Programming by a Sample: Rapidly Prototyping Web Applications with d.mix

Björn Hartmann, Leslie Wu, Kevin Collins, Scott R. Klemmer
Stanford University HCI Group
Gates Computer Science
Stanford, CA 94305
[bjoern, lwu2, kevinc, srk]@cs.stanford.edu

ABSTRACT
As an increasing number of web sites provide APIs, significant latent value for supporting developers' use of these APIs lies in the site-service correspondence: the site and its API offer complementary representations of equivalent functionality. We introduce *d.mix*, a tool that realizes this latent value, lowering the threshold for creating web mash-ups. With d.mix, users browse annotated web sites and perform a *parametric copy* of elements of interest. While a traditional copy contains web page elements, a parametric copy performs proxy-based rewriting of pages to select the *underlying programmatic calls* that yield those elements. Developers can paste this code and edit, execute, and share scripts on d.mix's wiki-based authoring environment. This approach speeds the creation of web applications while preserving the flexibility and high ceiling of script-based programming. An initial study with eight participants found d.mix to enable rapid experimentation, and suggested avenues for improving its annotation mechanism.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces—Graphical user interfaces.

General terms: Design, Human Factors

Keywords: programming by example modification, mash-ups, web services, prototyping

INTRODUCTION

With the advent of Service-Oriented Architectures (SOA) [7], the number and diversity of application building blocks that are openly available as web service APIs is growing rapidly. The web site programmableweb.com, which tracks availability and use of web APIs, reports 1720 different available APIs as of March 2007. These APIs promise developers a cornucopia of interface elements and data sources. Many of these web APIs are the programmatic interface to successful web sites, where the site and the associated API offer complementary views of the same underlying functionality. In essence, *the web site is the largest, functional example of what can be accomplished with an API.* To date, the potential value to developers that could be achieved by coordinating these representations has largely remained latent.

While web services have seen particular growth in the enterprise sector, rapid access to rich features and data also make web APIs a promising tool for prototyping and the creation of situated software: "software designed in and for a particular social situation or context" [31]. The small audience of situated software applications limits developer resources. As such, enabling faster and lower-threshold [25] authoring of these applications provides a catalyst for broader creation.

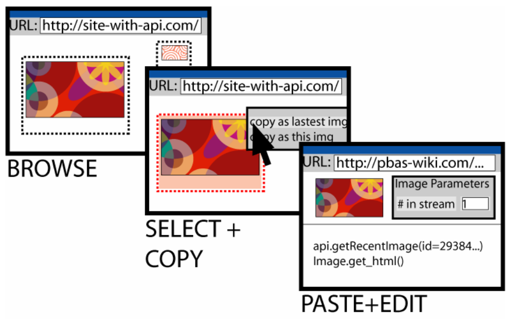

URL: http://site-with-api.com/
BROWSE
URL: http://site-with-api.com/
copy as lastest img
copy as this img
SELECT + COPY
URL: http://pbas-wiki.com/...
Image Parameters
# in stream 1
api.getRecentImage(id=29384...)
Image.get_html()
PASTE+EDIT

Figure 1. When programming by a sample, users first browse web sites that offer APIs through a proxy that adds annotations. They then select pieces they wish to copy and send them to the d.mix editor, where they can change parameters graphically or edit the underlying API source code.

As the number and scale of APIs and web services increase, and as programming entertains an ever-widening audience, more software is written by opportunistically combining pre-existing, high-level blocks of functionality. In these *mash-ups*, the program design resides in the glue layers that combine the different chunks of functionality. We define a web mash-up as a web application that recombines elements from at least two or more external web applications. Mash-ups are instances of the long tail [8] of software—the large numbers of small applications that cumulatively have a big impact. One of the broad shifts introduced in the mash-up paradigm is that the designer's effort and creativity are reallocated: less time is spent building an application up brick by brick, while more time and ingenuity is spent finding and selecting components, and then creating and shaping the "glueware" that interfaces them [16].

Two factors are currently hampering broader use of web APIs: the complexity of installing web application environments and the complexity of understanding and using web service APIs.

To enable flexible and rapid authoring of API-based web applications, this paper introduces *d.mix* (see Figure 1), a browser-based design tool with two notable attributes. The first is a programmable proxy system employing a *site-to-service map* that establishes the correspondence between elements shown on the site and the web service calls needed to replicate these data programmatically. This system enables users to create code that invokes web APIs by browsing the respective web site and visually specifying the web site elements they wish to use in their own application. The second contribution is a server-side *active wiki* that hosts scripts generated by the proxy. As a browser-accessible authoring tool, the active wiki provides a configuration-free environment for authoring and sharing of both source code and working applications. Together, these two components offer a perspective of how web developers could use the surface structure and social structure of the web as a means to democratize the tools of production [8].

The *d.mix* approach targets the growing group of web designers and developers that are familiar with HTML and scripting languages (*e.g.*, JavaScript and ActionScript), lowering the experience threshold required to build and share mash-ups. d.mix offers a completely graphical interaction path, from selecting samples to pasting them into a new page and changing their attributes using property sheets. Additionally, by virtue of displaying the actual underlying code to users, d.mix allows developers with sufficient technical expertise to drill down into code as needed.

To create a system that is felicitous with the practices of web developers, we employed a mixed-methods approach. First, each week for eight weeks, we met with web developers, using the d.mix prototype as a probe to elicit discussion on mash-up design. Second, to gauge the first-use experience of d.mix, we conducted a preliminary lab study with eight web developers. Third, we built example applications using our tool to explore a broader range of interaction designs.

The rest of this paper is structured as follows. To motivate the d.mix approach, we first present a short summary of research in information foraging and programming by demonstration. The next section introduces the main interaction techniques of our system through a scenario, followed by the implementation of d.mix. We then describe the two forms of evaluation undertaken: iterative feedback from web professionals and an initial laboratory study. We conclude with a discussion of related research and commercial systems, limitations of the current implementation, and an outlook to future work.

BACKGROUND

Our work draws on models of information foraging and research in programming by demonstration. We present a brief introduction to both areas here.

Information foraging

As the number and size of programming libraries swells, locating and understanding documentation and examples is playing an increasingly prominent role in developers' activities [32]. d.mix assists users with information foraging by shortening the time spent hunting for information patches. It co-locates two different kinds of information on one page: examples of what functionality and data a web site offers, together with information how one would obtain this information programmatically. Because problems often cut across package and function boundaries, example-based documentation provides value by aiding knowledge crystallization and improving information scent [29].

For this reason, examples and code snippets, such as those in the Java Developers Almanac, are a popular resource. This approach of documentation through example complements more traditional, index-based documentation. d.mix combines dynamic example generation with browsing of what the examples would look like based on the most complete example there is.

