Programming by a Sample: Rapidly Creating 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
Source-code examples of APIs enable developers to quickly gain a gestalt understanding of a library's functionality, and they support organically creating applications by incrementally modifying a functional starting point. As an increasing number of web sites provide APIs, significant latent value lies in connecting the complementary representations between site and service—in essence, enabling sites themselves to be the example corpus. We introduce *d.mix*, a tool for creating web mashups that leverages this *site-to-service* correspondence. With d.mix, users browse annotated web sites and select elements to sample. d.mix's sampling mechanism generates the *underlying service calls* that yield those elements. This code can be edited, executed, and shared in d.mix's wiki-based hosting environment. This sampling approach leverages pre-existing web sites as example sets and supports fluid composition and modification of examples. An initial study with eight participants found d.mix to enable rapid experimentation, and suggested avenues for improving its annotation mechanism.

ACM Classification: D.2.2 [Software Engineering]: Design Tools and Techniques—*User interfaces*. H5.2 [Information interfaces and presentation]: User Interfaces—*Graphical user interfaces*.
General terms: Design, Human Factors

Keywords: Programming by example modification, web services, mashups, prototyping

INTRODUCTION
Web hosting and search have lowered the time and monetary costs of disseminating, finding, and using APIs. The number and diversity of application building blocks that are openly available as web services is growing rapidly [7]. Programmableweb.com lists 2155 mashups leveraging 478 distinct APIs as of July 2007 (the most popular being Google Maps, Flickr, and Amazon). These APIs provide a rich selection of interface elements and data sources. Many serve as the programmatic interface to successful web sites, where the site and its 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.* However, the value that could be achieved by coordinating these representations has largely remained latent.

UIST'07, October 7–10, 2007, Newport, Rhode Island, USA.

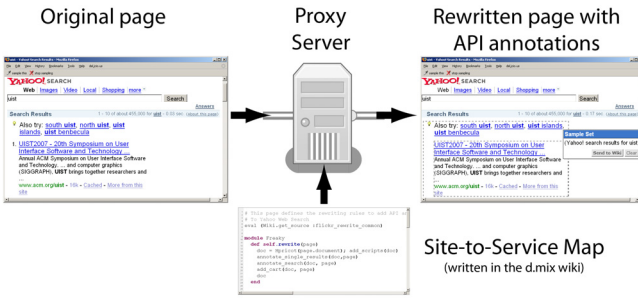


Figure 1. With d.mix, users browse web sites through a proxy that marks API-accessible content. Users select marked elements they wish to copy. Through a site-to-service map, d.mix composes web service calls that yield results corresponding to the user's selection. This code is copied to the d.mix wiki for editing and hosting.

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" [32]. The small audience of situated software applications limits developer resources. As such, enabling faster and lower-threshold [26] authoring of these applications provides a catalyst for broader creation.

An emerging approach that leverages web APIs for situated software is *mashups*, software created by combining elements from multiple third-party web services. Mashups are instances of the long tail [8] of software, the many small applications that cumulatively have a big impact. A broad shift of the mashup 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].

Integrating Site and Service
Two factors currently hamper broader use of web APIs: the complexity of understanding and using web services, and the complexity of installing web application environments.

To enable flexible and rapid authoring of API-based web applications, this paper introduces *d.mix* (see Figure 1), a web-based design tool with two notable attributes. The first

is a programmable proxy system providing a *site-to-service map* that establishes the correspondence between elements shown on the site and the web service calls needed to replicate these elements programmatically. This map enables users to create code by browsing web sites and visually specifying the 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. This 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 application development.

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 mashups. d.mix offers a graphical interaction path for selecting samples, pasting them into a new page, and changing their attributes using property sheets (see Figure 2). Additionally, by virtue of displaying the actual underlying code to users, d.mix allows developers with sufficient technical expertise to drill down into source code as needed.

