

From Scrapbook to Mashup: A Review of End-User Web Programming Tools and Supporting Toolkits

Michel Krieger
HCI Group
Stanford University
mkrieger@stanford.edu

October 1, 2007

Contents

1 Introduction	1
2 End User Web Programming Tools	3
2.1 Chickenfoot	4
2.2 Marmite	6
2.3 Koala	7
2.4 MashMaker	8
2.5 d.mix	9
2.6 Piggy Bank	10
2.7 Potluck	11
2.8 Hunter Gatherer	12
2.9 Internet Scrapbook	13
2.10 Summarizing Personal Web Browsing Sessions	15
2.11 Summary of End-user Web Programming Tools	16
3 Tools for Website Content Extraction and Manipulation	17
3.1 Solvent and Crowbar	17
3.2 GreaseMonkey	18
3.3 Anthracite	19
3.4 Citrine	20

1 Introduction

As hip-hop artists have taken samples and songs and united them - ‘mashed them up’ - in novel ways (Danger Mouse’s “Grey Album”, a combination of The Beatles’ White Album and Jay-Z’s Black Album, being the canonical example), web programmers, ranging from the novice to the expert, have in the last few years begun to take pieces of web content and unite them in

‘mashups’. The growth of these has dissolved the previous web paradigm of many websites united by hyperlinks, with each website functioning as a content silo. In its place, we see a union of data sources that are combined through a common front end, almost always remaining inside the web browser.

Beyond the technologies that have emerged as part of this trend, the growth of mashups have also created a new space for the exploration of end-user programming research. Microsoft Excel’s formula-based programming has previously been the prime example of how users might use programming to aid their everyday tasks, but is limited to the mathematical and financial realms. Mashups, on the other hand, involve tasks and websites with a broader appeal - maps applications such as Google Maps are employed every day by millions of people to get directions and routes, real estate listings are used by many house-hunters, and photo sharing websites such as Flickr are both a publishing house and a destination for diverse groups of users.

Part of the reason why the web is such a fertile ground for end-user programming is the easy availability, in most cases, of a website’s source code. In a traditional desktop application, as the source code is written in a compiled language such as C++, the end-user has little, if any, introspection available regarding the way content ends up on the screen. On the Web, since a page is rendered from its source on every load, programmers, and the tools they use, have the entire Document Object Model (DOM) available to them to remix and change as they please.

Examining the current state of end-user web programming tools will allow us to see what design decisions and tradeoffs have been made in existing tools, and how future tools might address these challenges in different ways. In this literature review, we examine the current state of end-user web programming tools - many of which are used to create mashups or

website re-mixes, or sample elements from multiple pages for a user’s benefit - as well as the tools that power these tools’ ability to ‘scrape’ existing web pages and build upon them. Focus is given to projects that have been written up in academic journals, since these projects and tools have a focus beyond ‘creating interesting web pages’, and instead focus on how end-user web programming tools might fit into the larger HCI context.

2 End User Web Programming Tools

This literature review orders these end-user Web programming tools in order of how close the user must get to the source code. Care has been given to address each tools’ design goals, assumptions, strengths, and tradeoffs. The tools’ relative abstraction from code is shown below:

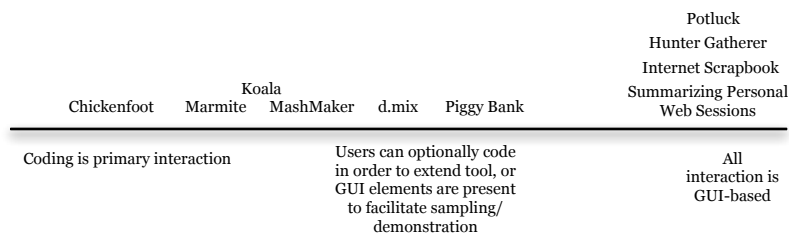


Figure 1: End-user Web Programming Tools’ Level of Abstraction from Code

Additionally, a timeline of the tools is provided below.