Programming by a Sample, or by Example?

d.mix's approach draws on prior work in *programming by example*, also known as *programming by demonstration* [12, 21, 27]. In these systems, the user demonstrates a set of actions on a concrete example—such as a sequence of image manipulation operations—and the system infers application logic through generalization from that example. The logic can then be re-applied to other similar cases.

The class of applications that d.mix addresses are those that employ web services. Through d.mix's site-to-service map, a designer can sample a portion of an extant web page; d.mix then replaces the surface attributes of that sample with the web service calls that generated the sample. While d.mix shares much of its motivation with programming-by-example systems, the approach is quite distinct. Instead of providing the computer with an example that the system then generalizes, designers specify logic through locating and parameterizing found examples. In this way, the task is more one of programming by *example modification*, which Nardi highlights as a successful strategy for end-user development [27]. Modification of a working example also speeds development because it provides stronger scaffolding than writing code tabula rasa.

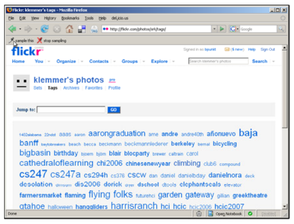
HOW TO PROGRAM BY A SAMPLE

A scenario will help introduce the main interaction techniques. We also encourage readers to watch the accompanying video, on the web at http://hci.stanford.edu/mashups.

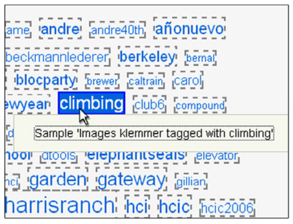
Jane is an amateur rock climber who frequently travels to new climbing spots with friends. Jane would like to create a page that serves as a lightweight web presence for the group. The page should show photos and videos from the latest outings. She wants content to update dynamically so she doesn't have to maintain the page. She is familiar with HTML and has some JavaScript experience, but does not consider herself an expert programmer.

Jane starts by browsing the photo and video sharing sites her friends use. Scott uses the photo site Flickr and marks his pictures with the tag "climbing." Drew also uses Flickr, but uses an image set instead. Karen shares her climbing videos on the video site YouTube. In short, this content spans multiple sites and multiple organizational approaches.

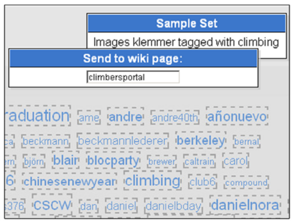
To start gathering content, Jane opens Scott's Flickr profile in her browser and navigates to the page listing all his tags (see Figure 2a). She then presses the "sample this" button in her browser bookmark bar. This reloads the Flickr page, adding dashed borders around the elements that she can



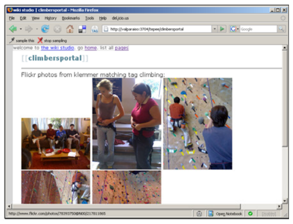
(a) Browse



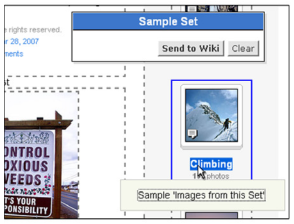
(b) Sample



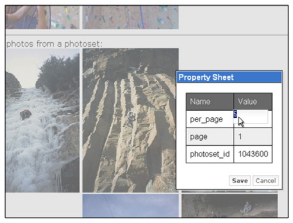
(c) Send to wiki



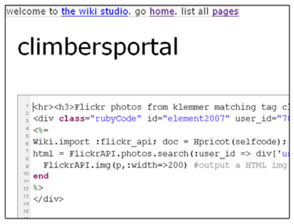
(d) Wiki executes copied script



(e) Browse & sample again



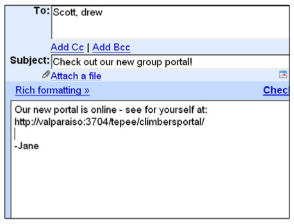
(f) Edit properties in wiki


welcome to the wiki studio. go home. list all pages

climbersportal

```
<hr><h3>Flickr photos from klemmer matching tag c
<div class="rubyCode" id="element2007" user_id="7
<%=
Wiki.import :flickr_api; doc = Hpricot(selfcode);
html = FlickrAPI.photos.search(:user_id => div['u
  FlickrAPI.img(p,:width=>200) #output a HTML img
end
%>
</div>
```

(g) Edit source code in wiki


To: Scott, drew

Add Cc | Add Bcc

Subject: Check out our new group portal!

Attach a file

Rich formatting »

Our new portal is online - see for yourself at:
http://valparaiso:3704/tepee/climbersportal/

-Jane

(h) Share URL

Figure 2. With d.mix, users switch between foraging for content and editing copies of that content in an active wiki environment.

copy into her *sampling bin*.

Jane right-clicks on the on tag "climbing," opening a context menu which offers her the choice to copy the set of images Scott tagged with that word (see Figure 2b). The copied item appears in her sampling bin, a repositionable floating layer on top of the page.

She selects *Send to Wiki* and enters a new page name, "ClimbersPortal" (see Figure 2c). Her browser now displays this newly created page in the d.mix programmable wiki. The visual representation dynamically requests the specified images; the textual representation contains the corresponding API call to the Flickr web service (see Figure 2d).

Continuing her information gathering, Jane samples Sam's climbing photo set on Flickr (see Figure 2e). Her wiki page now displays both Scott's photos and several images from Sam. Jane would like the page to display only the latest three images from each person. She right-clicks on Sam's images to invoke a property sheet which shows that the content came from a Flickr photo set and gives parameters for the user id associated with the set and for the number of images to show (see Figure 2f). Changing the parameters reloads the page and applies the changes.

Jane then opens Karen's YouTube video page. For Karen's latest video, d.mix offers two choices: copy this particular file, or copy the most recent video in Karen's stream. Because Jane wants the video on her page to update whenever Karen posts a new file, she chooses the latter option.

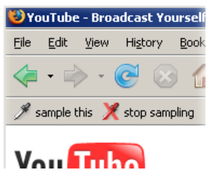
Next Jane would like to layout the images and add some text. She clicks on "edit source," which displays an HTML document, in which each of the three samples she inserted corresponds to a few lines of Ruby script, enclosed by a structuring *<div>* tag (see Figure 2g). She adds text and a table structure around the images. Remembering that Scott also sometimes tags his images with "rocks," she modifies the query string in the corresponding script accordingly.

When she is satisfied with the *rendered view* of her active wiki page, she emails the URL of the wiki page to her group members to let them see the page (see Figure 2h).

