Hacking, Mashing, Gluing: A Study of Opportunistic Design and Development

Björn Hartmann, Scott Doorley, Scott R. Klemmer
Stanford University HCI Group
Computer Science Department, Stanford, CA 94305-9035, USA
{bjoern, sdoorley, srk}@stanford.edu

ABSTRACT
This paper is about opportunistic practices in interactive system design: about copying and pasting source code from public online forums into one's own scripts; about taking apart consumer electronics and reappropriating their components for design prototypes; about "Frankensteining" software and hardware artifacts together by joining them with physical and digital hot glue and duct tape. It is about the hacks and prototypes of lowbrow experimentation, as opposed to highbrow design and engineering from the ground up. We combine these opportunistic practices under the moniker of "mash-up design." This paper presents results from an interview study with 14 professional and hobbyist "mashers" from three different design disciplines: Web 2.0 programmers, hardware hackers, and designers of interactive ubicomp systems. The paper analyzes commonalities and distills themes in opportunistic design through three lenses: first, the way mash-ups modify and combine pre-existing elements; second, the unique characteristics of opportunistic design as an activity; and third, looking at mash-ups as novel kinds of artifacts.

Author Keywords
Mash-ups, long tail, opportunistic programming, hacking

ACM Classification Keywords
D.2.11. [Software Engineering]: Software Architectures — *Patterns*. K.6.3. [Management of Computing and Information Systems]: Software Management — *Software Process*. D.2.2 [Software Engineering]: Design Tools and Techniques — *Modules and Interfaces*.

INTRODUCTION
This paper presents an investigation into a set of practices that run as a common thread through the disparate enterprises of web developers, hardware hackers, and builders of ubiquitous computing systems: opportunistic design and development. This paper is also about the *long tail* [1] of software (and hardware) in a networked world. The long tail comprises the "non-hits" in a genre — books, music, movies—that individually sell little, but collectively have a large impact.

Software has always had a long tail — the shell scripts of system administrators, the spreadsheets of financial workers — but three recent web-based trends have conspired to significantly fatten and lengthen the tail. The first is the ability of web dissemination to lower the distribution costs for software. The second is the ability of web search to lower the costs for finding compiled software, source code snippets, and problem solving tips. The third is the rise of publicly available web APIs, and the opening of commercial desktop applications through integrated scripting engines.

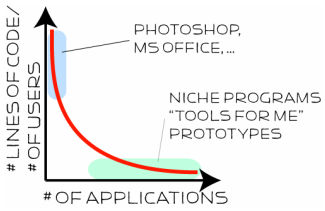


Figure 1. Different kinds of software systems call for different development tools and practices.

Physical sensing and actuation technologies have followed a different yet parallel trajectory. Today, embedded electronics are programmed on the PC; and prototyping tools have provided software abstractions to sensing and actuation to experts and novices alike [2, 13, 14]. While hardware itself cannot easily be copied and transmitted by digital means, common platforms and shared designs enable people or machines to replicate and program hardware.

The goal of our group's opportunistic programming research project is to understand how opportunistic programming is practiced today, how it differs from traditional software engineering (see Figure 1), and to design tools that both support existing practices and enable new ones. In this paper, we study how mash-up programming of software and hardware takes place in web development shops, design studios, and research labs today. This paper contributes an analysis of common themes in opportunistic design based on an interview study. We describe how designers choose between *deep* and *surface-level* integration of components; how mash-ups have *epistemic*, *pragmatic*, and *intrinsic* values for creators; how *shopping* becomes a central activity; and what user-experience challenges arise in *mixed-fidelity artifacts*.

A Short Etymology of Mash-ups
The term *mash-up* first surfaced in the electronic music community to denote the practice of taking elements of two or more existing songs and creating a new piece by rearranging, interspersing and superimposing samples from the different source songs. Mash-ups represent an extension of the *remix*, in which one producer takes a single track of

another producer and delivers a reinterpretation that balances the original work with elements specifically composed to accompany the source material. Mash-ups differ from remixes in that there are multiple sources and that the artistic effort lies in the arrangement, the "gluing together" of parts in novel ways, rather than the composition of additional new musical material.

Recently, computer science has adopted the term mash-up to refer to pieces of software created by programming against one or more public web APIs, also known as infrastructure services [5]. The most popular of these are Google Maps and Flickr. A typical mash-up — on programmableweb.com for example — shows data relevant to a local community, say coffee shops in San Francisco, using a Google or Yahoo map navigation UI.

We take a broad view on what constitutes mash-ups. We look beyond the web to examine the process of recombination and ad-hoc design across a spectrum of ubiquitous computing systems. Our working definition of a mash-up is "a *combination* of pre-existing, integrated units of technology, *glued together* to achieve new functionality, as opposed to creating that functionality from scratch." To help clarify the position of mash-ups in the design and engineering landscape, we contrast them briefly with traditional software engineering.

Waterfall Development and Opportunistic Programming

Broadly speaking, classical software engineering has primarily concerned itself with metrics such as performance, reliability, defects, and lines of code needed to produce an application [7]. And the domain of concern has largely been "big software" — the relatively small number of heavyweight applications that dominate mainframes and desktop computers. These large applications — operating systems, databases, word processors and image editors — often take years to develop and are generally created by dozens, if not hundreds, of developers.

Big software engineering has traditionally been organized around the waterfall method [24] — a development pipeline beginning with requirements gathering, and moving through design, implementation, verification, and maintenance. Recently, some in the field have begun moving toward more agile methods [3], which eschew the "big design up front" approach of the waterfall, in favor of an approach based on shorter plans, iteratively decided based on the exigencies of the software artifact and customer interactions with it. But really, even agile software is still largely concerned with the big stuff. Small software doesn't really need to be studied or supported, does it? And does small software provide much value anyway? Yes and yes.