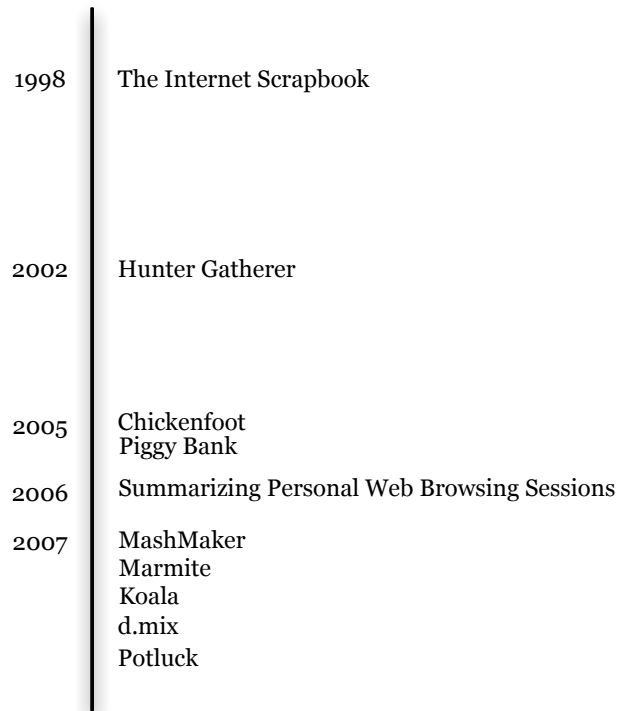


Figure 2: End-user Web Programming Tools Timeline

2.1 Chickenfoot

“A user should never have to view the HTML source of a Web page in order to customize or automate it.”

Developed at MIT CSAIL as an extension to the Mozilla Firefox browser, Chickenfoot works inside the browser as a development environment presented in a sidebar. By placing itself as a browser extension, Chickenfoot is instantly cross-platform (but not, currently, cross-browser). It also has close access to both the rendered Web site and its internal working, and the

user interface (chrome) of the browser itself. Chickenfoot is available as a download.

Chickenfoot scripts are written in Javascript, which requires some programming experience in its users. Thus, though it has a stated purpose of shielding users from a site's HTML source, it nonetheless has a command prompt as its primary mode of interaction. The abstraction in Chickenfoot is achieved by allowing end-users to write script using English keywords in lieu of DOM manipulation javascript. For example, on Google's home page, rather than writing:

```
document.getElementById('q')[0].value = 'dogs'
```

To enter the value "dogs" into the search box, a Chickenfoot programmer would only have to write:

```
enter('dogs');
```

The drawback of using keywords is that there is little to no indication of what keywords are available in the system as the system is being used. Instead, users must refer to documentation - the process is similar to that of using a command-line system interface such as the bash shell: if the user knows an action ("list all files in this directory") but not the command to perform it, they are forced to either guess, or look up the documentation.

A further drawback of keywords is that, in referring to Web site elements by the text that is displayed within them, Chickenfoot scripts make an assumption about the rendered contents of the page that may break when viewing a localized version of the page.

Ultimately, Chickenfoot's most valuable contribution is its keyword engine, which reduces the difficulty inherent in requiring users to enter code. Programmers familiar with Javascript can mix and match "plain" Javascript with the abstracted Chickenfoot commands when creating Web sites. These programmers appear to be the ideal audience for Chickenfoot. Unfortu-

nately, the evaluation methods in the Chickenfoot UIST publications focus entirely on the keyword aspect of the system, rather than whether the system can be used to efficiently automate everyday tasks on the Web.

2.2 Marmite

Marmite, an end-user Web programming tool developed at Carnegie Mellon, seeks its spiritual roots not in other Web service tools, but instead in Apple's Automator, a framework for automating actions in Mac OS X using a data flow model. This data flow model is the principal design choice made in Marmite, and informs the interaction its users will have with it.

Prior to the implementation of the actual Marmite system, its researchers decided to explore the data flow automation space, by conducting user studies with Apple's Automator, and paper prototype studies on lo-fi mockups of the Marmite system. It does not seem like these studies were used for much beyond validation and ideas for future directions, however.

