annotation Tools..... 12
 - Widgets 13
 - Web Scripting Environments 13
- Approach..... 14
 - Design Goals..... 14
 - Modularity 14
 - Flexibility 14
 - Simplicity..... 15
- Risks 15
 - Feature Creep 15
 - Saturated Space 15
 - Data Obfuscation 15
 - Performance 15
 - Security/Privacy 16
 - Target Audience 16
- Implementation 18
 - Modules 18

Site Profile Module	18
Tweak Module	20
Data Store Module	21
Annotated Example Script	22
Early Development.....	25
Developing for AJAX Applications	26
Challenges of Modifying AJAX Applications.....	27
Four Event Handling Designs	29
The Glass Pane System.....	29
Glass Pane: Dealing with Obscured Widgets	30
Glass Pane: Detecting Mouse Position	31
Multiple Glass Panes	33
Multiple Glass Panes: Browser Features.....	33
Multiple Glass Panes: Scrolling and Dragging.....	33
Multiple Glass Panes: Variants.....	34
The Root Filter System	34
The Leaf Filter System	34
Evaluation	36
Simple Sites	37
Craigslist.....	37
Google and Yahoo Search	37
Youtube.....	38
AJAX Applications.....	38
Google Calendar.....	38
Google Calendar: Tests	39
Google Calendar: Testing Summary.....	42

Other Calendar Applications	43
Google Documents.....	44
Remember the Milk	47
Discussion	48
Conclusion.....	50
Acknowledgements.....	51
Bibliography	52
Appendix A: API Description	55
API Functions	55
Tweak API Functions	55
Site Profile API Functions	56
Data Store API Functions	57
Appendix B: Hook Points.....	59
Appendix C: Implementation Data	61
Implementation Comparison	61
Google Calendar Results: SWAT Functionality.....	61
Google Calendar Results: Browser Functionality.....	62
Google Calendar Results: Application Functionality	62

Table of Figures

Figure 1. Augmenting Craigslist	7
Figure 2. SWAT Modules can be replaced individually.....	8
Figure 3. Yahoo Search result and DOM subtree.....	19
Figure 4. Identifying a Youtube Video.....	19
Figure 5. Interaction of SWAT Modules.....	21
Figure 6. Favorite Stars on YouTube.....	24
Figure 7. Accidentally Doing Two Things With One Click.....	28
Figure 8. Operation of the Glass Pane.....	30
Figure 9. Hidden widgets.....	31
Figure 10. Event Creation Error.....	32
Figure 11. Features of Google Calendar.....	39
Figure 12. An in-page popup message on Google Calendar.....	41
Figure 13. Google Calendar in a small window.....	42
Figure 14. Yahoo Calendar and Windows Live Calendar.....	43
Figure 15. Zoho Calendar.....	44
Figure 16. Google Documents with star widgets.....	45
Figure 17. Duplicate Widgets.....	46
Figure 18. Remember The Milk.....	47

Introduction

This thesis introduces a Stateful Web Augmentation Toolkit (SWAT) that gives users control over the presentation and functionality of web content.

The SWAT toolkit extends Chickenfoot, a Firefox browser scripting environment that offers a variety of automation and manipulation capabilities. SWAT allows programmers to identify data records in database-backed web sites. Records are nodes of data corresponding to rows in the database backend. Programmers can append additional functionality to those nodes, and the resulting code can be bundled up and installed by users without technical expertise.

Existing web customization tools are often hard-wired to solve a particular problem. Widgets can display a specific part of a site in a specific context. Mashups can represent content from one or more sites in a different view. Annotation tools can affix text to a page or page element. By building on a richly featured scripting environment, SWAT provides a flexible, open-ended framework for expanding the functionality of web sites and persisting information that the end user enters.

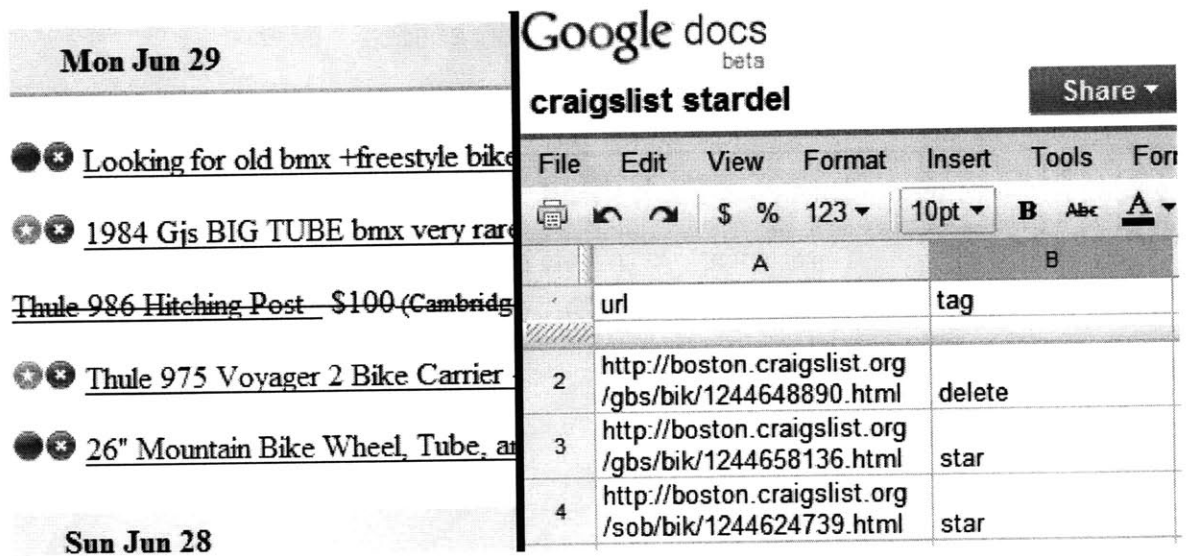


Figure 1. Augmenting Craigslist

In the image on the left, a SWAT augmentation has added two buttons to every listing on Craigslist- a favorite star button and a delete button. The star button lets users mark (or "star") noteworthy listings; here the second and fourth listing have been starred. The delete button lets users remove listings from search results. Here the third listing has been deleted.

Three modules are at work here: the site profile module determines which page elements represent Craigslist posts. The tweak module defines the look and behavior of the star and delete buttons, and inserts one of each beside each post. The storage module persists information to a data store (in this case, a google spreadsheet shown on the right).

SWAT is broken down into three modules: a Site Profile module that identifies data records, a Tweak module that defines the look and behavior of an interactive widget, and a Storage module that persists the widget state across pages and browser sessions. Default implementations are provided for each module, and these implementations adhere to an API that encompasses all communication between modules. A programmer can extend or replace any module to improve a system built with SWAT.

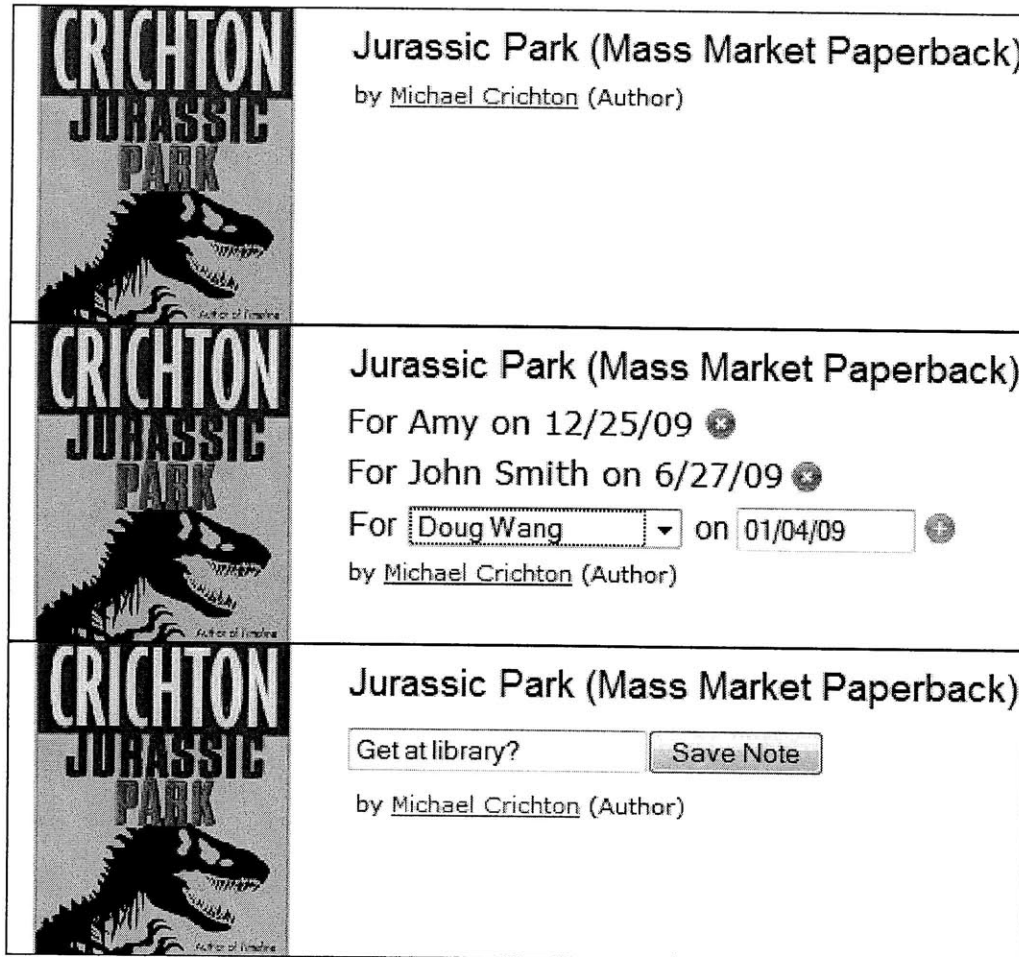


Figure 2. SWAT Modules can be replaced individually.

Here the same site profile module is mixed with two different tweak modules to modify Amazon.com in different ways. The first screenshot shows original content from Amazon.com. In the second image, SWAT has added a tweak that lets a user associate a name from their contact list and a date with a record (in this case, a book). In the third image, SWAT has added a tweak that lets the user attach a brief note to a record.

This paper describes methods for identifying records in web pages, using information from the document URL, the structure of the page's Document Object Model (DOM), identifying fields of

elements and content visible to the user. Records with detail pages can often be identified by the target URL of that page; this was found to be the most effective method across a variety of popular sites.

Special considerations are required for complex web applications that use Asynchronous Java and XML (AJAX). The frequent addition of records requires listening for changes to the DOM and updating the body of inserted widgets appropriately.

Five approaches are described and evaluated for handling clicks on a group of popular AJAX applications. These approaches include using one or more “glass” overlays to selectively divert click events, and using browser functionality stop the propagation of events. The glass overlays break some of the web applications below them, particularly ones dependant on mouse position. The systems that stop event propagation perform better, allowing more scroll and drag features to work properly.

SWAT is limited by its need to frequently re-insert the widgets that it has inserted into a page; the process of rapidly clearing and inserting is prone to race conditions and can leave pages in an invalid state. These problems could be mitigated by using a lazy reinsertion policy and clearing based properties common to all inserted widgets, rather than relying on a log of active widgets.

With SWAT, end users can customize sites far beyond where their content providers stopped, and can add functionality that logically connects different data sources, changes how and where data is stored, and redefines how they interact with the web.

Related Work

Structured Data on the Web

From the beginning, the World Wide Web has been designed to provide a large number of users with access to a large amount of data. The number of possible activities on the web has grown with the rising amount of available data and services. As user interests became more varied, a natural result was increased demand for customizing the browsing experience. Bookmarks and home pages gave users faster paths to sites that were important to them. Early tools aimed at personalizing web content changed HTML before it got to the browser, adding links and toolbars specific to a user^[4].

The structure of HTML often helps with data extraction. Data stored in the table tag, for instance, can be easily scraped if the way the table is being used is known. When the user does not know how a table is being used, the problem becomes more complicated- tables may be used for layout, information may be fragmented across multiple tables, and extraneous data (column headings, advertisements) may appear repeatedly in extracted data. Such data can be reliably classified extracted without user supervision, by estimating the structure of the data template and either building a system of logical constraints that describe that template or a system of probabilities describing the likelihood that the template has certain properties. Both have been shown to be effective in different cases; the former on clean, consistent data and the latter on data with minor inconsistencies^[13].

Database-backed web sites change the nature of information extraction. Rather than present a finite number of pages that can be analyzed and scraped, the pages on a database backed site represent queries. When a user accesses a page, a query is constructed, and data is retrieved and filled into a page template. The number of possible pages, then, is not limited by the time spent to create each page, but by the number of possible queries. For many sites, this is large enough to be effectively infinite; it is generally impractical to query every possible search string and analyze the resulting pages.

Assessing the structure of data on the web that is hidden behind a layer of queries is difficult; much of it cannot be indexed by search engines and is not directly linked from any page. This 'deep web' is estimated to contain 500 times the amount of information that the 'surface web' contains, although even estimating its size is a difficult task^[41].

One approach to determining the back-end structure of such sites is to repeatedly query a target site's search form, comparing query inputs and outputs with regular expressions to learn how the form input(s) are being used ^[2].

Just as database-backed web sites increase the number of possible pages, they reduce the variance in page structure across those pages. Templates create a de facto standard within a site: quirky variations that were once possible when pages were created individually are no longer possible when all pages are created by one process. By comparing the edit distance of DOM trees, the pages within a site can effectively be clustered into groups by format; by identifying data that varies between the pages within a group, important fields can be automatically extracted. This technique has proven very effective for extracting news articles from sites of unknown structure ^[1]. And just as repeated structure across pages gives information about important fields, repeated structure within a page (caused by templates used to repeatedly display product information or other records) can be used to identify and extract information about records ^[14]. The fact that both of these methods rely only on tag structure (and not the text content within tags) is a strong indication that DOM path is enough to identify where content is in a set of similar pages.

Mashups

The standardization of data formats also made it possible for site owners and others to create APIs and RSS feeds that provided data from news sites, blogs, and other dynamic sources of information. This meant that developers who only understood a small piece of a web site could build tools that could extract, manipulate, and represent the information in a variety of ways. This set the stage for mashups, web applications that restructure and re-present information from one or more web data sources.

Because mashups allow flows of structured information to be manipulated by third parties, they allow more innovative uses of data to come from end users. Research on mashups has focused on synthesizing information in order to display it more usefully, and tracking the state of information sources, both in real-time ^[7] and over a period of time ^[17].