Foraging for Examples

As the number and size of programming libraries swells, locating and understanding documentation and examples is playing an increasingly prominent role in developers' activities [34]. To aid users in foraging for example code, d.mix co-locates two different kinds of information on one page: examples of what functionality and what 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 [30]. 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 enables developers to dynamically generate code snippets for a web service API as they browse the canonical example of its functionality: the web site itself.

d.mix's approach draws on prior work in *programming by example*, also known as *programming by demonstration* [12, 21, 28]. 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.

While d.mix shares much of its approach with programming-by-example systems, it differs in the procedure for generating examples. Instead of specifying logic by demonstrating *novel* examples, with d.mix designers choose and parameterize *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 [28]. Modification of a working example also speeds development because it provides stronger scaffolding than writing code *tabula rasa*.

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 mashup design tools. Second, to gauge the initial experience of d.mix, we conducted a preliminary lab study with eight web developers. Third, we created applications with d.mix to explore a broader range of interaction designs.

The rest of this paper is structured as follows. We first introduce the main interaction techniques of d.mix through a scenario. Subsequently, we explain the d.mix implementation. We then describe the iterative feedback from web professionals, an initial laboratory study, and applications we created with d.mix. We conclude with a discussion of related research and commercial systems, limitations of the

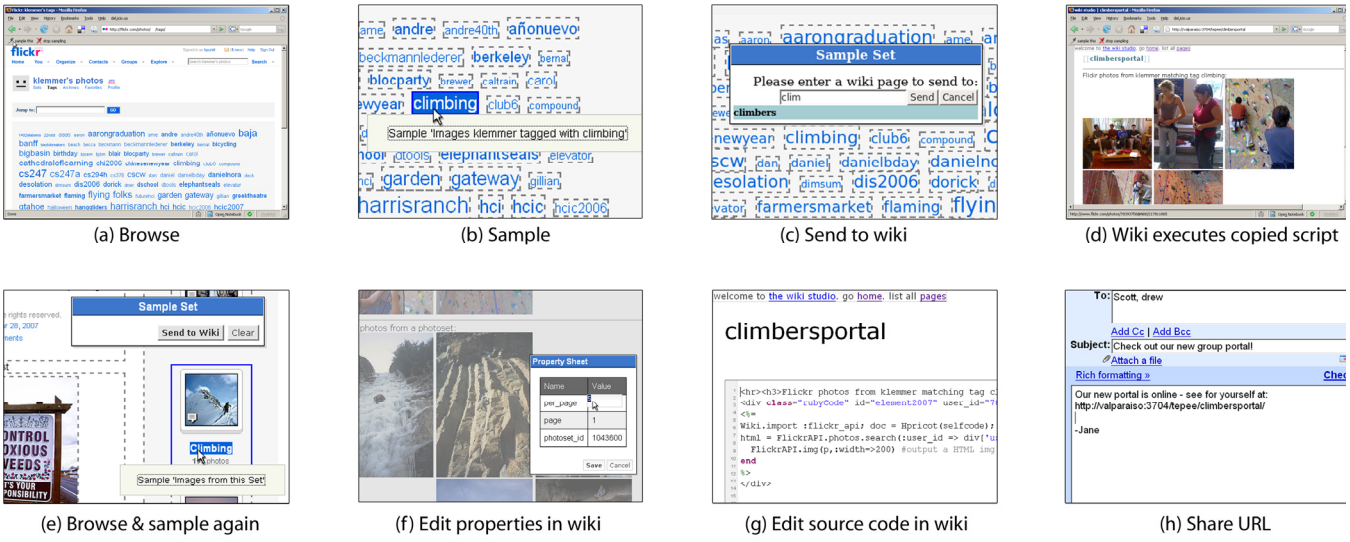


Figure 2. With d.mix, users switch between foraging for content and editing copies of that content in an active wiki environment.

current implementation, and an outlook to future work.

HOW TO PROGRAM BY A SAMPLE

A scenario will help introduce the main interaction techniques. 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 their 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. David uses the photo site Flickr and marks his pictures with the tag "climbing." Another also uses Flickr, but uses an image set instead. A third 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 David'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 copy into her *sampling bin*.

