Iterative Design and Evaluation of an Event Architecture for Pen-and-Paper Interfaces

Ron B. Yeh, Andreas Paepcke, Scott R. Klemmer
Stanford University HCI Group
Computer Science Department, Stanford, CA 94305
{ronyeh, paepcke, srk}@cs.stanford.edu

ABSTRACT
This paper explores architectural support for interfaces combining pen, paper, and PC. We show how the event-based approach common to GUIs can apply to augmented paper, and describe additions to address paper's distinguishing characteristics. To understand the developer experience of this architecture, we deployed the toolkit to 17 student teams for six weeks. Analysis of the developers' code provided insight into the appropriateness of events for paper UIs. The usage patterns we distilled informed a second iteration of the toolkit, which introduces techniques for integrating interactive and batched input handling, coordinating interactions across devices, and debugging paper applications. The study also revealed that programmers created gesture handlers by composing simple ink measurements. This desire for informal interactions inspired us to include abstractions for recognition. This work has implications beyond paper—designers of graphical tools can examine API usage to inform iterative toolkit development.

ACM Classification Keywords
D.2.2 [Software Engineering]: Design Tools and Techniques—*User interfaces*.
H.5.2. [Information Interfaces]: User Interfaces—*input devices and strategies; prototyping; user-centered design*.

Keywords
Toolkits, evaluation, augmented paper, device ensembles.

INTRODUCTION
Recent research has introduced techniques for combining pen-and-paper with interactive computing. These augmented interactions provide a fluid and flexible input interface for tasks such as documenting information in scientific research (*e.g.*, [26, 44]), sketching product designs, and composing music (see Figure 1). The primary attraction of designing augmented paper interactions is their embrace of existing practices, particularly in mobile, informal, and collaborative settings (*e.g.*, [17, 40]). To discover the best techniques to help developers create these systems, we can ask several questions. What aspects of graphical UI architectures can be adopted for creating paper applications? Which aspects of paper applications necessitate a departure from previous GUI design tools? Finally, what applications are developers interested in creating, and what do they do in practice? This paper addresses these questions; additionally, we hope our approach and findings will provide value to other areas of ubiquitous computing.

Figure 1. PaperToolkit provides an event-driven model, output to devices, and debugging techniques. Users have created tools for many tasks, including web design (*left*) and music composition (*right*). Going beyond the retrieval and form-filling tasks shown in prior paper + digital work, these apps explore real-time control of GUI elements and recognition of ink input.

In describing augmented paper interactions, it can be useful to delineate two approaches. The first builds digital interactivity on top of the drawing and writing tasks that users have traditionally engaged in with pen and paper. The other approach begins by regarding the pen as a command-specification device, exploring paper widgets, gestures, and asynchronously executed behaviors. In reality, most research and commercial systems draw from both of these approaches. Examples that draw more on the first approach include techniques for temporally coordinating multiple media—such as Audio Notebook's and LiveScribe's integration of written notes with captured audio [24, 37], Adapx's ruggedized system for capturing field notes [2], A-book's use of a PDA to help organize laboratory notes [26], and ButterflyNet's integration of field observations with photographs [44]. Work that exemplifies the second, command-centric approach, includes interactions such as taps of the pen (*e.g.*, to retrieve scientific citations [31]), gestures for editing printed documents (*e.g.*, [11, 22]), and gestures for creating and playing paper-based games [21].

This paper explores event-based architectures for paper + digital interactions, and describes PaperToolkit, a manifestation of the ideas. There are many enabling technologies for integrating paper and computation (*e.g.*, [9, 12, 15, 37, 40]). PaperToolkit, our implementation of the ideas in this paper, is built on top of Anoto's [3]; chosen for its reliability, mobility, high-resolution capture, and ability to distinguish pages. This technology employs a tiny camera, mounted inside the pen and pointed at the tip, to track pen motion across paper pre-printed with a dot-pattern. This vision-based tracking provides the location, force, and time of each stroke, either in real time (via Bluetooth) or in batched mode (via a wired dock). However, most aspects of PaperToolkit's architecture apply to alternate pen hardware.