Alan Kay, with Smalltalk [15], was the first to explicitly design a language for non-expert programmers (in Smalltalk's case, middle school students) and observe how that community used the language. Perhaps the next major advance toward opportunistic programming languages was introduced with the Tcl interpreted scripting language and the Tk windowing system [23]. The Tcl work suggests that developers are perhaps best served by distinguishing *systems* programming languages (such as C and its progeny) from *interface* programming languages, as the high-level scripting (such as Tcl) may be preferable for the latter. Today, high-level scripting languages such as Python and web-oriented languages such as PHP, JavaScript, and Ruby have replaced low-level systems programming languages in many contexts.

All Hype?

The long tail and mash-ups may both be a bit over-hyped currently, and certainly some of the hype will pass. The goal of this paper is not to be buzzword-compliant. Under our broad definition, many existing practices could be relabeled mash-ups, and part of the project of our design space analysis is to identify these precursors.

We suggest that two significant shifts from the traditional model of software engineering are redefining how individuals build ubiquitous computing systems in practice. First, the integration of bits and atoms in ubiquitous computing has introduced novel hardware — and its relationship with software — as domains of concern for interaction design and development. Second, recent shifts in the production, dissemination, and retrieval of software are reorienting the software development landscape itself towards opportunistic design. Many aspects of mash-up software development have diverged from the heavier-weight traditional development.

This paper presents an investigation into opportunistic design of ubiquitous computing systems through interviews with practitioners in three areas. The paper is structured as follows: we start with a brief review of related work, follow with a segmentation of the mash-up design space into four areas that will serve as a scaffold for our later discussion. We then present our study data and analysis.

RELATED WORK

A small body of prior work in HCI has investigated appropriation in design. MacLean *et al.* provide an overview of the challenges of end-user tailorability [19]. Moran's DIS 2002 keynote on everyday adaptive design [21], as well as workshops on design for hackability [11] and designing for community appropriation [20] have addressed modification, adaptation and appropriation of information technology by end users. Moran in turn takes inspiration from Brand's examination of the post-deployment life of artifacts [4]. In contrast, our work looks at the kinds of ad-hoc appropriation by designers themselves.

The computer systems community seeks to find technical means to enable combination of pre-existing technologies through frameworks for component-based software. The framing of development as comprising "components, scripts and glue" [25] is compatible with our view of mash-up programming. However, systems work still predominantly addresses large-scale development, and generally presupposes that all components will adhere to a proposed

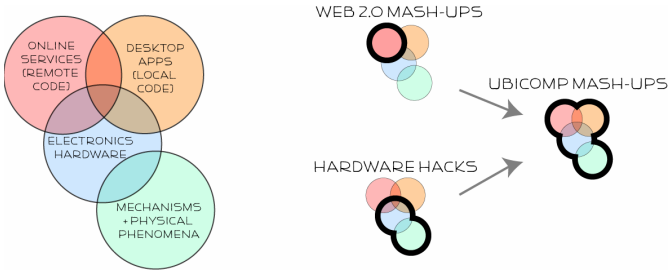


Figure 2. *Left*: Four ingredients of ubicomp systems. *Right*: Ubicomp mash-ups unite hardware and web practices.

architecture. We argue that, in practice, the parts chosen for integration often do not play nice with each other. In addition, our focus is on the user experience of mash-up programming, rather than the technological capabilities. In the larger picture, then, this project is more aligned with studies in the cognitive and psychological aspects of programming [8, 9] and with investigations of values and beliefs of designers (*cf.* [27]).

Papier-Mâché [18] introduced the approach of using fieldwork with *developers* as a means of finding opportunities for design tools. We report on our fieldwork and point out opportunities for tools support in this paper; we leave development of tools based on this study to future work.

LOCATING UBICOMP MASH-UPS

We are most interested in the nascent area of ubicomp mash-ups, which combine both software and hardware. As ubiquitous computing is about the confluence of bits and atoms—computing moving into the world—we look at the physical as well as the digital ingredients of mash-ups.

Moving from the physical to the digital domain, we can distinguish four types of components (see Figure 2). First, a mash-up can contain built or repurposed *mechanisms*, such as the movement mechanism of a toy doll. Second, sensors and actuators can interface with these mechanisms and with other physical phenomena; *electronics* such as analog circuits and embedded programmable microcontrollers provide the logic for sensors and actuators. Third, designers can leverage *off-the-shelf software* on their personal computers (be it a desktop, a PDA or a smart phone). These local applications may or may not offer hooks for programmatic automation through APIs or built-in scripting languages. Fourth, mash-ups can make use of *web infrastructure services* such as search and mapping APIs. At each of these four levels, designers can adopt pre-existing solutions, modify them, or build from scratch. As our study results will show, modification is often the strategy of choice.

This Isn't Totally New

Ubicomp mash-ups draw on different existing lineages of opportunistic design (see Figures 2, 3). Shell scripts and application macros have long been used as "glue" between desktop applications. Ousterhout [22] provides a good overview over the advantages of scripting languages for connecting pre-existing software components. In the tangible world of mechanisms and electronics, hobbyists and professional product designers alike take off-the-shelf products and cannibalize or repurpose them to fit new needs. The success of publications such as Make magazine attests to a recent upsurge in popular interest in adapting consumer electronics for daily living. More recently, the advent of open APIs for web services has spurred development of numerous services and sites that aggregate disparate data sets.

METHODOLOGY

We conducted semi-structured interviews with 14 practitioners in three areas of mash-up design. Four participants were involved in *Web 2.0 development*. Four participants had a focus on "*hardware hacking*"—working with mechanisms and embedded electronics: three were toy inventors, the other a hobbyist and technology writer. Six participants worked as *ubicomp designers*. Two of the ubicomp participants were academic researchers, two industry professionals, one a hobbyist electronic musician, and one an artist who creates interactive installations. Ten interviews were conducted individually, and two with pairs of participants. Eleven interviews were conducted in person; one participant in the ubicomp group was interviewed by phone and one pair in the Web 2.0 category was interviewed in writing online.

Interviews lasted 45 to 90 minutes. In the interviews, we began by asking participants to describe their work philosophy and general approach to problem solving, and then to focus on one particular recent project. To ground and structure the discussion, we asked participants to produce artifacts or visual representations (photographs or sketches) of this project. Specifically, we asked participant to describe the relationship between third party components they integrated and their own code; to describe how they arrived at the decision to include particular parts; and to reflect on tradeoffs and challenges experienced.