She right-clicks on the tag "climbing," to invoke a context menu. This menu offers the choice to copy the set of images that David has tagged with "climbing" (see Figure 2b). The copied item appears in her sampling bin, a floating layer atop 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 displays 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 David'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 showing that the content came from a Flickr photo set. This sheet 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.

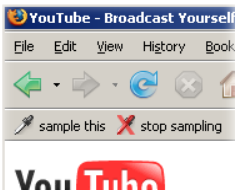
Next Jane would like to layout the images and add some text. She clicks on "edit source," which displays an HTML document, in which each of the three samples she inserted corresponds to a few lines of Ruby script, enclosed by a structuring <*div*> tag (see Figure 2g). She adds text and a table structure around the images. Remembering that David 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 the d.mix active wiki. This invokes the d.mix proxy, combining the target site's original web markup with annotations found using the *site-to-service map* (see Figure 1).

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. These maps describe the programmatically accessible components associated with a particular set of URLs (see Figure 3). Each map defines a partition of a web site into page types through pattern matches on URLs. For each page type, the map then defines the correspondence between the markup found on a page and the API method invocations needed to retrieve the equivalent content programmatically. It does so by searching for known markup patterns—using XPath and CSS selectors—and recording the metadata that will be passed to web services as parameters, such as a user or photo ID, a search term, or a page number.

For example, on Flickr.com, 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 retrieved 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. Second, d.mix's annotation visually augments the elements that can be sampled with a dashed border as an indication to the user.

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

d.mix is implemented in the Ruby programming language. We chose Ruby to leverage its meta-programming libraries for code generation and dynamic evaluation, and the freely available programmable Ruby HTTP proxy, the mouseHole [4].

Parametric Copy 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. 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 ascertain the items' *class*) as well as the content of the page (to ascertain their *ID*).

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 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 scrapes web pages as the developer visits them in order to extract the needed information for code generation. Scraping can be brittle because matching expressions can break when site operators change class or ID attributes of their pages. Still, it is common practice in web development [16] as it is often the only technique for extracting data without the site operator's cooperation. 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 mashup in the scenario. By scraping parameters first, d.mix's user-created pages can then make API calls at *runtime*, which tend to be more stable than the HTML format of the initial example pages.

We acknowledge that building rewrite and code generation rules procedurally is time-intensive and requires expertise with DOM querying through XPath or CSS. This barrier could be lowered through a declarative, domain-specific language. In addition. UI tools such as Solvent [19] that support building DOM selectors visually could allow markup selection by demonstration. Providing a smooth process for creating the site-to-service maps is important but somewhat orthogonal to this paper's contributions. 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 subsequent developers can leverage that effort.

Server-side Active Wiki Hosts and Executes Scripts

In d.mix's active wiki, developers can freely mix text, HTML, and CSS to determine document structure, as well as Ruby script to express program logic. When a developer enters a new page name in the *Send to Wiki* dialog, a wiki page of that name is created and the generated source code is pasted into that page. If the target wiki page already exists, the generated code is appended to it. The browser then displays a rendered version of the wiki page by executing the specified web API calls (see Figure 2d). 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.

To switch from the rendered view to the source view containing web markup and Ruby code, a user can click on the *edit* button, as in a standard wiki. The markup and snippets of script are then shown in a browser-based text editor, which provides syntax highlighting and line numbering. Clicking on the *save* button saves the document source as a new revision, and redirects the user back to the rendered version of the wiki page. The rapid switching between rendered view, and markup and application logic minimizes the cognitive friction involved in keeping track of the model and the rendered view of data-driven pages.

When evaluating Ruby code, the active wiki does so in a sandbox, to reduce the security risks involved. The sandbox cannot access objects such as the File class, but can maintain application state in a database and make web service calls through SOAP, REST, or other web service protocols.

