HUMAN–COMPUTER INTERACTION, 2009, Volume 24, pp. 315–366

ISSN: 0737-0024 print / 1532-7051 online
DOI: 10.1080/07370020902990428

Toolkit Support for Integrating Physical and Digital Interactions

Scott R. Klemmer[1] and **James A. Landay**[2]
[1]*Stanford University*
[2]*University of Washington*

ABSTRACT

There is great potential in enabling users to interact with digital information by integrating it with everyday physical objects. However, developing these interfaces requires programmers to acquire and abstract physical input. This is difficult, is time-consuming, and requires a high level of technical expertise in fields very different from user interface development—especially in the case of computer vision. Based on structured interviews with researchers, a literature review, and our own experience building physical interfaces, we created Papier-Mâché, a toolkit for integrating physical and digital interactions. Its library supports computer vision, electronic tags, and barcodes. Papier-Mâché introduces high-level abstractions for working with these input technologies that facilitate technology portability. We evaluated this toolkit through a laboratory study and longitudinal use in course and research projects, finding the input abstractions, technology portability, and monitoring facilities to be highly effective.

Scott R. Klemmer is an Assistant Professor of Computer Science at Stanford University, where he codirects the Human–Computer Interaction Group; his research enables designers and other innovators to create interactive media that integrates the physical and digital environments. **James A. Landay** is an Associate Professor in Computer Science & Engineering at the University of Washington, specializing in human–computer interaction; his current research includes automated usability evaluation, demonstrational interfaces, ubiquitous computing, user interface design tools, and Web design.

CONTENTS

1. INTRODUCTION

Although people's interaction with tools in the real world is highly nuanced and heterogeneous, graphical user interfaces map all of our tasks in the electronic world onto a small set of physical input devices. Recent advances in research and industry promise the ability to interact with computational systems through a rich variety of physical controls. However, the labor and

expertise required to create application-ready input from physical hardware impedes progress toward this dream—especially with computer vision. In addition, although iterative exploration can greatly improve designs, these development hurdles can make designing and evaluating multiple alternatives prohibitively costly.

These difficulties echo the experiences of developing graphical user interfaces (GUIs) 20 years ago, when substantial raster-graphics expertise was required. One of the earliest GUI toolkits, MacApp®, reduced Apple's development time by a factor of four or five (Myers & Rosson, 1992). We believe that similar reductions in development time, with corresponding increase in software reliability and technology portability, can be achieved through toolkit support for physical input hardware that provides distinct—and often richer—information than the 2D position and ASCII character stream that typify the keyboard-and-mouse paradigm.

The Papier-Mâché research introduces a modular, event-based architecture for working with heterogeneous physical input devices, enabling programmers with minimal hardware expertise to work with physical input, as GUI toolkits have enabled programmers who are not raster graphics experts to build GUIs. Papier-Mâché's library supports several types of physical input: interactive computer vision, electronic tags, and barcodes. By introducing input schemas that generalize across multiple technologies, Papier-Mâché improves application flexibility, minimizing the code changes required to migrate across input technologies. This facilitates exploring different versions of an application and performing comparative evaluations based on cost, performance, usability, and other important metrics.

In many ways, software development is largely about software debugging. A significant difficulty in program debugging is the limited *visibility* of application behavior (Détienne, 2001, §7.2). The novel hardware used in tangible interfaces, and the algorithmic complexity of computer vision, only exacerbate this problem. To support debugging, Papier-Mâché provides monitoring facilities that display the current input objects, image input and processing, and behaviors being created or invoked. The monitoring window also provides Wizard of Oz (WOz) (Kelley, 1984) generation and removal of input. WOz control is useful for simulating hardware when it is not available and for reproducing scenarios during development and debugging.

This research offers two methodological contributions. (a) This was the first research to employ fieldwork *with developers* as a basis for toolkit design. A toolkit is software where the "user interface" is an application programming interface (API) and the users are programmers. As part of our design process, we conducted structured interviews with nine researchers who have built tangible interfaces. (b) This research introduces a mixed-methods approach for evaluating development tools. Different usability methods yield different in-

sights (McGrath, 1994). For example, a laboratory study offers substantial control but often implies an artificial task. Observing longitudinal usage can offer more authenticity but limits many experimental controls. This article's mixed-methods approach comprises controlled laboratory study, monitoring of longer term use in projects, reimplementing existing applications, and traditional software engineering metrics. Aggregating data from these diverse methods afford a much richer picture of the usability of a programming tool.

This article is structured as follows. Section 2 covers related user interface toolkits and the toolkit design requirements we inferred from analyzing 24 applications. Section 3 discusses the findings of our structured interviews with developers. Section 4 describes the Papier-Mâché software architecture, including the event dispatch mechanism and monitoring interface. Section 5 presents the evaluations of Papier-Mâché. Section 6 concludes the article with a summary of contributions and future research directions.

Aspects of this work have been reported in Klemmer, Li, Lin, and Landay (2004)—this article expands significantly on the earlier publication by presenting much greater detail about the initial fieldwork, the architecture, and the studies. It also presents a much deeper discussion relating Papier-Mâché's approach to prior work and describes the applications that motivated the toolkit.

2. BACKGROUND

The difficulty of designing ubiquitous computing applications inhibits designers' ability to create and evaluate multiple alternatives (Carter, Mankoff, Klemmer, & Matthews, 2008). In each of the 24 applications we analyzed, at least one member of the project team was an expert in the sensing technology used. Contrast this with GUIs, where tools have helped shift the focus to interaction design by smoothing the development path:

> Tools help reduce the amount of code that programmers need to produce when creating a user interface, and they allow user interfaces to be created more quickly. This, in turn, enables more rapid prototyping and, therefore, more iterations of iterative design. (Myers, Hudson, & Pausch, 2000, p. 5)

Papier-Mâché's input architecture is inspired by software design patterns for handling input in GUIs. Model-View-Controller (MVC) is the predominant software design pattern for developing GUIs. It separates each widget into three pieces: A *controller* (input handler) sends input events to a *model* (application logic), and the model sends application events to a *view*. The MVC separation enables software input and display style to be altered independently of its underlying functionality (Krasner & Pope, 1988).

The Garnet toolkit introduced Interactors (Myers, 1990), which extended MVC to shield developers from issues such as windowing systems and separates view and controller more cleanly. Interactors are highly parameterized, minimizing the need to write custom input-handling code. Although Interactors have been extended to include gesture (Landay & Myers, 1993), and other modalities, such as speech, have been proposed (Myers et al., 1997), the Interactors architecture loses elegance beyond traditional wimp input. To see why, it is valuable to distinguish input *mode* (e.g., wimp, gesture, or speech) from *action* (e.g., select, create, or move). The original Interactors are a set of *action* types that span the wimp *modality.* Wimp, Gesture, and speech are distinct *modality* types, each of which can be used for many *actions.* The conflation of these axes with Interactors indicates that a more elegant architecture should separate mode from action. Papier-Mâché offers this separation.

2.1. Tool Support for Ubiquitous Computing

Several UI toolkits have recently introduced software abstractions for physical devices; we discuss two canonical examples here. Phidgets are programmable ActiveX controls that encapsulate communication with USB-attached physical sensors (Greenberg & Fitchett, 2001). The ActiveX controls, like Papier-Mâché's monitoring interface, graphically represent physical state. However, Phidgets primarily support tethered, mechatronic interfaces that can be composed of powered, wired sensors and actuators. In contrast, Papier-Mâché supports input from everyday, passive objects, for example, through computer vision. Phidgets facilitate the development of widget-like physical controls (such as buttons and sliders) but provide no support for the creation, editing, capture, and analysis of physical input, which Papier-Mâché supports. IStuff (Ballagas, Ringel, Stone, & Borchers, 2003) extended these ideas with wireless device support and, through the Patch Panel (Ballagas, Szybalski, & Fox, 2004), introduced fast remapping of input events so designers could control standard GUIs with novel input technologies. Papier-Mâché differs from iStuff in two important ways. First, like Phidgets, iStuff targets powered, rather than passive, input. For example, it is not possible to build computer vision applications using iStuff or Phidgets. Conversely, because Papier-Mâché only provides input support, it cannot be used to control a servomotor or other physical output device. Second, iStuff offers novel control of existing applications, whereas Papier-Mâché targets novel applications that leverage unique attributes of physical input.

A second related area is tools for computer vision applications. Perhaps most related, Crayons introduced a demonstrational programming technique where designers draw directly on camera input, selecting image areas (e.g., hands or notecards) that they would like the vision system to recognize (Fails

& Olsen, 2003). Crayons classifies images with pixel-level features using decision trees, exporting the resulting classifier as a Java library. Papier-Mâché's contribution is complementary to the training-interface contribution of Crayons; Papier-Mâché offers higher level recognition algorithms than Crayon's pixel-level classification, provides applications with higher-level object information, and most important introduces a richer event mechanism for fluidly integrating vision events into applications. Papier-Mâché's classifiers and event architecture also support ambiguity (Mankoff, Hudson, & Abowd, 2000).

Augmented Reality applications also leverage novel physical input devices, such as cameras. ARToolkit provides a library encapsulation for camera-based fiducial tracking (Kato, Billinghurst, & Poupyrev); its primary contribution lies in techniques for fiducial tracking rather whereas Papier-Mache focuses on software architectures for designing augmented interactions. DART offer a scripting-and-timeline interface for creating AR applications (MacIntyre, Gandy, Dow, & Bolter, 2004); it uses ARToolkit for its fiducial tracking. Although Papier-Mache is implemented largely in one language, DART partitions development: Designers specify interactions by scripting in Director, and developers add infrastructure components by programming in C. DART's split-language approaches provides choices that are more tailored to the separate concerns. However, it adds a "seam" between the environments, and the additional complexities of having to manage and coordinate two environments.

Papier-Mâché also draws on ubicomp software architectures more broadly. The main architectural similarity of The Context Toolkit (Dey, Salber, & Abowd, 2001) is that it does not just provide a software interface to physical sensors (à la Phidgets),; it "separates the acquisition and representation of context from the delivery and reaction to context by a context-aware application" (Dey et al., 2001, p. 100). In The Context Toolkit, a *widget* provides a device-independent interface to an input or output device, similar to Papier-Mâché's *InputSource*. Papier-Mâché provides richer monitoring and WOz facilities and supports interactive tangible interfaces, which The Context Toolkit does not.

Traditional UI software architectures only support input from a keyboard and mouse (or device that can generate equivalent events, such as a stylus). The OOPS toolkit (Mankoff et al., 2000) introduced architectural support for *ambiguous* input (input that requires interpretation) and for *mediating* that input (methods for intervening between the recognizer and the application to resolve the ambiguity). Ambiguity information is maintained in the toolkit through the use of hierarchical events. This hierarchy captures the relationship between raw input (such as mouse events), intermediate input (such as strokes), and potentially ambiguous derived values (such as recognition re-

sults). Ambiguity is an essential property of recognition-based input support. Papier-Mâché offers a lightweight form of ambiguity to illustrate that the architecture is ambiguity aware. A production implementation of Papier-Mâché should more fully support the ambiguity and mediation model introduced in OOPS.

2.2. Characterizing Physical Interaction Designs

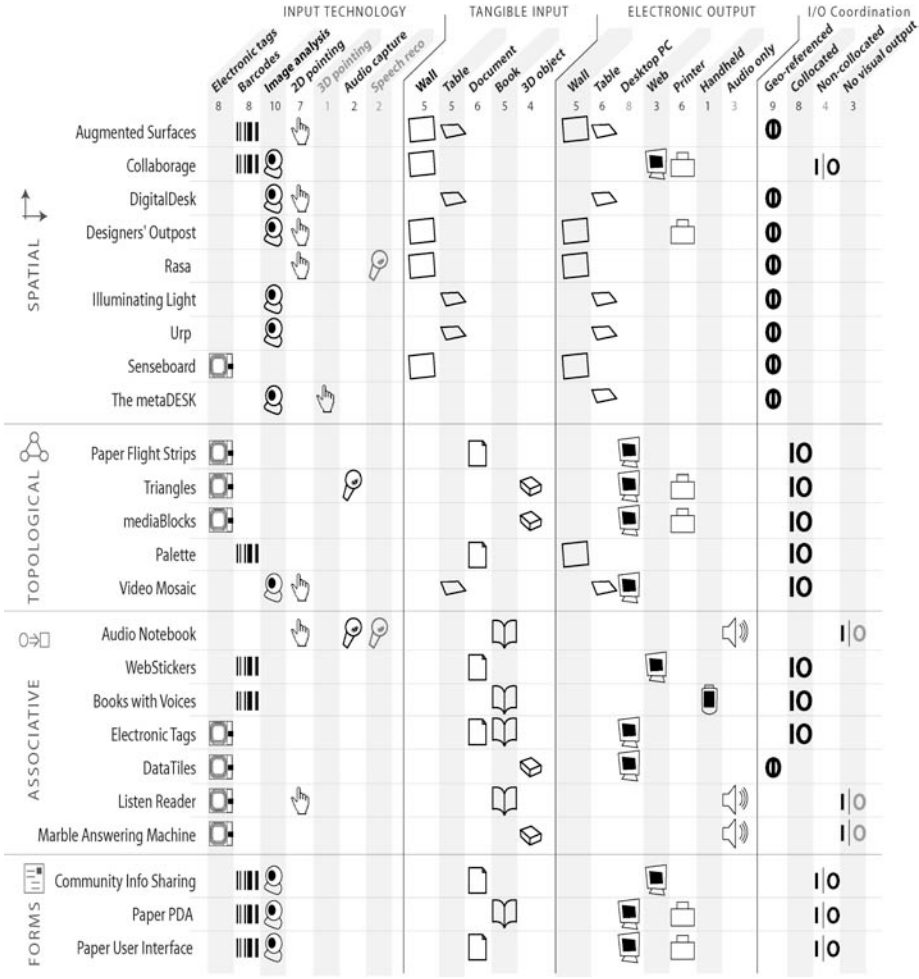
Recently, researchers have begun to propose more structured vocabularies for describing physical interfaces. Ullmer and Ishii (2001) foreground the interactional differences between the physical and digital representations; Fishkin (2004) pointed to the strength of metaphor and embodiment employed; and Shaer, Leland, Calvillo-Gamez, and Jacob (2004) emphasized how the discrete elements of the system (tokens) interact with continuous ones (constraints). Each of these frameworks has advanced our ability to articulate the user experience of physical interfaces and what makes different ones different. However, none of these frameworks foreground implementation concerns: That is our interest here.