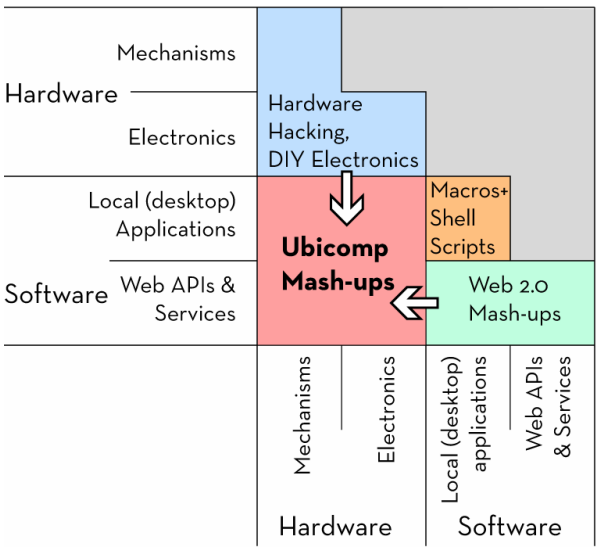


Figure 3. A classification of mash-ups based on the components they encompass

To become more familiar with the experience of working in this domain, we also created several different ubicomp mash-ups ourselves, and attended community events that brought designers of technology mash-ups together. While our personal experience is not the focus of this paper, we draw on these experiences in explaining some of the data.

SAMPLING MASH-UPS: WHO, WHAT, WHY

Here we review the material collected: who our participants are, what kinds of systems they build, and how and why they build them. We describe commonalities within groups.

Web 2.0 Programmers

We contacted participants who are active members of the Web 2.0 mash-up community. All four of these participants were professional programmers or web developers. Because of their background, they did not feel the technical aspects of mash-up programming were a hurdle.

Our first participant (W1) owns a cell phone software company. In his spare time, he independently developed a mash-up "to learn AJAX" (a client-server web technology). His mash-up website overlays restaurant and bar information on an interactive map (see Figure 4A). Users have the ability to build a graphical path, from one bar to the next to plan an evening out with friends. They can also send the paths they created onto a compatible mobile phone for mobile browsing. This mash-up combines three online services: CitySearch for entertainment reviews, Google Maps for mapping and navigation on the desktop, and Yahoo maps for mapping on a mobile device.

A second mash-up, written by participant W4, also builds on Google maps. For the past year, W4—who holds a master's degree in CS—has been developing a weather browser that aggregates weather forecasts from national and regional weather data providers and locates these forecasts on a map (see Figure 1B). His website features geo-referenced temperature readings for cities with microclimates (like San Francisco), real-time fog visualizations, integrated display of approximately 8,000 user-contributed webcam feeds and weather histories that allow users to view seasonal changes in weather for particular locales. He has recently branched out from his original concept and created special interest sites for winter sports enthusiasts, golf players, and campers. Together, these sites are generating enough traffic — and ad revenue—for him to contemplate turning this former side project into a full-time job.

Aiming solely at the emerging mobile market, two other participants, W2 and W3, built a mash-up that delivers relevant train schedules for three U.S. commuter rail systems to mobile phones through SMS or email (see Figure 4C). Users send a short message with a station name abbreviation to their system, which replies with upcoming train times. The system combines a SMS/email gateway with schedule data gathered from the individual rail companies. The two developers started the service while working in the same city and now maintain and update it through remote collaboration. The service is not a profit-making enterprise.

Screen Scraping vs. Web APIs

One major concern for our Web 2.0 participants is access to and strategies for getting data: "getting the data is the absolute hardest part" (W2). The surveyed mash-ups derived their value from integrating disparate data sets in ways not previously available. While two of the three projects used the open, documented infrastructure service of Google maps, all three projects resorted to *screen scraping* to gather at least part of their data. Screen scraping is a technique by which a program extracts text output meant for human consumption from the user interface output of another program. Two primary reasons were given for scraping: first, that APIs were simply not available for obtaining the desired data. Second, that web APIs, still in their infancy, are generally designed for smaller data requests, so that it is still easier to obtain large data sets by scraping. W4 reported building his own scraping toolkit so that it now takes him as much time to develop a scraper as it would to integrate an available API.

Software Architectures

Common to all projects is that they dedicate a web server to retrieve and cache large amounts of content from the different data sources they use. Participants spent significant effort building back-end architectures to scale to many simultaneous users. The back-end architecture design was the most structured, least ad hoc activity encountered in all of our interviews. While the spirit of web mash-ups is free-wheeling for obtaining data, the code written to serve the mash-up pages in the successful sites we reviewed is carefully engineered. We caution that, while mash-ups are in many ways the tail of software, there are still broad differences in engineering approaches within the mash-up field.

Business Models and Obstacles

All participants reported that their mash-ups started as side-projects to their daytime jobs as consultants, startup owners, and developers. However, two of the three projects expressed interest in turning the mash-up into a profitable business. With mash-ups, shifting from the personal sphere to the commercial sphere can be problematic for both legal and technical reasons. W1 reported that making money off "scraped" content is problematic because of licensing restrictions. W4 reported that he had to add redundant data sources, as individual weather providers could alter the format or withdraw their data stream at any point.

Hardware Hackers

In the physical / electronic design realm, we interviewed three toy inventors at two design companies and a hobbyist who refashions consumer goods into personalized tools and publishes instructions for these projects online.

The toy inventors' work consists of building prototypes that illustrate new interaction design concepts. They do not create finished products. The concepts are pitched for licensing or purchase to large toy company representatives who then further develop and manufacture the toys. Project schedules are very short, ranging from two days to less than a month.