Marmite, like Chickenfoot, is implemented as a Mozilla Firefox plugin, written in XUL and Javascript. The tradeoffs here are similar, with the added twist that Marmite relies extensively on user interface elements provided and customized from the Firefox widgets. Unlike Chickenfoot, Marmite is not provided for download, so we must infer its functionality and interaction from its associated publication.

Overall, the system appears to not have any resonance with the users it was tested on. One problem - the small sample size of participants (6) - was exacerbated by having two of the participants fail completely due to an unclear design in the first iteration. For the other participants, the fundamental model of the application - which had been validated by the Automator studies - appeared to fail when applied to this domain:

“These users were puzzled by the meaning of selecting inputs

and outputs. Users deleted operators they had successfully used not knowing that they were erasing data. These users believed that an operator was no longer needed once it had been used to produce a result in the spreadsheet view.”

It is hard to tell whether the usefulness of Marmite was limited by a few key design choices, or if the fundamental model does not resonate with users with little programming experience, or little experience with data flow-type programs. Some of its insights - such as suggesting “next” operators when users make a selection, and presenting graphical selection options for operators, can be carried forward into future threshold-reducing efforts.

2.3 Koala

A collaboration between MIT’s CSAIL and IBM’s Almaden Research Center, Koala is an end-user Web programming tool that, unlike many of the tools surveyed, tackles a specific sub-problem: automating business processes. The key concept behind Koala is to enable programming-by-demonstration through a ‘sloppy programming’ model that interprets ‘pseudo-natural language’ statements, rather than strict programming syntax. The four key components behind Koala are: a ‘recording’ mode to capture user actions; a pseudo-natural language syntax that is both human-readable and machine parse-able; links to data stores to automatically personalize actions with user-specific information; and finally, a wiki where users can share their Koala scripts. By enabling this easy wiki-like sharing, Koala is similar to d.mix, though the execution of scripts remains inside the Koala Mozilla Firefox extension, rather than the wiki itself.

The primary user interaction with Koala is executing a script written in the pseudo-natural language. This script may automate an action such as

ordering an item from an online business reseller. Any errors or ambiguities in the script are pointed out, and users then use a programming-by-demonstration mode to fix any of these problems. Any modifications made by users can be saved back to the Koala wiki. Through these steps, Koala builds on several staples of end-user programming, such as mixed-initiative execution - the concept that some steps are too complex to be worth automating, and are best performed by user. In Koala, if any step has the word “You”, then it is considered a user action step.

The system, as presented at CHI 2007, has not been thoroughly evaluated in user studies, and so for now remains a promising idea that leverages both programming-by-demonstration concepts and the Web’s structured document object model to create a collaborative automation tool. It extends Chickenfoot’s key contribution - keyword commands as a proxy for machine commands - to an extreme: the keyword commands are not a proxy, but the language understood by both humans and computers. Future systems that employ keyword commands as the programming model presented to its user may want to consider this approach.

2.4 MashMaker

“MashMaker is an interactive tool for editing, querying, manipulating, and visualizing ‘live semi-structured’ data.”

Born out of a collaboration between Intel Research Berkeley and Yahoo Research, MashMaker (currently in closed beta) attempts to leverage the strengths of the most ubiquitous end-user programming tool - the spreadsheet - and apply those to a ‘semi-structured’ view of the Web.

In a spreadsheet, one can usually program by demonstration - perform an action on one cell, and then have the spreadsheet apply that same action (with any necessary adjustments) on subsequent cells. MashMaker

takes this idea and applies it to Web mashups - a user looking up information about a series of houses can perform an action (such as finding nearby restaurants) to one house, using a filter widget, and then apply the same action to neighboring cells.

TODO - finish this section (after watching videos)

2.5 d.mix

Developed by researchers at Stanford's HCI Group, d.mix is a programming-by-sample system. End-user programmers can visit Web sites and selectively sample elements they would like to include in their 're-mixes'. These elements can be static - 'I want this particular picture from this user's Flickr photo page' - or dynamic - 'I want the three most recent YouTube videos posted with this tag'. d.mix is implemented as an 'active wiki' system, where pages are proxied through a Ruby wiki that transforms the pages in a manner conducive to sampling through d.mix.