We conducted a literature survey of existing systems employing passive, untethered input from paper and other everyday objects. To concentrate on this aspect of physical interaction design, we limited the purview of this survey—and the subsequent toolkit—in two important ways. First, this taxonomy omits interfaces that employ powered sensing and actuation, such as haptic feedback. Second, this taxonomy omits 3D sensing. These constraints offer a coherency of user experience interests, making it easier to compare the systems. In addition, our intuition was that these other areas have design requirements—such as the low-latency needed by force-feedback haptics—that would require different architectural support. Several of the taxonomies and toolkits mentioned earlier in this section target these other aspects of physical interaction design.

We selected 24 applications and categorized them by four traits: input technology, input form factor, output form factor, and how tangible input and electronic output are coordinated. In doing this analysis, we felt that four broad clusters emerged, based on the high-level ways these applications employed physical input: interactive surfaces (see Figure 1).

In *spatial* applications, users collaboratively create and interact with information in a Cartesian plane; the significance of the spatial relation of objects in the plane depends on the application. These applications include augmented walls, whiteboards, and tables (Jacob, Ishii, Pangaro, & Patten, 2002, #40; Klemmer et al., 2008, #5; McGee et al., 2002 #14; Moran et al., 1999, #39; Rekimoto & Saitoh, 1999, #26; Ullmer & Ishii, 1997, #68; Underkoffler & Ishii, 1999, #81; Underkoffler, Ullmer, & Ishii, 1999, #69; Wellner, 1993, #3).

Figure 1. **The rows of this diagram present the 24 applications in our literature survey, organized into four primary categories: spatial, topological, associative, and forms. Each column describes an attribute of the application: This attribute is listed textually at the top of the diagram. In the body of the diagram an icon is used to designate the presence of the column's attribute.**



A majority of these applications use computer vision, often in conjunction with image capture. The DigitalDesk used ceiling mounted cameras to track documents and hands on a physical desktop, with a ceiling mounted projector to electronically augment the real desk (Wellner, 1993). The metaDESK (Ullmer & Ishii, 1997) is a digital desk employing iconic physical objects as

controls to a map. Collaborage (Moran et al., 1999) uses computer vision to capture paper information arranged on a wall, enabling it to be electronically accessed (see Figure 2). These pieces of paper are tagged with glyphs, a type of 2D barcode. The electronic capture of paper information enables remote viewing (e.g., a Web page view of a physical in-out board) but not remote interaction.

Topological applications use the relationships between physical objects to control application objects (Gorbet, Orth, & Ishii, 1998; Mackay, Fayard, Frobert, & Médini, 1998; Mackay & Pagani, 1994; Nelson, Ichimura, Pedersen, & Adams, 1999; Ullmer, Ishii, & Glas, 1998). The simplest topological relationship, and the one that most of these systems use, is ordering. Palette (Nelson et al., 1999) uses paper notecards to order Microsoft PowerPoint® presentations. It was released as a product in Japan in 2000. VideoMosaic (Mackay & Pagani, 1994) and mediaBlocks (Ullmer et al., 1998) use physical objects to order segments of a video. Paper Flight Strips (Mackay et al., 1998) augments flight controllers' current work practice of using paper strips by capturing and displaying information to the controllers as the strips are passed around.

With *associative* applications, physical objects serve as an index or physical hyperlink to digital media. This usage of associative comes from Ullmer and Ishii, who used it to describe systems where "tangibles are individually associated with digital information and do not reference other objects to derive meaning" (p. 921). Examples include Back, Cohen, Gold, Harrison, and Minneman (2001); Holmquist, Redström, and Ljungstrand (1999); Ishii and Ullmer (1997); Klemmer, Graham, Wolff, and Landay (2003); Lange, Jones, and Meyers (1998); Rekimoto, Ullmer, and Oba (2001); Stifelman, Arons, and Schmandt (2001); and Want, Fishkin, Gujar, and Harrison (1999). Durrell

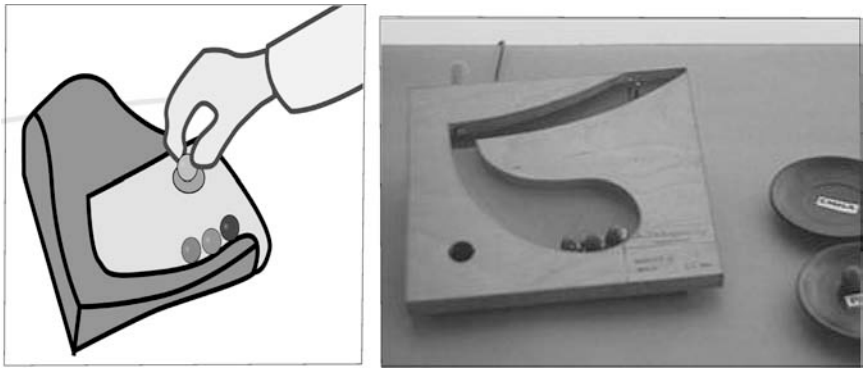
Figure 2. **Collaborage (Moran et al., 1999), a *spatial* TUI where physical walls such as an in/out board (left) can be captured for online display (right). Reproduced with permission.**

Bishop prototyped a marble answering machine (Ishii & Ullmer, 1997; see Figure 3) that would deposit a physical marble with an embedded electronic tag each time a message is left. To play a message, one picks up the marble and drops it into an indentation in the machine. Most associative applications employ either barcodes or electronic tags. Bishop created a partially functioning prototype using resistors embedded in marbles. Marbles could be identified by the unique resistance value. The Listen Reader (Back et al., 2001) is an associative system augmenting a paper book with an interactive soundtrack. Each RFID-tagged page has a unique soundtrack modified by the user's hand position. Hand tracking is accomplished via capacitive sensing. This coordination of reading and listening is highly compelling.

Forms applications provide batch processing of paper interactions (Grasso, Karsenty, & Susani, 2000; Heiner, Hudson, & Tanaka, 1999; Johnson, Jellinek, Rao, & Card, 1993). The Paper pda (Heiner et al., 1999) is a set of paper templates for a day planner. Users work with the planner in a traditional manner, then scan or fax the pages to electronically synchronize handwritten changes with the electronic data. Synchronization also executes actions such as sending handwritten e-mail.

The 24 applications we selected to examine have several attributes: physical input for arranging electronic content; physical input for invoking actions (e.g., media access); electronic capture of physical structures; coordinating physical input and graphical output; and an add, update, remove event structure—these events should contain information about the input (such as size and color) and should be easily extensible. In all of these applications, feedback is either graphical or auditory. Graphical feedback is sometimes geo-ref-

Figure 3. **The marble answering machine (Ishii & Ullmer, 1997), an associative TUI, uses marbles as a physical index to recorded answering machine messages. Left: Bishop's original sketch, redrawn by the author. Right: Bishop's prototype where resistors are embedded in marbles.**



erenced (i.e., overlaid), sometimes collocated but on a separate display, and sometimes noncollocated.

2.3. Evaluating Programming Tools and Languages

From 1974 through 1976, Goldberg and Kay taught Smalltalk to Palo Alto children aged 9 to 15 (Goldberg, 1977); this was the first work to observe nonexpert programmers developing software. They began by collecting the student's programs, and subsequently videotaped the programming sessions. Although the evaluation does not address particular language features, the student projects are highly compelling.

Broadly speaking, the subsequent 30 years have brought two significant methodological advances with direct import for our work here. In the 1980s, the attempt to understand how people program helped push scientific knowledge of cognition, and some researchers applied broader advances in cognitive psychology to understanding this domain. This work seeks to operationalize "ease of use" so that programming languages and toolkits can evaluated on readability, learnability, convenience, and comprehension (Shneiderman, 1986); this work also explores how reuse can and should occur. A seminal research system in this area was Alice, which employed user-centered methods in designing a 3D graphics environment (University of Virginia, 1995). More recently, researchers at the intersection of software engineering and human–computer interaction (HCI) have begun adapting design methods such as usability studies, heuristic evaluation, and design patterns. Using heuristics and patterns can minimize design errors, facilitate collaboration, and ease maintaining others' code. From this perspective, the success of a toolkit is judged by the extent to which it is leveraged to generate the solution.

For a comprehensive overview of this literature, see Détienne (2001) and Pane (2002); we highlight two particularly relevant projects here. The first is Pane's dissertation; it argues for usability as a primary criterion in programming system design and applies a user-centered design process to the programming system HANDS. Many of Pane's insights apply here, with the important distinction that Papier-Mâché is designed for professionals, whereas hands is designed for children. Second, the Cognitive Dimensions of Notations framework (Green & Petre, 1996) helps describe the usability attributes of programming languages. Notably, Clarke and colleagues have adopted it as both a usability inspection technique and for describing laboratory evaluation results (Clarke, 2001, 2004; Rodden & Blackwell, 2002).

Papier-Mâché's methodological contribution to this literature is its mixed-methods approach: combining fieldwork with lead designers to learn current practices, longitudinal observation, laboratory studies, and systems benchmarks. We believe this mixed-methods approach to be especially important

in emerging areas such as ubiquitous computing, where best practices are unclear and the range of ideas is much wider.

3. FIELDWORK INSPIRING PAPIER-MÂCHÉ

At the beginning of this project, we conducted structured interviews with nine researchers who have built tangible interfaces: four worked in academia, the other five in industrial research. There were 28 interview questions addressing general system questions, planning and organizational structure, software design, user and system evaluation, and difficulties in design and implementation. We conducted these interviews in person at the workplaces of researchers in the San Francisco Bay Area (three), and over the phone (one) or via an e-mail survey (five) otherwise. These researchers employed a variety of sensing techniques including vision, radio frequency and capacitance sensors, and barcodes. Here, we present the findings of this research, concentrating on the difficulties encountered in designing and developing these systems. To maintain the anonymity of interviewees, we use examples from our own research to illustrate our findings, rather than the actual systems built by interviewees.

3.1. Team, Process, and Goals

The size of the hardware and software development groups ranged from one to four (both the mean and median size was 3). Between one and five additional people were involved with interaction design and/or industrial design but not with software development (the mean was 2.4 and the median 2.7). Sometimes these conceptual contributors were the project manager or an advisor. Other times the conceptual contributors were colleagues with a different project as their primary focus.

In each of the three projects that employed computer vision, the team included a vision expert. Even with an expert, writing vision code proved challenging. In the words of one vision researcher, "getting down and dirty with the pixels" was difficult and time consuming. Writing code without the help of a toolkit yielded applications that were unreliable, brittle, or both. In addition, in two of the nonvision projects, the person who developed the tangible input was different from the person who developed the electronic interactions. In the remaining cases, the developers all had a substantial technical background and worked on both the physical and electronic portions. We speculate that if tools with a lower threshold were available to these individuals, then a larger portion of the team may have contributed to functioning prototypes, rather than just conceptual ideas.

Iterative implementation was alive and well among our interviewees. All of the interviewees' systems evolved from or were inspired by existing pro-

jects in their research groups. For two of these researchers, the evolution was from a virtual interface to a tangible interface. For two others, the tangible interface was an application or evolution of a physical input technology that the group had previously developed or had experience with. Four researchers had experience building physical interfaces prior to the project discussed in the interview. For these groups, the project we discussed was a continuation of work in this area. This next step was exploring an alternate point in a design space, exploring richer interactions, delivering greater use value, or exploring lower complexity. We see two primary reasons for this evolutionary iteration:

1. Good researchers often work in a similar area for a number of years, exploring different points in a design space.
2. Reusing existing code and/or technical knowledge lowers the threshold for application development.

Two of the interviewees began with paper prototypes, often trying out different scenarios to understand the interactions required before writing code. "The paper prototypes helped us understand the space considerations/constraints—Helped us work through the scenarios." One of these researchers also used physical objects (without computation) to think through the interaction scenarios, "to get an idea of what it would feel like to use our system."

The remaining seven interviewees began prototyping with technologies and tools that they were familiar with or had a low threshold, later exploring less familiar or higher threshold tools. Working with similar technologies and tools over a number of years affords fluency with a medium, in much the same manner as artists tend to work in the same media for long periods. An interviewee explained to us that he "was able to leverage the technology that we had earlier developed to build a prototype (with minimal functionality) in roughly eight weeks. Extended that work into the application that was fielded to end-users approximately two years later."

An alternate method of achieving fluency, or readiness-at-hand, is the use of tools with a low threshold. One researcher appreciated Max/MSP, the music and multimedia authoring environment because, "It's about I can make five versions of this by next Tuesday." This researcher chose to use Max for prototyping even though it was not appropriate for the final implementation.

We uncovered a somewhat surprising regularity in project duration. Six of the nine interviewees reported their project as lasting 2½ to 3 years. More specifically, several participants explained the the main research effort spanned about 1½ years and that there was a fast and furious period of prototyping at the beginning. The previous quote describing an 8-week prototype is an example of the fast-and-furious phase. The three remaining projects were shorter, lasting 6 months to a year.

Often, the interviewees redesigned aspects of their system architectures to support a wider range of behavior. This refactoring reflects positively on the developers; advocates of agile programming methods argue that software architectures should be simple at first, becoming more complex only as needed (Beck, 2000). This heavy refactoring also reflects negatively on the current state of software tools in this area. Much of the modularity that our interviewees introduced could be more effectively provided at the toolkit level, and indeed is supported in Papier-Mâché. Examples include changing the type of camera, switching between input technologies, or altering the mapping between input and application behavior.

The challenge of refactoring with limited development resources is that it often is "hacked"—the code is altered just enough to support the additional functionality, but not in a manner that is robust, flexible, maintainable, or understandable. One developer described their situation as "the code was way too complex at the end of the day" because there were "a lot of stupid problems" such as temporary files and global variables that inhibited reliability and malleability. These systems often rigidly relied upon all components being available and functioning properly and did not fail gracefully when most, but not all, of the components were working. In one case, the interviewee's group created an improved version of their core technology, but it was too late for use in the research prototype and happened only "after one of the engineers finished what I had begun." Toolkits can alleviate many of these development headaches.

We also spoke with the interviewees about their user experience goals. At a high level, they offered goals like "technology should make things more calm, not more daunting" and people are "torn between their physical and electronic lives, and constantly trying work-arounds." The primary motivation our interviewees had for building a tangible interface was the desire for a conceptual model of interaction that more closely matched user's behavior in the real world, often as one interviewee described it, "trying to avoid a computer."