IMPLEMENTATION

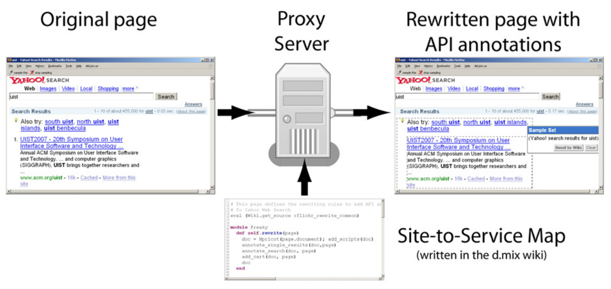
In this section, we describe d.mix's implementation for sampling, parametric copying, editing, and sharing.

"Sample This" button rewrites pages



d.mix provides two buttons, *sample this* and *stop sampling*, that can be added to a browser's bookmark bar to enable or disable sampling mode. *Sample this* is implemented as a bookmarklet—a bookmark containing JavaScript instead of a URL—that sends the current browser location to our active wiki. This invokes the d.mix proxy, combining the target site's original web markup with annotations found using our *site-to-service map* (see Figure 3).

It is important to note that the original web site need not provide any support for d.mix. The active wiki maintains a collection of site-to-service maps, contributed by knowledgeable developers. The site-to-service map describes the programmatically accessible components that are associated with a particular set of URLs (see Figure 4). For example, on the Flickr web site, pages of the form http://flickr.com/photos/<username>/tags contain a list of image tags for a particular user, displayed as a tag cloud. A user's tags can be accessed by calling the API method *flickr.tags.getListUser* and passing in a *user id*. Similarly, photos corresponding to tags for a given user can be re-



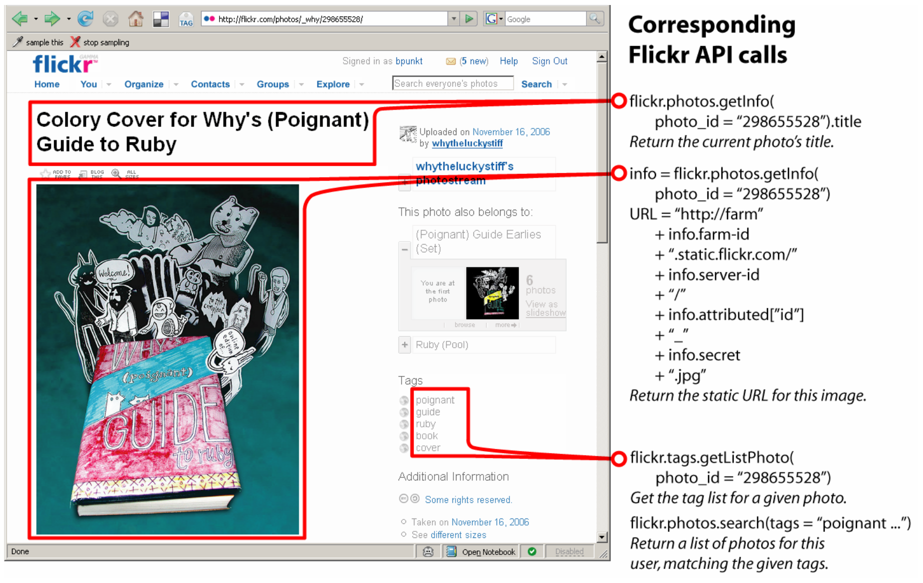
```
# This page defines the rewriting rules to add API an
# To Yahoo Web Search
eval (Wiki.get_source :flickr_rewrite_common)

module Freaky
  def self.rewrite(page)
    doc = Hpricot(page.document); add_scripts(doc)
    annotate_single_results(doc,page)
    annotate_search(doc, page)
    add_cart(doc, page)
    doc
  end
```

Figure 3. d.mix annotates web pages using an HTTP proxy.

trieved by a call to *flickr.photos.Search*.

When the user is in sampling mode, d.mix's programmable HTTP proxy rewrites the viewed web page, adding JavaScript annotations. These annotations serve two functions. First, d.mix uses the site-to-service map to derive the set of web service components which may be sampled from the current page. It does so by searching for known markup patterns—using XPath and CSS selectors — and recording the metadata that will be passed on to web services as parameters, such as a user or photo ID, a search term, or a page number. Second, d.mix's annotation visually augments the elements that can be sampled with a dashed border as an indication to the user.



flickr
Colory Cover for Why's (Poignant) Guide to Ruby
whytheluckystiff's photostream
poignant
guide
ruby
book
cover
Corresponding Flickr API calls

```
flickr.photos.getInfo(
    photo_id = "298655528").title
```
Return the current photo's title.

```
info = flickr.photos.getInfo(
    photo_id = "298655528")
URL = "http://farm"
    + info.farm-id
    + ".static.flickr.com/"
    + info.server-id
    + "/"
    + info.attributed["id"]
    + "_"
    + info.secret
    + ".jpg"
```
Return the static URL for this image.

```
flickr.tags.getListPhoto(
    photo_id = "298655528")
```
Get the tag list for a given photo.

```
flickr.photos.search(tags = "poignant ...")
```
Return a list of photos for this user, matching the given tags.

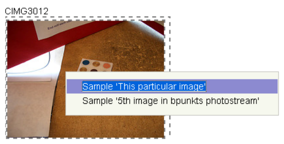
Figure 4. The site-to-service map defines a correspondence between HTML elements and web service API calls. This graphic highlights this mapping for three items on Flickr.

In the other direction, the "stop sampling" bookmarklet takes a proxy URL, extracts the client site URL and sets it as the new browser location, ending access through the proxy.

d.mix is implemented in the Ruby programming language. We chose Ruby to leverage the freely available programmable proxy, the mouseHole [4] and Ruby's metaprogramming libraries.

Parametric copy is achieved by generating web API code

An annotation of an HTML element (*e.g.*, an image on a photo site) comprises a set of action options. For each option, a right-click context menu entry is generated.



CIMG3012
Sample 'This particular image'
Sample '5th image in bpunkts photostream'

Associated with each menu entry is a block of source code, which in d.mix is Ruby script. The code generation routines draw both upon the structure of the page (to know what *class* of items are there) as well as the content of the page (which *specific* items are there).

As an example of how d.mix's source-code generation works, consider a "tag cloud" page found on Flickr. All tags are found inside the following structure:

```
<p id="TagCloud">
  <a href="…">Tag1</a>
  <a href="…">Tag2</a>…
</p>
```

The site-to-service mapping script to find each element and annotate it is:

```
@user_id=doc.at("input[@name='w']")["value"]

doc.search("//p[@id='TagCloud']/a").each do |link|
  tag = link.inner_html
  src = generate_source(:tags=>tag, :user_id=>@user_id)
  annotations += context_menu(link, "tag description", src)
end
```

In this code example, the Ruby code makes use of the Hpricot library [3] to extract the user's id from a hidden form element. It then iterates over the set of links within the tag cloud, extracts the tag name, generates source code by parameterizing a source code stub for *flickr.photos.search* and generates the context menu for the element.