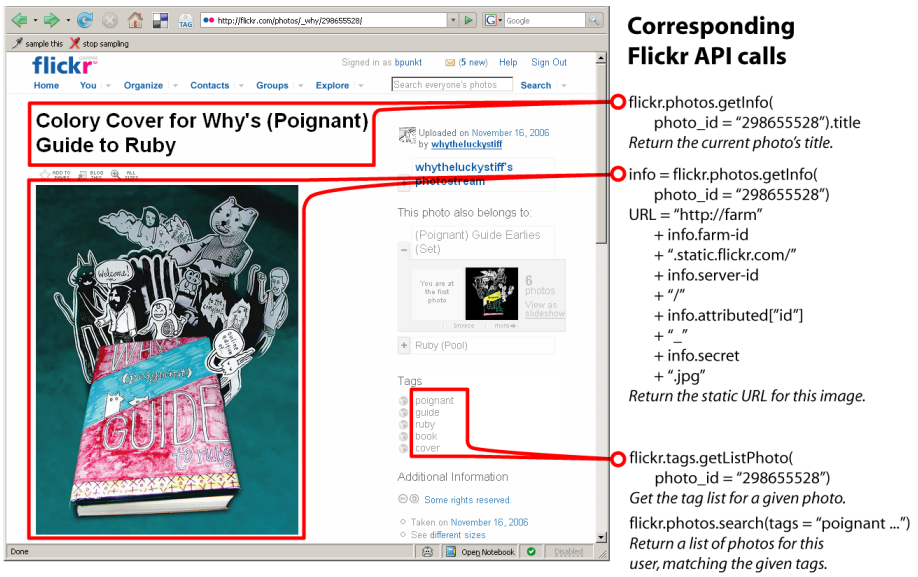


Figure 3. The site-to-service map defines a correspondence between HTML elements and web service API calls. This graphic highlights this mapping for three items on Flickr.

Pasted Material Can Be Parameterized and Edited

In comparison to a standard copy-and-paste of web content, the notable advantage of d.mix's *parametric copy* is that it copies a richer representation of the selected data. This allows an element's parameters to be changed after pasting. The d.mix wiki offers graphical editing of parameters through property sheets. The structure of these property sheets, implemented as floating layers in JavaScript, is determined during the code generation step. In our current implementation, properties correspond to the parameters passed to web service APIs. It may be valuable to provide additional parameters such as formatting commands.

Widget-based wiki platforms (*e.g.*, QEDWiki [10]) also offer parameter-based editing of their widgets—but they typically do not offer access to the underlying widgets' source-code representation. In contrast, d.mix generates Ruby script, which can be edited directly. d.mix does not currently support WYSIWYG wiki editing, but such functionality could be added in the future. Like other development environments, the active wiki offers code versioning.

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 that the 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.

The generated code makes calls into d.mix modules which broker communication between the active wiki script and the web services. For example, users' Ruby scripts often need to reference working API keys to make web service calls. d.mix modules provide a default set of API keys so that users can retrieve publicly accessible data from web services without having to obtain personal keys. 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 one can issue), we believe our solution works well for prototyping and for deploying situational applications with a limited number of users.

Built-in Sharing through Server-Side Hosting

An important attribute of the d.mix wiki is that public sharing is the default and encouraged state. An end-user can contribute a 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. When a d.mix user creates a new page that remixes content from multiple data sources, another end-user can just as easily remix the remix—copying, pasting, and reparameterizing the elements from one active wiki page to another.

Figure 4. 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.

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 existing web pages to improve 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 the event listing site *Upcoming.org* and the third-party calendaring site *30 Boxes*. By parsing an 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 4). 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 for one author and a colleague 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 filters can only be configured by administrators.

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 mashups 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 and "remix" functionality to extend into physical space. To explore d.mix design opportunities in web-enabled ubicomp applications, we augmented two smart devices in our laboratory to support API sampling: a camera that publishes a video feed of lab activity, and a network-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 5).

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 embed API annotations directly into the served web pages.

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 had 12 members. Informants included attendees of Ruby user group meetings, web developers at startup companies in Silicon Valley, and researchers at industrial research labs interested in web technologies.

Informants repeatedly raised scaling concerns for mashups. An early informant was a web developer at a 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 informant noted 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 mashup approach can be used to create widely distributed 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 mashups expands, informants were interested in how users and developers might locate relevant services. Several informants 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 recent work on Koala [22] has made strides in enabling sharing of end-user browser automations. We see further efforts in this direction as 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 kindled our interest in using a mashup approach to tailoring web applications for mobile devices (see Figure 4).