The central finding of *The Myth of the Paperless Office* (Sellen & Harper, 2001) is that users' work practices are much more successful, and much more subtle, than a naïve techno-utopianistic perspective might suggest. In addition, the book illustrates that although digital technologies change practices, they do not supplant our interaction with paper and physical objects. The interviewees, through design insights and their own observations of practice, came to a similar conclusion. This reverence for the nuanced success of everyday objects inspired much of the interviewees work. These interviewees had many of the same goals that our literature survey found, such as avoiding projector-based solutions to increase robustness to failure. An example of this appreciation for the success of our interaction with the physical world can be

seen through one researcher's goal that his group's system, "should mimic [users] current tools as closely as possible to begin with. Functionality can be added that is unavailable in the paper tool, but only after [our interface] captures their current work practice."

3.2. Acquiring and Abstracting Input

Employing physical objects as input requires hardware to sense the presence, absence, and manipulation of objects. Computer vision was a focus of this article's research—and our interviewees—because "it gives you information at a distance without a lot of hassle, wires, and instrumentation all over the place. It puts all the smarts in one device and instrumentation is limited. It also is possible to retrofit existing spaces." However, reliability concerns and development headaches can plague vision systems.

Our interviews explored how developers sought to understand the relative tradeoffs of alternate input approaches. Most of the interviewees either experimented with different input technologies or were interested in trying different input technologies for their application. As with tools and languages, the choice of sensing technologies sometimes also shifted between prototyping and implementation. One interviewee prototyped his spatial interface with a smart Board as the sensor. Later, he replaced the smart Board's touch sensor with vision for two reasons: First, smart Boards are expensive and bulky, whereas cameras are inexpensive and small. Second, the resistance-based smart Boards provide single-input of [x, y]. (The newer SMART Boards support multitouch interaction via cameras mounted in the corners, orthogonal to the input surface.) Vision offers a much richer input space. This vision task is exactly the kind of task that Papier-Mâché can support. Although vision offers many benefits, all of the interviewees shared the sentiment that, as one researcher explained, "the real-time aspects of camera interfacing were probably the hardest." Another researcher, after having completed his project lamented that "it's not always worth it to live at the bleeding edge of technology. … Make sure you have a very good reason if you choose to work on a problem whose solution requires pushing more than one envelope."

One interviewee explored several input options before settling on RFID because "we didn't know what to do" for input. As an experiment, "I had an intern that did a version with optical sensors (vision)." Another RFID user first "spent a lot of time looking into barcodes and glyphs, but they didn't seem right."

Rapid prototyping is a central advantage of tool support (Myers et al., 2000), and vision is an excellent technology for rapid prototyping of interactive systems. It is a highly flexible, software configurable sensor. There are many applications where the final system may be built using custom hard-

ware, but the prototypes could be built with vision. The one challenge, prior to Papier-Mâché, was the difficulty of rapidly prototyping vision-based applications.

When we asked for specific toolkit functionality requests, one researcher quipped, "More than two serial ports." In nearly all contemporary software architectures, a single mouse and keyboard are presumed to be the only input devices. Employing alternative input mandates a large software engineering effort. In some cases, the hardware- or software-imposed ceiling on the number of input devices prevented the researchers from realizing their ideal designs. As most hardware has migrated to USB or FireWire, it is becoming less true that the number of "serial ports" per se are a limiting factor, but the software's ability to handle and program these simultaneous inputs is still a limitation.

A general theme among interviewees was that acquiring and abstracting input was the most time consuming and challenging piece of development. This is not, as the cliché goes, a "small matter of programming." Acquisition and abstraction of physical input, especially with computer vision, requires a high level of technical expertise in a field very different from user interface development. These novel input technologies, especially vision, do not always function perfectly. We found that consequently, it is important to design a system where occasional errors do not prevent the system as a whole from functioning, and to provide feedback so that users can diagnose and help recover from system errors. An interviewee explained, "The sensing hardware is not perfect, so sometimes we had to change interactions a bit to make them work in the face of tracking errors." This error-aware design of physical interfaces is similar to the techniques used for voice user interface design, where limiting grammar size and providing confirmation feedback help systems minimize errors, and help users diagnose and recover from errors when they do occur.

3.3. Events and Constraints Are More Appropriate Than Widgets

Widgets are a software abstraction that encapsulates both a *view* (output) and a *controller* (input). Although some post-WIMP toolkits have hoped to provide an analogue to widgets (e.g., Dey et al., 2001), in practice *toolkit support for the view* is distinct from *toolkit support for the controller*, and with good reason: A particular piece of input can be used for many different types of output and vice versa.

In designing an architecture for physical interaction, we can draw on ideas from model-based interfaces (Szekely, 1996). In a traditional design tool, a developer might imperatively specify that a group of radio buttons should be

displayed in a pop-up menu. With model-based tools, developers specify an interface declaratively, and at a high level: A developer might specify that an application should present a widget for a user to designate an item from among a set. The tools would then select an appropriate widget (radio button, drop down menu, etc.). With the emergence of ubiquitous computing, and the increased number and heterogeneity of devices, model-based techniques have seen increased interest. Papier-Mâché's event structure and behaviors draw from this literature to provide a similarly high level of abstraction, allowing developers to specify objects, events, and behavior at a semantic level, for example, "for each Post-it note the camera sees, create a Web page."

Many of the interviewees we spoke with employed a similar event-based approach. Although each of these systems was built independently, without access to the source code of others, all interviewees settled on a small set of similar events for defining behaviors in applications. The basic events hinged on notifying application objects about the presence, absence, and modification of physical objects. These events were used to bind manipulations in the physical world to electronic behavior.

Broadly speaking, events can be categorized in three groups. The first is events that specify one or more *binary* values. Examples of this flavor include events that are triggered when a button is pressed or released, when a pointer enters or exits an area, and when an RFID tag or barcode moved into or out of a sensor's range. A related set is event properties that take on one of a small number of *discrete* states. The second group is events whose properties are one or more *continuous scalar* values. One example of this type is the planar position of the mouse; or for a vision-tracked object, its position, orientation, size, and other scalar values that the vision system reports. The third type is the rich *capture* of information from the world; for example, an audio recording, video recording, or photograph. These sources may have particular aspects of the recording semantically extracted.

Using these basic primitives, some of the interviewees created higher level events, and *constrained* the system behavior to be a function of these events. For example, one interviewee created a distance operator that measured the distance between objects on a surface and constrained the behavior to be a function of that distance.

3.4. Declaratively Authoring Behavior

In general, UI programming languages and software architectures employ either an *imperative* or a *declarative* programming style, and sometimes a combination of the two. Imperative code provides a list of instructions to execute in a particular order, for example, a Java program that counts the number of words in a text file via iteration. Declarative code describes *what* should be

done but not *how*. SQL is a well-known example of a declarative language: with SQL, database queries specify the information to be returned but not the technique for doing so. Spreadsheets are another highly successful example of declarative programming. For example, the value of a cell might be defined as the summation of the eight cells above it. This value updates automatically as the other cells change, and no "program" or control loop needs to be written to implement this functionality; it is handled by the system internally. Declarative programming has a model-based flavor, and is beginning to rise in popularity for programming Web services and GUIs (e.g., Microsoft's Windows Presentation Foundation).

Tangible interfaces *couple* physical input with electronic behavior; for example, a marble represents an answering machine message (Poynor, 1995). All nine of our interviewees described the behavior of their system as providing tangible input coupled with electronic behavior through event-based bindings or queries. This coupling can be either discrete (such as the marbles) or continuous (such as node-link diagrams); in general, constraint specifications such as these couplings can be more concisely and flexibly expressed declaratively than imperatively. The most common relationship we observed was a direct, 1:1 correspondence between physical elements—tagged with a barcode or RFID—and digital ones: in essence, physical hyperlinks to a particular piece of electronic behavior or media. These elements are bound together, and the behavior is executed through an event. With spatial applications the binding is parameterized through the location, size, shape, and other identifying characteristics of the object, and with topological applications the behavior is influenced by the relationships between the physical objects. Although our interviewees described their systems clearly using declarative terms, not everyone felt this declarative model was effectively implemented in their software. Several interviewees wished they had a more flexible method of defining bindings, making it easier to change the input technology and to explore alternative interactions for a given input technology.

Three of the nine applications provided the ability for multidevice, networked interaction. These systems were designed roughly around a distributed MVC architecture, where a database served as the server and central connection point. In these systems, the clients supported sensing input, information presentation, or both. Clients would report events to a server hosting the model, and then the server notified all of the other clients. In the most sophisticated system, the interaction clients were heterogeneous. Board clients reported data to a board server, and this server then sent events to applications, which were often Web apps but could also be devices like a printer. The Papier-Mâché toolkit targets single-machine applications. By virtue of the fact that Papier-Mâché integrates easily with the SATIN ink UI toolkit (Hong & Landay, 2000), there is a remote command system available for replicating

events between hosts. A toolkit that more explicitly supports distributed interaction and data storage could draw upon much of the literature in ubiquitous computing toolkits and is an area for future research.

3.5. Language, Architecture, and Reuse

Our interviewees used several different programming languages: C++ (three), Java (two), Prolog (one), Director (one), Visual Basic (one), and Python (one). Two of the non-Java teams have since switched to Java. Eight of the nine interviewees used a Windows PC as their development platform.

Most of the interviewees chose a programming language based on one particular requirement. They requirements cited were as follows:

1. *Technology integration*: Sometimes, the decision was made to ease integration with a particular piece of input technology, for example, the Designers' Outpost vision system was built in C++ because the OpenCV library it used was in C.
2. *Library support*: The majority of our interviewees chose a language based on the library use it facilitated. Two developers chose the Windows platform and a Visual Studio language specifically for their easy interoperability with Microsoft Office. A third interviewee was constrained to the Windows platform for integration reasons, and chose Director for its rapid development capabilities. Two of our interviewees decided on Java because of its rich 2D graphics libraries.
3. *Developer fluency*: For one interviewee, C++ was chosen "because [the lead software developer] knew it well." The same fluency sentiment was expressed differently by another interviewee that C++ "was our language of the time. Now we're Java."
4. *Rapid development*: One interviewee told us that "I used Python. This language make prototyping fast and easy. It can be too slow at times, but not too often thanks to Moore's law." Another interviewee, previously mentioned, settled on Adobe Director for rapid development reasons.

These four reasons—technology integration, library support, developer fluency, and rapid development—informed our choice of Java as a programming language. Java offers excellent library support, many developers are fluent in it, development time is very fast for a full-fledged programming language, and it was tractable for us to provide input technology integration. The double-edged sword of Java is its platform independence. Development and execution on multiple platforms provides a wide audience of developers and a flexibility of deployment, and this was a feature that one interviewee specifi-

cally requested. However, this independence comes at a performance cost and makes integration with novel input technologies more difficult.

User interface tools comprise two pieces:

> A library of interactive components and an architectural framework to manage the operation of interfaces made up of those components. Employing an established framework and a library of reusable components makes user interface construction much easier than programming interfaces from scratch. (Myers et al., 2000, p. 7)

Our interviews found that for this emerging area, each development team was creating an architecture, a set of library components, and an application (though the developers did not generally describe their work with such an explicit taxonomy). For all of our interviewees, Papier-Mâché would have eliminated the need to create a software architecture, with the exception of the distributed portion of the applications. Papier-Mâché would have also drastically minimized the amount of library code that needed to be written. Examples of library code that would have remained include particular vision algorithms, support for particular brands of RFID readers, and support for particular flavors of barcodes. Papier-Mâché would have also substantially reduced the amount of application functionality code, shifting the balance from creation from scratch toward composition of components.

3.6. Importance of System Feedback for Users and Developers

Good feedback is a central tenet of user interface design (Norman, 1990, chap. 4). One researcher found that "one key issue was that sensing errors were pretty mysterious from the users' perspective." Providing visual feedback about the system's perception of tracked objects helps users compensate for tracking errors. Feedback is particularly important to developers, because the complexity of their task is so high.

Debugging is one of the most difficult parts of application development, largely because of the limited visibility of dynamic application behavior (Détienne, 2001). The novel hardware used in tangible UIs, and the algorithmic complexity of computer vision, only exacerbate this problem. One interviewee had "the lingering impression that the system must be broken, when in fact the system was just being slow because we were pushing the limits of computation speed." The current state of debugging tools in this area is quite poor; another interviewee used Hyperterm, the Microsoft Windows command line tool designed for modem communication, to debug the serial communication of their input hardware.

Imperative software languages make it very difficult to see the flow of control of an application, especially one that is highly reliant on events. Control

flow moves rapidly between class files, making it so that understanding a particular behavior can be quite difficult. One interviewee explained that "the debugging of incorrect or missing elements within the fusion operation was the most cumbersome and time-consuming parts of the development process." This should not be taken as a critique of object-oriented design, but rather as a critique of relying solely on textual information for software understanding. Well-designed visual tools can be more effective for creating and debugging dataflow. Successful examples of visual dataflow graphs include the Max/MSP midi authoring system and spreadsheets (Burnett, Sheretov, Ren, & Rothermel, 2002; Jones, Blackwell, & Burnett, 2003).

Visualizations explaining the current state of an application are useful at short time scales, and Papier-Mâché provides this. An area for future research would be to provide logging and feedback of application behavior at longer time scales. One interviewee told us that he "put it up, and ran it for about six months in two or three locations in the building." To evaluate the robustness of the system, he then watched for failure mode. "These failure modes helped drive further development. This failure mode analysis is key." Another told us, "We were worried about robustness. So I made a prototype and left it in the hall for months." There are three broad areas where long-term monitoring could help: software errors (such as crashes), recognition errors, and usability errors (where the software behaved as expected, but the user interface was poor). The first could be addressed by self-evaluating software techniques similar to those of Liblit and colleagues (Liblit, Aiken, Zheng, & Jordan, 2003). The second could be addressed through logging user mediations. The last could be addressed by logging access to help systems (when available) and undo actions, or by introducing a user feedback system (a simple example would be an "I don't understand" button that could log application state).

Several participants specifically asked us for better error diagnostic tools. One researcher gave us some example questions that she hoped tools would solve: "This crashed, what happened? Why won't it boot? How far does it get?" Interviewees also asked us for the ability to remotely administer and diagnose deployed systems. They wanted to be able to find out answers to questions such as, "Which sensors did they use? In the way you think or something else completely?"