Similar to how Piggy Bank relies on a collection of site scraping scripts, d.mix functions by proxying Web sites through server-side active wiki scripts that perform the necessary transformations to prepare a page for sampling. This server-side approach allows for these transformation scripts to be updated without requiring all d.mix clients to have to update themselves. Also, it means the transformation scripts themselves can be edited collaboratively in a wiki environment.

d.mix's most important contributions are the notion of a 'parametric copy', and its lowered threshold for dealing with Web APIs. The 'parametric copy' extends the traditional programming-by-demonstration concept of copying and pasting a sample, and instead adds the parameters required to generate the sample in the first place. This relates directly to the second contribution - by copying the API parameters necessary for generating a

sample, d.mix reduces the need for end-user programmers to look through potentially confusing API documentation. Instead, they have to simply take the results of that API's work, and sample it. This is analogous to how many of the site scraping technologies take advantage of a Web browser's rendered view of the page - d.mix takes advantage of Web services' rendered view of data.

2.6 Piggy Bank

Though the promise of the Semantic Web has been present for over a decade, end-users have yet to benefit from it. Though reducing the Web to structured, bare component would provide a rich input to the mashup and remixing tools present today, placing data in RDF (Resource Description Framework) format is rarely performed by webmasters.

Acknowledging the slowness of adoption of the Semantic Web, the researchers behind Piggy Bank decided to provide an 'upgrade path' for the Web as it is today. Implemented as a Firefox extension, Piggy Bank is a Semantic Web browser that tries to read a site's RDF contents, but falls back into user-submitted screen scrapers if semantic information is unavailable. The programming aspects of Piggy Bank that merit its inclusion in this review are twofold: users can write and submit screen scrapers for sites they visit often and wish to experience through Piggy Bank, and users can share the information mashups they create in Piggy Bank with other users, thus becoming authors.

The key motivation behind Piggy Bank - and indeed much of the Semantic Web - is similar to that of early content extraction mechanisms for the Web: breaking down content into pages, as a Web browser does, is too coarse. The researchers behind Piggy Bank claim this is a holdover from the physical artifacts that informed the concept of Web 'pages':

“But just as the earliest automobiles looked like horse carriages, reflecting outdated assumptions about the way they would be used, information resources on the Web still resemble their physical predecessors.” (p 1)

While Piggy Bank is one of the most feature-rich of the tools surveyed, its introduction of multiple new concepts to users - RDF descriptions of data, saving of snippets of content that are generated by screen scrapers, and the notion of information pieces rather than pages - is overwhelming upon first use. The project is an important leap forward in bringing the Semantic Web closer to reality - by providing a well-defined ‘upgrade path’ for existing Web sites - but does not seem to have much consideration for the user experience of sampling this Semantic Web, and thus remains inaccessible to those users who might wish to use the rich information collected in a meaningful way.

2.7 Potluck

While tools such as Piggy Bank and Solvent may bring us closer to the Semantic Web, a vision that will enable easy re-mixing of data, the authors of Potluck claim that this in itself is not enough - even semantically tagged data can be messy, and joining data from two sources can be messy without some assistance.

With a stated audience of non-programmers who want to mash-up data, the Potluck system - implemented as a Java server and a Javascript/D-HTML front-end - is one of the few tools surveyed that provides pleasant surprises rather than unpleasant ones. As long as one begins with sources of data that are correctly tagged (a large assumption, but one that is beyond the scope of the tool), mixing data from different sources into common records is a simple, drag-drop process. Viewing different facets of the

mashed-up data is achieved through toggle boxes.

Through a user study (described in brief in the paper) the researchers behind Potluck found that, though programmers and non-programmers were both equally able to learn and use the tool, the programmers were far more excited about its possibilities and interface. There are two large take-aways to gain from Potluck: first, that providing clear affordances and a simple, pleasantly surprising user interface (even if it is not central to the research) will aid participants in unearthing the real strengths and weaknesses of the system, rather than getting caught up on the surface of the tool. Second, getting non-programmers to appreciate the value of end-user Web programming tools is an unsolved challenge that future tools should explore.