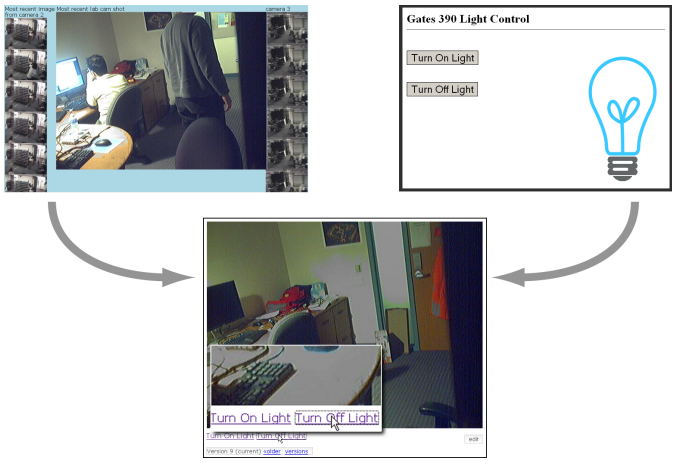


Figure 5. An example of a d.mix ubicomp mashup: web services provide video monitoring and lighting control.

Informants also raised the broader implications of a mashup 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 require about 10 lines per element; this was met with a positive response. For future work they suggested that d.mix could 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 college education; four had completed graduate school. Four had a computer science education; one was an electrical engineer; three came from the life sciences. Recruiting developers with Ruby experience proved difficult—only four 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 scripting language. Four participants had some familiarity with web APIs, but only two had previously attempted to build a mashup.

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 the YouTube 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 6 shows two pages that one participant, who was web design-savvy but a Ruby novice, produced. After completing the tasks, participants filled out a qualitative questionnaire on their experience and were 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 6). 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 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 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 request is not trivial because creating new annotations requires manual programming effort. However, we believe the amount of effort is reasonable when amortized over a large number of users. Other shortcomings fall into four categories. First, coexistence of two different sampling strategies caused confusion about how to sample from a data set. Second, participants had difficulty switching between multiple languages interspersed in a single wiki page. Third, documentation and error-handling in the wiki was insufficient compared to other tools. Fourth, wiki-hosted applications may not scale well beyond prototypes for a few users.

Inconsistent models for sampling

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, 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 shortcoming 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 mix of multiple programming languages in a single document introduces both flexibility and confusion for web developers.

d.mix's property sheet 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

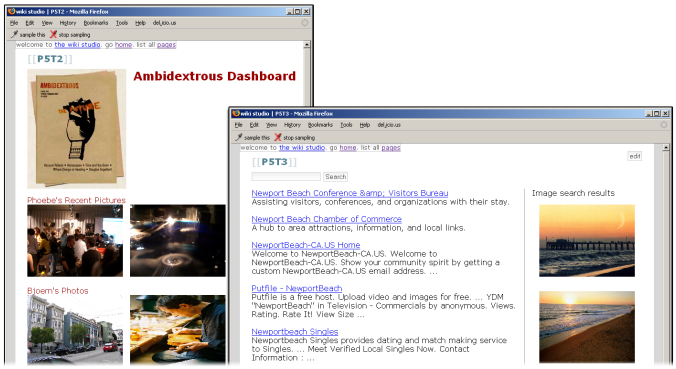


Figure 6. Two pages a participant created during our user study. *Left image*: Information dashboard for a magazine editor, showing recent relevant images of magazine photographers. *Right image*: Meta-search engine showing both relevant web pages and image results for a search term.

confused by this wrapping and unsuccessfully tried to insert Ruby variables into the <*div*> tag.

Lack of documentation & error handling

Many participants requested more complete documentation. One participant asked for more comments in the generated code explaining the format of API parameters. A related request was to provide structured editors inside property sheets that offered alternative values and data validation.

Participants also commented that debugging their wiki pages was hard, since syntax and execution errors generated "incomprehensible error messages." d.mix currently catches and displays Ruby exceptions along with source code that generated the exception, but it does not interpret or explain the exceptions.