3.7. Summary

The threshold for developing robust tangible interfaces in the absence of tools is significant: it discouraged experimentation, change, and improvement, limiting researchers' ability to conduct user evaluation, especially longitudinal studies. One interviewee avoided these studies because his team

lacked the resources to "add all the bells and whistles" that make a system usable. The results that had the greatest effect on the Papier-Mâché architecture were (a) all of the interviewees used an event-based programming model, (b) interviewees experimented with different input technologies and that each of these prototypes was built from scratch, and (c) understanding the flow of an application and debugging failures is quite difficult.

4. THE PAPIER-MÂCHÉ ARCHITECTURE

Our literature survey, experiential knowledge, and fieldwork data show that a toolkit for tangible input should support

- Techniques for rapidly prototyping multiple variants of applications as a catalyst for iterative design
- Many simultaneous input objects
- Input at the *object* level, not the *pixel* or *bits* level
- Heterogeneous classes of input
- Uniform events across the multiple input technologies, facilitating rapid application retargeting
- Classifying input and associating it with application behavior
- Visual feedback to aid developers in understanding and debugging input creation, dispatch, and the relationship with application behavior

4.1. Introduction

These goals provided the basic framing for the architectural decisions in Papier-Mâché. Papier-Mâché is an open-source Java toolkit written using the Java Media Framework and Advanced Imaging (JAI) APIs. It abstracts the acquisition of input about everyday objects tracked with a camera, or tagged with barcodes or RFID tags. These three technologies span the vast majority of input needs of our 24 inspiring applications. The exceptions are systems that employ tethered 3D tracking (Fitzmaurice, Ishii, & Buxton, 1995), speech input (McGee, Cohen, Wesson, & Horman, 2002), or capacitive sensing (Back et al., 2001). This library supporting input technologies also illustrates Papier-Mâché's capability for a developer to originally implement a system with one technology and later retarget it to a different technology. The need for rapidly retargeting input encouraged our use of event-based bindings—rather than widgets—as an architecture for tangible interaction.

We explain the Papier-Mâché architecture using two examples: an RFID implementation of Bishop's marble answering machine (Ishii & Ullmer, 1997), and a simplified version of PARC's Collaborage In/Out Board (Moran et al., 1999) using computer vision and barcodes. For each of these applica-

tions, a developer has two primary tasks: declaring the input that he or she is interested in and mapping that input to application behavior.

4.2. Input Abstraction and Event Generation

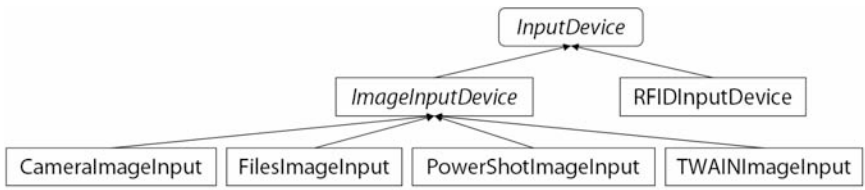
Papier-Mâché represents physical objects as *Phob*s. (In this article, Java class names are designated in italics.) The input layer acquires sensor input, interprets it, and generates the *Phob*s. A developer is responsible for selecting input types, such as RFID or vision. He or she is not responsible for discovering the input devices attached to the computer, establishing a connection to them, or generating events from the input.

These "accidental steps" (Brooks, 1987) are time consuming and require substantial hardware and computer vision expertise, a field very different from user interface development. For example, the marble answering machine developer adds his or her application logic as a listener to an RFID reader but does not need to manage a connection to the hardware. Similarly, the Collaborage developer tells Papier-Mâché that he or she is interested in receiving computer vision events with a video camera as the source.

A piece of physical hardware that is attached to the computer needs to implement the *InputDevice* marker interface (see Figure 4, top row). A marker interface is a design pattern where a programmer creates an interface without any methods; implementing a marker interface is a technique for tagging the implementing class as providing a certain type of support. Using the inheritance mechanism as a tag provides for both compile-time and run-time verification that class instances provide the necessary support. In this case, each input device has a unique API for dealing with the underlying hardware. The *InputDevice* marker interface tags the implementing classes as being responsible for input acquisition.

For each category of input hardware, there is a class implementing the marker interface that provides a general API to Papier-Mâché for handling

Figure 4. **The inheritance hierarchy for physical input devices. Each device class encapsulates a physical input. The InputDevice is a marker interface: it is an interface class that contains no methods. Classes implement the marker interface to denote that they represent a physical device.**



that type of input (see Figure 4, middle row). The *ImageInputDevice* and *RFIDInputDevice* are the two library examples that Papier-Mâché includes for general device types.

For each device class, there are several different APIs currently available. Papier-Mâché abstracts the varying aspects of particular members of an equivalent device class. For example, different types of cameras have different types of APIs for image acquisition. These specific device styles use the public interface and event dispatch from their superclass, adding acquisition functionality that is particular to their type (see Figure 4, bottom row). Device drivers obviate the need to create implementations for all camera models; only one library element needs to be written for each imaging standard. The two main imaging standards are webcams and twain. The Papier-Mâché library supports webcams through the Java Media Framework. Java Media Framework supports any camera with a standard driver, from inexpensive Webcams to high-quality 1394 (FireWire) video cameras. Papier-Mâché supports twain capture, an alternate protocol designed for scanners but increasingly used for digital cameras, through Java twain. In addition, Papier-Mâché supports the use of simulated input through the *FilesImageInput* class. Simulated input is useful for creating and testing code when a camera or the environment is not available and for unit-test-style repeated verification of functionality of a system as it changes over time.

Consumer digital cameras are designed for use by a human photographer. In our research, we have also found it useful to control these high-resolution low-frame-rate cameras computationally, both for structured image capture and for computer vision. However, currently, there is no widely adopted computational control standard for this class of device. In a few years, twain and/or the Windows Imaging Acquisition standard will likely emerge as a commonly adopted standard. In the interim, we have provided the *PowerShotImageInput* class in the Papier-Mâché library. This acquires images from Canon's PowerShot cameras. We chose these cameras because they are high quality, are readily available, and have the best developer support for computational control. Papier-Mâché calls native Windows code for controlling these Canon digital cameras (the binding of native code to Java code is accomplished through the Java Native Interface) and for communicating with the Phidgets RFID tags (this is accomplished through the IBM Bridge2Java system, which is a tool that automatically generates JNI stubs for ActiveX objects).

Event Generation

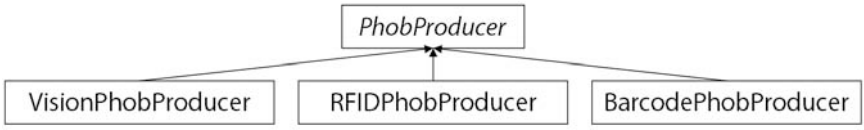
Once the developer has selected an input source, Papier-Mâché generates events representing the addition, updating, and removal of objects from a

sensor's view. Event types are consistent across all technologies. Providing high-level events substantially lowers the application development threshold and facilitates technology portability. The class responsible for this event generation is the *PhobProducer* (see Figure 5, top row). *PhobProducer* is an abstract class—it contains the event dispatch mechanisms and maintains the set of objects currently in a sensor's view but not the techniques for interpreting and packaging input from an *InputDevice*. These techniques are delegated to the subclasses (see Figure 5, bottom row). There is a 1:1 mapping between *InputDevice* instances and *PhobProducers* instances. The separation of input acquisition from event dispatch is an important one.

The bottom level of the *InputDevice* hierarchy—the library classes that wrap particular types of input devices—can be seen as high-level devices in the tradition of UIMS systems. It is quite possible, as with mouse-based UIMS tools, that improvements in the device driver space will minimize the need for the bottom level of the *InputDevice* hierarchy. (The top two levels of the hierarchy compose the Papier-Mâché architecture and will continue to be needed.) The RFID community is headed in this direction and creating standards that cover the "air interface" (how tags and readers communicate), and industry appears to be slowly headed toward an xml standard for how RFID hardware and applications communicate.

While all technologies fire the same *events*, different technologies provide different *types of information* about the physical objects they sense. RFID provides only the tag and reader IDs. Vision provides much more information: the size, location, orientation, bounding box, and mean color of objects. Size, location, and orientation are computed using image moments (Freeman et al., 1998). Because this set is commonly useful, but not exhaustive, *VisionPhob*s facilitate developer extension by storing a reference to the image containing the object. Application developers can use this for additional processing. Barcodes contain their ID, their type: EAN, PDF417, or CyberCode (Rekimoto & Ayatsuka, 2000), and a reference to the *VisionPhob* containing the barcode image. The *BarcodePhob* class includes an accessor method that returns this *VisionPhob*, allowing developers to access all of its information, such as location and orientation. Encouraged by mass proliferation of digital cameras in

Figure 5. **The inheritance hierarchy for PhobProducers. Producers are paired with InputDevices; they take input from a device and generate PhobEvents. The abstract PhobProducer base class manages the event listeners and the production of events.**



mobile phones, recent years have seen significant advances in fiducials. Fiducial tracking libraries—such as ARToolkit—can be integrated into Papier-Mache as an alternate barcode library element.

RFID Events

Generating RFID events requires minimal inference. Each reader provides events about tags currently placed within range. We currently use Phidgets (Greenberg & Fitchett, 2001) RFID readers, which sense only one tag at a time. The inclusion of Phidgets demonstrates the architecture's ability to handle RFID tags and enables users to rapidly develop RFID-based interfaces. If a final implementation required a particular brand of RFID (or a reader that supported simultaneous tag reads), it would be fairly straightforward for a developer to add an additional RFID library element.

When a tag is placed within range of a reader, Papier-Mâché generates a *phobAdded* event. Each subsequent sensing of the same tag generates a *phobUpdated* event. If the reader does not report a tag's presence within a certain amount of time, Papier-Mâché infers that the tag has been removed, generating a *phobRemoved* event. This inference technique was introduced by Want et al. (1999). RFID events contain both the tag id and the reader id. Applications can use either or both of these pieces of information to determine application behavior.

Vision Events

Generating vision events requires more interpretation of the input. Image analysis in Papier-Mâché has three phases: camera calibration, image segmentation, and event creation and dispatching. Application developers can override each of these processing steps if they are so inclined. The contribution of the Papier-Mâché research is not in the domain of recognition algorithms; we drew the vision techniques directly from the literature. Papier-Mâché's technical contribution is a software architecture that provides a high-level API for the use of computer vision in the user interface so that non-vision experts can build vision-based interfaces, and a separation of concerns between UI design and algorithm design.

We have implemented camera calibration using perspective correction—an efficient method that most contemporary graphics hardware, and the JAI library, provide as a primitive. More computationally expensive and precise methods exist, see (Forsyth & Ponce, 2003, chap. 1–3) for an excellent overview of the theory and methods.

The segmentation step partitions an image into objects and background; see Forsyth and Ponce (2003, chap. 14–16) for an overview of image segmen-

tation. There are two broad classes of segmentation techniques: stateless techniques that process images individually without regard to earlier information, and stateful techniques that take into account prior information. The Papier-Mâché library includes an example of each category of *SegmentationTechnique*: Edge detection (Canny, 1986) is stateless and background subtraction is stateful. Vision developers can create additional techniques by implementing the *SegmentationTechnique* interface. Each segmentation technique takes as input a raw camera image and generates a bilevel image where white pixels represent object boundaries and all other pixels are black. Edge detection finds points in an image with abrupt changes in pixel values. An example is the Collaborage reimplementation developed by Andy Kung; it used edge detection to find the 2D barcode glyphs that individuals moved on a wall to indicate whether they were *in* or *out*.

At a high level, background subtraction works by comparing the current camera image to a prior camera image or an aggregate of prior images. The theory is that the constant portions of an image represent the background (e.g., the wooden surface of a desk) and the changed portions of an image represent the foreground (e.g., documents placed on the desk). In practice, comparing against an aggregate performs better than comparing against a static background image because aggregates enable the inclusion of slowly changing information as background information (e.g., as the sun moves across the sky the light on the desk changes). A standard technique for creating an aggregate image is to use an exponentially weighted moving average filter (Forsyth & Ponce, 2003).

At each time step **t**, the current image I_t is subtracted from the aggregate image I_A; the resulting difference image $I_{A\text{-}t}$ represents the change in the scene. A new aggregate $I_{A'}$ is then computed by a weighted addition of the current image I_t with the earlier aggregate I_A: the current image is given weight α and the earlier aggregate is given weight $(1 - \alpha)$; the value of α is between 0 and 1. The technique is called "exponential weight" because this recursive addition is equivalent to the summation $\sum_{n=0}^{t} (I_n \times \alpha^{t-n})$. The fraction or "weight" of each individual image I_n is α raised to the exponent t - n; the weight of old images approaches 0 in the limit. Lederer and Heer (2004) developed the initial version of Papier-Mâché's background subtraction code for ceiling-mounted camera tracking of individuals in an office for their All Together Now system.

Labeled foreground pixels are grouped into objects (segments) using the connected components algorithm (Horn, 1986). We create a *VisionPhob* class for each detected object. At each time step, the vision system fires a *phobAdded* event for new objects, a *phobUpdated* event for previously seen objects, and a *phobRemoved* event when objects are removed from view.

Of course, there are many more sophisticated techniques for scene and object recognition than edge detection, and users of Papier-Mache would clearly benefit from their inclusion in its library. The included algorithms are valuable primarily for illustrating how recognition components integrate architecturally with the other portions of the toolkit. The included techniques are certainly sufficient for many, and more important the basic object information that is used to compute behavior is largely consistent across segmentation techniques. Applications that require specific domain information (e.g., the species of an animal) could accomplish this by extending the *VisionEvent* class. For all of our inspiring applications, this would not be necessary.