Mashups can be further personalized by allowing the user to filter data, tag data or display the tagging that other users have done, and change how the data is displayed ^[16].

Existing Mashup Tools

Many of the things mashups do have traditionally been out of reach of end users, because they require a creator to understand APIs, write large amounts of code, and set up a server or database. Although the barriers have been reduced, the complexity of creating mashups still keeps away many users who would otherwise benefit from being able to do so^[15].

A number of tools have been created recently to make the process of creating mashups easier for users. Most focus on making the experience easier for nontechnical users, by creating a simple drag and drop UI that allows flows of data to be directed from sources to sinks in a customizable network. Yahoo Pipes^[19] allows arbitrary feeds of data to be fed into data visualizations, which can be selected from a finite list of options in a toolbar. Microsoft Popfly^[18] offers similar features, and lets users develop more complicated modules in visual studio to be plugged into the network of flowing data. Marmite^[5] extracts content from web pages and displays that data in spreadsheet view, making data transformations clearer for the user. Mashmaker^[6] is designed to aid development with collaboration/sharing features and suggestions for data extraction.

Limitations of Mashups

Because the focus of so many existing mashups is on presentation, almost all existing mashups are stateless: they offer a view of data that is independent of the data itself. By design no element of the source data, be it posts on craigslist or photos on flickr, is preserved in the mashup. This limits the space of customizations to a minimal set of choices: what site to monitor, what terms to search for, how often to update, and so on. Mashups that allow users to enter data to a collection pass that data to the underlying site, and cannot handle data that the site was not designed to take.

Other Existing Technologies

There are other web content enhancement tools besides mashups. This section describes three classes of such tools and discusses how this project is different from these existing technologies.

Annotation Tools

Annotation tools let users attach notes to arbitrary web pages. Notes can be anchored to a pixel position or an arbitrary HTML element, and the state is retained by the tool itself, stored in the browser or on a third party server.

Annotation tools often have a social aspect; where comments on pages can be shared with the public at large^[23] or a select group of friends^[24]. Annotation tools most commonly store free text, and do not

change their functionality across sites or users. Like annotation tools, SWAT can be used to associate text with pages or elements on pages. SWAT expands the functionality of annotation tools by enabling site-specific behavior and the input of structured content.

Widgets

Widgets are small, self-contained displays of web data that are easily installed/embedded. They can provide a small view into many kinds of data—unread emails, today's news stories, weather, etc.

Most modern web portals, including iGoogle^[25] and My Yahoo!^[26] have a prominent section where users can freely add, remove, and drag widgets around the screen. Widgets also appear in Mac OSX^[27] and in the Vista sidebar^[28] (branded there as “gadgets”).

Widgets often provide some small level of end-user configuration, usually focused on a search string, category/region of interest, or display properties like size and color of output. Most of the control, then, remains with the creator of the widget, and as with mashups, widgets offer a view of data without retaining or allowing special operations on that data; that is, they are stateless.

Like widgets, SWAT can provide a configurable view into different sources of information. SWAT expands the functionality of widgets by allowing end users to persist data.

Web Scripting Environments

Web scripting environments are powerful, general purpose tools that let users manipulate Javascript and HTML nodes by writing scripts to interact with a web page. Existing products like Greasemonkey^[22] supply large sets of tools with broad capabilities. SWAT is built on a web scripting environment called Chickenfoot^[21]. SWAT allows access to all existing Chickenfoot functions, but also supplies a framework for augmenting web pages in a structured, standardized way.

Approach

This section describes the design of the Stateful Web Augmentation Toolkit, risks of this design and attempts to mitigate them. In the course of implementing SWAT, some adjustments and corrections were made; they are outlined in the Implementation section later in this document.

The SWAT toolkit is built on Chickenfoot, a browser scripting environment that offers a variety of automation and manipulation capabilities. SWAT allows programmers to identify data records in database-backed web sites. Records are nodes of data corresponding to rows in the database backend. Programmers can append additional functionality to those nodes, and the resulting code can be bundled up and installed by users without technical expertise.

Strictly speaking, the functionality of browser extensions created with Chickenfoot and SWAT is the same as the functionality of Chickenfoot scripts without SWAT. The scripted pattern detection, DOM (Document Object Model) manipulation and file I/O in SWAT uses existing Chickenfoot faculties. The important new feature SWAT introduces is a group of high-level abstractions that enable the conception and creation of interactions with data records on database backed web sites. The SWAT toolkit helps users create such browser extensions in a standardized and modular way. These abstractions are described further in the Modules section of this paper.

Design Goals

This project is motivated by the following design goals:

Modularity

The code for SWAT is separated by interfaces into three modules; each module has several default implementations supplied. As programmers use the system to make browser extensions, they can switch out modules independently, often with a change to only a single line of code. By isolating code, less useful parts of the system can be separated and replaced without throwing everything out or re-architecting the system.

Flexibility

Instead of a library that is highly optimized for a few narrow use cases, SWAT aims to be a general purpose platform. Although it provides a few sample implementations of widgets and storage mechanisms, SWAT is a robust framework that can accommodate many pieces beyond those provided.

Short of replacing provided widgets and storage mechanisms, programmers can use hook points to redefine select parts of their behavior.

Simplicity

Any tool like this one will be daunting for some users. SWAT provides a simple and intuitive programming interface/user interface, so creating and using browser extensions is an easy and transparent process.

Risks

Five risks were considered during the design of the SWAT system. This section describes those risks and plans made to minimize them. Later in this document, the Evaluation and Analysis sections describe the problems that were actually encountered and ways of dealing with them.

Feature Creep

There are many ways to extend this project to new kinds of functionality, so an effort was made to limit the project scope to a subset of the possible features. This kept implementation time bounded and kept the project focused on answering the research questions at hand.

Saturated Space

Web research is a popular field, and many other people have done work that has some functional overlap with this project. To ensure that the tool produced is a valuable and unique one, existing work was considered before and during development.

Data Obfuscation

SWAT can only work if extensions can get data from the web sites they aim to enhance. The structure and presentation of that data is wholly determined by the content providers who operate web sites.

Content providers have an incentive to provide data to the users who access their site. But they may not have any incentive to provide tools like SWAT with data about the layout and structure of their site. In some cases, they may deliberately obfuscate information that a browser extension would need, for any one of several reasons— security, keeping data about their business model away from competing web sites, and avoiding unflattering price comparisons.

Performance

An interface that allows programmers to interact with database-backed web sites as if they were databases will allow many database interactions that were not previously possible. Rather than

operating on the data presented by a single page fetch, as many web tools do, this could result in arbitrarily complicated operations as programmers try to join or merge tables. If this data was retrieved on the fly, enormous sets of data might need to be fetched and combined, resulting in a sluggish and unsatisfying end-user experience.

This is a risk that is partially borne by the programmers who use SWAT; if it is flexible and transparent in the way it handles data, programmers will be able to choose how they want to get data, and more clever implementations will have better performance.

Security/Privacy

Storing user information is a sensitive job, especially if some end users are using SWAT to manage their data because they don't trust content providers with it. SWAT must keep storage mechanisms and privacy options transparent to both programmers and end users.

Another privacy concern, relevant to the second use case outlined earlier in this document, is that data injected into the DOM can technically be accessed by the content provider, who can examine the page and post data back using AJAX. This problem could be addressed by giving the end user control over when a page could post data, although this would slow or break many rich AJAX sites. The problem could also be addressed by posting new content in a layer outside of the page (but graphically overlaid above the page), although this would make it hard to position added content in-line with the content of the page. For the purposes of this project, this privacy concern was considered out of scope.

Target Audience

SWAT is useful to two sets of users: Programmers and End Users.

Programmers are users with an intermediate or high level of technical skill; they understand what mashups do and would be able to use existing mashup toolkits without much difficulty. They can use SWAT to create browser extensions and make them available to End Users.

End Users are users with a low level of technical skill; they are able to browse the Internet and understand the basic functionality of the sites they visit, but need not understand HTML, Javascript, etc. They install browser extensions to use the features that programmers create.

These two groups of users closely reflect the groups that interact with existing mashup tools. Hopefully the same users who make mashups using existing tools will be willing and able to make browser

extensions with SWAT, and the same users who view the resulting mashups will be willing and able to use those browser extensions.

Implementation

This section describes the process of implementation; the problems encountered and adaptations that were made. There are four sections; first, Modules introduces the three key pieces of every SWAT browser extension. Then, an Annotated Example Script shows how these pieces fit together. Early Development deals with uniquely identifying information and adjusting the module APIs. Finally, Developing for AJAX Applications discusses the challenges that more complicated interactive web sites bring as they add functionality once reserved for desktop applications.

Modules

SWAT consists of three modular components: one to identify records in pages served by database-backed web sites, one to embed widgets in those sites, and one to manage storage of data that end users create. This section outlines how each of these modules work. More information about their implementation can be found in the

Implementation section.

Site Profile Module

To build an extension for an existing web site, programmers first need to identify what parts of the site are relevant to their extension. To do this, the site profile module can examine any information about a page that is available in the browser. Currently implemented site profiles use page URLs, content on the page, and the structure of the DOM to identify records on a page.

The part of the page that a programmer identifies represents the manifestation in HTML of a record (row) in the site's back-end database. Just as a database row holds different information about a record in each column, the HTML node holds a number of fields of structured information corresponding to the record it is displaying.

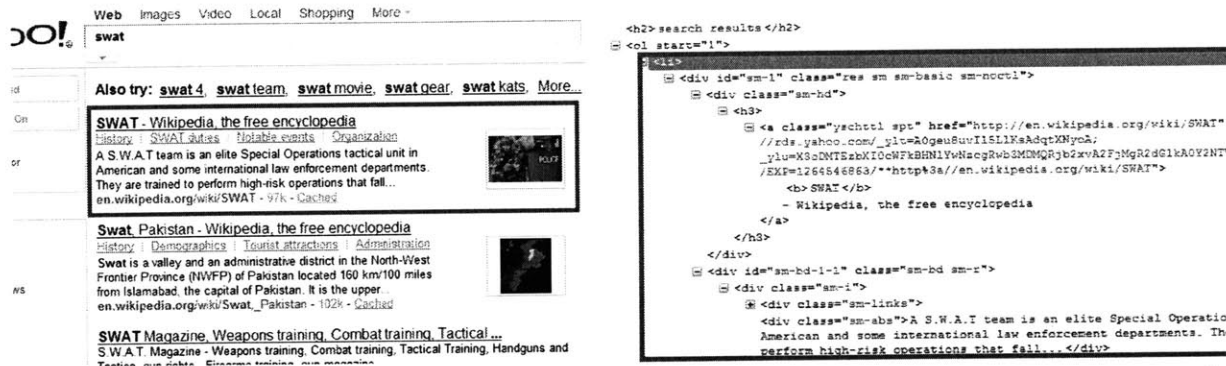


Figure 3. Yahoo Search result and DOM subtree.

On the left, the first search result is highlighted with a red frame. On the right, the corresponding subtree in the DOM is highlighted. The fields that make up the data record can be reconstructed from data in the DOM subtree.

A programmer does not need to identify and store all information related to a record, only the pieces that are relevant to the task at hand. Within a record, a programmer can specify relevant subfields with relative or absolute paths in the DOM subtree. Fields may be transformed as they are extracted from the page, for instance, long description strings may be truncated, and rating information may be converted from an image to an integer.

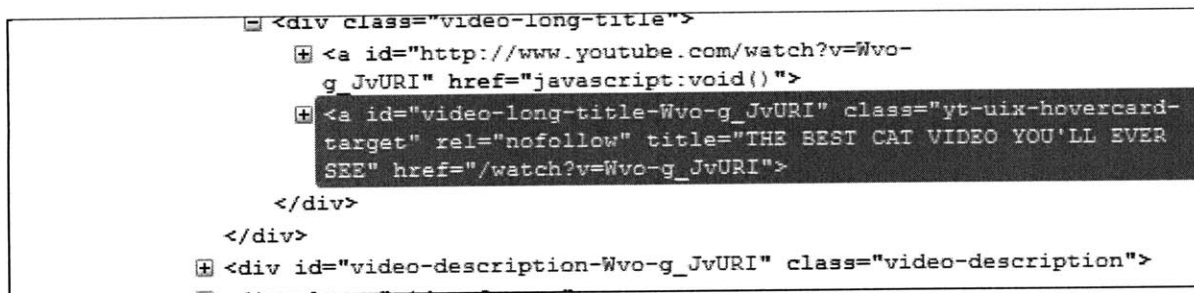


Figure 4. Identifying a Youtube Video.

This image shows a portion of the DOM on a listing of youtube videos. Here the link to play a video contains a unique identifier for the record in the anchor's destination. The identifier for this video is "Wvo-g_JvURI".

The programmer must also identify one field that will uniquely identify a record, corresponding to a primary key in the assumed data model. This field may consist of data that is transformed into a new format. It may also be created from the amalgamation of several other fields, just as some databases use multiple columns to define a table's primary key.

Using this information, SWAT can be used to locate nearly any regularly-formatted records present on a page, and separate those records into fields.

Tweak Module

Once programmers have clearly defined what it is in a web site that they want to manipulate, they need to be able to specify how they want to change that data. This module gives the users the ability to associate a widget with a record, or some subset of the instances of a record, based on rules that take into account data in that record, its position on the page, end user input, etc.

To create a flexible development framework, widgets must be as flexible as possible. Developers should be able to create their own widget to store any kind of data. To make the system easy to get started with, though, it is important to provide developers with a few preset widgets that they can use out of the box. To that end, SWAT provides default implementations of these widgets:

- Text Field – attach arbitrary text to a record. This reproduces the functionality of existing annotation tools. This widget allows hyperlinks.
- Favorite Star – flag or ‘star’ an item with a small star icon. Because it is simple and takes up very little space, this widget was used for evaluating sites.
- Pick List – pick from a list of options presented in a drop-down style. This can be populated with values set by the programmer or end user, or those values could be drawn from an XML feed or a file on the user's system.
- Date – associate a date with a record.

SWAT could be extended to include additional widgets in the future:

- Time/Time Range – associate a time or time range with a record.
- Rating Tool – rate a record.
- Image – associate a graphic with a record.
- Chart / Visualization / Map – display data in a more complicated view.

Each tweak module also has a connect function, which lets it create a mapping between its own fields (the state of the buttons or drop-downs it inserts), the fields in the site profile module, and fields in the data store module.