In essence, the d.mix mapping code is performing on-the-fly web scraping of pages the developer is visiting to extract the needed information for code generation. While scraping can be brittle—matching expressions can break when site operators change class or ID attributes of their pages, it is also common practice in web development [16] since it is often the only way to extract data without cooperation of site operators. An important design decision in d.mix is to scrape at *authoring-time*, when the designer is creating pages such as the Flickr-and-YouTube mash-up in the scenario. By scraping parameters first, d.mix's user-created pages can in turn make API calls at *run-time*, which tend to be more stable than the HTML format of the initial example pages.

We acknowledge that building these rewrite rules is time-intensive and requires expertise with DOM querying through XPath or CSS. However, UI tools such as Solvent [19] that support building DOM selectors visually could allow much of it to happen by demonstration. Providing a smooth process for creating the site-to-service maps is important, but is somewhat orthogonal to the contributions of this paper. As such, we leave it to future work. For this paper, the salient attribute is that the site-to-service map need be created only once per web site. This can be performed by a somewhat expert developer, and then all designers wishing to use that site can leverage that effort.

Server-side active wiki hosts and executes scripts

d.mix's active wiki is a space where developers can freely mix text, HTML, and CSS to determine document structure, as well as Ruby script to express program logic. Whenever a developer enters a new page name in the "Send to Wiki" dialog on a sampled page, a new wiki page of that name is created (if needed) and the generated source code is pasted into that wiki page. The developer is then shown the rendered version of the wiki page, in which the web API calls that d.mix generated are executed and their result is shown.

To see the associated web markup (HTML / CSS) and Ruby code, a user can click on the “edit” button as in any other kind of wiki. The markup and snippets of script are then shown in a browser-based text editor, which has rudimentary syntax highlighting and line numbering. When the user clicks on the “save” button, the wiki source is saved, as a new revision, and the user is redirected to the rendered version of the wiki page. In this rendered version, HTML, CSS, and JavaScript tags take effect, and the embedded Ruby code is evaluated by a templating engine, which returns a single string for each snippet of Ruby code.

When evaluating Ruby code, the active wiki does so in a sandbox, to reduce the security risks involved. The sandbox has limited access to objects such as the File class, but can maintain application state in a database or make web service calls through SOAP, REST, or other web service protocols.

In traditional web interface design, the user interface designers create a mockup in HTML which is later thrown over the wall to the front-end engineers (or vice versa). By contrast, the active wiki allows web UI designers to quickly switch between rendered view, markup, (meta)data, and application logic, with less cognitive friction involved in keeping the mappings between the Model, View, and Controller—the active wiki keeps track of this for them.

Pasted material can be re-parameterized and edited

In comparison to a standard copy-and-paste operation, the notable advantage of our *parametric copy* is that an element’s properties can be changed after the fact. To provide rapid editing of the most common parameters of a pasted element—namely those passed to a web service, our wiki offers graphical editing of parameters through property sheets, implemented as floating layers in JavaScript.

Widget-based wiki platforms (*e.g.*, [10]) also offer parameter-based editing of their widgets —but typically do not offer access to the underlying widgets’ source-code representation. In contrast, d.mix generates Ruby script, which can be edited directly.

Like other development environments, the active wiki offers versioning and importing of code living elsewhere on the wiki. It does not yet support WYSIWYG wiki editing, but such functionality could be supported in the future.

As a test of the complexity of code that can be written in a wiki environment, we implemented all site-to-service mapping scripts as wiki nodes. This means the wiki scripts used to drive the programmable proxy and thus create new wiki pages are, themselves, wiki pages. To allow for modularization of code, a wiki page can import code or libraries from other wiki pages (analogous to “#include” in C or import in Java).

The generated code makes calls into Ruby modules that we define, which broker communication between the active wiki script and the web services. For example, users’ Ruby scripts must reference working API keys, which are often needed to make web service calls to popular web APIs.

While using a small static number of web API keys would be a problem for large scale deployment (many sites limit the number of requests you can issue in an hour), we believe our solution works well for prototyping and for deploying situational applications with a limited number of users.

Sharing is built-in as applications are hosted server-side.

An important attribute of the d.mix wiki is that public sharing is the default and encouraged state. An end-user can contribute their own site-to-service mapping for a web site they may or may not own, or simply submit small fixes to these mappings as a web site evolves. If an end-user makes use of d.mix to remix content from multiple data sources, another end-user can just as easily remix the remix—copying, pasting, and parameterizing the elements from one active wiki page to another.

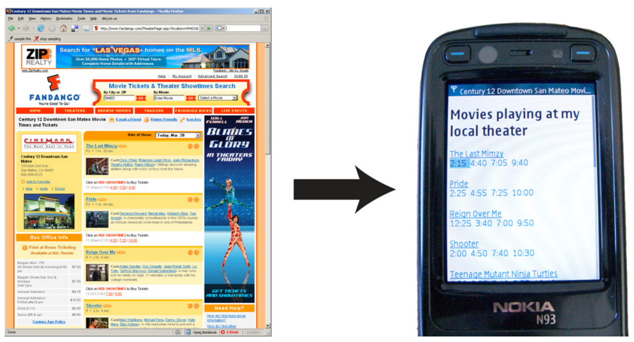
ADDITIONAL APPLICATIONS

In this section we review additional applications of d.mix beyond the use case demonstrated in the scenario.

Existing web pages can be virtually edited

The same wiki-scripted programmable HTTP proxy that d.mix employs to annotate API-enabled web sites can also be used to remix, rewrite, or edit *any* web page, document, or web application to improve a site’s usability, aesthetics, or accessibility, enabling a sort of *recombinant web*. As an example, we have created a rewriting script on our wiki that provides a connection between a popular event listing site and a third-party calendaring web application. By parsing the event’s microformat on the event site and injecting a graphical button, users can copy events directly to their personal calendar. Because this remix is hosted on our active wiki, it is immediately available to any web browser.

Another example is reformatting of web content to fit the smaller screen resolution and lower bandwidth of mobile devices. Using d.mix, we wrote a script that extracts only essential information—movie names and show times—from a cluttered web page. This leaner page can be accessed through its wiki URL from any cell phone browser (see Figure 5). Note that the reformatting work is executed on the server and only the small text page is transmitted to the phone. d.mix’s server-side infrastructure made it possible to develop, test, and deploy this service in 30 minutes. In contrast, client-side architectures such as Greasemonkey [2] do not work outside the desktop environment, while server-side proxies can only be configured by administrators.