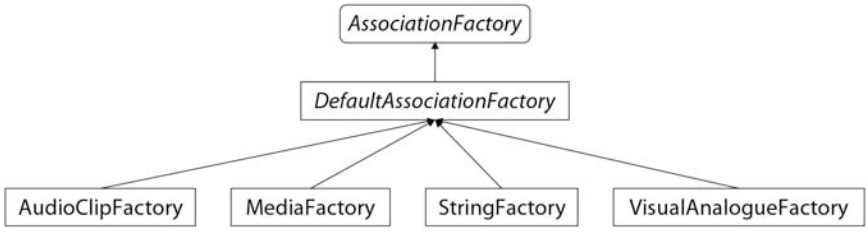
4.3. Declaratively Associating Input With Behavior

Papier-Mâché provides three levels of abstraction for handling behaviors associated with the objects it detects. (a) *PhobEvent* instances carry information about objects detected by a *PhobProducer*. (b) *AssociationFactory* instances provide a mechanism for creating and modifying application logic (*AssociationElts*). (c) The *BindingManager* is built using the two previous primitives: it receives all *PhobEvents* from all *PhobProducers* and uses *AssociationFactory* instances to create *AssociationElts*. We discuss each of these architectural abstractions in turn.

Events are the basic input dispatch primitive: At this level, developers manually instantiate producers and devices, register themselves as listeners, and receive the events that these producers generate. All application logic, and the relationships between input and application logic must be programmed manually.

A factory is a design pattern that provides an interface for creating families of related objects (Gamma, Helm, Johnson, & Vlissides, 1995, pp. 87–96). In Papier-Mâché, an *AssociationFactory* (see Figure 6) creates *AssociationElts*. The

Figure 6. **The inheritance hierarchy for factories: objects that create *AssociationElts* from *Phob* input. The top level is the *AssociationFactory* interface. The middle level is the *DefaultAssociationFactory* abstract class; this class provides the ability to be *VisuallyAuthorable* and the ability to serialize to XML using JAXB.**



AssociationFactory is an interface that contains a method that creates an *AssociationElt* and returns it. In some cases, the *AssociationElt* created is parameterized by capture from the current environment. An example of this type is the marble answering machine, where each *AudioClip* created records an audio clip from a microphone. In other cases, the created *AssociationElt* is parameterized by the properties of the *Phob* passed in. All of the inspiring spatial interfaces require this behavior. These systems use the location (and often the orientation) of the physical object to control spatial aspects of graphical objects. In the remaining cases, when an *AssociationElt* is created, it prompts the user with a dialog box to specify its parameters. The WebStickers system for using barcodes as physical hyperlinks, if written with Papier-Mâché, might pop up a dialog box asking the user to specify a URL. Some of the factories (such as the *AudioClipFactory*, *MediaFactory*, and *StringFactory* provided in the Papier-Mâché library) are agnostic to the type of input device that created the *Phob*. Others require that a particular type of information be available in the *Phob*. The *VisualAnalogueFactory* requires a *VisionPhob*, as it uses the location and orientation of the *Phob*. Whereas the creation and invocation of behaviors is handled through the factory, developers must manually handle the management of multiple devices and/or multiple behaviors.

Earlier, we introduced the technique of declaratively authoring application behaviors by binding a specific set of physical input to a particular piece of application logic. This need inspired the *BindingManager* class in Papier-Mâché. The binding manager automatically registers itself with all available *PhobProducers*, manages the flow of events, and automatically creates behaviors. The binding manager contains *classifier/behavior* pairs and it is the recipient of these events. It invokes application behavior for each element the classifier matches. Developers select *PhobProducer*(s) that will create input events, *ObjectClassifier*(s) that select a subset of generated input, and *AssociationElt*(s) that the factory should create.

The *BindingManager* contains a map data structure that maintains past and present bindings and creates new bindings in response to physical input. The manager listens for new *PhobEvents*. When a *Phob* is first seen or updated, the *PhobProducer* fires an event with the *Phob* as its payload. The *BindingManager* receives this event and compares it to the table of classifiers.

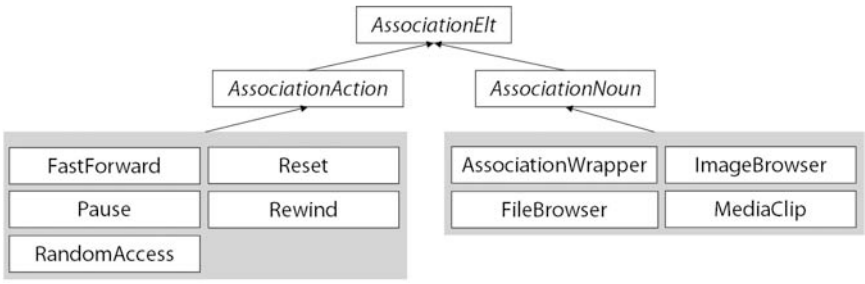
Here, the developers' primary goal is instantiating and parameterizing classifiers and behaviors. Many of our inspiring applications can be created solely by parameterizing existing library classes. This parameterization-based approach is inspired by Interactors (Myers, 1990), and lends itself to visual programming (Chang, 1990)—a subsequent section describes a visual tool. For more complex functionality, developers can implement their own application logic by creating a custom behavior.

We now illustrate how developers employ this declarative programming style in Papier-Mâché, using the In/Out board as an example. In this system, a barcode id represents a person, and its location represents whether they are *in* or *out*. Developers author these representation mappings by implementing a *BehaviorFactory*, which listens to events from the input sources. The factory receives a callback to create a new *AssociationElt* (see Figure 7) representation instance (e.g., a "person") for each new *Phob* created, and an update callback to modify that element's state (e.g., whether they are in or out) each time a *Phob* is updated.

Each *AssociationElt* either represents a particular piece of content (we call these *Nouns*) or they operate on a piece of content (we call these *Actions*). This distinction is also used in Fishkin's survey of tangible user interfaces (Fishkin). Operationally, a *Noun* can be the selection focus of an application, whereas an *Action* controls the current selection focus. In the In/Out board, each person would be a *Noun*.

The Papier-Mâché library includes four common types of nouns and five common media manipulation actions (see Figure 7). The *FileBrowser* wraps files and URLs, the *ImageBrowser* wraps images, and the *MediaClip* wraps audio and video files. All of the topological and associative applications can be built with these three nouns, with the exception of Paper Flight Strips (Mackay et al., 1998), which requires air traffic control information. The fourth noun, *AssociationWrapper*, is more general purpose. It wraps any functionality that the developer provides. For example, an *AssociationWrapper* can wrap a *JPanel* or other graphical element. An *AssociationWrapper* would be used to wrap each person in the In/Out board. The five media manipulation actions in Papier-Mâché's library were also chosen because media manipulation operations cover a majority of the behavior of our inspiring appli-

Figure 7. **The inheritance hierarchy for associations. Associations are the elements in the Papier-Mâché architecture that input is bound to. These elements can either be nouns or actions. The Papier-Mâché library includes five common media manipulation actions, and four common types of nouns.**



cations. *FastForward*, *Pause*, and *Rewind* perform their respective action on a *MediaClip*. *RandomAccess* moves the current position of a *MediaClip* to a designated place. *Reset* moves the current position of a *MediaClip* to the beginning.

4.4. Switchable Classes of Underlying Technology

The Papier-Mâché library encapsulates three classes of physical devices: electronic tags (e.g., RFID tags), barcodes, and image analysis. The first two involve manually tagging objects: before an object can be used with the system, it must be properly "suited up." The visual signature of an object can also be used as a tag (e.g., using the content of paper documents as a tag signature), with the caveat that this higher level recognition task may at times decrease robustness. The main benefit is that any object can be appropriated for use. The main drawback is that because these "tags" are human generated, not machine generated, there are no guarantees that the tag-space is well partitioned, or even partitionable. Two blank documents have the same signature, for example.

Image analysis deserves some comment because it is substantially more flexible than the other channels. Papier-Mâché supports the use of vision for pure recognition, for pure capture, and for using both together for structured capture, such as capturing the contents of recognized documents. Image capture—acquiring input from an image source—is the first step in any vision system. In addition, some interfaces use the raw capture as simply an image, and no further processing is required. The Peripheral Display Toolkit (Matthews, Dey, Mankoff, Carter, & Rattenbury, 2004) is an example of a system that used Papier-Mâché for its flexible and low-threshold image acquisition API. This project's use of Papier-Mâché is described in further detail in Section 5.4.

One benefit of Papier-Mâché's vision architecture is that it provides a separation of concerns. Application developers can quickly develop a functional prototype using the provided libraries. Because the architecture is already provided, vision developers can work in parallel (or after the completion of the UI) to customize or replace the underlying vision algorithms as dictated by the domain.

4.5. How Papier-Mâché Differs From a GUI Input Model

Papier-Mâché events have some similarities to GUI events, but they also differ in important ways. We use the vision implementation of these events to illustrate this. Applications receive *VisionEvents* from an *ImageSourceManager* by registering *VisionListeners*. A *VisionListener* receives events about all objects larger than a specified minimum size. This minimum size constraint is solely for performance; it avoids an inappropriately large number of events from

being generated. *VisionEvents* have a similar API to Java's *MouseEvents*. There are several important differences, however.

1. A mouse is a *temporally multiplexed* (Fitzmaurice et al., 1995), generic input device; the meaning of its input is constructed entirely through the graphical display. In traditional GUIs there is always exactly one mouse (though some research systems have extended this, providing multiple mice). The behavior of moving the mouse or pressing a mouse button changes over time, as a function of the mouse's position and the application state. In contrast, tangible interfaces nearly always employ multiple input devices, and these inputs are *spatially multiplexed*, as in the knobs and sliders of an audio control board. The audio board contains many knobs: Each knob is always available and always performs the same function. In most physical interfaces, the functionality of an input object is conveyed by its physical form factor, markings on the object, and the object's location. In these systems, the input devices are lightweight; multiple objects appear and disappear frequently at runtime. Although *MouseEvents* offer only position and button press updates, *VisionEvents* offer Add, Update, and Remove methods.
2. With a traditional mouse, the only input is (x, y) position and button presses. With physical objects on a plane, the captured information is position (x, y), orientation (?), size, shape, and visual appearance. Papier-Mâché provides bounding box, edge pixel set, and major and minor axis lengths as shape information. It provides the mean color as well as access to the source image data, for visual appearance.
3. Although the position of a mouse and the state of its buttons is unambiguous, the information retrieved through computer vision is often uncertain. To address this, Papier-Mâché provides a lightweight form of classification ambiguity (Mankoff et al., 2000). In Papier-Mâché, classifiers are responsible for reporting ambiguity; this is currently achieved through a scalar confidence value.
4. Similarly, with computer vision, the raw input (a camera image) contains a richness unavailable in the high-level events. These high-level events are an appropriate match for most of a developer's goals, but there are two cases where access to the original source data are beneficial: when a developer would like to conduct additional processing beyond object detection (such as recognizing an object as a unique instance, rather than simply a member of a class) or when a developer would like to capture the raw image data for subsequent display. To accommodate this, Papier-Mâché provides access to the original pixel data along with the region of interest that the object was located in.

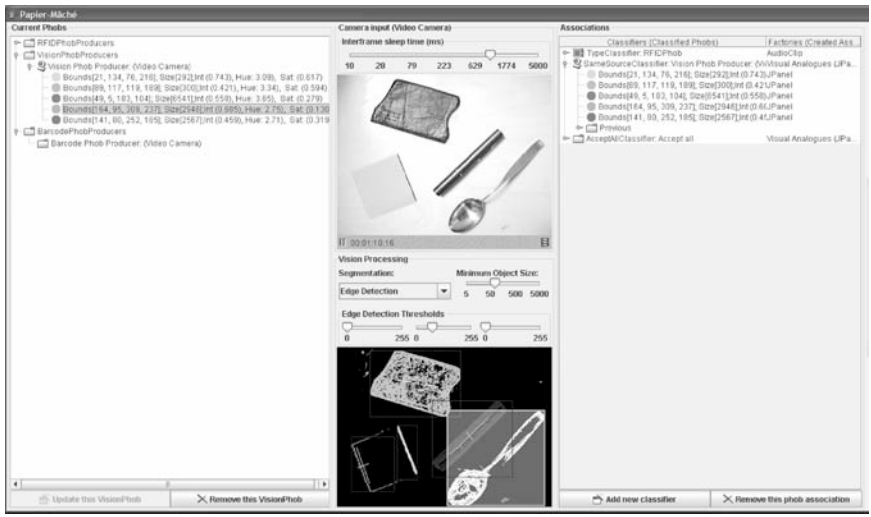
4.6. Program Monitoring: Application State Display

In addition to the Java API, Papier-Mâché provides application developers monitoring facilities (see Figure 8). It displays the current input objects, image input and processing, and behaviors being created or invoked through the binding manager.

At the left-hand side of the monitoring window, Papier-Mâché displays a three-level tree. This allows developers to see the current state of the system. The top level presents the *PhobProducer* types. These are the broad input classes: RFID, vision, and barcode. The second level presents the *PhobProducer* instances. These are instances of objects creating input events; each instance is listed beneath its type. The bottom level presents the currently visible *Phob*s. Each *Phob* appears in the hierarchy beneath the producer that sensed it. The *Phob* displays a summary of its properties; *VisionPhob*s also have a circular icon showing their mean color.

Raw camera input is displayed at the top of the second pane. At the bottom of the second pane is the processed image; it displays each object's outline, bounding box, and orientation axis. Clicking on an object in either the "Current Phobs" view or the vision view highlights it in both views.

Figure 8. **The monitoring window. In the first column, each current object appears in the hierarchy beneath the producer that sensed it. The second column displays the vision input and output. The third column displays classifiers (in this figure, RFID tags are associated with audio clips, and vision objects with graphical analogues).**



Papier-Mâché provides the WOz control through the *add* and *remove* buttons at the bottom left of the monitoring window. Selecting a producer type (the top level of the hierarchy) and pressing the add button creates a new *PhobProducer*. The system queries the available devices and presents a dialog box allowing the user to create a producer that uses real input or a "fake" producer that will be controlled by the user's WOz input. When a producer is selected, the remove button will remove it from the system, and the add button will create a new *Phob* with the selected producer as the source. For example, with computer vision, selecting a *VisionPhobProducer* and pressing *add* generates a *Phob* with a reference to the camera's current image. When a *Phob* is selected, it can be removed by pressing remove, and its information can be updated by pressing update. In all cases, the created events appear exactly the same as if they had come from the sensor. This WOz control is useful when hardware is not available and for reproducing scenarios during development and debugging.

For example, Andy Kung did not always carry a camera while creating the Collaborage rewrite on his laptop. He saved several camera frames of the whiteboard in different states onto his laptop, and this allowed him to test the application when the camera was not connected to his computer. It also allowed him to perform repeated unit tests with identical input to verify that the code was functioning correctly.