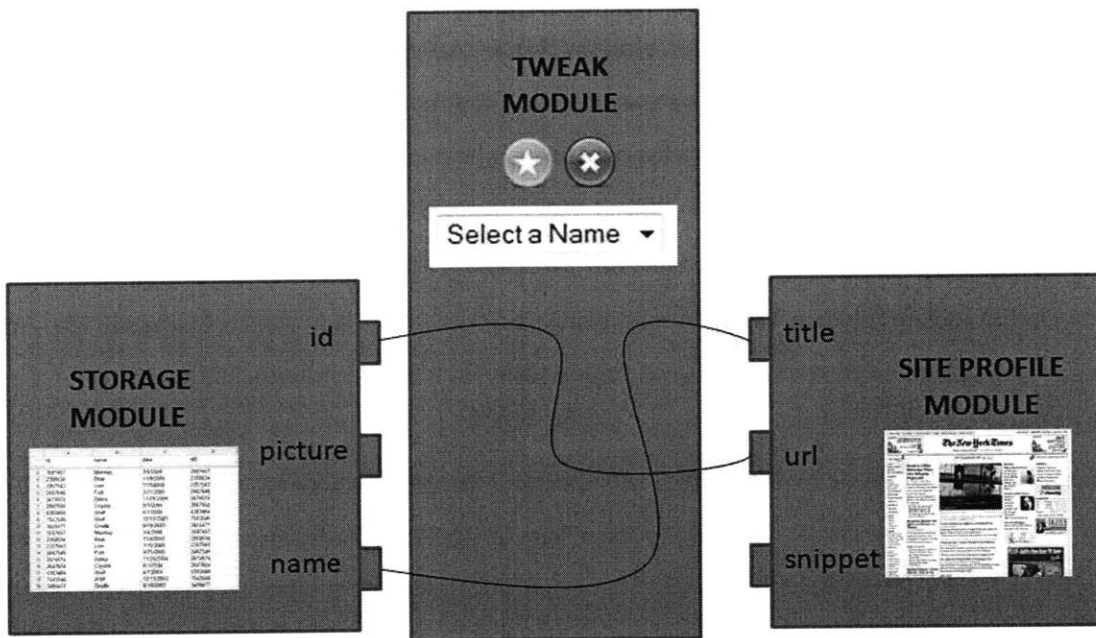


Figure 5. Interaction of SWAT Modules.

This diagram shows the behavior of the tweak module's connect function. The connect function creates mappings between fields in the data store module, the tweak module, and the site profile module.

Data Store Module

When end-users use a widget to add data, SWAT stores that data. Storage is a modular part of the SWAT toolkit, so programmers can decide for themselves where the data that is added by their end users should be stored.

The data store module can also interface with information from a source outside the content provider's database, such as an instant messenger buddy list or a schedule posted online. This information can be used by a tweak to make a widget with a drop-down list of friends or appointments.

As with the tweak module, the data store module is flexible enough that developers can create their own storage mechanisms. SWAT also provides a few preset ones to make early development easier. Out of the box storage mechanisms that are provided are:

- Comma Separated Values file (CSV) – Stores a flat file on the local filesystem.
- Google Spreadsheet – Stores data in the cloud. This makes it easy to add records, sort and search during development. Some end users might appreciate these features as well.

In the future, storage modules could be implemented to supply additional storage mechanisms:

- Hosted database – A hosted database could be a very flexible solution, but this would put a large burden on developers who didn't already have a server running to support their extension.
- Hosted database service – With a storage service, like Amazon S3, the server would already be set up, but using it would require a paid account, something that would turn off some developers.
- Within the Target Web Site – Some sites have free text fields designed for comments or notes that could be used to store XML or comma-separated values. The storage cost would be low but this might disrupt some users who wish to use such fields for their intended use.
- In-Browser – This is the safest and most private, keeps data tied to the browser. The end-user can't access their data on other computers (or on other browsers).

Annotated Example Script

This section will go over functionality of a working script built with the SWAT library. The script displays favorite stars next to videos on youtube search listings and on pages on which the videos play. It runs each time a page is loaded whose URL begins with a certain prefix (eg, "http://www.youtube.com/")

```

1 // youtube_eval.js
2 base_url = "c:\\chickenfoot\\"
3 include(base_url + "common\\swat_library.js")
4
5 // Data Store
6 my_store = make_csv_data_store(base_url + "youtube_store.csv",
7 ["url"])
8
9 // Site Profile
10 my_site = make_site_profile_youtube_search()
11 my_site2 = make_site_profile_youtube_player()
12
13 // Tweaks
14 my_tweak = make_tweak_star()
15 my_tweak.connect("unique_id", my_site, "url", my_store, "url")
16 my_tweak.connect("unique_id", my_site2, "url", my_store, "url")
17 my_tweak.run()

```

Lines 2-3 define the location of the library on the local filesystem and include it.

Line 6 creates the data store. In this script, this data store is a CSV file stored locally. The location is a string indicating the path to the CSV file (here the CSV happens to be near the SWAT library; it could be anywhere). The second argument defines the headers for the stored data. Here there is only one: the URL of a video.

```
1 URL
2 http://www.youtube.com/watch?v=s13dLaTIHSg
3 http://www.youtube.com/watch?v=nTast5h0LEg
4 http://www.youtube.com/watch?v=J--aiyznGQ
5 http://www.youtube.com/watch?v=SN1VcgRrEM8
6 http://www.youtube.com/watch?v=w0ffwDYo00Q
7 http://www.youtube.com/watch?v=TZ860P4iTaM
8 http://www.youtube.com/watch?v=kvBiSW5QFKY
```

Contents of the data store file, youtube_store.csv. This file has only one column, but any number of columns is possible.

Lines 9-10 create the site profiles. This script display stars on pages of two different formats: youtube search listings and pages on which videos play. Since records appear differently on the two kinds of pages, the script uses a different profile to find videos in each format. The browser page is not touched at this point; these profiles don't extract data from the page until the tweak's run function is called (line 16)

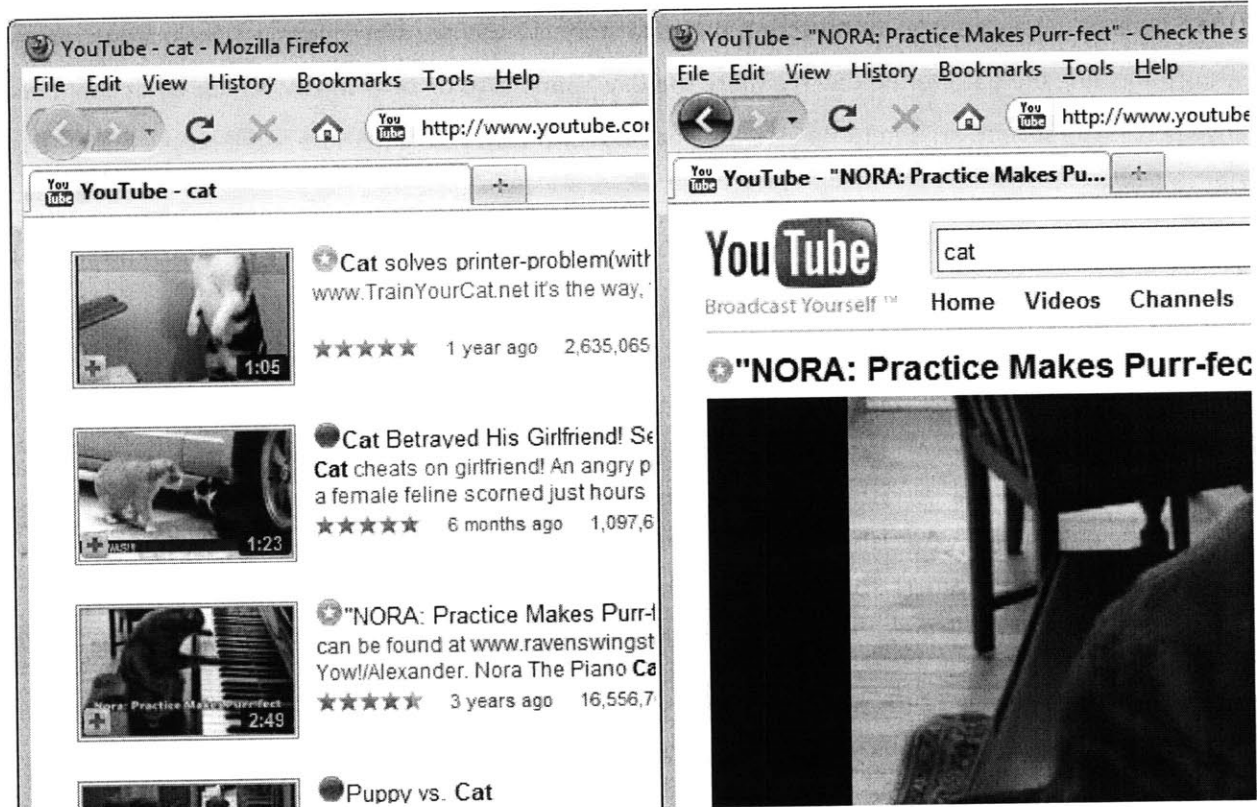




Figure 6. Favorite Stars on YouTube.

The example script adds favorite stars (which appear as  or ) to videos in YouTube's search results and video player page. A record starred in either view will be starred in the other.

Line 13 instantiates the star tweak. This object comes with a connect and run function.

Lines 14-15 connect the modules together. The star tweak is to be mapped to the two site profile modules. In each case the same data store will be used to store results. This means videos starred in the player will be starred in the search results, and vice-versa. Further, the star's unique ID field is to be mapped to the URL field in each of the site profiles, which in turn is to be mapped to the URL column in the data store.

Line 16 runs the tweak. The tweak tries to match each site profile against the newly loaded page content, searching for matching records. Each of those records is in turn located in the data store. A star widget is created and given properties that represent the state of the record in the data store. In this case, a stored record appears as a highlighted yellow star (); all others appear as grayish blue circles (). Click listeners are added to the widgets so they can respond to user interaction and persist changes to the data store accordingly.

With the modules pre-defined, only these nine lines of code are necessary to seamlessly make a substantial change to a web site. Better still, minor changes to this simple script can swap out preset modules to change the front-end or back-end.

For more information on the module APIs that this script uses, see Appendix A: API Description.

Early Development

There are many ways that records on the web could conceivably be identified - by a unique ID field, a combination of fields that could uniquely identify an element, and by a detail page that it linked to. It was this last approach, finding a record-specific link, that was the most fruitful. An informal survey of several sites in each category found that news sites, video sites, search engines, product search results, social networks and document editing sites, in the overwhelming majority of cases, had a unique URL associated with records that could identify the content. The following table shows some examples of these URLs:

Site	Type of Content	Record	Example URL
bbc.co.uk	News	Article	http://news.bbc.co.uk/2/hi/south_asia/8478386.stm
www.hulu.com	Video	Video	http://www.hulu.com/watch/110395/the-simpsons-meeting-his-hero
bing.com	Search	Search Result	http://en.wikipedia.org/wiki/Cat
buy.com	Shopping	Product	http://www.buy.com/prod/sandisk-4gb-cruzer-usb-2-0-flash-drive/q/loc/101/210758047.html
facebook.com	Social Network	Profile	http://www.facebook.com/wmatthew
docs.google.com	Document Editing	Document	http://docs.google.com/Doc?docid=0AZn_E9DEIuTCZGhieDQzY21fNTlka24ycm1jag

One consequence of this was that many sites had two kinds of pages- listings of records, sometimes in set lists and more often appearing as search results, and detail pages, devoted to a single record. The URLs of detail pages generally started with a special prefix (such as 'http://www.youtube.com/watch?'), which made detail pages easy to detect.

In its original conception the site profile module was a single entity that could find content on any page in a domain. For example, in the example script in the previous section, a single site profile would identify content on YouTube search results and YouTube player pages. This approach resulted in bloated site profiles for complicated sites; if a developer wanted to create a browser extension that only affected lists of search results (for instance, to inject Hulu links into YouTube search results), he or she would be burdened by SWAT code dealing with identifying videos on YouTube player pages.

To address this, site profiles were redefined to focus on one particular view of site content. One site profile would handle finding and parsing search result records, and another would handle player pages.

To make multiple site profiles work, the tweak module's connect function had to be redefined to allow it to be run multiple times. Rather than storing a single profile and data store, it would now hold a list of profiles, each with associated fields mapped to corresponding fields in the tweak.

As code was improved, testing entailed refreshing a web page and re-running the script, to see if a new feature worked correctly. If the page was not refreshed, extra copies of widgets would get inserted, doubling and tripling up, and generally messing the page up. It was frustrating to not have a programmatic way to deal with this. Programmatically keeping track of the widgets that had been inserted into the page in an array improved the situation; with the addition of a clear function these widgets could be removed at the beginning of a script to restore a page to its original state.

The clear function was later added to the site profile API, and the general philosophy was adopted that whenever possible, code should be able to be run again any number of times without causing undesired results. This made things a lot easier later on, when pages got more complicated and the number of things triggering a reinsertion increased.

Developing for AJAX Applications

Once you know where records are and how you want to change them, augmenting a static web page is simple. You insert widgets into the DOM, add a listener to those widgets, and respond to events when widgets are clicked. This works well for sites like Craigslist and Google Search, where content displayed in the browser only changes when the user navigates to a new URL.

Unfortunately, recent years have seen the rise of a more dynamic kind of web page, characterized by the ability to asynchronously request server data and display it to the user without the user navigating to a different page. This development technique, called AJAX (Asynchronous Javascript And XML), has

profound implications on the functionality that a web page can provide. This document refers to web pages using AJAX as AJAX web applications, or just AJAX applications. Pages that do not use AJAX are referred to as static web pages.

With the ability to request data that is stored remotely, AJAX applications are freed of some of the constraints of browser memory size and bandwidth. Dealing with large datasets is now faster and easier, as the page only needs to store and display a subset of the data (the part the user is currently looking at). By prefetching the right data, AJAX applications can give an experience that is almost as good as desktop applications, with less waiting for pages to load and render. Sometimes the user experience of an AJAX application is even better than that of a desktop application, as application backends are limited by the size of data centers, not personal computer hard drives.

Challenges of Modifying AJAX Applications

Ephemeral Content

What implications does the rise of AJAX web applications have for this project? A user can load a static page and read its contents in their entirety before navigating to another page. An AJAX application lends itself to a more continuous form of browsing: fluidly transitioning between searching for records, reading their details, and creating or deleting them without changing pages. As such, records are constantly being added and removed while the browser displays a single page. The initial page load is no longer an all-important milestone; the page must be analyzed when other events occur.