This work offers three contributions: First, this research builds on prior augmented paper platforms [11, 35] that have abstracted development pragmatics such as producing Anoto-enhanced paper, acquiring pen data, and digitally rendering captured ink. PaperToolkit is similar to this prior work in its bookkeeping of the correspondence between *interactive* paper elements and their location in the Anoto coordinate space. However, PaperToolkit's architecture is more flexible than these prior systems, introducing techniques for integrated real-time and batched input handling, coordinated interactions across multiple devices, and rich debugging of augmented paper applications.

Second, this paper reports on the usage of PaperToolkit, and how it has evolved in response to our findings. We provided the toolkit to a semester-long undergraduate HCI class at another university. The class comprised 69 students in 17 groups, mostly computer science juniors and seniors. The usage patterns we present were distilled through discussions with students and a review of the final projects.

Third, the paper contributes a method for user-centered toolkit design through static source-code analysis. Our findings provided ideas for architecture and API revisions.

The paper is organized as follows. We first introduce the core PaperToolkit architecture and describe how applications are created with it. We then describe the user study, present findings about toolkit usage, and detail how we improved the toolkit in response to the results of the study. We then describe this paper's relationship with prior work, and close by suggesting opportunities for future research.

THE PAPERTOOLKIT ARCHITECTURE

PaperToolkit addresses the problem of creating, debugging, and deploying paper + digital applications. In these interfaces, one or more people use digital pens and paper to capture and organize information, and issue commands to a computer via pen gestures and paper widgets. Visual or audio feedback is presented to the user on a nearby PC or handheld device (see Figure 2). Alternatively, a user may work without a PC nearby; his pen input is batched for later processing. Paper interfaces come in many forms, including datasheets for scientists, notebooks for designers, and large maps and posters for engineers. The question is how developers program the input handling and feedback for the UI.

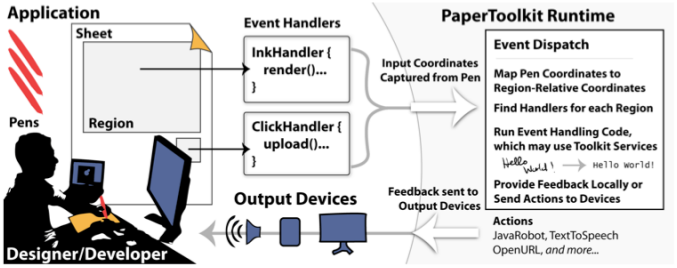

Figure 2. In PaperToolkit, input arrives from pens (*left*) and is sent to handlers (*middle*). Output is displayed on the local machine or routed to devices (*bottom*). Event dispatch and UI construction are modeled after GUI architectures, to help programmers create paper + digital applications rapidly.

PaperToolkit helps programmers accomplish this by providing methods to create paper forms, abstractions to handle multi-device events, and techniques to develop and debug faster. The abstractions are distributed across seven main concepts, which were developed based on iterative feedback from our developers, both internal and external to our lab. The concepts are summarized in the following table:

	Concept	Examples	Useful in Other Apps
Paper UI	Application Runtime and Event Handling	PaperToolkit, ClickHandler	○
	Paper UI Construction and Printing	Sheet, Region, Printer	◒
	Coordinating Ensemble Interactions	Device, OpenURL, LiveWhiteboard	●
Pen	Pen Data Capture, Manipulation, Rendering	Ink, InkRenderer, InkXMLParser	◒
	Ink Stroke Metrics and Recognition	InkUtils, HandwritingRecognizer	●
Tools	Development and Debugging Tools	SaveAndReplay, PaperUIDesigner	○
	Units, Coordinate Conversion, Utilities	Inches, CoordinateConverter	●

While all seven areas support paper applications, the ones marked by dark circles are also valuable in other domains. For example, a mobile application may use the *Device* architecture to send feedback from a phone to a PC.

Scenario: Designing a Paper-Based Blog

Karen is building a paper-based blogging system. A user writes blog entries with a digital pen, and taps a paper button to wirelessly transmit the entries to a handheld device, which uploads them to a web site.