Papier-Mâché offers developers control over two axes that directly affect performance. The first is the time that the image-processing thread should sleep between processing of images. Applications requiring interactive feedback benefit from a short sleep time (5 or 10 msec); applications where there is no interactive feedback could reduce processor load by opting for a longer sleep time (perhaps half a second). The second choice developers must make is the minimum size of objects. This choice helps limit a flood of events about "objects" that may simply be noise in the image sensor. The slider in the middle of the monitoring window controls this parameter (minimum object size in pixels). To aid developers in making this choice, the size of currently detected objects is listed with each object on the left-hand panel and the size of all objects that match a classification is displayed with each object on the right-hand panel. Developers then choose a value that is safely below the smallest item.

4.7. Visually Authoring and Modifying Application Behavior

Papier-Mâché's graphical interface provides both monitoring information and authoring information. Interacting with the visual authoring facilities allows developers to create and modify the application's runtime code. Papier-Mâché uses xml as a persistent representation of the visually authored program. This is accomplished via JAXB, a tool for serializing Java objects.

To create new behaviors, developers select the technology that will provide the input on the left-hand side of the monitoring window and press the "add new classifier" button in the lower right of the monitoring window. This invokes a dialog box to create a new binding between a class of physical input and an application behavior (see Figure 9). A developer selects an input classifier on the left-hand side of the dialog; the set of available classifiers is based on the technology selected in the monitoring window. The developer then selects the type of application behavior that should be created on the right-hand side. The list comprises the set of all *AssociationElt* objects that have registered themselves with the monitoring window. By default, this is the five *AssociationActions* and four *AssociationNouns* that the Papier-Mâché library provides.

To provide custom behaviors, developers can implement and register their own *AssociationElts*, and then specify the physical input parameters they are interested in. One such dialog—for finding objects of a specified color—is shown in Figure 10. The parameter specification dialogs were created by De Guzman, Ramírez, and Klemmer (2003). Each dialog provides a GUI where developers specify each of the classifiers parameters; visual feedback about the currently specified class is presented in the lower left. The parameters of the classifier are initially set to the currently selected *Phob* in the monitoring window. For example, when the developer begins creating a new classifier in Figure 9, a pair of olive-brown sunglasses is selected. Upon selecting MeanColorClassifier, the corresponding dialog appears with olive-brown as the selected color (see Figure 10). This color-based classifier offers control over the mean color and the tolerance, specifying the span of colors to include. In the future, it may be more appropriate to have this value selected automatically by the system using techniques similar to Crayons (Fails & Olsen, 2003).

Figure 9. **The dialog box for creating a new binding between input and behavior.**

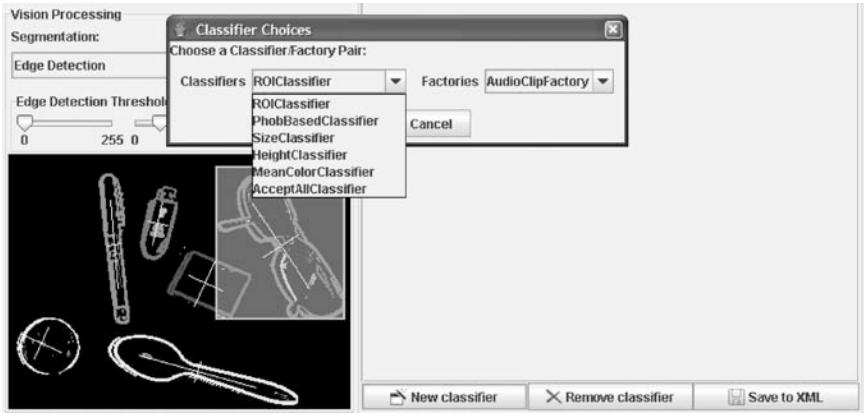
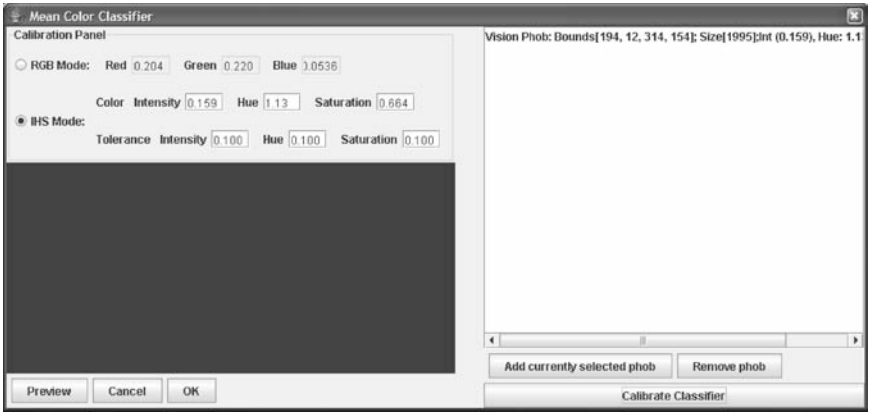


Figure 10. **A dialog box where developers specify the color of objects of interest. Dialog box designed by De Guzman and Ramírez (De Guzman, Ramírez, & Klemmer, 2003).**



5. EVALUATION

This section describes a mixed-methods evaluation of the Papier-Mâché architecture. We hope the results shed useful light on architectural considerations for integrating physical and digital interactions. In addition, this research introduced a mixed-methods approach for evaluating design and usability issues of a software API. The results of these studies help to illustrate the different insights produced through different methods and how composing the results of different techniques yields a fuller use picture of a software API.

Although there has certainly been prior work on evaluating software tools, this area is more limited than might be expected, perhaps because, as Détienne (2001) wrote,

> The dominant problems have been perceived as technical rather than as related to the usability of the systems. The introspective approach, which is the common approach in this field, carries the illusion that usability problems are automatically handled: tool developers will use their own experience as the basis for judging the usefulness of the tools they develop. (p. 118)

The laboratory study helped us understand the novice use of Papier-Mâché in a controlled setting. The results of this study demonstrate that even first-time users could build tangible interfaces and easily adapt applications to another technology. Testing with novice users provides a lot of usability information, such as the understandability of class names, the quality of documen-

tation, and where the system model is different than users' initial conceptual model (Norman, 1983).

We also examined how developers used Papier-Mâché in their own work. This technique has the opposite set of trade-offs from a laboratory study. The developers chose their own tasks, offering a realism and breadth unavailable in the laboratory. The time scale was much longer, ranging from 1 week to several months. However, it is difficult to directly compare results between projects precisely because they are all different.

A limitation of our fieldwork is that the researchers we interviewed were technology experts in their area. One of the goals of the Papier-Mâché architecture was enable a larger community to design physical interactions. The laboratory study and project use overcome this difficulty by evaluating a developer's first experience with programming (in the laboratory), and their longer use of the tool (in the developers' own applications).

5.1. Performance

On a dual Pentium III computer running Microsoft Windows® XP, the vision system runs at 5.0 frames per second without monitoring and 4.5 fps with monitoring, at a CPU load of 80%. With the vision system and two RFID readers, the performance is 3.0 fps. This performance is sufficient for applications where physical input serves as a hyperlink or other style of discrete token; here, a few hundred milliseconds latency is generally not an issue. Where tangible input provides a continuous, interactive control, the user experience would clearly benefit from increased performance. This benchmark data was collected in 2003; needless to say, contemporary hardware is significantly faster.

The vast majority of this computation time is in the image processing code. Although Papier-Mâché's code is reasonably optimal, Java is not a language known for its speed. The JAI architecture partially addresses the traditional performance limitations of Java: JAI is released as both a pure-Java cross-platform addition and with platform specific performance packs. At runtime, image manipulation operations use the native performance pack code if it is available, and use cross-platform code otherwise. Porting Papier-Mâché to Microsoft's C# language would retain the benefits of the Papier-Mâché architecture and programming using managed code (e.g., garbage collection and security), and gain a significant performance increase.

5.2. Lowering the Threshold: A Simple Application

In addition to measuring how rapidly applications built with the toolkit execute, it is important to measure how rapidly a developer can build an appli-

cation. A metric of the threshold for using a toolkit is the number of lines of code required by a basic application. The following Java code comprises the complete source for a simple application that graphically displays the objects found by the vision system. It is only four lines of code, three of which are constructor calls.

Have the vision system generate objects from camera input.

1. PhobProducer prod = new VisionPhobProducer (new CameraImageInput());

Create a factory that associates each object seen by the camera with a JPanel. The factory creates a JPanel for each object seen and adds the JPanel to the specified window.

2. AssociationFactory factory = new VisualAnalogueFactory(new PMacheWindow(prod, CALIBRATE), JPanel.class);

Create a binding manager that will receive events; this map contains the factory.

3. BindingManager bindingMgr = new AssociationMap(factory);

Attach the binding map to the camera, which will create, update, and remove JPanels according to what the camera sees.

4. prod.addPhobListener(bindingMgr);

This simple example illustrates that the threshold for creating an application with Papier-Mâché is quite low. Subsequent examples demonstrate the small code size for more involved applications.

5.3. In-Lab Evaluation

We conducted a controlled evaluation of Papier-Mâché to learn about the usefulness of our input abstractions, event layer, and monitoring window. Seven graduate students in our university's computer science department participated in the study. (We excluded HCI students because of potential conflicts of interest.) All participants had experience programming in Java.

We began each evaluation session by demonstrating an application associating RFID tags with audio clips, including an explanation of the monitoring window. We then asked the participant to read a seven-page user manual introducing the toolkit. Next, we gave participants a warm-up task and two full tasks. The evaluation was conducted in our lab on a 400 MHz dual Pentium ii running Windows XP with the Eclipse 2.1.1 IDE. We verbally answered questions about Java and Eclipse; for toolkit questions, we referred participants to the user manual and online documentation. We asked participants to "think aloud" about what they were doing, and we videotaped the sessions and saved participants' Java code for further review.

The warm-up task was to change an application that finds red objects so that it finds blue objects. The first full task was to change an In/Out board written using computer vision to use RFID tags instead. The second full task was to write an application that used RFID tags to control a slideshow. One

tag represented a directory of images; the two other tags represented *next* and *previous* operations.

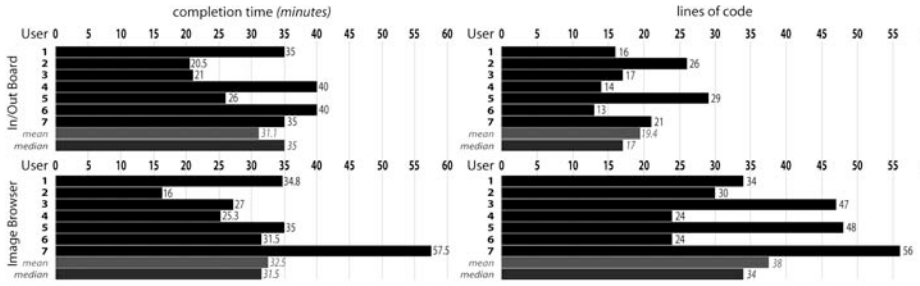
Every participant completed every task, though not without moments of difficulty. In our first task, participants converted an In/Out board from vision to RFID in a mean time of 31 minutes using a mean of 19 lines of code (see Figure 11). This shows that technology portability is quite achievable with Papier-Mâché.

Participants appreciated the ease with which input could be handled. In addition to their verbal enthusiasm, we noted that no one spent time looking up how to connect to hardware, how input was recognized, or how events were generated. In our second task, participants authored an RFID-based image browser in a mean time of 33 min using a mean of 38 lines of code. Note that participants on average wrote code twice as fast in the second task as in the first, indicating that they quickly became familiar with the toolkit. Two of the participants directly copied code; one said, "So this is like the marble answering machine [in the user manual]."

Ironically, the warm-up task—changing a colored-object finder from red to blue—proved the most challenging. The problem was that the classifier took a color parameter represented in the intensity-hue-saturation color space, highly effective for image analysis but not intuitive to most computer scientists, who are used to the RGB color space. Participants had difficulty even though we explained that the color space was intensity-hue-saturation, not RGB. Once a color in the proper color space was found, it took less than a minute to make the change. Ideally, these parameters should not be specified textually at all. These results inspired our investigation of visual authoring tools.

Overall, participants found the monitoring window to be very useful. For the warm-up task, they used it to understand the (confusing) color classifier. For the In/Out board task, they used the monitoring window to get informa-

Figure 11. **The task completion times and lines of code for the seven users in the Papier-Mâché laboratory study.**



tion about the attached RFID readers. When participants had errors in their code, they also used the monitoring window to verify that the input was not the source of these errors.

We also uncovered several usability issues. The most glaring was an inconsistency in naming related elements: the superclass was named *PhobGenerator*, a subclass *RFIDReader*, and the accessor method *getSource*. The term *generator* is also inconsistent with how similar classes in the Java library are named (Java uses the term *producer* for similar classes). We addressed these issues by renaming the abstract superclass *PhobProducer*, the subclass *RFIDPhobProducer*, and the accessor method *getProducer*(). Other points of confusion highlighted places where our documentation was insufficient. We have since addressed these usability issues by improving the API, documentation, and method names based on the feedback from this study.

5.4. Applications Using Papier-Mâché

A more open-ended, longitudinal evaluation of Papier-Mâché was conducted by observing its use in class and research projects at UC Berkeley. Between February 2003 and May 2004, nine groups of graduate and undergraduate students used Papier-Mâché for their class and research projects: two groups in a graduate HCI course in the spring of 2003, four groups in a graduate ubiquitous computing course in the fall of 2003, and three other groups in the 2003–2004 academic year.

Two groups in the spring 2003 offering of the graduate HCI class at UC Berkeley built projects using Papier-Mâché.

Physical Macros (De Guzman & Hsieh, 2003; see Figure 12) is a *topological* interface for programming macros, such as "actions" in Adobe Photoshop®. In this system, users compose physical function blocks that represent image editing functions. When examining their code, we found that presenting geo-referenced visual feedback was a substantial portion of the code. Reflecting on this, we realized that many of our inspiring applications, including The Designers' Outpost (Klemmer, Everitt, & Landay, 2008), also require this feature. For this reason, we introduced bindings where the location of physical input and electronic output could be coordinated.

SiteView (Beckmann & Dey, 2003; see Figure 13) is a *spatial* interface for controlling home automation systems. On a floor plan of a room, users create rules by manipulating physical icons representing conditions and actions. The system provides feedback about how rules will affect the environment by projecting photographs onto a vertical display. SiteView employs a ceiling-mounted camera to find the location and orientation of the thermostat and the light bulbs, and three RFID sensors for parameter input (weather, day of week, and time).