Listeners can be added for DOM modification events, and when a node is added or changed, the page can be reconsidered and updated if necessary.

To keep from doing unnecessary work, some activities are only done the first time they are needed. For example, establishing a connection to the data store is done once per browser tab session, the first time the data store is accessed. Most other activities- reading over the page, inserting widgets, adding listeners- must be done many times over if a user navigates through a lot of data on one AJAX page.

Event Handling

Crucially, AJAX web applications tend to employ a complex set of UI components. Instead of letting the user navigate through links and use the occasional print or email button, AJAX applications must have the drop-down menus, scrolling areas, dialog boxes and so on of desktop applications. This requires a large number of event listeners. These listeners form a complex ecosystem of event handling- listeners

receive clicks in a specified order defined by the position of their elements in the DOM, and some listeners may stop the propagation of an event or modify it, affecting other listeners downstream.

Augmenting web content means adding new page elements and listeners into this ecosystem. This must be done carefully. If a click is acted upon by SWAT widgets and the underlying web application, the resulting behavior- doing two things with one click- will create a confusing and frustrating user experience.

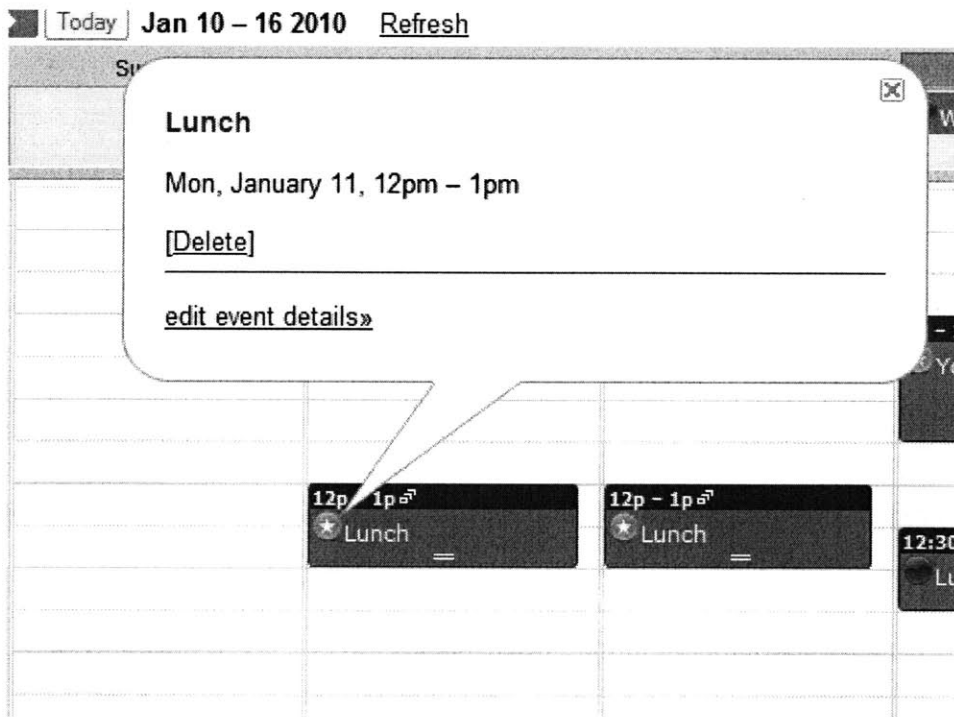


Figure 7. Accidentally Doing Two Things With One Click.

Clicking a widget to star a record toggles the star and triggers an unwanted response from the application.

Figure 7 illustrates one example of an AJAX application and SWAT responding to the same click. Here a star widget has been embedded into each appointment in Google Calendar. SWAT has added a listener to each star widget and Google Calendar has added a listener to each appointment. If a user tries to click a star, this click changes the state of that star, but it also triggers the containing appointment's default click behavior. In this case, that behavior creates a large popup that obscures the rest of the calendar. The popup's close button is small and far from the cursor position, yielding a frustrating experience. Worse yet, the standard action that would normally close the popup- a second click in the same spot-

would both close the popup and click the widget again. The star represents a boolean state, so a second click reverts the change made by the first.

To differentiate it from other event handling techniques described later in this document, this naïve approach of inserting listeners that allow event propagation will be referred to as the Leaf Listener system, as it adds listeners to DOM leaf nodes (widgets).

Positioning widgets outside of a record's target click area is a losing proposition, as the area around records often has a click listener of its own. In the case of this calendar application, clicks anywhere around an appointment trigger the creation of new calendar appointments at the corresponding day and time.

This project seeks to find a general-purpose solution for modifying web pages; a solution that involves changing behavior by adding widgets and listeners without breaking existing functionality. The coming sections describe a variety of approaches for the event handling problem and discuss their strengths and weaknesses.

Four Event Handling Designs

This section describes the design of four event handling systems that are aimed at addressing problems described in the previous section: Glass Pane, Multiple Pane, Root Filter and Leaf Filter. Each system is a different implementation of the Tweak module. The other modules and the interfaces between them remain unchanged.

These systems are intended to solve technical problems while preserving as much functionality of the underlying AJAX applications as possible. Each system is general enough to be applied to many AJAX applications without modification.

SWAT's modular design makes it easy to implement a module in multiple ways and compare the efficacy of the different implementations. Each system described here was implemented and evaluated; the results are described in the Evaluation section of this document.

The Glass Pane System

The problem with the leaf listener system is that it allows the underlying web application to act on clicks it isn't supposed to receive. A straightforward way to keep a web page from acting on clicks is to catch all clicks, trapping some and passing the rest to the web application's event listeners.

SWAT can capture all mouse clicks by creating a “glass pane” – a transparent div element that covers an entire page. When clicked, the pane extracts mouse coordinates from the mouseEvent and decides if a widget has been clicked by comparing widget bounding boxes to mouse coordinates. To do this, the glass pane needs access to a list of widgets that have been added to the page. This list is created when widgets are added to the page during the tweak.run function.

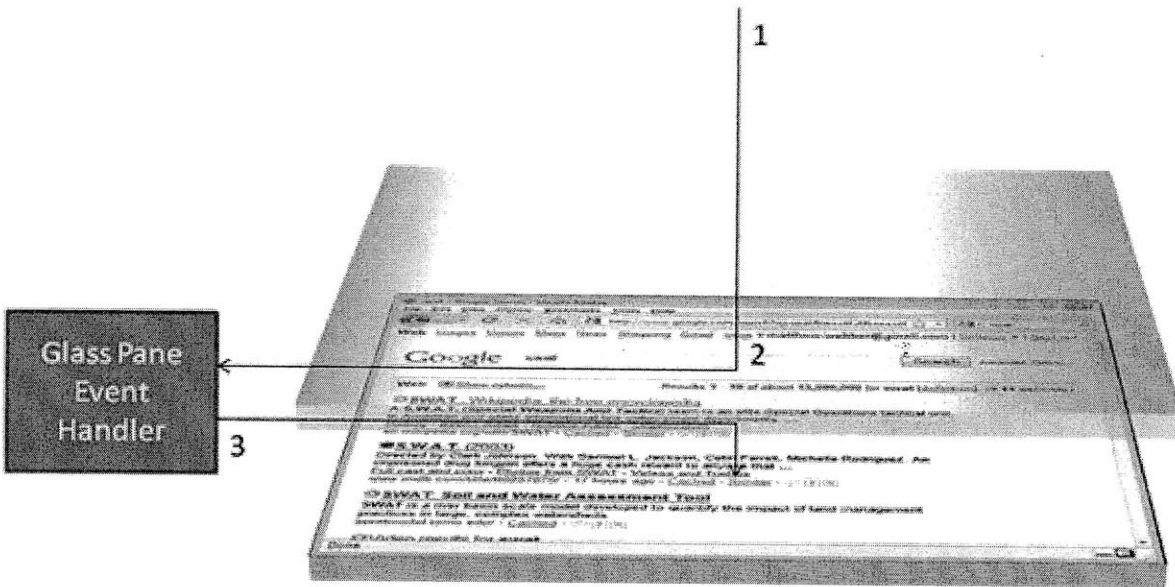


Figure 8. Operation of the Glass Pane.

This diagram shows how the Glass Pane approach handles click events. 1. The user clicks a point on the screen. 2. The click lands on the glass pane covering the page and is fed into the glass pane’s click handler. If the click is over a SWAT widget, the handler calls the appropriate widget function directly (not shown). 3. If the click is not over a SWAT widget, the handler creates a second click event and dispatches it to the appropriate target element.

When the glass pane finds a widget whose bounding box contains the click coordinates, it directly calls the function that that a click listener on that widget would have called. If the glass pane finds no such bounding box, it creates a second click event, copies appropriate attributes from the previous click event, and dispatches the second click event directly to the element at the coordinates of the original mouse click.

Glass Pane: Dealing with Obscured Widgets

The glass pane system addresses the problem of double-counted clicks; with a glass pane clicks on a widget will only trigger that widget, and not the AJAX application. Unfortunately, when widgets scroll out of view or are obscured by part of an application, the glass pane does not recognize them as hidden.

Clicks at the widget's location are intercepted by the glass pane and passed to the widget. If a button or link is obscuring the widget, the button or link will appear unresponsive, while the (invisible) state of the widget changes with each click.

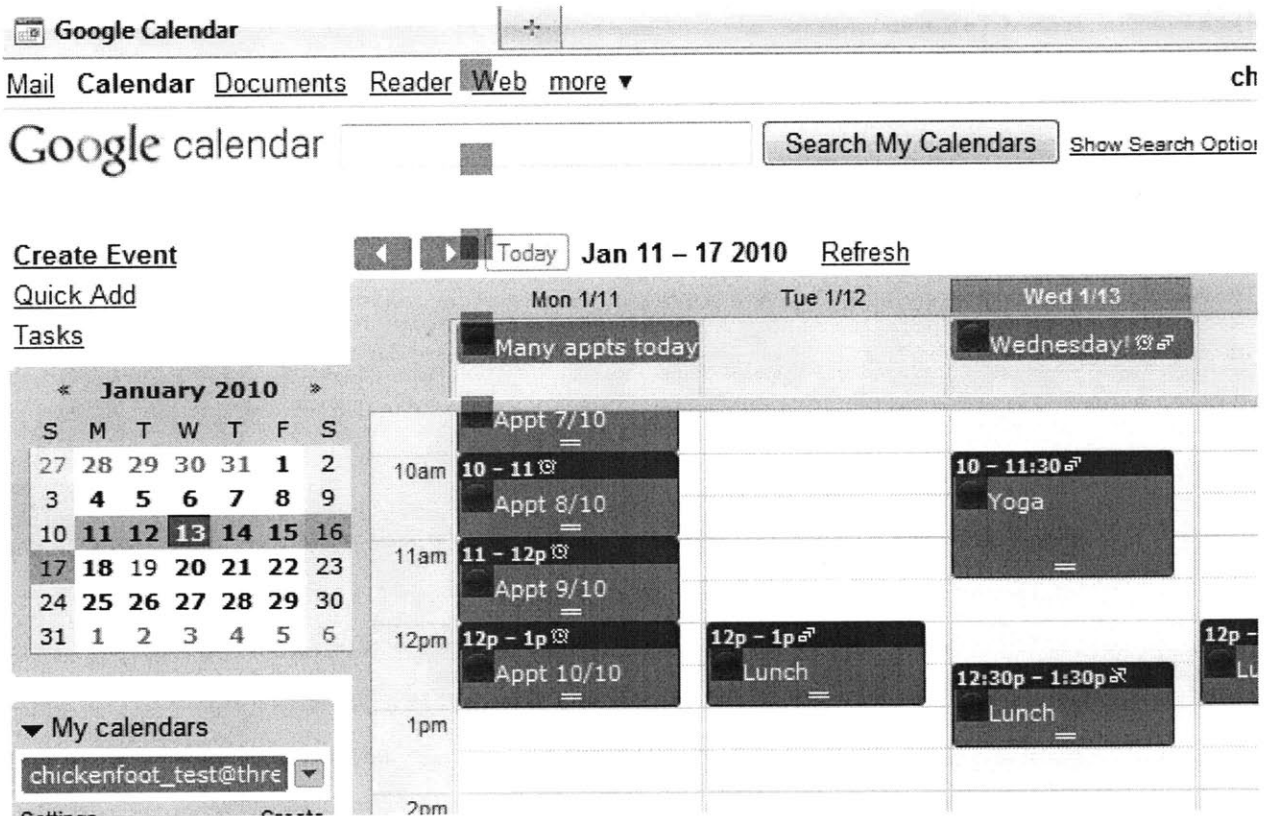


Figure 9. Hidden widgets.

Areas where clicks will be caught by the glass pane are highlighted in blue for emphasis; they are normally transparent. Note the interference with textboxes, links and buttons in the calendar application.

This problem could be remedied by measuring the bounding box of each ancestor of a widget element. The intersection of these bounding boxes would determine the boundaries of the visible section of the widget, and would enable the glass pane to distinguish between clicks on visible and invisible widgets. This remedy was not implemented in this version of the SWAT toolkit.

Glass Pane: Detecting Mouse Position

Another weakness of the glass pane system is the need to duplicate mouse events that it intercepts when those events do not affect a SWAT widget. If mouse position information is not properly replicated, clicking on a spot on the page will have an unintended effect.