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.

RELATED WORK

d.mix draws on existing work in three areas: tools for end-user modification of the web, tools that lower the threshold of synthesizing new web applications, and research on 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. 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, 27]). 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 source code. Supporting direct editing of the 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, Chickenfoot and Koala focus 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 stored and executed 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—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 the second category, 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 [37] employ a dataflow approach for working with web services.

Yahoo! Pipes draws on the dataflow approach manifest in Unix pipes, introducing 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 "mashup 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 offers a graphical language for sequentially composing web service data sources and transformation operations; the interaction style is somewhat modeled after Apple's Automator system for scripting desktop behaviors. 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 dataflow 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, and thus cannot be shared. An additional distinction is that the Marmite programming model is one of *tabula rasa* composition, while d.mix's is based on example modification. Clearly, both approaches have merit, and neither is globally optimal. A challenge of composition, as the Marmite authors note, is that users have difficulty "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 construct web applications in a hosted wiki environment. 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 shows that programmers often create new functionality by finding an example online or in a source repository [13, 16, 20]—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 [34] 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 [30] involved in locating example code, their approaches are largely complementary.

Several tools have offered structured mechanisms for "deeply" copying content. Most related to d.mix, Citrine [35] 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.

In other domains, Live Clipboard [29] and Hunter Gatherer [31] aided in copying web content; Clip, connect, clone enabled copying web forms [14]; and WinCuts [36] and Façades [33] replicate regions of desktop applications at the window management level. Broadly speaking, d.mix differs from this prior work by generating code that retrieves content from web services rather than copying the content itself.

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 authoring by sampling. 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. Extending d.mix to the logged-in web comprises two concerns: sampling from pages that require authentication to view, and then subsequently performing authenticated API calls to retrieve the content for remixed pages. To sample from logged in pages, both client-side solutions, *e.g.*, a browser extension that forwards a full DOM to d.mix to be rewritten, and server-side solutions, *e.g.*, through utilizing of a "headless" browser [25] are possible. To perform API calls authenticated with private tokens, the private variable mechanism of Koala [22] could be adapted. Implementation of private data would also require addition of access permissions through user accounts to the d.mix wiki.

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 share their customization scripts [23]. It would be valuable to study code sharing practices on the web.

CONCLUSIONS

We have introduced the technique of programming by a sample through d.mix. 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 and Michael Krieger for programming and video production help; whytheluckystiff for Ruby support; and Wendy Ju for comments. This research was supported through NSF grant IIS-0534662; a Microsoft New Faculty Fellowship; a PC donation from Intel; and a SAP Stanford Graduate Fellowship for Björn Hartmann.

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*. 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. Papazoglou and D. Georgakopoulos. Vol. 46.

8 Anderson, C., *The Long Tail*: Hyperion. 256 pp. 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 *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, *et al.*, Reflective Physical Prototyping through Integrated Design, Test, and Analysis. In *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*. 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 *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 Mazzocchi, S. and R. Lee, *Crowbar*, 2007. http://simile.mit.edu/wiki/Crowbar

26 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.

27 Myers, B. A., Peridot: Creating User Interfaces by Demonstration. In *Watch what I do: programming by demonstration*. MIT Press. pp. 125-53, 1993.

28 Nardi, B. A., *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA: MIT Press. 178 pp. 1993.

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

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

31 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. 172-81, 2002.

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

33 Stuerzlinger, W., O. Chapuis, D. Phillips, and N. Roussel, User Interface Façades: Towards Fully Adaptable User Interfaces. In *UIST 2006: ACM Symposium on User Interface Software and Technology*. 2006.

34 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.

35 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.

36 Tan, D., B. Meyers, and M. Czerwinski, WinCuts: Manipulating arbitrary window regions for more effective use of screen space. In *CHI 2004: ACM Conference on Human Factors in Computing Systems*. 2004.

37 Wong, J. and J. Hong, Making Mashups with Marmite: Towards End-User Programming for the Web. In *CHI 2007: ACM Conference on Human Factors in Computing Systems*. 2007.