Figure 12. **The Physical Macros class project: a wall-scale, topological TUI. At left, a set of physical operation cards placed on the SMART Board; the resize operator is parameterized with an electronic slider. At top right, the image resulting from the operations. At bottom right, the set of physical operation cards available to the user.**

Figure 13. **SiteView, a *spatial* UI for end-user control of home automation systems. Left: A physical light-bulb icon on the floor plan, with projected feedback above. Right: The six physical icons.**

The thermostat is distinguished by size; the bulbs are distinguished by size and color. In general, the system worked well, but human hands were occasionally picked up. This inspired our addition of an event filter that removes objects in motion. With this in place, human hands do not interfere with rec-

ognition. SiteView is roughly 3,000 lines of Java code. SiteView's input code comprises about 30 lines of calls to Papier-Mâché. As a point of comparison, the Designers' Outpost was built with OpenCV and required several thousand lines of vision code to provide comparable functionality. We consider this substantial reduction in code to be a success of the API.

Four students in the fall 2003 offering of a graduate course on ubiquitous computing at UC Berkeley used Papier-Mâché for a 1-week miniproject. The goals of the miniprojects were tracking laser pointers, capturing Post-it notes on a whiteboard, invoking behaviors such as launching a Web browser or e-mail reader, and reading product barcodes. The quotes are drawn from the students' project reports.

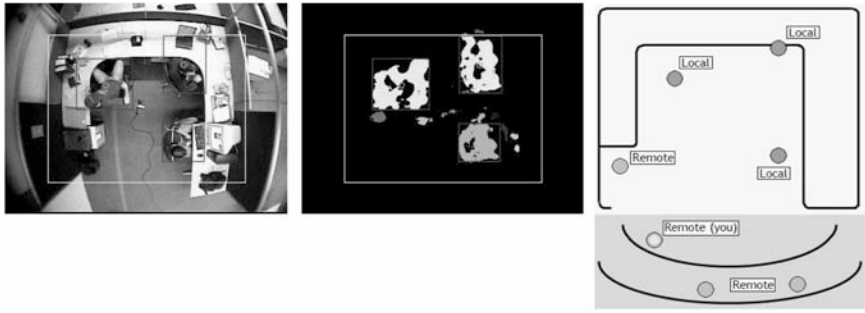
These programmers were impressed with the ease of writing an application using Papier-Mâché. One student was amazed that "it took only a single line of code to set up a working vision system!" Another student remarked, "Papier-Mâché had a clear, useful, and easy-to-understand API. The ease with which you could get a camera and basic object tracking set up was extremely nice." The students also extended the toolkit in compelling ways. One student's extension to the monitoring system played a tone whenever an object was recognized, mapping the size of the recognized object to the tone's pitch. This provided lightweight monitoring feedback to the recognition process.

These projects also unearthed some shortcomings of the Papier-Mâché library's current vision algorithms. For example, the system tended to lose track of an object and then immediately find it again, causing the undesired firing of *phobRemoved* and *phobAdded* events. One student observed that vision algorithms are inherently ambiguous and requested better ways of dealing with the ambiguity. The vision requirements for our inspiring applications and for the projects created here can be reliably handled by contemporary techniques. The challenge is that these techniques are more computationally intensive than the techniques currently included with the Papier-Mâché library, indicating that the Java language would probably not be appropriate. In addition, Papier-Mâché should offer a richer model of ambiguity and support techniques for mediating ambiguous input such as those introduced by Mankoff (Mankoff et al., 2000).

Three other Berkeley projects have used Papier-Mâché. The first is ObjectClassifierViews (De Guzman et al., 2003), which provides a set of graphical user interface dialogs that allow users to create classifiers and modify their parameters. This work inspired us to integrate their code into Papier-Mâché and to provide a mechanism for saving applications created visually.

The second is All Together Now (Lederer & Heer, 2004; see Figure 14), an awareness tool where the locations of individuals in a space are captured through computer vision and presented abstractly on a Web page. Remote in-

Figure 14. **ATN captures a bird's-eye video feed of the physical space (left), locates people using computer vision (middle), and displays local actors' positions (orange) in a virtual space (right) shared with remote actors (green). Non-participating remote actors are placed in an observation deck. Each remote actor's circle is marked with a yellow core in his personal view. (Picture on right is annotated for grayscale printers). Image from Lederer and Heer (2004).**



dividuals can "interact" with the local individuals by placing a marker of themselves on the space. Prior to All Together Now, the Papier-Mâché library only included edge detection, a stateless vision technique. The complexity of this scene and the low fidelity of the camera make stateless techniques impractical. Lederer and Heer implemented the background subtraction algorithm to overcome this. We incorporated their background subtraction code into the Papier-Mâché library. This experience showed us that it is possible for individuals interested in "getting under the hood" to change the vision algorithms used by Papier-Mâché and that its overall architecture is modular enough to easily accommodate new algorithms.

The last application that used Papier-Mâché is the Peripheral Display Toolkit (Matthews et al., 2004), a tool that lowers the threshold for developing peripheral displays—systems that provide ambient awareness of information (such as the fluctuation of a stock price). Peripheral Display Toolkit uses the image acquisition portion of Papier-Mâché as one of its input sources; it then abstracts this input and renders aspects of the input to an ambient display. All of our vision-based inspiring applications use continuous image processing. Peripheral Display Toolkit's needs are distinct in two ways: (a) It does not use the built-in processing, only the acquisition, as it does its own processing to find motion in images, and (b) it needs new images so sporadically that it is more appropriate to ask for them than to have them pushed at a regular interval. This use of Papier-Mâché demonstrates that the input acquisition and vision processing are sufficiently distinct that the former could be used without the latter. It also encouraged us to include the ability to request images, rather than enforcing an event-driven model.

5.5. Inspiring Applications Rewritten With Papier-Mâché

To understand Papier-Mâché's ability to build some of the inspiring applications, Jack Li and Andy Kung—then undergraduates working with the authors—reimplemented key aspects of three applications: the marble answering machine (Poynor, 1995), Books with Voices (Klemmer et al., 2003), and Collaborage (Moran et al., 1999).

Bishop's marble answering machine is an associative interface where physical marbles correspond to answering machine messages. The physical-to-digital associations in this application are straightforward for a developer to create with Papier-Mâché: excluding comments, white space, and imports, it comprises 18 lines of code. Note that this prototype does not communicate with the telephone system; it uses the audio system of a desktop PC. When an RFID tag is seen for the first time, the user records a message to it. Each subsequent time that a tag is seen, that recorded message is played back. Li also developed an alternate version that uses two readers: one reader designates recording, the other designates playback. This enables tag reuse. This marble answering machine implementation demonstrates a prototype that is more realistic than the original designer of the system was able to create.

Books with Voices links physical transcripts to the recorded interviews they were derived from. Jack Li implemented two alternate versions of this application: one uses RFID tags and the other uses barcodes. This simplified version of the application handles the user interaction but not the document creation software. Excluding comments, white space, and imports, the RFID version comprises 22 lines of code and the barcode version 30 lines. The difference between the two versions lies in the initialization. The barcode version uses barcodes detected in a camera image. The developer must specify in the initialization what camera they would like to use for input, and then connect that input to the *BarcodePhobProducer*. This minimal difference between versions of this application helps suggest that Papier-Mâché facilitates technology retargeting.

The Collaborage application connects physical documents on walls with database information. Andy Kung reimplemented a version of the Collaborage In/Out board, including connecting to a SQL database back-end. This example is the most complex of the three. It consists of three files: *Run.java* is the primary application file, comprising 146 lines of code; *Network.java* provides the connection to the SQL database, comprising 136 lines of code; *Log.java* prints a time-stamped log as elements are shifted between *In* and *Out*, comprising 81 lines of code. This example helps illustrate that Papier-Mâché can be used to build more complex applications, and that it can be integrated with other tools that are needed to build these applications.

6. CONCLUSIONS AND FUTURE WORK

This research demonstrated that an event-based software architecture employing high-level input abstractions can lower the threshold for tangible user interface development. This architecture also supports switching input technologies with minimal code changes. These architectural contributions are embodied in the Papier-Mâché toolkit, which makes debugging easier through monitoring facilities that include WOz control. An important benefit of this low-threshold, flexible architecture is that it opens development in this area to a wider community and enables rapid, iterative design.

6.1. Summary of Contributions

The Papier-Mâché toolkit introduced a novel software architecture that lowered the threshold for working with physical input in interaction design. This architecture provides high-level events for input technologies such as computer vision, and separates the acquisition of device input from the interpretation of device input. This modularity enables different members of an equivalent device class—such as cameras with different APIs—to use the same interpretation framework. It also enables vision developers to create different algorithms that perform the image interpretation task. The application receives information about this interpreted input through events. The event architecture is equivalent across all technologies, making it possible to rapidly explore different input technology alternatives. The binding manager contains *classifier/behavior* pairs and it is the recipient of these events. It invokes application behavior for each element the classifier matches. This manager facilitates multiplexed input, and, as with all elements of Papier-Mâché, it is instrumented such that its relevant information appears in the monitoring window. The monitoring window provides feedback about application behavior: input, how input is interpreted, and what behaviors are created.

The results of the multiple evaluation techniques suggest that Papier-Mâché provided a low-threshold architecture and enabled developers to rapidly switch input technologies. They also demonstrated that the monitoring facilities helped developers better understand application behavior. The longitudinal use of Papier-Mâché demonstrated its ability to support novel applications and that its modularity separating input acquisition, input interpretation, and application behavior enabled developers to incorporate and/or modify each of these independently. This longitudinal use also suggested that more sophisticated vision algorithms would lower recognition errors: We are hopeful that increased technology transfer from computer to HCI will aid this.

Second, this article introduced improved user-centered methods for the design and evaluation of software tools, including fieldwork with developers

as a basis for the design of software tools, to learn what software developers are really doing and what tool support would be beneficial. The fieldwork provided us with an important understanding of developers' successes, limitations, design practices, and requirements for tools supporting tangible interaction.

Last, this work demonstrated a mixed-methods approach to designing software tools, comprising controlled laboratory study, monitoring of longer term use in projects, and systems metrics. Each method provided different information about the usability of Papier-Mâché. For example, the laboratory study facilitates comparing results across participants and observing Papier-Mâché's use in others' work enabled us to understand longitudinal use of the toolkit in a wider variety of developer-selected applications. This application of multiple methods offers a much fuller picture of a system's usability.

6.2. Limitations

Papier-Mâché attempted to satisfy three disparate groups of users: those interested in very rapid interaction prototyping, those interested in more detailed interaction implementation, and those interested in computer vision algorithms. When architecture trade-offs forced us to privilege the needs of one of these groups, we chose those interested in interaction implementation. The results show that we were successful in this endeavor. Our work on visual programming shows that there is a space of applications that can be prototyped very rapidly through a visual UI. This UI has a very low threshold, but also a low ceiling. The difficulty with any prototyping tool that generates a different format than the programming language does is that there is a seam between the two. Visual programs written with Papier-Mâché cannot be easily extended with Java code. Although some environments support both visual authoring and scripting (such as Adobe Director®), there is much work to be done on more seamlessly integrating these two programming styles.

Although prior work has largely concentrated on aiding the technologists (e.g., Intel's OpenCV; Bradski, 2001), Papier-Mâché concentrates on supporting novel interaction design. This article provides tools that enable a wider range of developers to create tangible interfaces. In addition, the separation of concerns between vision development and interaction design in Papier-Mâché does indeed enable the two groups to work simultaneously. However, the trade-off is that Java is an awkward language for computer vision. For those solely interested in *prototyping vision algorithms*, Matlab may be a better choice, primarily because of its language-level support for computations involving arrays and matrices and because of its extensive mathematical libraries. For those interested in *writing production code*, native languages such

as C and C++ may be preferable because, although lacking in vision-appropriate language constructs, they are fast at runtime. A general open problem in ubiquitous computing software tools is that the component pieces of these heterogeneous applications are best individually served by different programming models. One solution is to offer each community its ideal programming model, but this has the drawback that an individual must alternate between many languages and tools to accomplish a single task or that applications can only be built by large groups. Balancing these design issues is an area of future research.

6.3. Future Work

We believe user-centered design of software tools to be a ripe area for future work, especially as computing moves beyond the desktop. In particular, ubiquitous computing tools would benefit from tools that provide integrated support for design, evaluation, and analysis. The SUEDE system for designing speech user interfaces (Klemmer et al., 2000) first introduced this integrated support. Ubicomp tools in this vein would support evaluation of both WOz and functioning systems. Such tools could also provide visualizations illustrating the user experience and system performance aspects of applications, and when recognition errors occur or when users have difficulty understanding the system (these could be flagged by the user or by an observer). Logging behavior over periods of extended behavior and visualizing that information is also an important area for future research.

Second, the heterogeneous technologies used in ubiquitous computing suggest research on improved methods for collaboration through design tools. Tools should aid conversations by affording designers some understanding of the technical constraints of a system and technologists an understanding of the user needs, without requiring that either be an expert in the other's domain. This is especially true for recognition-based technologies, where the perplexity of the system (a term used in speech UI design to describe grammar size) has an impact on recognition rate.

Last, the heterogeneity of ubicomp technologies resists easy encapsulation in a small component library. This stands in contrast to GUIs, where a standard set of widgets span nearly all common applications. Addressing this remains an opportunity for research; model-based design techniques may be of benefit here. Continued work on model-based techniques could aid designers in exploring applications of different form factors with radically different input and output technologies. This would benefit both designers' abilities to explore alternatives and work iteratively and their ability to create interfaces that can be customized for individual situations and user needs.

NOTES

Acknowledgments. We thank James Lin, Jack Li, and Andy Kung for their implementation assistance.

Authors' Present Addresses. Scott R. Klemmer, Stanford University, 353 Serra Mall, Stanford, CA 94305. E-mail: srk@cs.stanford.edu. James A. Landay, University of Washington, Computer Science & Engineering, 642 Paul G. Allen Center, Box 352350, Seattle, WA 98195. E-mail: landay@cs.washington.edu.