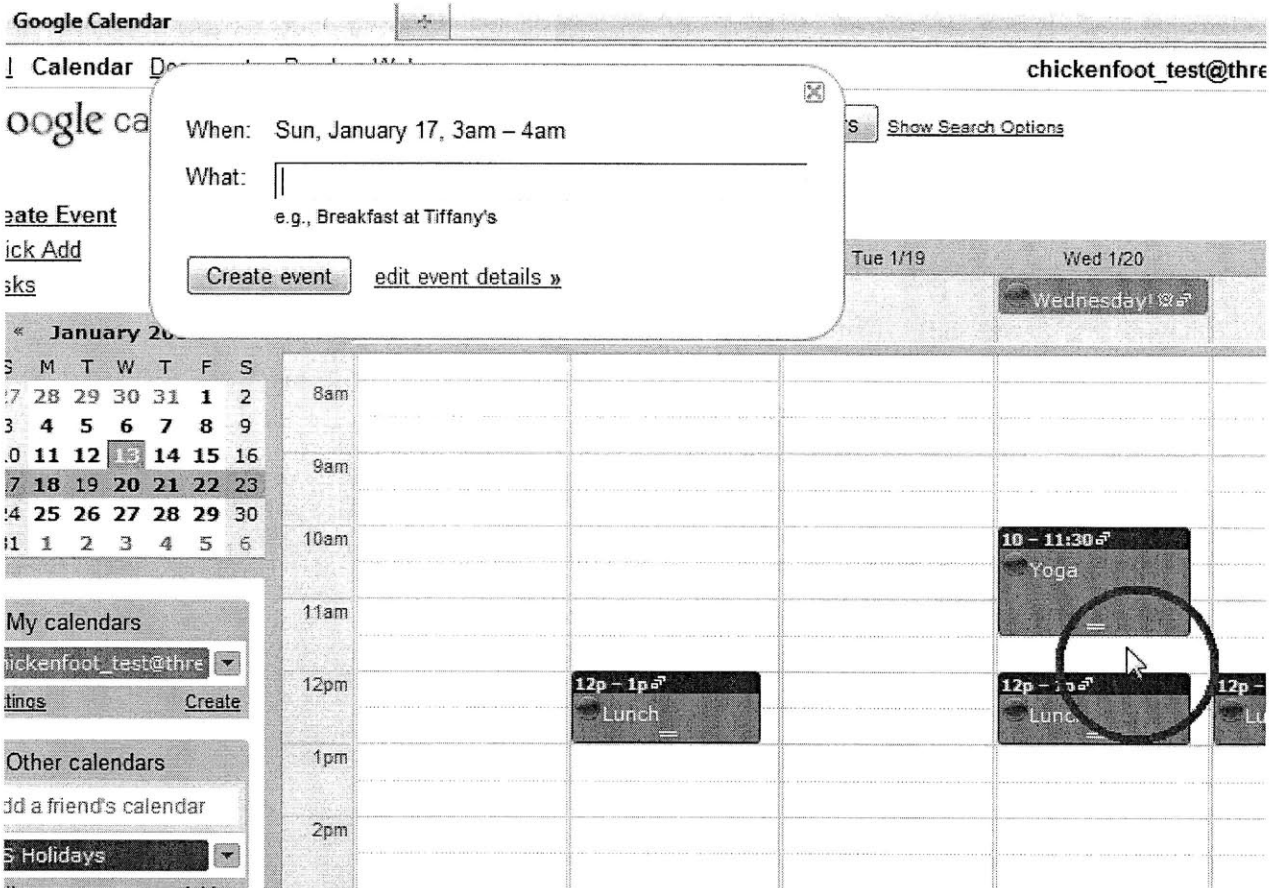


Figure 10. Event Creation Error.

Here a click immediately below the Wednesday ‘Yoga’ appointment (red circle added for emphasis) is not replicated with appropriate coordinates. The calendar application assumes the click occurred at the far upper left of the calendar pane. The application positions the creation window and populates the date and time accordingly.

In the current implementation of SWAT, page and screen coordinates are copied from the original click to the duplicate, but mouse position is not always properly replicated. This breaks a number of features; some menus and record creation mechanisms do not work properly. For instance, Figure 10 illustrates the result of clicking a spot on Google Calendar to create a new appointment. Without coordinates, the application creates an appointment with an incorrect day and time (Google Calendar derives these fields from the position of the mouse click).

Without accurate information about the mouse position of events, it is also not possible to capture and transmit scroll and drag events. Application widgets that require mouse dragging, like scrollbars, sliders and moving calendar appointments are not currently possible with the Glass Pane system, with one exception: popup messages tend to be added after the glass pane and appear above it; they receive all

clicks intended first and react to clicks appropriately, even if there is a widget embedded in the page below the popup.

Multiple Glass Panes

In the Multiple Glass Panes system, SWAT inserts one glass pane for each widget. Rather than spanning the entire page, these panes only cover their widget's bounding box. This system is designed to minimize interference with mouse events that are meant for the AJAX application- these can go directly to the target page element as they would normally.

To position the glass panes correctly, the widget is first put into the flow of the document. The widget's position is measured and used to place the glass pane. Then a listener is added to the glass pane which triggers the appropriate method in the widget below.

As with the single glass pane, the problem with clicking obscured widgets remains: a widget that has scrolled out of view can still be triggered erroneously, without any visual indication.

Multiple Glass Panes: Browser Features

While maximizing and restoring window size can be handled by listening for window resize events, continuously resizing the window (triggering a constant stream of resizes) causes a few display jitters as the glass panes are repositioned over and over. Occasionally this leaves the glass panes in a slightly incorrect state, a few pixels away from where they should be. This is undetectable and frustrating, though the "two widgets" method described in the variants section below can be used to improve the experience to detectable and frustrating.

Multiple Glass Panes: Scrolling and Dragging

By allowing the majority of mouse events directly access to the web page, web application features work better with the Multiple Glass Pane System than with the single glass Pane system. In addition to all the web application features that worked with the single glass pane, mouse position is now correctly set. Dragging works properly, so page elements can be moved where allowed by the web application.

Scrolling presents several difficulties. In its current state, SWAT monitors the top-level window object for scroll events. Using an interior scrollbar (to scroll a content page but not the entire web page) does not trigger a scroll event for this object, so glass panes and widgets get out of sync and stay that way until something triggers an update. Mouse scrolling can be caught by listening for a DOMMouseScroll event, but as with page resizing, the high frequency of repositioning glass panes sometimes leaves them out of

sync. When the mouse is over a glass pane, the pane absorbs scroll events, and though the `DOMMouseScroll` still fires and triggers an update of the SWAT widgets, it does not move the calendar internal window. Thus mouse-scrolled content stops dead in its tracks when a glass panel slides under the cursor.

Multiple Glass Panes: Variants

One temptation is to simplify click passing by putting the widget in the flow of the document, measuring its position, then moving it onto its glass pane where it can receive clicks directly. But the widget needs to stay in the flow of the document so that HTML elements are pushed aside to make space for it, and so that page resizes and content scrolling affect the position of the widget.

A modification that works a little better is to switch the place of the widget and glass pane, so the widget can directly receive clicks and the glass pane can fill space in the document.

Yet another possible variant is using two widgets, with a listener on the top widget and both reflecting the state of the corresponding record. The advantage here is that if the two layers get out of sync, it is immediately obvious (since neither layer is transparent); the failure is easier to diagnose but no easier to resolve.

The Root Filter System

The idea of a glass pane is to introduce a filter that can stop certain events from propagating and allow others through. There is already a point that all events go through- the document root node. From there, events propagate down to the target element, then bubble back up through the document hierarchy to the root.

By adding a single event listener to the root, all click events can be monitored. The listener can stop the propagation of events that are heading toward one of our widgets, and let the rest through. Since events naturally pass from the root to child subtrees of the DOM, this system does not need to simulate the click- it can just allow it to continue where it is already going. With only one listener, this event handling system requires the fewest listeners and DOM modifications, and can be implemented in the least number of lines of code.

The Leaf Filter System

One concern with the root filter is that by passing all events through a single function at the root node, SWAT creates a single point of failure (or performance degradation) that could affect all clicks, including

ones that aren't relevant to the inserted SWAT widgets. Removing a single point of failure reduces the potential impact of a single error and increases compatibility with other systems that may have large event flows or strict requirements for event propagation speed. For this reason, the leaf filter system may be the best choice for situations where the flow of events through the document root is a performance bottleneck.

Just as the Multiple Glass Pane system distributes the properties of the Glass Pane to each widget, the Leaf Filter distributes event-stopping listeners to the widgets that have been inserted. The Leaf Filter system uses one listener for each widget in the page, and stops events when they reach that widget. Clicks on application components do not reach these filters, so they are not affected.

The risk of the Leaf Filter system is that a web application will act on events as they are dispatched downward toward the target, before they reach (and can be stopped by) the leaf filter system.

Evaluation

With the development phase of this project complete, an evaluation was performed on the finished product. The process used during development to train the SWAT toolkit on a few sites served as a template for the full-scale evaluation.

The initial plan for this evaluation was quantitative, with large groups of sites determined procedurally. By taking the top 20 most trafficked sites from the web traffic reporting system Alexa^[38] and the top 20 most commonly mashed up sites from a directory of mashups^[39], three groups of sites were constructed- those that were popular, commonly mashed up, and those that fit both descriptions. A scoring method was then defined for each of five categories of desired results. With the averaged scores for each group, conclusions could be drawn about whether SWAT had achieved its objective of expanding the end-user programmable web.

This strategy was adapted to a more qualitative approach for several reasons. First, each site provided its own unique quirks and technical challenges. Metrics quickly became out of date and inappropriate for the sort of differentiating factors that came up. Second, although SWAT helps the user along with utility functions and some premade modules, it is far from automatic. Focusing on building the right browser extension for each site was time consuming, and it soon became clear that the evaluation would have to be a superficial examination of many sites or an in-depth exploration of a few.

Finally, the rigor of a quantitative evaluation was dampened by a number of subjective factors. Sites had to be pruned from the list of evaluation candidates if they were in a foreign language, because it was necessary to understand content to determine if records were duplicates of each other. Other sites were removed for being duplicates of English-language sites. Others were portals into a broad range of discrete sub-sites that did not share record structure or content; these too were removed.

Instead, a more qualitative evaluation was done on a smaller group of sites. These sites encompassed a broad variety of functionality- for instance, to do lists, search engines, and video content. They also included AJAX applications from a number of providers- although Google has an AJAX application for nearly every purpose, alternatives were sought out where possible.

This is a list of sites that were evaluated:

Simple Sites

Craigslist^[33]

Google Search^[34]

Yahoo Search^[35]

Youtube^[36]

AJAX Applications

Google Docs^[37]

Live Calendar^[31]

Google Calendar^[40]

Yahoo Calendar^[38]

Zoho Calendar^[30]

Remember the Milk^[32]

Most of the sites that were evaluated had been touched on little or not at all during development. In general, the toolkit performed well, although this document focuses mostly on the failures, what they indicate about the structure of web content and how they could be addressed in a future revision. There is, after all, less to discuss about the systems that worked exactly as expected.

Simple Sites

The sites that in this section are not really all that simple- each gets over a billion views each month, stores millions (if not billions) of records, and is under constant development. These sites are not without AJAX or javascript-powered UI features. Still, identifying and augmenting their records can be done without the click-handling systems described in the Developing for AJAX Applications section.

Craigslist

Craigslist^[33] is widely known for its simple design and minimal use of CSS and javascript. SWAT was trained on this site, and experienced no problems. Clicks on widgets changed their state; clicks on site links also had the desired effect.

A post can appear in two places on craigslist- on a search listing and a result page. Posts can be identified by URL, or by a unique ID number (which is contained in the URL).

Google and Yahoo Search

Of course URLs identify records on these sites; web pages are the records. Google^[34] and Yahoo^[35] both keep the search results on their site fairly clean and straightforward. Unlike Craigslist (or Youtube), there isn't a detail page¹ with additional information about a record; all of the information about the record appears in the search result.

¹Although some pages are cached.

Both Google and Yahoo embed special results in their search results- for weather, video and image results. These were not anticipated and the site profile module had to be expanded to accommodate them, although when they went undetected the system failed very gracefully, adding widgets to the records it found and ignoring the rest.

Youtube

Youtube^[36] lists videos in several ways- on the record detail page (in this case, the page where the video plays), in the search results, in playlists, and in other lists that appear around the site. Videos can be linked in comments and in other videos.

The browser extension built for YouTube matched the video player page and video in the search results page. The other list features were not matched, but there are no significant technical limitations on matching them- it would be a straightforward process to expand the site profile scripts to handle the remaining lists of videos on YouTube.

AJAX Applications

Google Calendar

The evaluation performed on Google Calendar was the most structured and exhaustive. Such an evaluation was needed on at least one AJAX site to give a clear view of the relative merits of the event handling systems outlined in the earlier section, Four Event Handling Designs. As an AJAX application, Google has a number of properties that make it well-suited for this purpose:

- **Complex.** Google Calendar presents a variety of event handling problems. There are several different ways of viewing records, and a number of different tools, menus and modes. In some views dozens of records can be shown at once.
- **Difficult to mash up.** A primary motivation of this project is expanding the user-controlled web. Google Calendar is not currently popular with the mashup community.
- **User-Generated Content.** Certain tests can only be performed when records can be freely created, modified, and destroyed. For instance, one such test asks, if a record is destroyed and re-created, will SWAT treat it as the same record or as a new record? Another test asks, what will SWAT do if two records are created with the same values for some fields? To answer these questions naturally it was necessary to be able to create records.

- Popular. Obscure sites that fit the above criteria might focus this evaluation on a small or arcane subset of web-application related problems. Google Calendar has a large user base and is under active development.

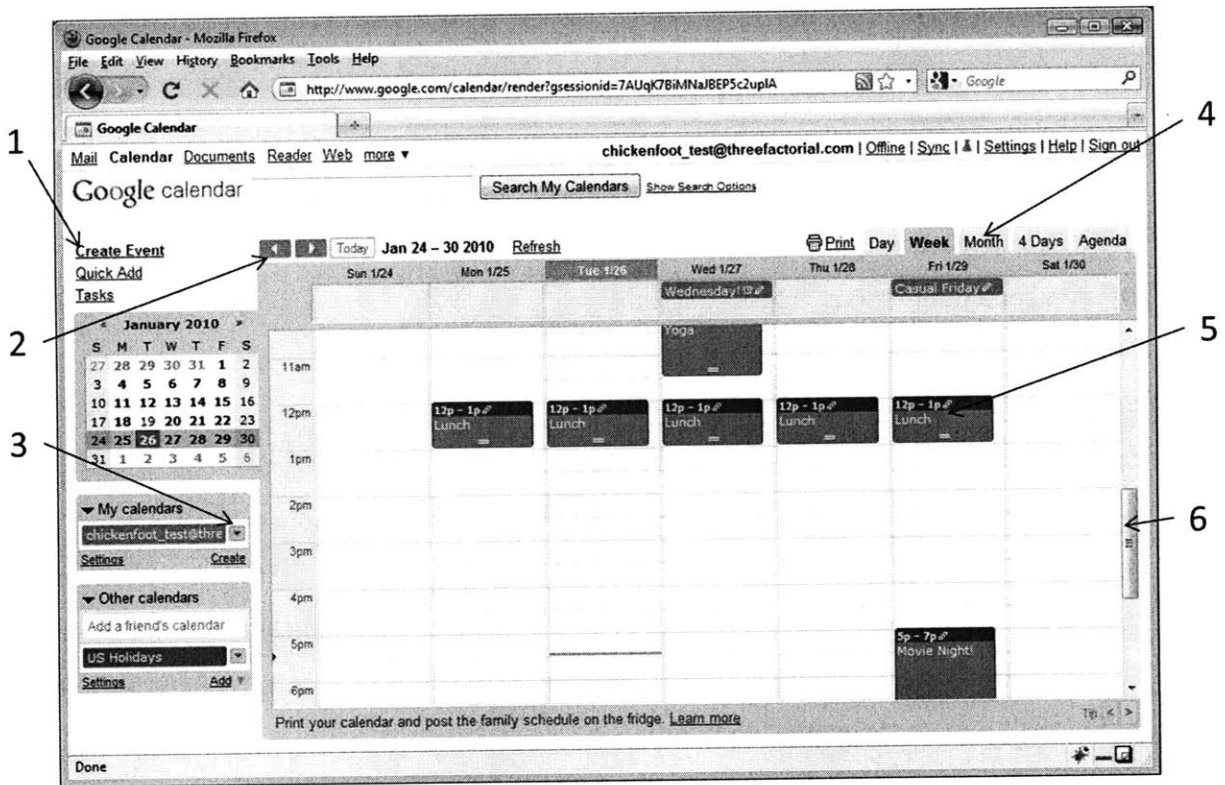


Figure 11. Features of Google Calendar.