Century 12 Downtown San Mateo Movi...

Movies playing at my local theater

The Last Mimzy
2:15 4:40 7:05 9:40

Pride
2:25 4:55 7:25 10:00

Reign Over Me
12:25 3:40 7:00 9:50

Shooter
2:00 4:50 7:40 10:30

Teenage Mutant Ninja Turtles

Figure 5. The rewriting technology in d.mix can be used to tailor content to mobile devices. Here, essential information is extracted from a movie listings page.

Beyond web-only applications

The scenario presented in this paper focused on data-centric APIs from successful websites with large user bases. While such applications present the dominant use case of mash-ups today, we also see opportunity for d.mix to enable development of situated ubiquitous computing applications. A wide variety of ubicomp sensors and actuators are equipped with embedded web servers and publish their own web services. This enables d.mix's fast iteration cycle to extend the "remix" functionality into physical space. To explore d.mix design opportunities in web-enabled ubicomp applications, we augmented two smart devices available in our lab to support API sampling: a camera that publishes a feed of lab activity, and a web-controlled power outlet. Combining elements from both servers, we created a wiki page that allows remote monitoring of lab occupancy to turn off room lights if they were left on at night (see Figure 6).

More important than the utility of this particular example is the architectural insight gained: since the web services of the camera and power outlet were open to us, we were able to modify their web pages and embed API annotations with the services. This proof of concept demonstrated that web service providers can integrate support for API sampling directly into their pages, obviating the need for a separate site-to-service map on the d.mix server.

FEEDBACK FROM WEB PROFESSIONALS

As d.mix matured, we met weekly with web designers to obtain feedback for a period of eight weeks. Some of these meetings were with individuals, others were with groups; the largest group was 12. We mostly recruited informants at professional events; informants included attendees of several Ruby programming language user groups, web developers at startup companies in Silicon Valley, and researchers at industrial research labs interested in web technologies.

Perhaps the most important issue raised by informants was one of scale. An early informant was a web developer at a Bay Area calendaring startup. He was most interested in the technology to allow rewriting of third party pages through scripts shared on a wiki. He saw performance as well as legal hurdles to grow our approach to many simultaneous users. Another team voiced similar concerns, particularly about scaling issues arising from the limits imposed by web services as to how many API calls a user can make. Scaling concerns are clearly central to the question of whether a mash-up approach can be used to create wide-distribution web applications; however, they are less critical for tools such as d.mix that are primarily designed for prototyping and situated software.

As the reach of mash-ups expands, informants were interested in how users and developers might locate relevant services. Several informants, including a JavaScript developer at a web-based instant-messaging startup, suggested that it was important to consider how tools might aid users in finding new components. They noted that while services are rapidly proliferating, there is a dearth of support for search and sensemaking in this space. Mackay [23] and MacLean [24] have explored the social side of end-user-created software—and the recent Koala work has made strides in

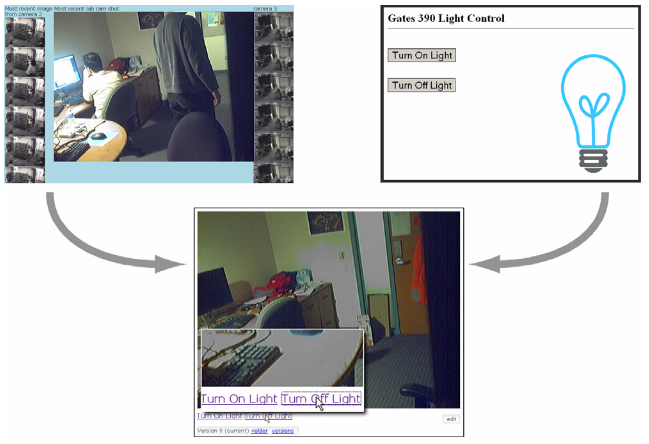


Figure 6. An example of a d.mix ubicomp mashup: web services provide video monitoring and lighting control.

this direction for the web [22]—we believe further efforts in this direction to be a promising avenue for future work.

Informants saw the merits of the d.mix approach to extend beyond the PC-based web browser. A researcher at an industrial research lab expressed interest in creating an "elastic office," where web-based office software is adapted for mobile devices. This focus on mobile interaction encouraged our interest in using a mash-up approach to tailoring web applications for mobile devices (see Figure 5).

Informants also raised the broader implications of a mash-up approach to design. A user experience designer and a platform engineer at the offices of a browser vendor raised end-user security as an important issue to consider. At a fashion-centered web startup, a web developer brought our attention to the legal issues involved in annotating sites in a public and social way.

Our recruiting method yielded informants with more expertise than d.mix's target audience; consequently, they asked questions about—and offered suggestions for raising—the ceiling of the tool. In a group meeting with 12 web designers and developers, informants expressed interest in creating annotations for a new API, and asked how time-consuming this process was. We explained that annotation in d.mix requires 5 to 10 lines per element; this was met with a positive response. A suggestion they offered for future work was for d.mix to fall back to HTML scraping when sites lack APIs.

EVALUATION

We conducted a first-use evaluation study with eight participants: seven were male, one female; their ages ranged from 25 to 46. We recruited participants with at least some web development experience. All participants had some level of college education; four had completed graduate school. Four participants had a computer science education; one was an electrical engineer; three came from the life sciences. As recruiting developers with Ruby experience proved difficult, only 4 participants had more than a passing knowledge of this scripting language. Everyone was familiar with HTML; six participants were familiar with JavaScript; and six with at least one other high-level scripting language. Four participants had some familiarity with web APIs, but only two had previously attempted to build a mash-up.

Study Protocol

Study sessions took approximately 75 minutes. We made three web sites with APIs available for sampling—Yahoo! web search, the Flickr photo sharing site, and YouTube, a video sharing site. For each site, d.mix supported annotations for a subset of the site's web API. For example, with Flickr, participants could perform full-text or tag searches and copy images with their metadata, but they could not extract user profile information. Participants were seated at a single-screen workstation with a standard web browser. We first demonstrated d.mix's interface for sampling from web pages, sending content to the wiki, and editing those pages. Next, we gave participants three tasks to perform.

The first task tested the overall usability of our approach—participants were asked to sample pictures and videos, send that content to the wiki, and change simple parameters of pasted elements, *e.g.*, how many images to show from a photo stream. The second design task was similar to our scenario — it asked participants to create an information dashboard for a magazine's photography editor. This required combining data from multiple users on the Flickr site and formatting the results. The third task asked participants to create a meta-search engine—using a text input search form, participants should query at least two different web services and combine search results from both on a single page. This task required generalizing a particular example taken from a website to a parametric form by editing the source code d.mix generated. Figure 7 shows two pages that one participant produced using d.mix. After completing the tasks, participants filled out a qualitative questionnaire on their experience and were also debriefed verbally.