At the time of our visit, H3, who holds a master's degree in product design, worked on a toy that functioned as a flashlight with sound effects. To make the concept tangible, she bought a pair of plastic monkeys from a toy store because they had a similar opening mechanism to the one she envisioned for her toy (see Figure 4D). She then embedded a tactile switch into the mechanism's lever. This switch was used to trigger light and sound effects using external electronics. She kept the monkey's form factor, even though thefinal product would have a radically different appearance. While the aesthetics (the "toyness") of the packaging mattered, the fact that it was not a flashlight was not relevant for her client demonstration. A previous project prototype combined a toy car body with plastic rocket engines from a model plane kit to create a new flying car (see Figure 4E). A lever switch underneath the chassis was added to detect when the car was lifted from the ground and when it was put back down. A circuit board inside the car triggered playback of sound samples whenever the switch state changed.

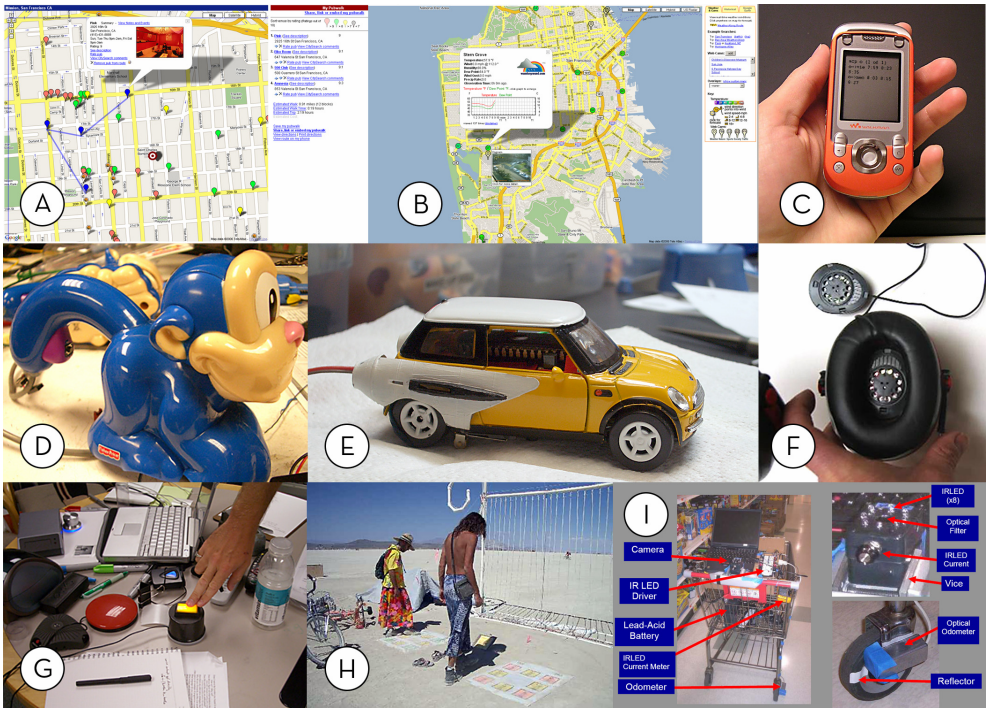


Figure 4. Examples of participants' mash-ups, with individual projects labeled by letter.

At the second toy company, participants H1 and H2 described how they prototyped a handheld wireless controller for a TV game: the *barrel* of the controller was taken from a soda bottle. The *grip* was built from a wireless mouse that uses a gyroscope to sense tilt, transforming that tilt data into cursor movement. The two pieces were integrated into one unit through custom-made plastic molds. The cursor and click stream from the wireless mouse was then used to animate graphics on a laptop (used as a stand-in for a television set) running Macromedia Flash.

In contrast to the rough-and-ready prototypes [28] of the toy designers, participant H4 builds his hardware-based mash-ups for long-term private use. A designer of 3D printing technology, his self-professed strength is to "make a machine that barely works in two to three weeks." Many of the artifacts he uses daily were created by modifying consumer goods. He has documented his activity through more than 50 project descriptions on a how-to web site. One project he brought to the interview was a pair of jackhammer hearing protection earmuffs that he retrofitted with a pair of free airline headphones to listen to audio books in noisy environments (see Figure 4F). According to H4, this design offers better noise reduction than commercial noise-canceling headphones while being significantly cheaper.

To Buy or Not to Buy

For all three toy inventors, an integral part of their core practice was visiting large retail stores to purchase interesting new toys; these would be later disassembled in their shop. We identified three different strategies of reappropriating store-bought toys: first, designers *extract mechanisms* ("remember that freaky belly movement?") and reuse them in different skins; second, designers *keep the shell* of a toy but embed new electronics into it ("because it immediately looks like a toy"); third, designers *fuse different shells* (*e.g.*, a metal toy car and plastic model plane rocket engines to create a new flying car), then embed their own electronics.

H4 saw the tailoring of existing artifacts as a partial rejection of consumer culture. The self-sufficiency of 'do it yourself' offers a degree of personal satisfaction (it is intrinsically satisfying) as well as a level of personalization and lasting novelty not available in mass-produced artifacts. H4 reuses existing parts because "you don't want to invent the wheel—the wheel exists." Also for him, the economies of scale that mass produced consumer goods leverage are incentives. Picking existing parts is cheap — "it's never cheaper to start from scratch to make your own."

Ubicomp Designers

Our ubicomp developers used mash-ups as prototypes and proof of concept deliverables, but also as a way to design and implement site-specific tool for a single user or a small community. For brevity, we only mention a subset here.

Participant U1—one of two academics in our study—is a PhD candidate in mechanical engineering and a self-taught programmer. He worked on a system for design teams to annotate their printed documents with short video mes-

sages. In his laptop-based functional prototype (see Figure 4G), users push a dedicated physical button to initiate video message recording. After the recording completes, the system prints a small self-stick label displaying a snapshot of the video and a barcode. The user then attaches this barcode to the document described in the video. If another user wants to access the video, she waves the barcode in front of the same camera. Using the barcode as a key, the system plays back the specified video for the user. U1 relied heavily on commercial off-the-shelf-software (COTS) since this path offered the "easiest way to plug things together." In his working prototype, no less than five different COTS packages are controlled through AppleScript. For example, he scripted QuickTime to record and play back video, and the Excel spreadsheet software is used as a database. To convey the complexity of this project, Figure 5 shows our redrawn version of his system architecture sketch.