1. Events can be created using the Create Event link or by clicking on an empty spot in the center pane. 2. Navigation arrows let the user skip forward week by week. 3. Drop-down menus show menu options when clicked. 4. Appointments can be displayed in Day, Week, Month, 4 Days, and Agenda views. 5. This appointment could be dragged downward to change its time to later in the day, or dragged to the right to move it to the next day. 6. The inner pane of content has its own scroll bar. Note that two appointments are partially scrolled out of view.

Google Calendar: Tests

A battery of 18 tests was drawn from issues encountered during development; test procedures were designed to reliably reproduce problematic behavior. Although these formed the bulk of the evaluation, additional tests were added for features fundamental to the application.

Four tests evaluated the features introduced by SWAT:

1. Click to star a record.
2. Change between weekly and monthly view; ensure stars stay correlated.

3. Partially obscure an appointment, ensure that widgets are properly obscured.
4. Try to click an obscured widget.

Twelve tests were centered on using features of the calendar application. These tests were:

1. Use the Google toolbar links at the top of the page.
2. Use one of Google Calendar's dropdown menus.
3. Switch views using the day, four day, week, and month view tabs.
4. Click the forward and backward buttons to navigate week-to-week.
5. Create an appointment by using the "Create Event" link.
6. Create an appointment by clicking on the calendar.
7. Drag an appointment to change its day/time.
8. Delete an appointment.
9. Scroll the inner window, using the scrollbar.
10. Scroll the inner window, using the mouse wheel.
11. Open Google Calendar in two browser windows. Add an appointment in the first window. The appointment should appear in the second window and be recognized by SWAT as a record.
12. Trigger an in-page popup message (here "in-page" refers to an absolutely positioned div, rather than a javascript alert window. See **Error! Reference source not found.****Error! Reference source not found.**). Ensure the popup works properly.

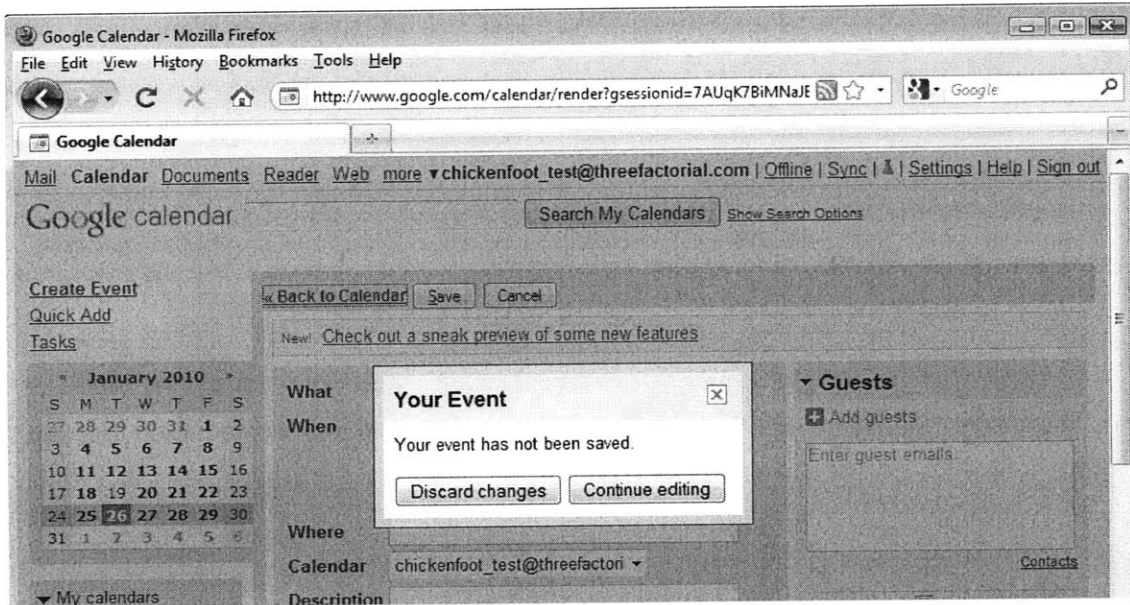


Figure 12. An in-page popup message on Google Calendar.

Two tests evaluated features of the browser application:

1. Maximize and restore the outer window, ensure widgets still work properly.
2. Continuously resize the outer window, ensure widgets still work properly.

The browser window's scrollbars were not tested because Google Calendar's content resizes to fit the available space; a browser scrollbar for the whole page only appears when the content is extremely small. At such scales, Google calendar itself does not have enough room to properly display its content.

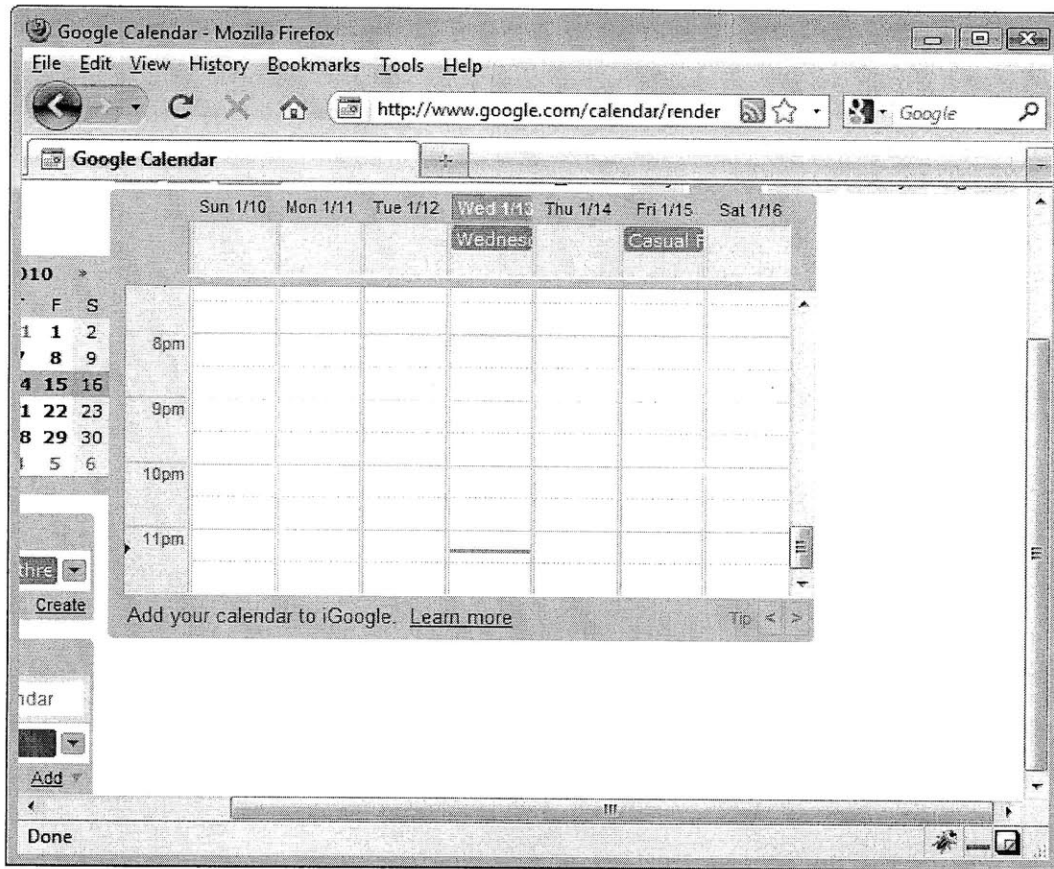


Figure 13. Google Calendar in a small window.

Google Calendar scales its content down to fit the window. When the window is smaller than a minimum size, the application no longer fills the page properly. This screenshot shows Google Calendar's default behavior, without modification by SWAT.

Google Calendar: Testing Summary

The four click handling systems described earlier² are laid out in the following chart.

	Single Point of Control	Multiple Points of Control
Use Glass Panes to control click flow	Glass Pane (failed 7/18 tests)	Multiple Glass Panes (failed 4/18 tests)
Modify Events in Listeners to control click flow	Root Filter (passed all tests)	Leaf Filter (passed all tests)

² Not including the leaf listener system. The leaf listener serves as a baseline against which the other systems are compared.

The two systems that passed all tests are on the bottom row. The most obvious conclusion, then, is that modifying events in listeners is more effective than using glass panes. More generally, it seems the glass pane approach failed because it required re-implementing too much of the browser's default behavior. In an attempt to improve click performance, features like scrolling and elements hiding other elements were broken. Browsers have a lot of complicated behavior and subtle dependencies, so it's best to use as much of the browser's structure as possible.

Along the other axis, there seemed to be less difference between a single centralized point of control and multiple local points of control. For more discussion of these results, see the Discussion section. For a more detailed look at the data gathered during Implementation, see Appendix C: Implementation Data.

Other Calendar Applications

Google Calendar is not the only web calendar application, and given the complexity of the space, it seemed fitting to evaluate other AJAX applications that serve a similar purpose.

The first calendar application considered was Yahoo Calendar^[29]. Unfortunately, this application was more "Web 1.0" than anticipated, loading a new page with every click and doing little if any asynchronous exchange of data. Since the purpose of evaluating a calendar application was to see how SWAT worked with listeners and AJAX controls, it seemed reasonable to find a different calendar to examine.

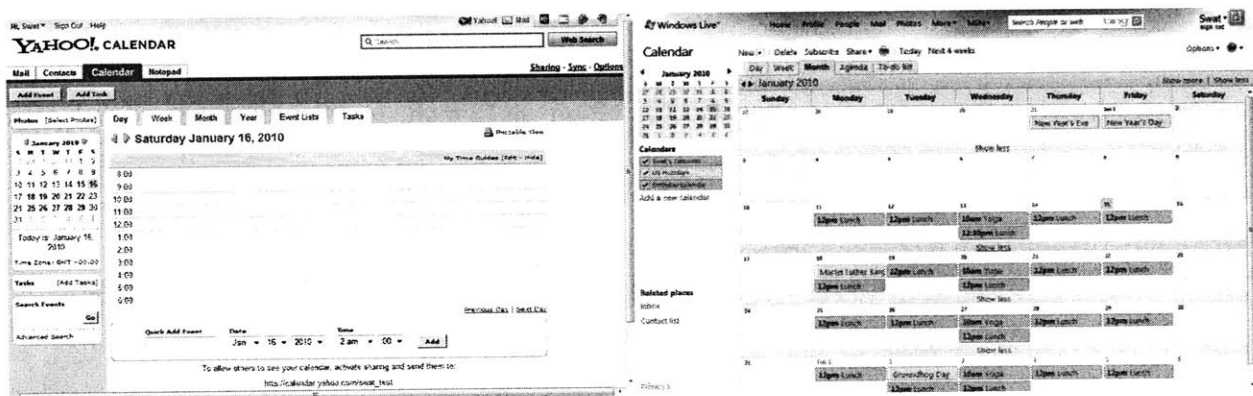


Figure 14. Yahoo Calendar and Windows Live Calendar.

The next calendar considered was Microsoft's Windows Live Calendar^[31], a more sophisticated AJAX application. Calendar appointment elements had no identifying markings- no unique ID, date label, or other information besides the text of the appointment and the position of the element. Identifying

records here requires using the relative position of column headers and date labels to ascertain the date associated with a record. A SWAT browser extension for this site was not built.

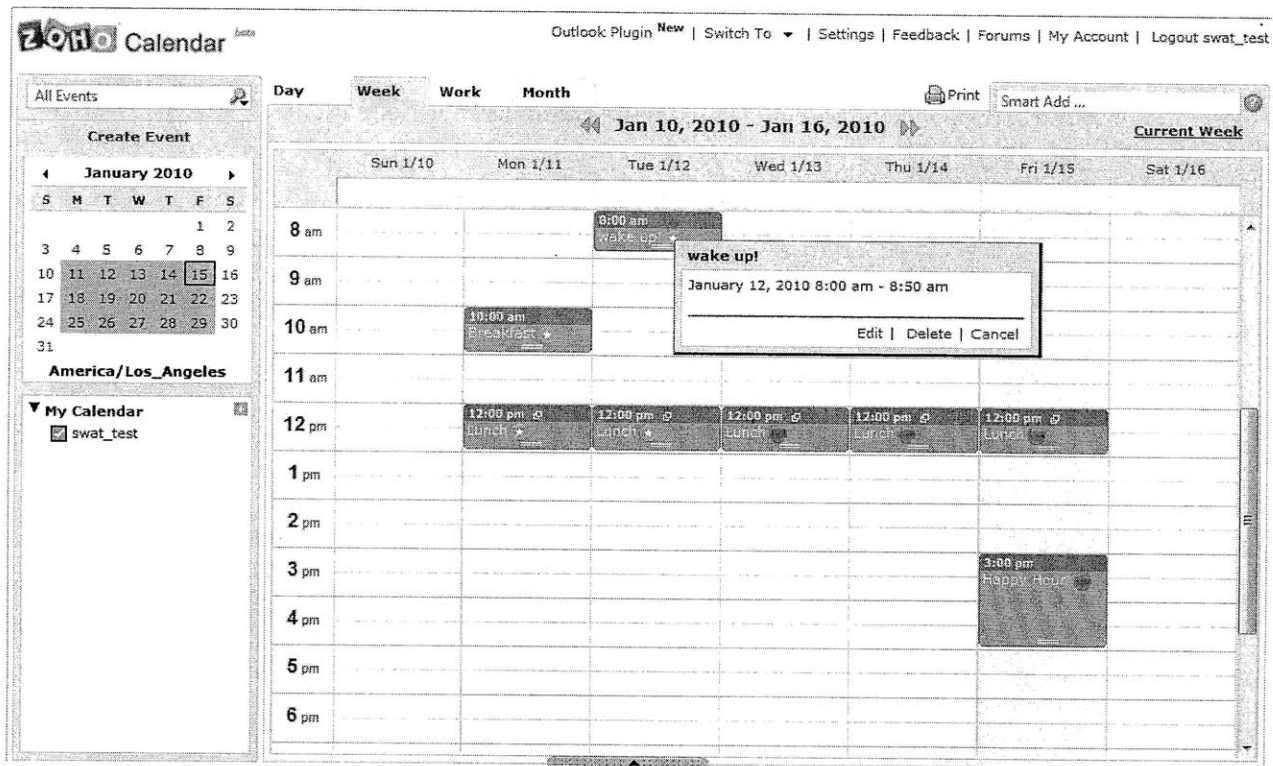


Figure 15. Zoho Calendar.