2.8 Hunter Gatherer

Developed by m.c. shraefel and team at the University of Toronto, Hunter Gatherer is an end-user Web programming tool that focuses on the activity of “sampling” from Websites. The tool predates the emergence of Web mashups as a trend, as it was created in 2002, but there are parallels between the “sampling” activity present in the Hunter Gatherer research and the sample/remix pattern found in Web mashups. One strong insight from Hunter Gatherer is that, though sampling was a task with multiple applications, it was infrequently performed in the real world. They identify two such applications:

“a journalist might want to build a collection of different newspaper coverage of the same story. A student might build a heterogeneous collection to reflect her current term, including courses, professors, gym hours and so on.”

Hunter Gatherer seeks partial inspiration from WYSIWYG editors such as Microsoft Front Page, that often feature the ability to take images dropped

from a Web browser, while retaining the information about that image's origins. However, most of these editors do not know where dropped text comes from, which is a shortcoming that Hunter Gatherer addresses.

A user interacting with Hunter Gatherer writes no code whatsoever. Instead, users choose to sample sections of Websites by selecting them (bounding boxes and highlights show which sections are selectable) and then editing their collection in a Collection List.

Perhaps the strongest element of the Hunter Gatherer research is its creators' desires to explore not only the efficiency/feasibility of their tool, but also its affect, as defined by Andrew Dillon. To this end, they deployed a Hunter Gatherer field study where users were allowed to use the tool 'free style'. Performing this study was important in understanding how Hunter Gatherer might fit into a users' Web usage pattern - and, in some cases, even change that pattern:

“most users responded that the tool/concept had indeed become part of their way of thinking about gathering information on the Web.”

By keeping the interaction entirely in the GUI, Hunter Gatherer provides a very low threshold - any user familiar with Microsoft Word or a similar editor should be able to pick up the tool quickly - but also a low ceiling, as users have little to no control over how the finished product looks.

2.9 Internet Scrapbook

An early example of Web page extraction tools, Internet Scrapbook was developed in 1998 by a team at C&C Media Research Laboratories. Like Hunter Gatherer, the research was performed before the advent of Web mashups, but the extraction techniques in Internet Scrapbook have been influential on future research.

Users interact with Internet Scrapbook (Scrapbook) by copying and pasting context from browsers (at the time, Internet Explorer and Netscape Navigator) into Scrapbook. Scrapbook then uses adaptive heuristics to learn where on the source page that pasted content came from. Their primary insight regarding Web page layout (that remains mostly true today) is that heading and titles tend to remain static, even as content changes. A glance at current popular news sites - CNN.com, Google News, the New York Times online - shows that this is particularly true for sites with constant updates in hard-coded sections.

Scrapbook allows for user freedom by not requiring that end-user programmers select text in a particular way. Instead, it performs best when a user select the entire content snippet (ie. an article) but will try to account for partial matches as well. The system tries to identify patterns either through a header pattern, or through a tag pattern, though preference (in the heuristics) is given to the header pattern matching. Later content extraction systems have tended towards the tag pattern, as it is friendlier towards XPath expressions, or have abandoned content extraction as the primary sampling technique, preferring instead access through APIs.

Almost ten years on, Scrapbook's contributions are still valuable. Particularly, its authors identified not only which sections of Web sites tend to remain constant, but why this is so:

“First, changing the structure of Web pages is a heavy burden for information providers and, furthermore, this is costly. Second, Web sites must ensure the readability of their Web pages. If the document structure is changed often, readers cannot easily find their target information.” (p 8)

2.10 Summarizing Personal Web Browsing Sessions

The University of Washington's Summarizing Personal Web Browsing Sessions (SPWBS) research focuses on an update of the technologies explored a decade ago by Internet Scrapbook and Hunter Gatherer. Like those systems, the SPWBS system is entirely GUI oriented, with no coding required or available. It also relies heavily on the browser's rendered DOM tree for locating elements in a Web page's structure.