On a PC, Karen writes a Java program to create a *Sheet* object, a large *Region* to capture the user's handwriting, and a small *Region* to act as the upload button (*e.g.*, the sheet in Figure 2). She adds two event handlers: an *InkHandler* to capture notes, and a *ClickHandler* to detect the pen tap. When Karen prints the paper UI, PaperToolkit augments the interface with the dot pattern. At runtime, the *InkHandler* receives the user's strokes from the pen's wireless connection. When the user taps the button, Karen's code retrieves the handwritten ink strokes, sends it through handwriting recognition, renders it to a JPEG, and uploads both the recognized text and the image of the handwriting to the user's blog. This programming approach (see Figure 3)

```
1  public class SimplePaperApp {
2     private static int numStrokes = 0;
3     public static void main(String[] args) {
4        Application app = new Application();
5        Sheet sheet = app.createSheet(8.5, 11);
6        Region inkReg = sheet.createRegion(1, 1, 4, 4);
7        Region tapReg = sheet.createRegion(1, 6, 1, 1);
8        Device remote = app.createRemoteDevice();
9        inkReg.addEventHandler(
10          new InkHandler() {
11             public void handleInkStroke(InkEvent e) {
12                numStrokes++;
13             }});
14       tapReg.addEventHandler(
15          new ClickAdapter() {
16             public void clicked(PenEvent e) {
17                remote.displayMessage(numStrokes);
18             }});
19       app.run(); // starts event dispatch loop
20    }
21 }
```

Figure 3. This PaperToolkit program, written in Java, includes all the core concepts. Every application is defined in terms of *Sheets*, *Regions*, and *Handlers*, which correspond to GUI Windows, Components, and Event Handlers. PaperToolkit also includes a *Device* abstraction, which helps programmers coordinate feedback across multiple computers. This program contains two regions, which will appear as rectangles when printed on a sheet of paper. When a user writes in the inking region, and taps the button region, the application sends feedback to a remote device. A GUI programmer would find this approach familiar (compare lines 14-18 with Java Swing). Additionally, inter-device interactions are abstracted (lines 8 & 17). The *Device* object handles message passing, without requiring a developer to program socket communications.

builds on the Java Swing [38] and Windows Forms [27] architectures. However, PaperToolkit distinguishes itself by providing API support for integrating paper tools into mobile and collaborative environments.

Implementation

The core event loop receives input from multiple pens. It translates hardware pen coordinates to page locations, and dispatches events to the corresponding handlers. To support easy debugging of pen interactions, all event data is sent over sockets as XML, and also logged to the file system.

This toolkit was developed over multiple iterations. It comprises 242 classes, and is implemented primarily with Java SE 6, with pieces built on other platforms. For example, batched pen input is handled through a .NET component, as Anoto synchronization is implemented in Windows code. For rendering to paper, PaperToolkit uses PS/PDF: paper UIs and dot patterns are rendered using the Java EPS [28] and iText PDF libraries [25]. Handwriting recognition uses Microsoft's Tablet PC recognizer, and the gesture handler uses Wobbrock's $1 recognizer [41].

Exploring the Design Space

Once we built a working version of the toolkit, we began to use it in our own lab to experiment with augmented paper. This process served two purposes. First, we could rapidly identify deficiencies in the toolkit, and improve its abstractions iteratively. Second, we gained an intuition about the applications and features that were interesting, yet difficult to implement. This section highlights some of the projects we implemented, and how they cover points in the paper-digital design space. For more details, see [42]. We explored three areas—tables, walls, and notebooks.

Collaborative Scenarios using Paper on Digital Tables

Through three projects (~19 Java classes each) involving a touch-sensitive digital table [8], we learned the importance of supporting *multiple users*, coordinating *multiple devices*, and translating between *multiple coordinate systems*.

We created a tabletop design environment [6] (see Figure 4, *right*) that used PaperToolkit to capture writing from multiple users, render ink to a canvas, and send drawings to a printer. A second project, by a visiting researcher, captured ink written on sticky notes [16]. It recognized handwriting from multiple users, and associated the annotations to a map displayed on the table. Cross-out gestures deleted annotations. These two projects supported multiple users, and recognition of handwriting and stylus gestures.