Note the similarity to Google Calendar's layout.

Finally, Zoho Calendar^[30] was considered. This application eschews frames and does plenty of asynchronous data exchange. The layout and functionality are very similar to Google's, and calendar appointment elements are identified a very clear notation- an 18-digit identifier followed by the date of the appointment. This made it relatively easy to uniquely identify records for this application.

SWAT features and browser features worked as well as they had with Google calendar, with one exception- clicks continued to trigger Zoho's dialog box (seen in Figure 15), even when the root filter click handling system was used. This behavior was likely because Zoho was listening for other mouseEvents- like mouseDown and mouseUp- that were not trapped by SWAT.

Google Documents

Google Documents^[37] is a site that lets users upload, create, and edit documents and spreadsheets. This evaluation focused on the directory listing of documents, an application that reproduces much of the functionality of a desktop file browser like Windows Explorer.

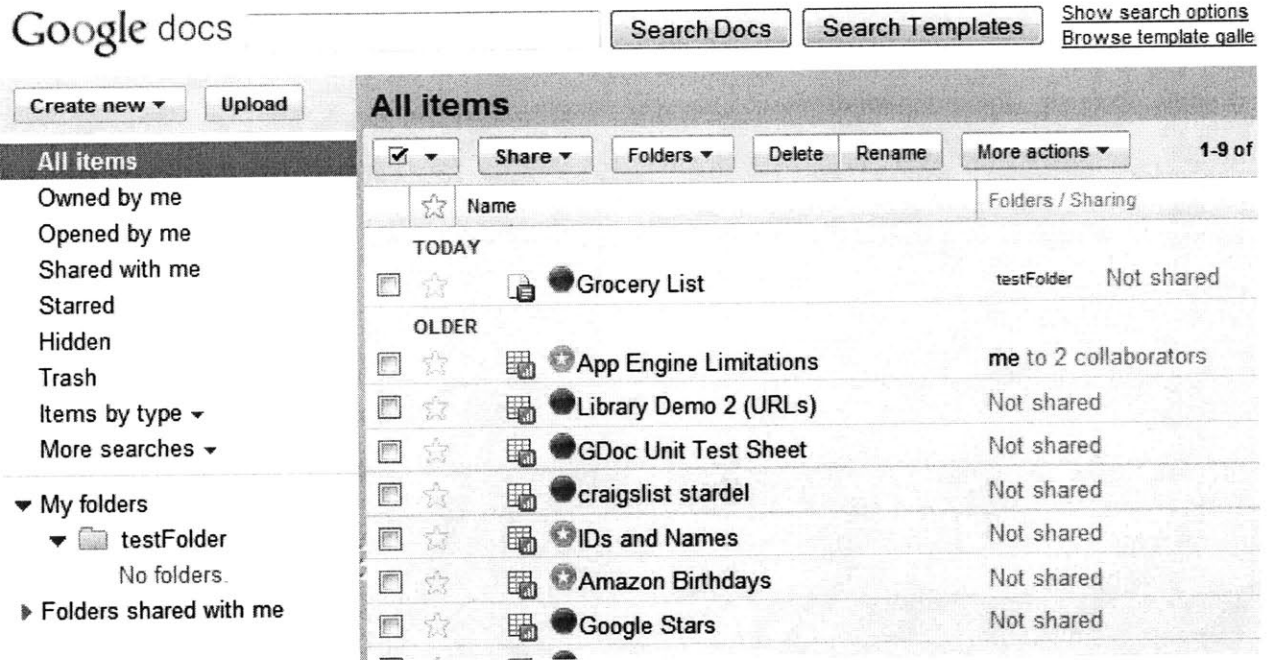


Figure 16. Google Documents with star widgets.

Note the application's native star widgets to the right of the column of checkboxes.

Google Documents already has a "star" feature that lets the user mark a document or folder. Previous sites were evaluated with a star; for consistency this was done here as well. Although two stars looks extraneous, the focus of this part of evaluation is the interaction with the application; the star widget is a simple module and could be replaced with something different without affecting the event listening or DOM restructuring performance.

The leaf filter system which had worked for Google Calendar did not work here; clicking widgets triggered both the widget and the application. This is likely because the Google Docs application tracks events with a listener in the document root node, intercepting them before they get to the leaf listeners installed on widget elements.

Fortunately, the root filter system appeared to address these problems. SWAT features worked perfectly- clicking a star starred a record without the application knowing about the click. Obscured appointments rendered properly and did not register unwanted clicks. Starred records stayed starred across views.

Browser features also worked- resizing, restoring and maximizing the window were correctly detected and content was repositioned accordingly. Like Google Calendar, Google docs scales to fit the browser window, so testing the outer window scroll bar was not possible.

All critical browser applications worked, including adding and deleting records, moving documents to folders (by dragging), and scrolling the internal panel (using either the mouse scroll wheel or dragging the scrollbar thumb).

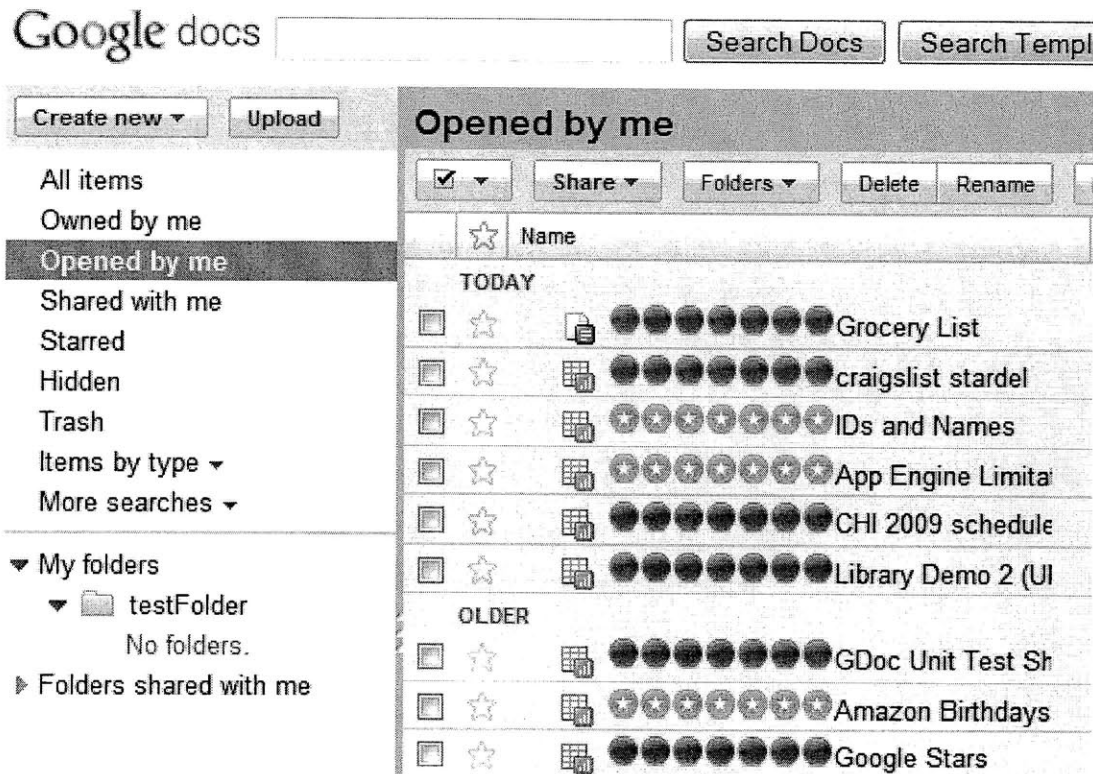


Figure 17. Duplicate Widgets.

Clearing old widgets failed when the view (see menu on left) was changed rapidly.

The only problem encountered here followed a number of changes in rapid succession, such as sorting a list of documents or switching between views. Old widgets would not be erased, leading to the creation of multiple widgets for each record, a number that would grow steadily as navigation continued. This was likely caused by a race condition in the SWAT toolkit.

Remember the Milk

Remember the Milk^[32] is a task list manager. This was the most complicated site that passed all tests. Widgets were added to the records (tasks), which had a globally unique ID number and appeared in a consistent place in the DOM across all views.

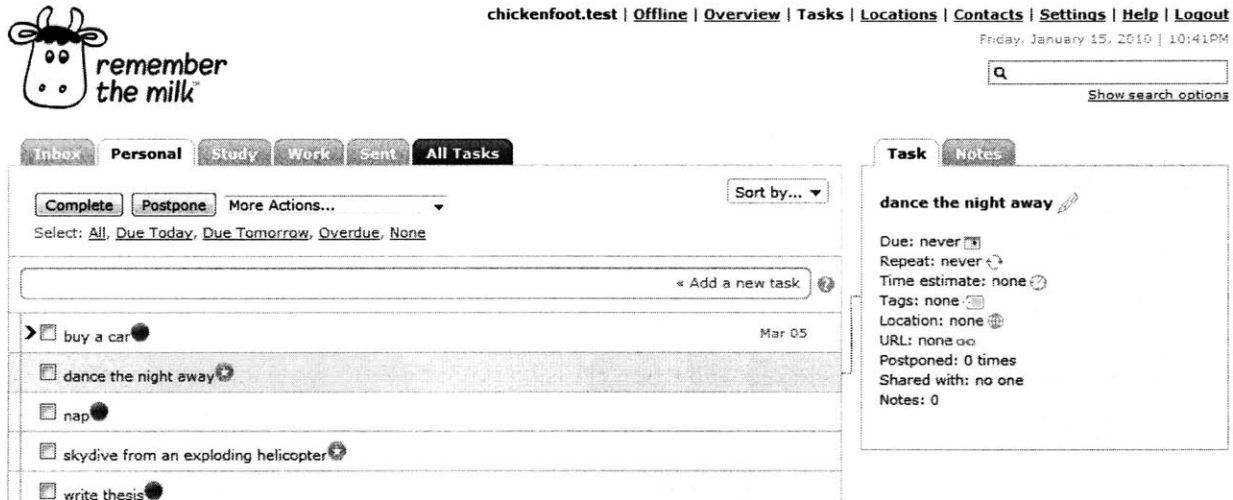


Figure 18. Remember The Milk.

Note task detail window, which appears when a task is moused over and changes the DOM with each appearance/disappearance.

Scrolling, dragging, clicking, and resizing worked fine, and the stars felt like a natural part of the site.

One feature that caused a few jitters was the task detail window, seen in the screenshot above. Instead of triggering on a click, it appeared every time a task was moused over- and disappeared on every mouseout event. Because the DOM was being changed with each event, mousing over the list of tasks sometimes caused a flickering delay as SWAT struggled to reinsert the star widgets over and over.

Discussion

Many challenges were encountered in the creation of the SWAT toolkit. Some are resolved, some are outstanding. This section will describe the pieces of the end user programming landscape that are significant: the strengths that are unique to this work and the problems that undermine it.

The proper interception, handling and distribution of clicks was a major concern during this project, especially on complex AJAX applications. On web 1.0 sites, where navigation through data and transitions between web pages were the same, there were few click events and few entities competing for them. Event traffic was completely different in AJAX applications. On some, dozens of DOM-changing events could be fired without a single click.

Clearing and reinserting widgets worked well when clicks and other events were infrequent. When events fired more frequently, content flickered and the state of the inserted widgets sometimes became invalid. Invalid state persisted until an event fired that would clear and reset. Occasionally an invalid state would persist until a page refresh or navigation to a new page; in testing invalid state never persisted through these events.

SWAT catches a number of events that do not have an effect on the widgets it has inserted; aggressive clearing means widgets are often removed and reinserted “just in case”. A better design would adopt a lazy update policy, where a survey of the current state and the desired state is done first, and the minimal number of changes is made to convert one to the other.

SWAT relies too heavily on an “additions” array that holds a list of changes to the web page. This array is used to clear changes to the page. When the clearing code was interrupted or crashed on an unexpected value, the system failed harder than it needed to.

A better design would put a common feature on all inserted widgets, like a unique ID or ID prefix. If an xpath query was used to return the results that the change array was meant to hold, forgotten page elements would be captured on the next clearing round.

Such a design would also help direct clicks faster. The root filter click handling system is particularly inefficient when click frequency is high (it iterates through the additions array, checking each element against the one that was clicked). Although the primary objective was proving things to be possible (and not optimizing speed), some actions did go noticeably slower when SWAT was in place.

One common thread in these concerns is that the handling of page interactions is a complicated task worthy of more attention. The original three-module design focused on storing data, parsing a site, and creating widgets that could augment records. The site profile module was originally meant to be the only site-specific module, but as click handling became more of a concern, different versions of tweak modules were implemented, each with different click handling systems, to better match the needs of sites. In retrospect, it appears that a site profile API function dealing with page interaction would be a good addition to SWAT. Such an “interaction profile” could define nuanced policy for which page events to pay attention to, and which to ignore.

An open question at this point is: How will SWAT browser extensions hold up over time? Much of the evaluation of sites took place in a fairly small time window; when sites are redesigned will SWAT scripts need minor changes or complete rewrites? If a site changes the way a field displays or whether an index appears in a certain DOM element, some scripts will fail silently- new unique IDs won't match old ones, and record annotations will disappear, their only record in the not-very-prominent storage module.