Successes

On a high level, all participants understood and successfully used the workflow of browsing web sites for desired content or functionality, sampling from the sites, sending sampled items to the wiki, and editing items. Given that less than one hour of time was allocated to three tasks, it is notable that all participants successfully created dynamic pages for the first two tasks. In task 3, five participants created working meta-search engines (see Figure 7). However, for three of the participants without Ruby experience, its syntax proved a hurdle; they only partially completed the task.

Our participants were comfortable with editing the generated source code directly, without using the graphical property editor. Making the source accessible to participants allowed them to leverage their web design experience. For example, multiple participants leveraged their knowledge of CSS styles to change formatting and alignment of our generated code to better suit their aesthetic sensibility. Copy and paste within the wiki also allowed participants to reuse their work from a previous task in a later one.

In their post-test responses, participants highlighted three main advantages that d.mix offered to them compared to their existing toolset: elimination of setup and configuration barriers; enabling of rapid creation of functional web application prototypes; and lowering of expertise threshold.

First, participants commented on the advantage of having a browser-based editing environment. There was "minimum setup hassle," since "you don't need to set up your own server." One participant's comments sum up this point succinctly: "I don't know how to set up a Ruby/API environment on my web space. This lets me cut to the chase."

Second, participants also highlighted the gain in development speed. Participants perceived code creation by selecting examples and then modifying them to be faster than writing new code or integrating third party code snippets.

Third, participants felt that d.mix lowered the expertise threshold required to work with web APIs because they were not required to search or understand an API first. A web development consultant saw value in d.mix because he felt it would enable his clients to update their sites themselves.

Shortcomings

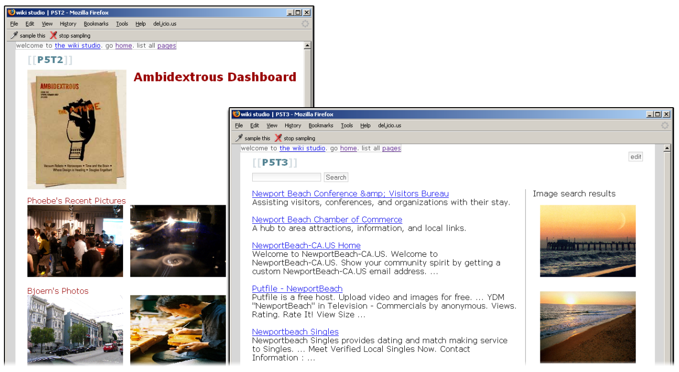
We also discovered a range of challenges our participants faced when working with d.mix. Universally, participants wished for a larger set of supported sites. This is a not a trivial request since annotation of web pages requires developer work. A longer public deployment is needed to gauge whether d.mix users can and will generate their own site-to-service maps on the wiki.

Other shortcomings discovered can be categorized into conceptual problems related to the action of sampling; difficulty of multi-language development; insufficient error-handling support in the wiki; and lack of documentation.

Inconsistent model of our sampling implementation

Participants were confused by limitations in what source elements were "sampling-aware." For example, to specify a query for a set of Flickr images in d.mix, the user currently must sample from the *link* to the image set, not the *results*. This suggests that the d.mix architecture should always enable sampling from both the *source* and from the *target* page. Also, where there is a genuine difference in effect, distinct highlighting treatments could be used to convey this.

Participants complained about a lack of visibility whether a given page would support sampling or not. Since rewriting pages through the d.mix proxy introduces a page-load delay,


wiki studio | P5T2 - Mozilla Firefox
File Edit View History Bookmarks Tools Help del.icio.us
sample this stop sampling
welcome to the wiki studio. go home. list all pages
[[P5T2]]
Ambidextrous Dashboard
Phoebe's Recent Pictures
Bjoern's Photos
wiki studio | P5T3 - Mozilla Firefox
File Edit View History Bookmarks Tools Help del.icio.us
sample this stop sampling
welcome to the wiki studio. go home. list all pages
edit
[[P5T3]]
Search
Image search results
Newport Beach Conference &amp; Visitors Bureau
Assisting visitors, conferences, and organizations with their stay.

Newport Beach Chamber of Commerce
A hub to area attractions, information, and local links.

NewportBeach-CA.US Home
Welcome to NewportBeach-CA.US. Welcome to NewportBeach-CA.US. Show your community spirit by getting a custom NewportBeach-CA.US email address. ...

Putfile - NewportBeach
Putfile is a free host. Upload video and images for free. ... YDM "NewportBeach" in Television - Commercials by anonymous. Views. Rating. Rate It! View Size ...

Newportbeach Singles
Newportbeach Singles provides dating and match making service to Singles. ... Meet Verified Local Singles Now. Contact Information : ...

Figure 7. Two pages a participant created during our user study. *Left image*: Information dashboard for a magazine editor, showing recent relevant images of magazine photographers. *Right image*: Meta-search engine showing both relevant web pages and image results for a search term.

participants browsed the web sites normally, and only turned on the sampling proxy when they had found elements they wished to sample. Only after this action were they able to find out whether the page was enhanced by d.mix. One means of addressing this is to provide feedback within the browser as to whether the page may be sampled; another would be to minimize the latency overhead introduced through the proxy so that users can always leave their browser in sampling mode.

Multi-language scripting

Dynamic web pages routinely use at least three different notation systems: HTML for page structuring, JavaScript for client-side interaction logic, and a scripting language such as Ruby for server-side logic. This mixing of multiple programming languages in a single web page introduces both flexibility and confusion for web developers.

d.mix's property sheets implementation exacerbated this complexity. It wrapped the generated Ruby code in a HTML *<div>* element whose attributes were used to construct the graphical editor, but were also read by the Ruby code inside the tag to parameterize web API calls. Participants were confused by this wrapping and unsuccessfully tried to insert Ruby variables into the *<div>* tag.

Lack of documentation & insufficient error handling

Many participants requested more complete documentation. One participant asked for more comments in the code explaining the format of API parameters. For example, two participants struggled to modify an image-search call to support multiple parameters. A related request was to provide structured editors in the graphical property sheets that offered alternative values and validated data entry.

Participants also complained that debugging their wiki pages was hard. Several participants complained about the "incomprehensible error messages" that syntax and execution errors generated. d.mix currently catches and displays Ruby sandbox exceptions, along with the source code that generated the exception.

How to go beyond the wiki environment?