HCI Editorial Record. First manuscript received June 1, 2006. Revisions received March 5, 2007 and March 3, 2008. Accepted by Brad Myers.—*Editor*

REFERENCES

Back, M., Cohen, J., Gold, R., Harrison, S., & Minneman, S. (2001). Listen reader: An electronically augmented paper-based book. *CHI Letters, 3*(1), 23–29.

Ballagas, R., Ringel, M., Stone, M., & Borchers, J. (2003). iStuff: A physical user interface toolkit for ubiquitous computing environments. *CHI Letters, 5*(1), 537–544.

Ballagas, R., Szybalski, A., & Fox, A. (2004). The patch panel: Enabling control-flow interoperability in ubicomp environments. *Proceedings of Percom 2004: IEEE International Conference on Pervasive Computing and Communications.* Los Alamitos, CA: IEEE Computer Society.

Beck, K. (2000). *extreme programming eXplained: mbrace change.* Reading, MA: Addison-Wesley.

Beckmann, C., & Dey, A. K. (2003). SiteView: Tangibly programming active environments with predictive visualization. *Proceedings of Fifth International Conference on Ubiquitous Computing.* New York: Springer.

Bradski, G. (2001). *Open source computer vision library.* Retrieved from http://www.intel.com/research/mrl/research/opencv

Brooks, F. P. (1987). Essence and accidents of software engineering. *IEEE Computer, 20*(4), 10–19.

Burnett, M., Sheretov, A., Ren, B., & Rothermel, G. (2002). Testing homogeneous spreadsheet grids with the "What you see is what you test" methodology. *IEEE Transactions on Software Engineering, 28*(6), 576–594.

Canny, J. F. (1986). A computational approach to edge detection. *Transactions on Pattern Analysis and Machine Intelligence, 8*(6), 679–698.

Carter, S., Mankoff, J., Klemmer, S. R., & Matthews, T. (2008). Exiting the cleanroom: On ecological validity and ubiquitous computing. *Human–Computer Interaction, 23*(1), 47–99.

Chang, S.-K. (Ed.). (1990). *Principles of visual programming systems.* Englewood Cliffs, NJ: Prentice Hall.

Clarke, S. (2001). Evaluating a new programming language. *Proceedings of Workshop of the Psychology of Programming Interest Group.* Poole, UK: Bournemouth University.

Clarke, S. (2004). Measuring API usability. *Dr. Dobbs Journal, 29,* S6–S9.

De Guzman, E., & Hsieh, G. (2003). *Function composition in physical chaining applications.* Berkeley, CA: UC Berkeley.

De Guzman, E. S., Ramírez, A., & Klemmer, S. R. (2003). *ObjectClassifierViews: Support for visual programming of image classifiers.* Berkeley, CA: UC Berkeley.

Détienne, F. (2001). *Software design—Cognitive aspects* (F. Bott, Trans.). London: Springer Verlag.

Dey, A. K., Salber, D., & Abowd, G. D. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction, 16*(2–4), 97–166.

Fails, J. A., & Olsen, D. R. (2003). A design tool for camera-based interaction. *CHI Letters, 5*(1), 449–456.

Fishkin, K. P. (2004). A taxonomy for and analysis of tangible interfaces. *Personal and Ubiquitous Computing, 8*(5), 347–358.

Fitzmaurice, G. W., Ishii, H., & Buxton, W. (1995). Bricks: Laying the foundations for graspable user interfaces. *Proceedings of CHI: Human Factors in Computing Systems.* New York: Addison-Wesley.

Forsyth, D. A., & Ponce, J. (2003). *Computer vision: A modern approach.* Upper Saddle River, NJ: Prentice Hall.

Freeman, W. T., Miyake, Y., Tanaka, K.-i., Anderson, D. B., Beardsley, P. A., et al. (1998). Computer vision for interactive computer graphics. *IEEE Computer Graphics and Applications, 18*(3), 42–53.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software.* New York: Addison-Wesley.

Goldberg, A. (1977). *Smalltalk in the classroom* (No. SSL 77-2). Palo Alto, CA: Xerox Palo Alto Research Center.

Gorbet, M., Orth, M., & Ishii, H. (1998). Triangles: Tangible interface for manipulation and exploration of digital information topography. *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems.* New York: ACM Press.

Grasso, A., Karsenty, A., & Susani, M. (2000). Augmenting paper for community information sharing. *Proceedings of DARE 2000: Designing Augmented Reality Environments.* New York: ACM Press.

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments. *Journal of Visual Languages and Computing, 7*(2), 131–174.

Greenberg, S., & Fitchett, C. (2001). Phidgets: Easy development of physical interfaces through physical widgets. *CHI Letters, 3*(2), 209–218.

Heiner, J. M., Hudson, S. E., & Tanaka, K. (1999). Linking and messaging from real paper in the paper PDA. *CHI Letters, 1*(1), 179–186.

Holmquist, L. E., Redström, J., & Ljungstrand, P. (1999). Token-based access to digital information. *Proceedings of Handheld and Ubiquitous Computing. First International Symposium, HUC'99.* New York: Springer.

Hong, J. I., & Landay, J. A. (2000). SATIN: A toolkit for informal ink-based applications. *CHI Letters, 2*(2), 63–72.

Horn, B. K. P. (1986). Binary images: Topological properties. In *Robot vision* (pp. 65–89). Cambridge, MA: MIT Press.

Ishii, H., & Ullmer, B. (1997). Tangible bits: Towards seamless interfaces between people, bits and atoms. *Proceedings of CHI: Human Factors in Computing Systems.* New York: ACM Press.

Jacob, R., Ishii, H., Pangaro, G., & Patten, J. (2002). A tangible interface for organizing information using a grid. *CHI Letters, 4*(1), 339–346.

Johnson, W., Jellinek, H., Jr., L. K., Rao, R., & Card, S. (1993). Bridging the paper and electronic worlds: The paper user interface. *Proceedings of INTERCHI: Human Factors in Computing Systems.* New York: ACM Press.

Jones, S. P., Blackwell, A., & Burnett, M. (2003). A User-centred approach to functions in Excel. *Proceedings of SIGPLAN International Conference on Functional Programming.* New York: ACM Press.

Kato, H., Billinghurst, M., & Poupyrev, I. (2000). *ARToolKit.* Retrieved from http://www.hitl.washington.edu/artoolkit/

Kelley, J. F. (1984). An iterative design methodology for user-friendly natural language office information applications. *ACM Transactions on Office Information Systems, 2*(1), 26–41.

Klemmer, S. R., Everitt, K., & Landay, J. A. (2008). Integrating physical and digital interactions on walls for fluid design collaboration. *Human-Computer Interaction, 23*(2), 138–213.

Klemmer, S. R., Graham, J., Wolff, G. J., & Landay, J. A. (2003). Books with voices: Paper transcripts as a tangible interface to oral histories. *CHI Letters, 5*(1), 89–96.

Klemmer, S. R., Li, J., Lin, J., & Landay, J. A. (2004). Papier-Mâché: Toolkit support for tangible input. *CHI Letters, 6*(1), 399–406.

Klemmer, S. R., Sinha, A. K., Chen, J., Landay, J. A., Aboobaker, N., & Wang, A. (2000). SUEDE: A Wizard of Oz prototyping tool for speech user interfaces. *CHI Letters, 2*(2), 1–10.

Krasner, G. E., & Pope, S. T. (1988). A description of the model-view-controller user interface paradigm in the Smalltalk-80 system. *Object Oriented Programming, 1*(3), 26–49.

Landay, J. A., & Myers, B. A. (1993). Extending an existing user interface toolkit to support gesture recognition. *Proceedings of Adjunct Proceedings of INTERCHI '93: Human Factors in Computing Systems.* New York: ACM Press.

Lange, B. M., Jones, M. A., & Meyers, J. L. (1998). Insight lab: An immersive team environment linking paper, displays, and data. *Proceedings of CHI: Human Factors in Computing Systems.* New York: ACM Press.

Lederer, S., & Heer, J. (2004). All together now: Visualizing local and remote actors of localized activity. *Proceedings of Extended Abstracts of CHI: Conference on Human Factors in Computing Systems.* New York: ACM Press.

Liblit, B., Aiken, A., Zheng, A. X., & Jordan, M. I. (2003). Bug isolation via remote program sampling. *Proceedings of PLDI: SIGPLAN 2003 Conference on Programming Language Design and Implementation.* New York: Association for Computer Machinery.

MacIntyre, B., Gandy, M., Dow, S., & Bolter, J. D. (2004). DART: A toolkit for rapid design exploration of augmented reality experiences. *CHI Letters, 6*(2), 197–206.

Mackay, W. E., Fayard, A.-L., Frobert, L., & Médini, L. (1998). Reinventing the familiar: Exploring an augmented reality design space for air traffic control. *Proceedings of CHI: Human Factors in Computing Systems.* New York: ACM Press.

Mackay, W. E., & Pagani, D. (1994). Video mosaic: Laying out time in a physical space. *Proceedings of Multimedia, 2,* 165–172.

Mankoff, J., Hudson, S. E., & Abowd, G. D. (2000). Providing integrated toolkit-level support for ambiguity in recognition-based interfaces. *CHI Letters, 2*(1), 368–375.

Matthews, T., Dey, A. K., Mankoff, J., Carter, S., & Rattenbury, T. (2004). A toolkit for managing user attention in peripheral displays. *CHI Letters, 6*(2), 247–256.

McGee, D. R., Cohen, P. R., Wesson, R. M., & Horman, S. (2002). Comparing paper and tangible, multimodal tools. *CHI Letters, 4*(1), 407–414.

McGrath, J. E. (1994). Methodology matters: Doing research in the behavioral and social sciences. In R. M. Baecker, W. Buxton, J. Grudin, & S. Greenberg (Eds.), *Readings in human–computer interaction: Toward the year 2000*, 2nd edition. San Francisco: Morgan Kaufmann.

Moran, T. P., Saund, E., van Melle, W., Gujar, A., Fishkin, K. P., & Harrison, B. L. (1999). Design and technology for Collaborage: Collaborative collages of information on physical walls. *CHI Letters, 1*(1), 197–206.

Myers, B. A. (1990). A new model for handling input. *ACM Transactions on Information Systems, 8*(3), 289–320.

Myers, B., Hudson, S. E., & Pausch, R. (2000). Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction, 7*(1), 3–28.

Myers, B. A., McDaniel, R. G., Miller, R. C., Ferrency, A. S., Faulring, A., et al. (1997). The amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering, 23*(6), 347–365.

Myers, B. A., & Rosson, M. B. (1992). Survey on user interface programming. *Proceedings of CHI: Human Factors in Computing Systems.* New York: ACM Press.

Nelson, L., Ichimura, S., Pedersen, E. R., & Adams, L. (1999). Palette: A paper interface for giving presentations. *Proceedings of CHI: Human Factors in Computing Systems.* New York: ACM Press.

Norman, D. A. (1983). Some observations on mental models. In D. Gentner & A. L. Stevens (Eds.), *Human–computer interaction in the new millennium* (pp. 7–14). Hillsdale, NJ: Erlbaum.

Norman, D. (1990). *The design of everyday things.* New York: Doubleday.

Pane, J. (2002). *A Programming System for Children that is Designed for Usability.* Unpublished doctoral dissertation, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Poynor, R. (1995). The hand that rocks the cradle. *I.D., 42,* 60–65.

Rekimoto, J., & Ayatsuka, Y. (2000). CyberCode: Designing augmented reality environments with visual tags. *Proceedings of Designing Augmented Reality Environments (DARE 2000).* New York: ACM Press.

Rekimoto, J., & Saitoh, M. (1999). Augmented surfaces: a spatially continuous work space for hybrid computing environments. *Proceedings of CHI: Human Factors in Computing Systems.* New York: ACM Press.

Rekimoto, J., Ullmer, B., & Oba, H. (2001). DataTiles: A modular platform for mixed physical and graphical interactions. *CHI Letters, 3*(1), 269–276.

Rodden, K., & Blackwell, A. (2002). Class libraries: A challenge for programming usability research. *Proceedings of Workshop of the Psychology of Programming Interest Group* Isleworth, UK: Brunel University College.

Sellen, A. J., & Harper, R. (2001). *The myth of the paperless office.* Cambridge, MA: MIT Press.

Shaer, O., Leland, N., Calvillo-Gamez, E. H., & Jacob, R. J. K. (2004). The TAC paradigm: Specifying tangible user interfaces. *Personal and Ubiquitous Computing, 8*(5), 359–369.

Shneiderman, B. (1986). Empirical studies of programmers: The territory, paths, and destinations. *Proceedings of First Workshop on Empirical Studies of Programmers.* Norwood, NJ: Ablex.

Stifelman, L., Arons, B., & Schmandt, C. (2001). The audio notebook: Paper and pen interaction with structured speech. *CHI Letters, 3*(1), 182–189.

Szekely, P. (1996). Retrospective and challenges for model-based interface development. *Proceedings of Eurographics Workshop on Design, Specification and Verification of Interactive Systems.* New York: ACM Press.

Ullmer, B., & Ishii, H. (1997). The metaDESK: Models and prototypes for tangible user interfaces. *Proceedings of UIST: User Interface Software and Technology.* New York: ACM Press.

Ullmer, B., & Ishii, H. (2001). Emerging frameworks for tangible user interfaces. In J. M. Carroll (Ed.), *Human-computer interaction in the new millennium* (pp. 579–601). New York: Addison-Wesley.

Ullmer, B., Ishii, H., & Glas, D. (1998). mediaBlocks: Physical containers, transports, and controls for online media. *Proceedings of SIGGRAPH 98: 25th International Conference on Computer Graphics and Interactive Technique.* New York: ACM Press.

Underkoffler, J., & Ishii, H. (1999). Urp: A luminous-tangible workbench for urban planning and design. *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems.* New York: ACM Press.

Underkoffler, J., Ullmer, B., & Ishii, H. (1999). Emancipated pixels: Real-world graphics in the luminous room. *Proceedings of SIGGRAPH: Computer Graphics and Interactive Techniques*. New York: ACM Press.

University of Virginia. (1995). Alice: Rapid prototyping for virtual reality. *IEEE Computer Graphics and Applications, 15*(3), 8–11.

Want, R., Fishkin, K. P., Gujar, A., & Harrison, B. L. (1999). Bridging physical and virtual worlds with electronic tags. *Proceedings of CHI: Human Factors in Computing Systems.* New York: ACM Press.

Wellner, P. (1993). Interacting with paper on the DigitalDesk. *Communications of the ACM, 36,* 87–96.