A community of interested individuals could keep modules up to date and distribute them by subscription, as some ad-blocking systems do today with blacklists. There is an unfortunate chicken-and-egg problem here; such a community is unlikely to develop unless the scripts are popular; to be popular they must be fairly robust by themselves.

On a positive note, some pieces of the system worked better than expected. Identifying records by target URL worked on a large number of sites; this only broke down on AJAX applications. The SWAT API proved useful and robust: even as unpredicted problems required rewriting modules, the purpose of those modules and their interaction stayed constant. Also, throughout the evaluation there were not observed problems involving persisting data or getting it back from storage modules, though the storage systems were not stress tested with multiple browsers making changes at once.

Conclusion

This thesis introduces SWAT, a powerful, extensible toolkit built on the Chickenfoot web scripting environment that aims to restore content control to end users. Its modular design lets programmers mash up SWAT code with their own, or that of other programmers.

SWAT has been evaluated on a range of sites, and uncovering problems and solutions for a number of implementation issues in this space. The increasing prevalence of AJAX web applications and the resulting change in the nature and volume of events necessitates sophisticated strategies for dealing with event traffic. Four such event handling strategies were outlined and evaluated in this paper.

The results in this paper can serve as both a blueprint for future development and a survey of the limitations of making modifications to web applications that are, from the content provider's perspective, unsolicited and unauthorized. Ultimately, there is a clear value to users in the ability to organize and control the information they consume, and that value will motivate future work in the direction this paper has started in.

Acknowledgements

This project wouldn't have been possible without the support and advice of many people.

Thanks to my parents for teaching me the importance of ideas, reading to me and being willing to discuss anything, anytime, until my questions ran out. Thanks to my sister for laughing at my jokes and listening to my stories.

Thanks to iTweek on deviantArt for the beautiful circular buttons.

Thanks to Rob Miller for all his feedback, for guiding me toward many of the best ideas in this paper and helping me to identify and improve the weaker ones, and for drawing a great group of people together for a constant exchange of all sorts of ideas.

Thanks to that group of people- my lab mates and friends in MIT's User Interface Design Group. Greg Little, for sharing code and insight on Javascript, Mike Bolin, for technical advice about Firefox's quirks, Lydia Chilton, for coming up with the SWAT acronym, and everyone who helped brainstorm ideas over tea. Thanks to MIT for fostering an environment that makes all of these things possible.

Bibliography

1. De Castro Reis, Golgher, Silva and Laender. Automatic Web News Extraction Using Tree Edit Distance. WWW, 2004.
2. Wang and Lochovsky. Data Extraction and Label Assignment for Web Databases. WWW 2003.
3. Bogart, Burnett, Cypher and Scaffidi. End-User Programming in the Wild: A Field Study of CoScripter Scripts. VL/HCC, 2008.
4. Barrett, Maglio, and Kellem. How to Personalize The Web. CHI, 1997.
5. Wong and Hong. Marmite: End-User Programming for the Web. CHI, 2006.
6. Ennals and Garafalakis. Mashmaker: Mashups for the Masses. SIGMOD, 2007.
7. Zang, Rosson, and Nasser. Mashups: Who? What? Why? CHI 2008.
8. Huynh, Mazzocchi, and Karger. Piggy Bank: Experience the Semantic Web Inside Your Web Browser. ISWC, 2005.
9. Dontcheva, Drucker, Salesin, and Cohen. Relations, Cards, and Search Templates: User-Guided Web Data Integration and Layout. UIST 2007.
10. Huynh, Miller and Karger. Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities. UIST 2006.
11. Goldman, Bernstein, Van Kleek, and Miller. Databasket: Coherent Shared Data for the Web. Unpublished.
12. Dontcheva, Drucker, Wade, Salesin, and Cohen. Summarizing Personal Web Browsing Sessions. UIST 2006.
13. Lerman, Getoor, Minton, and Knoblock. Using the Structure of Web Sites for Automatic Segmentation of Tables. SIGMOD 2004.
14. Zhai and Liu. Web Data Extraction Based on Partial Tree Alignment. WWW 2005.
15. Zang and Rossen. What's in a Mashup? And why? Studying the perceptions of web-

active end users. VL/HCC 2008.

16. Wong and Hong. What Do We “Mashup” When We Make Mashups? WEUSE IV 2008.

17. Adar, Dontcheva, Fogarty, and Weld. Zoetrope: Interacting with the Ephemeral Web. UIST 2008.

18. Microsoft Popfly. <http://popfly.ms>

19. Yahoo Pipes. <http://pipes.yahoo.com>

20. oSkope visual search. <http://www.oskope.com/>

21. Chickenfoot for Firefox. <http://groups.csail.mit.edu/uid/chickenfoot/>

22. Greasemonkey. <http://www.greasespot.net/>

23. Blogger Web Comments for Firefox.

<http://www.google.com/tools/firefox/webcomments/index.html>

24. Warichu Annotation Tool. <http://www.warichu.com>

25. iGoogle. <http://www.google.com/ig>.

26. My Yahoo! <http://my.yahoo.com>

27. Apple Dashboard Widgets. <http://www.apple.com/downloads/dashboard>

28. Windows Sidebar and Gadgets. <http://www.microsoft.com/windows/windows-vista/features/sidebar-gadgets.aspx>

29. Yahoo Calendar. <http://calendar.yahoo.com>

30. Zoho Calendar. <http://calendar.zoho.com>

31. Windows Live Calendar. <http://windowslive.com/Online/Calendar>

32. Remember The Milk. <http://www.rememberthemilk.com>

33. Craigslist. <http://www.craigslist.org>

34. Google Search. <http://www.google.com>

35. Yahoo Search. <http://search.yahoo.com>

36. Youtube. <http://www.youtube.com>

37. Google Docs. <http://docs.google.com>

38. Alexa. <http://www.alexa.com>

39. Programmable Web. <http://www.programmableweb.com>

40. Google Calendar. <http://calendar.google.com>

41. He, Patel, Zhang, and Chang. "Accessing the Deep Web: A Survey". Communications of the ACM 2007

Appendix A: API Description

API Functions

Tweak API Functions

This section describes the functions and properties in the Tweak API. The first function in each module API is of the form `make_MODULE()`. These

`make_tweak_<TWEAK NAME>()`

Return a newly created tweak object. All tweak objects implement the other tweak API functions described in this section. The tweak object can insert widgets into a web page, and has a property called “fields”, listing the names of the data fields that those widgets control. Those fields can be mapped to fields in the site profile and data store modules.

Takes no arguments.

fields

A property of every tweak object, this array holds field name strings. Each field is a part of the state tracked by the tweak; this part of the state corresponds to a piece of data that can be identified on the site by the site profile and to another that will be stored in its own column by the data store when a record is persisted.

`connect (tweak_field, profile, profile_field, store, store_field)`

Connect a tweak to a specified site profile and data store. For each a field is specified. When the tweak is run, a widget will be inserted that controls the value of `tweak_field`. The initial condition of this widget is determined by the value of `profile_field`. When it is changed, the value will be stored in the data store’s `store_field`.

This function can be run more than once to create multiple connections. For instance, a complicated tweak that deals with compound data (eg, a name+calendar appointment pair) would need to have the connect function run once for the name field and once for the appointment field. Once connected, the user could perform a single action on the tweak that would display both fields from the profile and persist them to a single record in the data store.

tweak_field: the relevant field name in this tweak. Must be a string in the tweak's fields array.

profile: the site profile of the target web site.

profile_field: the relevant field name from the site profile's field_headers array.

store: the data store that will hold persisted records.

store_field: the relevant field name from the data store's headers array.

This function does not return anything.

run()

Iterate through all records in the site profile(s) connected to this tweak. For each record, insert a tweak widget at the corresponding insert point if appropriate. Outfit each widget with an event listener that will call some tweak-specific function when clicked.

clear()

Remove all widgets that have been inserted into the page by this tweak. This function is useful on AJAX-rich sites like calendar applications, where the page structure may change so radically that the tweak needs to start from scratch on new page content.

Site Profile API Functions

This section describes the functions and properties in the Site Profile API.

make_site_profile_<PROFILE_NAME>()

Return a newly created site profile object. All site profile objects implement the other site profile API functions described in this section.

This function does not get information out of the page; that is not done until get_records is called.

Takes no arguments.

field_headers

A property of every site profile object, this array holds field name strings. Each field is a distinct piece of data that can be extracted from the site for each record there. For instance, on a shopping site fields might include the item's name, price, and text describing the item.

url_pattern

A property of every site profile object, this string describes the URLs of pages that this profile is designed to get data from. For example, a profile for Google search result pages might have a url pattern of "http://www.google.com/search?*"

get_records()

Analyze the current web page and find all records there.

Returns an array record objects. Each record contains an HTML element corresponding to the DOM subtree where the record's data is located, and an array of extracted fields, with one entry for each string in the field_headers array. Any decoding or interpretation that is necessary to transform data on the web page into a useful format is done in this step; after this point the data should be in an accessible, usable format.

Data Store API Functions

This section describes the functions and properties in the Storage Profile API.

make_data_store_<STORE NAME>(store_location, headers)

Return a newly created data store object. All data store objects implement the other data store API functions described here.

This function checks the store_location. If this file/web page/database does not exist, it tries to create it and sets up the appropriate column headers.

store_location

A property of every data store object, this object describes the location of the data store. The specifics of this object varies across implementations. It may be a path to a local file, a URL to a web page, or other information about where to persist data.

headers

A property of every data store object, this array of strings represents column names. Each column in a data store holds a distinct piece of data for each record stored there.

get_all_records()

Return a list of all records in the data store.

add_record(record)

Adds the specified record to the data store.

delete_record(record)

Deletes the specified record from the data store.

Appendix B: Hook Points

Users who wish to go beyond the preset module implementations can implement their own modules, but a facility is provided to let them make minor changes without changing the library source code. At a series of hook points in the code, dummy functions are called which can be overridden by the user. This section describes the locations and purposes of these functions.

The functions include the names of their module to avoid namespace collisions; users may wish to modify different modules in different ways.

By default, all of these functions return their input unchanged.

hook_modify_tweak_<TWEAK NAME>(widgetString)

Before any widget is inserted into the DOM, this function is called on the HTML string that makes up the widget. When overridden, this hook point lets the user modify the widget, for example, wrapping it with a border or replacing a set of checkboxes with radio buttons.

hook_filter_get_records_store_<STORE NAME>(records)

At the end of a data store's `get_all_records` function, this function is called on the result set. When overridden, this hook point lets the user modify the data coming out of the data store. For example, the user could filter out records with a certain property, split records into parts, or add new records. A user who wanted to write to their data store using some other system might use this hook point to change escape characters or handle a different data encoding.

hook_filter_set_records_store_<STORE NAME>(records)

The complement to the previous function, this function is run at the beginning of the `add_record` and `delete_record` functions. It is passed an array containing one record. A user could use this to modify the data going into the data store by filtering, splitting or adding records.

hook_filter_get_records_profile_<PROFILE NAME>(records)

At the end of the site profile's `get_records` function, this function is run on the set of results. This function can be used to reduce the number of elements on a page that are matched as records. For example, a user might want to only see a limited number of complicated or slow-loading widgets on a

search result page. Since the site profile module does not add records to a page, there is no corresponding set records function.

user_defined_modules.js

Included at the end of the top-level `swat_library.js`, this file is where users can override functions and include their own list of module definitions while keeping their changes limited to the hook subdirectory. By default `user_defined_modules.js` is empty.

By isolating all user changes to a single subdirectory, all user-introduced changes can be removed or backed up at once, and the library can be updated or reverted to a previous version without destroying user changes.

Appendix C: Implementation Data

Implementation Comparison

	Leaf Listener	Glass Pane	Multi Glass	Root Filter	Leaf Filter
Inserted Widgets	In flow of document	In flow	In flow (variant: on top)	In flow	In flow
Inserted Panes	None	One per page	One per button	None	None
Listeners	One per button	One for glass pane	One per glass pane	One at root	One per button
Application Clicks	Normal	Simulated	Normal	Normal, after root lets them through	Normal
Lines of Code*	134	198	189	163	136

*This measure only in the number of lines in the relevant tweak module implementation; it does not include storage module code, site profile code, or code in common utility functions.

Google Calendar Results: SWAT Functionality

This table describes the results at the end of development. Each cell reports whether the feature listed worked properly under the given listening system.

	Feature	Leaf Listener	Glass Pane	Multi Glass	Root Filter	Leaf Filter
SWAT Functionality	Click to star a record	No, application reacts to click	Yes	Yes	Yes	Yes
	Stars stay correlated across views	Yes	Yes	Yes	Yes	Yes
	Widget obscured;	Yes	Yes	Yes	Yes	Yes

	star properly obscured?					
	Clicking obscured widget	Yes	No, click registers with widget	No, click registers with widget	Yes	Yes

Google Calendar Results: Browser Functionality

Feature	Leaf Listener	Glass Pane	Multi Glass	Root Filter	Leaf Filter
Scroll browser window	Test not run; at this scale the application does not work properly anyway.				
Maximize/restore browser window	Yes	Yes	Yes	Yes	Yes
Continously resize browser window	Yes	Yes	Shows jitters, occasionally leaves glass panes in wrong position	Yes	Yes

Google Calendar Results: Application Functionality

Feature	Leaf Listener	Glass Pane	Multi Glass	Root Filter	Leaf Filter
Regular Google toolbar links	Yes	No, links don't respond to clicks	Yes	Yes	Yes
Google toolbar dropdown widget	Yes	Does not respond to clicks	Yes	Yes	Yes
Switch tabs (day,month...)	Yes	Yes	Yes	Yes	Yes
Next/prev week nav	Yes	Yes	Yes	Yes	Yes

buttons					
Create event link	Yes	Yes	Yes	Yes	Yes
Create event by clicking on calendar	Yes	Popup appears, but new appt time and dialog box placement are wrong	Yes	Yes	Yes
Drag appointment	Yes	Dragging does not work	Yes	Yes	Yes
Delete appointment	Yes	Yes, with slight display error	Yes	Yes	Yes
Scroll the inner window	Yes	Not scrollable	Breaks; glass panes and widgets get out of alignment	Yes	Yes
Mouse scroll wheel	Yes	Scrolling event not passed properly	Scrolling occasionally ends up in misaligned state, when mouse is over a widget or many scroll events are fired in short succession.	Yes	Yes
New content arrives (edits in	Yes	Yes	Yes	Yes	Yes

another browser)					
In-page popup messages	Yes	Yes	Yes	Yes	Yes