Mash-ups have also found their way into corporate work. Participant U3 described a recent project where he designed an indoor location system for smart shopping carts; the client was his employer's retail store group. This positioning system employed computer vision. To test the quality of the vision data, he attached a custom-built optical rotation sensor to the wheel of a shopping cart and soldered its contacts to the left button of a gutted PC mouse, so that each revolution yielded one click (see Figure 4I). By keeping track of the total number of clicks on the PC, he was provided with reasonable ground truth data about the total distance traveled. He estimates that using this mouse mash-up, he completed testing in a quarter the time it would have taken to build a distance tracker from scratch.

U4, "a software engineer by day and software artist by night" has been developing his own musical programming language and graphical environment for producing and performing electronic music for the past decade. Since 2002, he has built audio installations that he shows at the annual Burning Man festival. While spending years on building his software from the ground up, his use of physical controllers is more opportunistic: "you can choose what level of effort you want to put in — you can buy the next

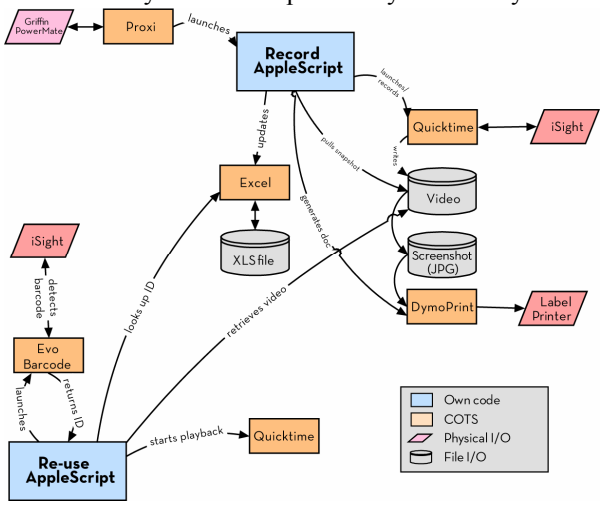


Figure 5. System diagram of U1's project.

level of integration." An important "bridge" he found was a converter that allows him to connect controllers built for proprietary game consoles to a PC USB port. Through this adapter, he has connected multiple "Dance Pad" floor mats to his synthesis program (see Figure 4H).

Screen Poking

As Web 2.0 programmers employ *screen scraping* — instrumenting the surface properties of web pages — to harvest information from online databases, ubicomp programmers use *screen poking* —generating mouse and keyboard events by computational or electronic methods—as a means to remote control software. In addition to U3's appropriation of a mouse button for counting turns of a measuring wheel, U1 initially used the macro software Automate as a means of controlling desktop applications by computationally injecting synthetic mouse and keyboard events, and U4 purchased a hardware converter that transformed the output of pressure-sensing dance pads into Windows platform game controller events. These "glueware" techniques are chosen for similar reasons as screen scraping: APIs are sometime unavailable, other times do not yield the desired information, and still other times are more time-consuming that surface-level instrumentation.

How Choices are Made

What metrics do designers use to select the elements of a mash-up? Our ubicomp participants reported three strategies: relying on experience, searching online forums, and the decree of a supervisor. We briefly discuss the first two.

Participants reported integrating technologies into their project that they had experience with from prior work, enabling them to leverage their proficiency in a medium and hedge against unforeseen shortcomings. Communal experience also proved valuable: for example, U1 reported that he integrated a custom hardware-switch-to-USB interface because other members of his research lab had previous success with them.

Participants also leveraged descriptions from online sources, and this experience of "shopping" for preexisting solutions yields a very different pattern of time usage than developing systems from scratch. For example, U1 reported that the time it took to search for appropriate components exceeded the time it took to then write scripts to integrate the found components into his project. For U4, the search for the right USB adapter that would glue controllers to his software took months: he reported doing research on at least a dozen different models and then buying six different models until he found a one that satisfied his requirements.

Caveats

We conclude our tour of mash-ups with two cautionary tales from our participants. The first illustrates the potential downsides of opportunistic design; the second reminds us that mash-ups are not a one-size-fits-all solution.

For an interactive museum installation, U5, a professional A/V systems developer, took a computer vision system intended for industrial process monitoring to track visitor's

hand movements over a projection display. Buying packaged functionality and writing thin wrapper layers promised significant time and cost savings over developing an in-house solution. However, because of variations in lighting conditions, the out-of-box system did not perform as expected. Trying to compensate for these problems proved hard as the team had only limited access to internals of the vision processing system. In the end, the group spent more time and resources trying to patch up these problems than development from scratch would have taken.

Finally, we talked to an independent artist who produces large interactive public installations that involve combinations of computer vision and projection, as well as sensing and actuation. He was trained as both an artist and a computer scientist and previously held an industrial research position. This participant was the only person who followed a top-down design approach that does not include any mash-ups of existing technologies. This is an important counterexample to our narrative. U6 consciously engineered all his projects from the ground up. Having full control over aesthetics and behavior is part of his conception of how interactive art should be created.

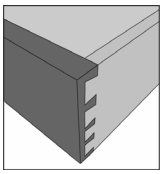
THEMES IN OPPORTUNISTIC PROGRAMMING

This section discusses three themes that emerged from our interview data. First, we look at a different ways that mash-ups modify and combine pre-existing elements. We then consider the unique characteristics of mashing as an activity, and conclude with a look at mash-ups as novel artifacts.

Combinations: The Core of Mash-Ups

One of the broad shifts introduced in the mash-up paradigm is that the designer's effort and creativity are reallocated: rather than building an application up from scratch, brick by brick, much time and ingenuity becomes concerned with finding and selecting components, and then creating and shaping the "glueware" that interfaces them.