Participants valued the active wiki for its support of rapid prototyping. However, because of a perceived lack of security, robustness and performance, participants did not regard the wiki as a viable platform for larger deployment. One participant remarked, "I'd be hesitant to use it for anything other than prototyping" and two others expressed similar reservations. Our motivation was to target situational applications with a small number of users. A real-world deployment would be needed to determine if the wiki is a suitable platform for deploying situational web applications.

Usability problems

Two smaller usability problems that disrupted participants' work were also discovered: from experience with shopping carts on commerce web sites, participants expected the sampling bin to be persistent across different pages within a web site. Participants also wished that the "send to wiki" dialog offered a drop-down list of existing wiki pages instead of requiring them to enter a full page name each time.

RELATED WORK

d.mix draws on existing work in three areas. First, it draws on research for end-user modification of the web. Second, it relates to tools that lower the threshold of synthesizing web applications. Third, d.mix relates to projects that deal with locating, copying, and modifying program documentation and examples. We discuss each area in turn.

Tools for end-user modification of web experiences

Greasemonkey [2], Chickenfoot [9] and Koala [22] are client-side Firefox browser extensions that enable users to rewrite web pages and automate browsing activities. Greasemonkey enables the use of scripts that alter web pages as they are loaded; users create these scripts manually, generally using JavaScript to modify the page's Document Object Model (DOM). Chickenfoot builds on Greasemonkey, contributing an informal syntax based on keyword pattern matching; the primary goal of this more flexible syntax was to enable users with less scripting knowledge to create scripts. Koala further lowers the threshold, bringing to the web the approach of creating scripts by generalizing the demonstrated actions of users (*e.g.*, [11, 26]).Of this prior work, Koala and d.mix are the most similar. d.mix shares with Koala the use of programming-by-demonstration techniques and the social-software mechanism of sharing scripts server-side on a wiki page. d.mix distinguishes itself in three important ways. First, Chickenfoot and Koala are end-user technologies that shield users from the underlying representation. d.mix's approach is more akin to visual web development tools such as Adobe Dreamweaver [1], using visual representations when they are expedient, yet also providing access to the code. Supporting direct editing of source enables experts to perform more complex operations; it also avoids some of the "round-trip" errors that can arise when users iteratively edit an intermediate representation. Second, prior work focuses on automating web browsing and rewriting web pages using the DOM in the page source—they do not interact with web service APIs directly. In contrast, d.mix leverages the web page as the site for users to demonstrate content of interest; d.mix's generalization step maps this to a web service API, and stores *API calls* as its underlying representation. Third, with d.mix, the code is actually executed server-side, in addition to being stored server-side. In this way, d.mix takes an infrastructure service approach to support end-user remixing of web pages. This approach obviates the need for users to install any software on their client machine, and the increasing use of the web as a software platform provides evidence as to the merit of this approach.

Tools for end-user synthesis of web experiences

In addition to tools that support *modification* of a web page's DOM, there are several tools that lower the expertise threshold required to create web applications that *synthesize* data from multiple pre-existing sources. Most notably, Yahoo! Pipes [6], Open Kapow [5], and Marmite [34] are tools that employ a dataflow approach for working with web services.

Yahoo! Pipes also offers a visual node-and-link editor for manipulating web data sources. It focuses on visually rewriting RSS feeds. Open Kapow offers a desktop-based visual editing environment for creating new web services by combining data from existing sites through API calls and screen scraping. Services are deployed on a remote "mash-up server." The main difference between these systems and d.mix is that Kapow and Pipes are used to create web services meant for programmatic consumption, not applications or pages intended directly for users.

The Marmite browser extension draws on the dataflow approach manifest in Unix pipes and more recent visual tools, such as Apple's Automator. The sources in Marmite consist of calls to web services and the use of Marmite's screen scraper. Perhaps Marmite's strongest contribution to end-user programming for the web lies in its linked representation of program implementation and state: the implementation is represented through visual data flow and the current state is visualized as a spreadsheet. The user experience benefit of this linked view is an improved understanding of application behavior. Unlike d.mix, Marmite applications run client side. An additional distinction from d.mix is that the end-user approach of Marmite is based on visual dataflow. One of the challenges of data flow, as the Marmite authors note, is that users have "a hard time knowing what operation to select"—we suggest that the direct manipulation embodied in d.mix's programming-by-demonstration approach ameliorates this gulf-of-execution [18] challenge.

IBM's QEDWiki uses a widget-based approach to constructing web applications in a hosted wiki environment. QEDWiki's widgets are similar to Marmite's data sinks. This approach suggests two distinct communities—those that create the widget library elements, and those that use the library elements—echoing prior work on a "tailoring culture" within Xerox Lisp Buttons [24]. d.mix shares QEDWiki's interest in supporting different "tiers" of development, with two important distinctions. First, d.mix does not interpose the additional abstraction of creating graphical widgets; with d.mix, users directly browse the source site as the mechanism for specifying interactive elements. Second, d.mix better preserves the underlying modifiability of remixed applications by exposing script code on demand.

Finding and appropriating documentation and code

The literature has shown [13, 16, 20] that programmers often create new functionality by finding an example online or in a source repository—less code is created tabula rasa than might be imagined. Recent research has begun to more fully embrace this style of development. The Mica system [32] augments existing web search tools with navigational structure specifically designed for finding API documentation and examples. While Mica and d.mix both address the information foraging issues [29] involved in locating example code, their approaches are largely complementary.

Several tools have supported mechanisms for copying web content and interface widgets in a structured manner [14, 28, 30]. Most related to d.mix, Citrine [33] introduced techniques for structured copy and paste between desktop applications, including web browsers. Citrine parses copied text, creating a structured representation that can be pasted in rich format, *e.g.*, as a contact record into Microsoft Outlook. d.mix extends idea of structured copy into the domain of source code. With d.mix however, the structuring is performed by the extensible site-to-service map as opposed to through a hard-coded set of templates.

LIMITATIONS AND FUTURE WORK

This section discusses limitations of the current implementation of d.mix and implications for future work.

The primary concern of this paper is an exploration of the approach of authoring by sampling, not with the details of a public deployment of such a tool. As such, there are security and authentication issues that a widely-released tool would need to address. Most notably, the current d.mix HTTP proxy does not handle cookies of remote sites as a client browser would. This precludes sampling from the "logged-in web" —pages that require authentication beyond basic API keys.

A second limitation is that using d.mix is currently limited to sites that are amenable to web scraping—*i.e.*, those that generate static HTML, as opposed to sites that rely heavily on AJAX or Flash for their interfaces.

Third, a comprehensive tool should offer support both for working with content that *is* accessible through APIs and content that *is not* [16]. d.mix could be combined with existing techniques for scraping by demonstration.