The SPWBS system's primary interaction occurs when a user sees a particular element he or she would like to sample on the Web. They can then choose to sample that element - for example, a restaurant listing - and show the system which pieces of metadata are where. The selection is done on the DOM tree listing itself - thus, users have insight into the layout of the page, without having to write the XPath expressions needed to capture that element in the tree. One interesting facet of the SPWBS system is that future captures of similar listings can 'back-propagate' into the information about the first sample. For example, if the user identified where the name of the restaurant, but not its location, when sampling the first restaurant, but then later sampled a second restaurant from the same site where they identified the location, the system automatically fetches a location for the first restaurant as well.

Perhaps the most significant contribution of the SPWBS system is its pre-made layout templates that are optimized for particular views - for example, mobile versus desktop. In the traditional Model-View-Controller model, the SPWBS system is one of the few in this review that address different views (d.mix allows for different views, but they generally must be authorized by hand through HTML and CSS). Future systems would do well to allow for an easy template system for exporting user's mashups.

2.11 Summary of End-user Web Programming Tools

As evidenced by the timeline at the beginning of this section, interest in end-user Web programming tools from an academic standpoint has spiked in the last three years and is likely to continue. With the Web and the current DOM manipulation and site scraping tools available, there is now a fertile new ground for exploring end-user programming. This programming can be as simple as data disambiguation in Potluck, or as complex as automating a business flow in Koala.

In general, there is a close correlation between each tool's abstraction from code, and its resulting threshold and ceiling. For example, though Hunter Gatherer's threshold for use is very low - bounding box are drawn on existing Web pages and can be selected with a mouse - users have little-to-no control over the resulting gathered data's display. Tools in the middle of the spectrum, such as d.mix, allow for experienced programmers to dive into the code and change the way their pasted/sampled content is rendered, but still present a disconnect between the programming-by-sample nature of the parametric copy and the eventual editing of source code. One promising compromise is the work done in Summarizing Personal Web Browsing Sessions, as device-specific and context-specific templates are provided in order to display mashed-up content. Providing these, perhaps with a source code fall-back such as d.mix, might account well for the 80% case of just wanting the content displayed in an aesthetically pleasing fashion, and the 20% case of wanting to tweak that display.

Ultimately, what few of these tools explored is the way in which end-user Web programming tools can integrate with non-programmers' (or beginning programmers') existing work-flow and behavior. The argument that many more jobs will require programming in the near future is a compelling one - but this projection seems to hold little sway with users today. The tools

in this review have fanned out and experimented with different paradigms for end-user programming on the Web - from programming-by-sample, to Koala's programming-by-demonstration, to Chickenfoot's Javascript programming - and evaluated the effect (and affect) of each. Next, research might look into longitudinal 'free-style' experimentation studies, a longer deployment of these tools, or ethnographic work into non- and beginning programmers' existing mashup-like or mashup-requiring habits to investigate what areas are worth researching next.

3 Tools for Website Content Extraction and Manipulation

In tandem with the development of end-user Web programming tools in the past decade, the frameworks that these tools rely on to parse Web content and find similarities between document object structures have evolved as well. In this section, we review a few popular libraries and frameworks, sorted in order of development. Since these tools are often developed as a function of other projects, or are developed in a non-academic context, non-refereed publications and tools without publications are included. These are reviewed mostly with an eye towards extensibility for research work in the immediate future.

3.1 Solvent and Crowbar

Developed by MIT as part of the Simile project (that also includes Piggy Bank), Solvent's goal is to produce Javascript screen scrapers that integrate with Piggy Bank. Solvent screen scrapers are written in Javascript, and are produced, edited, and debugged through a Mozilla Firefox extension similar to Joe Hewitt's Firebug.

Users begin constructing screen scrapers by using the “Capture” function to highlight elements on the page that contain important information, that will later be transformed into semantic data. Solvent automatically generates the XPath necessary to grab the selected element, and then allows the user to add variables to selected elements (that will later be used when writing the processing script). For example, if we’re scraping over a row of tabular data, and one of the elements for a particular scrape are defined, Solvent can automatically generate the Javascript necessary to iterate over each element.

Like much of the Simile project, Solvent’s primary contribution is enabling a bridge between the current, non-semantic Web and a hopeful future where more data is stored as RDF, rather than scraped into RDF. Though its default output is RDF, it is effective as a scraping toolkit and could be extended to output in a format friendlier to different re-mixing projects.