Dovetail Joints vs. Hot Glue



In our interviews, we saw two distinct approaches to glue. In the first approach, two components explicitly support combination through a shared interface. They are aware of each other. This allows for *tight integration.* We use the metaphor of the carpenter's *dovetail joint* to label these deep combinations. Dovetail joints are documented extension and integration points provided in the system architecture—APIs in software, breakout headers and connectors in electronics, mounting holes in hardware. The use of AppleScript (U1) and Web APIs (W1,4) are examples of dovetail joints.

In contrast, "hot glue" combinations adjoin components that are either incompatible, don't know about each other, or don't support each other. Hot glue can be applied to almost anything, but it has limited adhesive power—all it can offer is surface-level integration. Screen scraping and screen poking are examples of hot glue joints. Importantly, the designer's intent is not directly visible in the code generated by hot glue: the record is the trace of actions that reflect the intent, not the semantics themselves.

The trade-offs between these two architectural approaches are exemplified by U1's experience. He designed two separate versions of his document annotation system, and while the core functionality was similar, all of the component pieces were replaced. The first version was scripted using Network Automation's AutoMate 6 software. AutoMate enables users to record human interaction with GUI widgets and to parameterize and replay those actions programmatically. While this system succeeded as an experience prototype, it was not robust enough for any unsupervised deployment: the shallow glue provided by AutoMate's screen-based scripts proved to be too brittle.

Seeking a more robust system, U1 switched to a Macintosh platform so that he could use AppleScript, and this platform switch mandated an entirely different set of software applications. AppleScript allowed him to leverage application-specific APIs. While the deeper glue that AppleScript provides is significantly cleaner for expressing logic than Automate, it also has limited *reach*: for example, U1 found no programmatic means for uploading the video clips to an online media-sharing site.

Appropriation

The discussion of gluing approaches hinged largely on technical considerations. Here, we turn more to the socio-technical issue of the relationship between the designed *intent* of the constitutive elements and that of the resulting mash-up. At times, these intents are felicitous. However, at other times, mash-ups appropriate technologies, repurposing them as building blocks toward a different goal.

In Eglash's words, appropriation is *the extent to which a violation of a technology's intended purpose occurs* [10]. This violation is easy to see in toy hacking: toys were intended for children to play with, not for designers to take apart. Similarly, in the digital realm, screen scraping re-appropriates output intended for human consumption as program input. In contrast, using Web 2.0 APIs such as Google Maps is *not* an act of appropriation because the providers of the API give explicit permission to use the service in new contexts. Similarly, applications written for Apple's operating system expose API hooks to AppleScript to enable programmatic control by end-user automation scripts.

It is notable that in the Web 2.0 space, where the general trend has been to open up infrastructure services to allow reuse without appropriation, all of our participants still resorted to screen scraping techniques. There are valid business reasons not to make all company data available for automatic processing by others through APIs. Simultaneously, those same business reasons make capturing the data valuable for third parties. We conclude that *support for both tight and loose coupling* (hot glue and dovetail joints) is

needed—opportunistic design is based on integrating existing artifacts that best fulfill a functional or informational need, regardless of their programming interface or licensing agreement.

Mashing as a design activity

Next, we turn our attention to the activity of creating mash-ups: when, how and why is mashing preferable to other approaches of design and development; and what kind of value do practitioners derive from it?

Epistemic, pragmatic, and intrinsic values

We found that mash-ups provided both pragmatic and epistemic value to our participants. An artifact is pragmatic to the extent that it enables actual use, and epistemic [16] to the extent that the artifact serves as a locus of communication with other stakeholders — clients, team members, and users — and provides information that can drive future design [17]. For some participants, creating mash-ups also held *intrinsic* value generated by the activity itself, rather than from the utilitarian or educational value of the outcome.

Pragmatic decisions for mash-ups are made if mash-ups are more efficient or effective to reach a known goal than other techniques. We saw an example of this earlier, where participant U3 estimated that by repurposing a mouse button to fire a 'click event' with each revolution of a wheel, he was able to complete the sensing part of his project in a quarter of the expected time. Furthermore, incorporating existing pieces allows designers to leverage functionality that they could not build themselves — framed this way, the set of existing technologies in the world can be thought of as a vast library that can be used to lower the threshold for development. For example, U4 did not have sufficient technical knowledge to build his own physical music controller, and it was through adapters that he was able to leverage commercially available game controllers.

Other times, mash-up design is employed as a means of exploration, learning, or inspiration. This *epistemic* activity was most prevalent among our toy inventors, who chose mash-ups as effective means to illustrate new concepts. What their clients paid for was the idea, *prototyped* through the mash-up, not the implementation. Furthermore, rapidly creating prototypes provides designers with concrete artifacts they can expand on, react against, modify, and transform. This *conversation with materials* (as opposed thinking in the abstract) is an important strategy of successful reflective practitioners [26].

In the *intrinsic* case, mash-ups are created purely as an end in and of themselves. Intrinsic value is derived from the joy of exercising a craft ("what a great way to spend an afternoon") or from a personal ideology ("recycling is my form of protest against rampant consumer culture"). Our interviews suggest that intrinsic activity is most common among hobbyists. The move towards community appropriation and hackability by end-users falls, at least partially, in this category. As Galloway writes, end-user mash-ups are valuable because they are empowering, "DIY culture involves creating your own world amid the dominant culture, thereby putting power back in the hands of individuals." [11].

What is notable about this taxonomy is that the division is based on the *intended use* of the artifact, the developer's motivation rather than an attribute of the artifact itself.

Shopping for functionality

"The most radical possible solution for constructing software is not to construct it at all." –F. Brooks [6]

Brooks identified the drastic departure from existing practice that buying instead of building involves. But how exactly does the activity of designing and developing *change* when no "new" software is created? Glueware addresses the integration aspect. However, before integrating parts, participants reported spending significant time on finding and acquiring their ingredients. In fact, some participants reported that this was the most challenging or the most time consuming part of their process. U1 described the processes of searching for components and determining if and how they could be integrated into his design as "the main part of the whole thing." Or, as U3 put it: "The real challenge is finding the interface between the problem and commercially available stuff."