Lastly, while d.mix is built on wikis, a social editing technology, we have not yet evaluated how use by multiple developers would change the d.mix design experience. Prior work on desktop software customization has shown that people do share their customization scripts [23]. It would be worthwhile to study to what extent this holds to rewriting the web, and what characteristic differences there are in this domain. It is our goal to have an open deployment in the future to study these questions.

CONCLUSIONS

We have introduced the technique of programming by a sample through d.mix, a tool that embodies this technique. d.mix addresses the challenge of becoming familiar with a web service API and provides a rapid prototyping solution structured around the acts of sampling content from an API-providing web site and then working with the sampled content in an active wiki. Our system is enabled on a conceptual level by a mapping from HTML pages to the API calls that would produce similar output. On a technical level, our system is enabled by a programmable proxy server and a sandbox execution model for running scripts within a wiki. Together with our past work [15, 17] we regard d.mix as a building block towards new authoring environments that facilitate prototyping of rich data and interaction models.

ACKNOWLEDGMENTS

We thank Leith Abdulla for programming and video production help, whytheluckystiff for Ruby support, and Wendy Ju for comments on this paper.

REFERENCES

1 *Dreamweaver*, 2007. Adobe Inc. http://www.adobe.com/products/dreamweaver

2 *Greasemonkey*, 2007. http://greasemonkey.mozdev.org

3 *Hpricot, a fast and delightful HTML parser*, 2007. http://code.whytheluckystiff.net/hpricot

4 *The MouseHole scriptable proxy*, 2007. http://code.whytheluckystiff.net/mouseHole

5 *Open Kapow*, 2007. Kapow Technologies. http://www.openkapow.com

6 *Pipes*, 2007. Yahoo! http://pipes.yahoo.com

7 *Service-oriented computing*. Communications of the ACM, M.P. Papazoglou and D. Georgakopoulos, ed. Vol. 46.

8 Anderson, C., *The Long Tail*: Random House Business. 2006.

9 Bolin, M., M. Webber, P. Rha, T. Wilson, and R. C. Miller, Automation and customization of rendered web pages, in *UIST 2005: ACM Symposium on User Interface Software and Technology*. 2005.

10 Curtis, B., W. Vicknair, and S. Nickolas, *QEDWiki*, 2007. IBM Alphaworks. http://services.alphaworks.ibm.com/qedwiki/

11 Cypher, A., EAGER: programming repetitive tasks by example, in *CHI: ACM Conference on Human Factors in Computing Systems*. 1991.

12 Cypher, A., ed. *Watch What I Do - Programming by Demonstration*. MIT Press: Cambridge, MA. 652 pp., 1993.

13 Fairbanks, G., D. Garlan, and W. Scherlis, Design fragments make using frameworks easier, in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 2006.

14 Fujima, J., A. Lunzer, K. Hornb, and Y. Tanaka, Clip, connect, clone: combining application elements to build custom interfaces for information access, in *UIST 2004: ACM Symposium on User Interface Software and Technology*. 2004.

15 Hartmann, B., L. Abdulla, M. Mittal, and S. R. Klemmer. Authoring Sensor Based Interactions Through Direct Manipulation and Pattern Matching. In Proceedings of *CHI 2007: ACM Conference on Human Factors in Computing Systems*, 2007.

16 Hartmann, B., S. Doorley, and S. R. Klemmer, *Hacking, Mashing, Gluing: A Study of Opportunistic Design and Development*. Technical Report, Stanford University Computer Science Department, October 2006.

17 Hartmann, B., S. R. Klemmer, M. Bernstein, L. Abdulla, B. Burr, A. Robinson-Mosher, and J. Gee. Reflective physical prototyping through integrated design, test, and analysis. In Proceedings of *UIST 2006: ACM Symposium on User Interface Software and Technology*, 2006.

18 Hutchins, E. L., J. D. Hollan, and D. A. Norman. Direct Manipulation Interfaces. *Human-Computer Interaction* **1**(4). pp. 311-38, 1985.

19 Huynh, D. and S. Mazzocchi, *Solvent Firefox Extension*, 2007. http://simile.mit.edu/wiki/Solvent

20 Kim, M., L. Bergman, T. Lau, and D. Notkin, An Ethnographic Study of Copy and Paste Programming Practices in OOPL, in *Proceedings of the 2004 International Symposium on Empirical Software Engineering*. 2004, IEEE Computer Society.

21 Lieberman, H., ed. *Your Wish is my Command*. ed. Morgan Kaufmann. 416 pp., 2001.

22 Little, G., T. A. Lau, J. Lin, E. Kandogan, E. M. Haber, and A. Cypher. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In Proceedings of *CHI 2007:ACM Conference on Human Factors in Computing Systems*, 2007.

23 Mackay, W. E., Patterns of sharing customizable software, in *CSCW 1990:ACM conference on Computer-supported cooperative work*. 1990.

24 MacLean, A., K. Carter, L. Lövstrand, and T. Moran, User-tailorable systems: pressing the issues with buttons, in *CHI 1990: ACM Conference on Human Factors in Computing Systems*. 1990.

25 Myers, B., S. E. Hudson, and R. Pausch. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction* **7**(1). pp. 3–28, 2000.

26 Myers, B. A., Peridot: Creating User Interfaces by Demonstration, in *Watch What I Do: Programming by Demonstration*. MIT Press. pp. 125-53, 1993.

27 Nardi, B. A., *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA: MIT Press. 1993.

28 Ozzie, R., *Live Clipboard*, 2007. http://www.liveclipboard.org/

29 Pirolli, P. and S. Card. Information Foraging. *Psychological Review* **106**(4). pp. 643-75, 1999.

30 schraefel, m. c., Y. Zhu, D. Modjeska, D. Wigdor, and S. Zhao. Hunter Gatherer: Interaction support for the creation and management of within-web-page collections. In Proceedings of *International World Wide Web Conference*. pp. pp. 172-81, 2002.

31 Shirky, C., *Situated Software*, 2004. http://www.shirky.com/writings/situated_software.html

32 Stylos, J. and B. Myers. A Web-Search Tool for Finding API Components and Examples. In Proceedings of *IEEE Symposium on Visual Languages and Human-Centric Computing*. pp. 195-202, 2006.

33 Stylos, J., B. A. Myers, and A. Faulring, Citrine: providing intelligent copy-and-paste, in *UIST 2004: ACM Symposium on User Interface Software and Technology*. 2004.

34 Wong, J. and J. Hong. Making Mashups with Marmite: Re-purposing. In Proceedings of *CHI 2007:ACM Conference on Human Factors in Computing Systems*, 2007.