Also relevant to Solvent is the Crowbar project, which implements a headless (Firefox/XULRunner) Web browser as RESTful Web service. This is significant because it allows scripts running outside the browser environment to access the serialized DOM of a Web page, parsed by the same rendering engine as Firefox.

3.2 GreaseMonkey

Perhaps the most important, and certainly the most popular framework responsible for launching interest in the Web as an end-user programming environment is GreaseMonkey. Developed as an extension to Mozilla Firefox, GreaseMonkey allows for the execution of Javascript after a Web page has loaded, and provides graphical tools to enable and disable scripts, change the order in which they execute, or modify them. Similar implementations

exist for the Safari, Opera, and Internet Explorer browsers, though scripts are rarely cross-platform due to browser idiosyncrasies.

Scripts produced in GreaseMonkey are often named ‘user-scripts’, and one repository of such scripts - userscripts.org - contains over 8,000 of such scripts. They range from the mundane - fixing small annoyances in Web pages or removing certain ads - to complete user interface overhauls and mash-ups, such as embedding Google’s RSS Reader into its mail application.

GreaseMonkey is valuable for having kick-started the end-user Web programming trend, though the tool itself does not provide much value to academic endeavors. It would be placed at the far left of our abstraction spectrum from the previous section, and provides no assistance whatsoever to non-programmers - indeed, even for programmers, debugging GreaseMonkey scripts is an often difficult process that requires the use of other tools such as Firebug.

3.3 Anthracite

Anthracite, published by Metafy, is a commercial screen scraping utility with tight Mac OS X integration. Anthracite is a standalone application, as opposed to the many Mozilla Firefox-based scrapers. The advantages of this approach are apparent upon first loading Anthracite: while Firefox plugin-based utilities have to work within the confines of Firefox’s own XUL interface markup language, a stand-alone application can leverage native user interface toolkits (in this case, Cocoa) to allow for richer visualizations and an icon-based, fully drag-drop user interface.

Rather than work from the DOM tree upwards, Anthracite takes a high-level approach. Much like Automator or other data-flow metaphor applications, Anthracite works as a series of objects that manipulate data. ‘Source’

objects are the starting point; ‘Processor’ objects transform the content or extract elements using a variety of conditionals (XPath, Regular Expressions, “Text Before/After”, and others) and are then piped to a ‘Results’ object. The output can then be viewed in a browser, as it is (by default) HTML.

Anthracite comes with a few predefined objects - such as an ‘Extract Links’ processor - and others can be defined and saved. Since the ‘Input’ object can take multiple Web sites or files, Anthracite can be used to parse multiple pages at a time. Overall, Anthracite is one of the most polished Web content extraction tools available, and it’s unfortunate that it’s closed-source nature makes it difficult to extend beyond its own sandbox.

3.4 Citrine

Though not specifically a Web content extraction tool, Carnegie Mellon’s Citrine system is worth an examination because it tackles one of the paradigms that is most applicable to the end-user Web programming realm: copy-paste. Citrine works to enhance the cut/copy/paste cycle by detecting what sort of data has been copied, and what a useful output format for that data is given the data and the application it is being inserted into.

Citrine’s major departure from other copy-paste extensions is that it works, in its authors’ words, “between the regular copy and paste actions” (p 2). Thus, it requires no modification to the source and target applications. In a similar way, end-user Web programming tools must also act between source and target in a way that requires no modification - beyond the tools’ own - to the pages and services it draws data from, or insert data into.

Another way in which Citrine’s work is relevant to Web tools is its plug-in framework. Much like screen scrapers must be adapted for particular Web sites’ layout, Citrine needs to understand what sort of objects an ap-

plication expects in order to be able to paste into them. For example, Microsoft Outlook accepts the pasting of 'Contact' objects, which are normally only constructed within Outlook. With Citrine, an ordinary address or telephone number present on a Web site is automatically transformed into Contact meta-data, and a Contact object can be pasted into Outlook.

Ultimately, the work done in Citrine should inform the design of any end-user Web programming tool that has 'sampling' or 'copying' as a primary interaction sequence, even if the project itself may not be directly applicable.