In a top-down, waterfall development paradigm, shopping for functionality versus integrating existing solutions becomes a cost/benefit tradeoff. Given knowledge of those tradeoffs, the unit selection decision of what to buy becomes merely a "small matter of purchasing." In our study though, we encountered more bottom-up driven design, where searching and acquiring pieces were used as inspirational and direction-giving activities that steered projects in one direction or another. This suggests that shopping itself can take on an epistemic function.

Our toy inventors reported frequent toy store trips without a concrete shopping list for a current project in mind. U4 did the same at electronics retail stores. We find three reasons for shopping without a project. First, it builds awareness of the state of the art, and allows designers to update their mental map of what is available. Second, acquiring ahead of time reduces the cost of search later. Like squirrels gathering nuts before the winter, designers reported stockpiling mechanisms to have them ready-to-hand later. "We collect [mechanical] movements… [During a project, one of us will say] 'Remember that freaky belly movement?' " (H1&2). Third, designers browsed in search of inspiration that may launch new projects: "I go on shopping trips and think about repurposing objects... I'll walk around Walgreens and look at objects and think, 'what could this be?" (H3). The important epistemic and pragmatic aspects of shopping for mash-up design suggest further investigation into tools that support search and acquisition.

Searching for bridges

In multiple instances, participants reported finding crucial connecting pieces for their mash-ups in fields only tangentially related to their own. U4 discovered that a MIDI-to-

relay interface used by church organ builders was what he needed to trigger lights based on music commands for his Burning Man installations. In our own work, we have discovered that the most straightforward way to interface discrete digital inputs to PCs is to use hardware developed by arcade game enthusiasts. This community builds their own arcade cabinets with PCs replacing dedicated electronics. Multiple vendors sell boards to interface arcade joysticks and push buttons to the game software.

Adapters and bridges are well-known design patterns for software engineers [12] and glueware often instantiates these patterns in software or hardware. However, we want to focus here on the social side: the bridges that allow these connections to be found in the first place. While web search was universally used, effective search requires prior knowledge of the space of opportunity. Community sources play an important role: U1 chose to integrate two different external button interfaces into his project because he was peripherally aware that other researchers in his building had used those particular models successfully. Scaling community awareness to the internet is a factor behind the success of how-to sites that chronicle DIY projects such as instructables.com.

Short timelines, small audiences

The activity of mashing is also often characterized by two trends: it tends to happen on very short timelines, and the artifacts created are intended for small audiences. Many mash-ups we encountered were built quickly, and many discarded as quickly afterwards. The emphasis on speed is a good match for designers wanting to rapidly prototype multiple ideas, consultants operating on compressed project schedules, and hobbyists with limited leisure time. Similarly, for these constituencies, the audience of users of a mash-up is small: the design team, a single client, oneself.

Web mash-ups have a different set of traits: they operate continuously and their success is measured in the number of users they attract. If mash-ups are to scale, robustness and maintenance have to be addressed — tracking how successful web mash-ups grapple with these issues could provide valuable lessons for guiding mash-ups design in other areas.

While it is certainly fast to get up and running with mash-ups, completing the "last mile" — fine-tuning application logic and interaction design — can be quite difficult as desired functionality and offered features of existing components diverge (see Figure 6). This trade-off is exemplified by U5's experience of working with a shrink-wrapped computer vision package: afterwards, she felt that a custom-written solution would have been faster and more flexible. U1 also reported getting stuck on mundane details such as a missing hook for a print option that was not exposed in AppleScript, stonewalling his efforts.

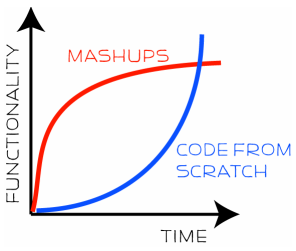


Figure 6. Mash-ups provide more functionality up front, but the "last mile" may be slow.

On the other hand, building from scratch incurs a large initial cost as developers have to write their own tooling. In exchange, flexibility is preserved and they can leverage their own tools later in the project cycle. The "sweet spot" for rapid, disposable mash-ups that our interviews found is consistent with this analysis. It also suggests an opportunity for design tools that leverage opportunistic development early on while preserving more flexibility later on.

Mash-ups as mixed-fidelity artifacts

We conclude our analysis of themes by looking at the artifacts that opportunistic design produces. What kind of objects are mash-ups? And how do they differ from other products?

Affordances of mixed-fidelity prototypes

Mash-ups are made of disparate, heterogeneous pieces, and each individual component brings its own architecture, functionality, and level of polish with it. Hence, they operate as *mixed-fidelity* artifacts. Fidelity is a slippery term, and we distinguish two different perspectives that lead to different sets of concerns here. From a designer's perspective, the fidelity is an affordance of a design tool or medium, in that it helps structure the conversation with the material [26]. Low-fidelity media can only capture and express the gestalt of a concept, while high-fidelity media allow detailed insight into tradeoffs and alternatives.

From a user's perspective however, fidelity is a property of the designed artifact. It is the degree to which a prototype exhibits the affordances (actionable properties between artifact and actor) of the final object being designed. Mixed fidelity here results in a potentially discontinuous user experience as imported functionality can offer either too many or too few interactive properties. For example, using a game controller for music synthesis can raise the expectation that all possible interaction opportunities on the controller will have some effect, as they would in a game. Similarly, by basing a web mash-up on Google maps, one automatically imports all interaction techniques of that application into the mash-up, even if some of them are not felicitous with the intended application. The extent to which a mismatch between perceived and offered functionality is problematic in practice depends on whether the designer can provide guidance. We speculate that large-scale, unsupervised deployments are likely to be more problematic than individual, designer-led demonstrations.

The semiotics of media

Distinct from the actionable properties are the perceived values and meanings beaconed by an artifact. Mashing enables designers to easily buy into a product or service *genre*. Leveraging existing materials can scaffold user expectations, for better or for worse. Toy designer H3 preferred to cannibalize existing toys not just because of the readily available functionality, but because these objects already "look like a real toy." Building from scratch in her

shop was technically feasible, but the aesthetics would not be as convincing in client meetings. This automatic import of the residual meanings of parts into a project also has a downside — mixed messages are hard to avoid. This multiplicity of messages may be the reason that U6 ultimately shunned any opportunistic design for top-down engineering — for his artistic vision he required full control over the communicative aspects of his works — something that mash-ups cannot provide.

CONCLUSION

In this paper, we placed opportunistic design in a larger software engineering context and presented a descriptive account of how mash-ups are used in practice today in three different areas: Web 2.0 development, hardware hacking, and ubicomp system building. We analyzed three common themes in mash-up design: how components are combined, what the characteristics of the activity of opportunistic design are, and how mash-ups are unique artifacts. Future work consists of developing concrete intervention strategies to probe how introduction of software and hardware tools can support opportunistic design and mitigate against problems of scale, robustness, and communicative ambiguity. We hope this paper will be a first step towards establishing community interest in and support for opportunistic design and development practices.

ACKNOWLEDGMENTS

We thank Intel for donating PCs for this research, and our study participants for sharing their time and insight.

REFERENCES

1 Anderson, C., *The Long Tail*: Random House Business. 2006.
2 Ballagas, R., M. Ringel, M. Stone, and J. Borchers. iStuff: a physical user interface toolkit for ubiquitous computing environments. CHI: ACM Conference on Human Factors in Computing Systems, *CHI Letters* **5**(1). pp. 537–44, 2003.
3 Beck, K., *Extreme Programming Explained - Embrace Change*. The XP Series: Addison Wesley. 190 pp. 1999.
4 Brand, S., *How Buildings Learn: What Happens After They're Built*. Reprint ed: Penguin Non-Classics. 256 pp. 1995.
5 Brewer, E. A. Lessons from Giant-Scale Services. *IEEE Internet Computing* **5**(4): IEEE Educational Activities Department. pp. 46-55, 2001.
6 Brooks, F. P., *The mythical man-month: essays on software engineering*. Anniversary ed. Reading, Mass: Addison-Wesley. 322 pp. 1995.
7 Clements, P., R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*. SEI series in software engineering. Boston: Addison-Wesley. 323 pp. 2002.
8 Détienne, F., *Software Design - Cognitive Aspects*. Practitioner Series, R. Paul: Springer. 139 pp. 2001.
9 Ducheneaut, N. Socialization in an Open Source Software Community: A Socio-Technical Analysis. *Comput. Supported Coop. Work* **14**(4): Kluwer Academic Publishers. pp. 323-68, 2005.
10 Eglash, R., Appropriating Technology, in *Appropriating Technology: Vernacular science and social power*, R. Eglash, *et al.*, Editors. University of Minnesota Press. p. 401, 2004.
11 Galloway, A., J. Brucker-Cohen, L. Gaye, E. Goodman, and D. Hill, Design for hackability, in *Proceedings of the 2004 conference on Designing interactive systems: processes, practices, methods, and techniques*. 2004, ACM Press: Cambridge, MA, USA.
12 Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series: Addison-Wesley. 395 pp. 1995.
13 Greenberg, S. and C. Fitchett. Phidgets: easy development of physical interfaces through physical widgets. UIST: ACM Symposium on User Interface Software and Technology, *CHI Letters* **3**(2). pp. 209–18, 2001.
14 Hartmann, B., S. R. Klemmer, *et al.* Reflective physical prototyping through integrated design, test, and analysis. In Proceedings of *UIST 2006: ACM Symposium on User Interface Software and Technology*. Montreux, Switzerland, 2006.
15 Kay, A. C. The early history of Smalltalk. In Proceedings of *The second ACM SIGPLAN conference on History of programming languages* Cambridge, MA: ACM Press. pp. 69-95, 1993.
16 Kirsh, D. and P. Maglio. On distinguishing epistemic from pragmatic action. *Cognitive Science* **18**. pp. 513-49, 1994.
17 Klemmer, S. R., B. Hartmann, and L. Takayama. How Bodies Matter: Five Themes for Interaction Design. In Proceedings of *Design of Interactive Systems*. State College, PA, 2006.
18 Klemmer, S. R., J. Li, J. Lin, and J. A. Landay. Papier-Mâché: Toolkit Support for Tangible Input. CHI: ACM Conference on Human Factors in Computing Systems, *CHI Letters* **6**(1). pp. 399–406, 2004.
19 MacLean, A., K. Carter, L. Lövstrand, and T. Moran, User-tailorable systems: pressing the issues with buttons, in *Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people*. 1990, ACM Press: Seattle, Washington, United States.
20 March, W., M. Jacobs, and T. Salvador, Designing technology for community appropriation, in *CHI '05 extended abstracts on Human factors in computing systems*. 2005, ACM Press: Portland, OR, USA.
21 Moran, T. P., Everyday adaptive design, in *Proceedings of the conference on Designing interactive systems: processes, practices, methods, and techniques*. 2002, ACM Press: London, England.
22 Ousterhout, J. K. Scripting: Higher Level Programming for the 21st Century, *IEEE Computer*, March, 1998.
23 Ousterhout, J. K., *Tcl and the Tk toolkit*: Addison-Wesley Reading, Masspp. 1994.
24 Royce, W. W. Managing the development of large software systems: Concepts and techniques. In Proceedings of *IEEE WESTCON*. Los Angeles, CA: IEEE. pp. 1-9, 1970.
25 Schneider, J.-G. and O. Nierstrasz, Components, Scripts and Glue, in *Software Architectures – Advances and Applications*, L. Barroca, J. Hall, and P. Hall, Editors. Springer. pp. 13-25, 1999.
26 Schön, D. A. and J. Bennett, Reflective Conversation with Materials, in *Bringing Design to Software*, T. Winograd, Editor. ACM Press: New York, 1996.
27 Sharp, H., H. Robinson, and M. Woodman. Software Engineering: Community and Culture. *IEEE Softw.* **17**(1): IEEE Computer Society Press. pp. 40-47, 2000.
28 Wong, Y.-Y. Rough and ready prototypes: Lessons from graphic design. In Proceedings of *Extended Abstracts of CHI: Conference on Human Factors in Computing Systems*. Monterey, CA: ACM Press. pp. 83-84, 1992.