The third project, by a Masters student, explored fluid multi-device input [5]. For example, a user can set a pivot point with his pen and rotate/zoom a photo around that pivot using his other hand. PaperToolkit provided the handlers to locate the pen while writing, rotating, and zooming. From this project, we learned the importance of creating an abstract input device, to allow the toolkit to receive data from a digital pen, mouse, or fingertip. This device and its handlers help to coordinate transformations; while pen input is physically co-located with finger input, the pen and table each report coordinates that must be reconciled.

Remote Collaboration around Large Paper Displays

The second area explored large paper prints for collaboration. We created FeedReader, a wall poster that displayed articles in an RSS feed. Multiple users write in comments (captured by *InkHandlers*); the ink is transported to a web site where remote participants can view the discussion. *ClickHandler*s retrieved articles to a nearby display, and the *InkRenderer* created JPEGs for upload to the web. We also created BuddySketch, which supported video conferencing between scientists by providing real-time sharing of draw-

Figure 4. We built applications (such as the Twistr tabletop game, *left*) to explore the design space of paper interfaces. Our students have used the toolkit to produce research, such as a collaborative design environment integrating sketches on paper with manipulations on a digital table [6] (*right*). These applications support multiple users (*e.g.*, four pens in Twistr).

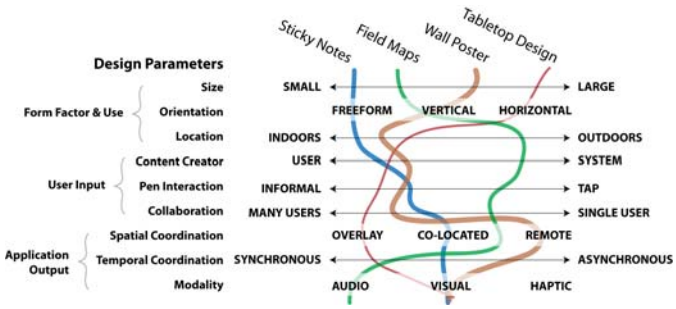

Figure 5. We show how four of the projects we built cover the design space of paper interactions. One area we did not explore was haptic feedback (*e.g.*, [23]). This design space borrows elements from prior taxonomies (*e.g.*, [10]).

ings. These projects taught us to coordinate between paper and devices (*e.g.*, triggering remote display of digital ink).

Mobile Field Notes and Maps

In the third area, we looked at how handheld maps could fit into a field biologist's ensemble. For example, users can search GPS-tagged field data with circle gestures, specifying the center and radius of the search in a single motion. The map comprises two Java classes, and communicates with a database over a Web API. We have also explored larger maps. For example, the Audio Guide allows scientists to retrieve audio annotations on a map while out in the field. A pen tap selects a region, and nearby audio notes are played to a wireless earpiece the user wears. These projects suggested a need for GPS support and gesture recognition.

Figure 5 summarizes the paper+digital design parameters we considered. While not all the categories are orthogonal (*e.g.*, outdoor use tends to imply smaller sizes), they illustrate the application areas we have encountered.

EVALUATION BY DEPLOYMENT AND CODE ANALYSIS

We conducted a study of the core toolkit elements (UI construction, pen input, and event handlers) with 69 programmers (17 teams) as part of an undergraduate HCI class at UC Berkeley (the authors did not teach the class). The students developed applications using PaperToolkit, beginning in the eighth week of the 14-week semester. The first author held in-person sessions to answer questions, receive feedback, and record observations of toolkit usage.

First, we observed that the toolkit's GUI-like approach worked well. It is notable that 17 teams of students with no experience in building paper interfaces were able to build working projects in six weeks. Because PaperToolkit built on established conventions, students who had programmed GUIs before could leverage existing patterns instead of learning new ones. Additionally, some students *learned GUI programming* as part of this introductory course (one team said that "none of us had developed event-driven programs prior to this"). For these students, the similarity to Java Swing meant that they did not have to learn two different programming models.

Second, we learned that developers desired event-based access to *both* real-time and batched pen data. PaperToolkit primarily targets synchronous pen interactions, so it dispatched events only when the pen was in real-time mode. Operating in *batched mode*—where data resides on the pen until it is uploaded—eases the deployment of mobile applications by eliminating the need for a wireless connection to a nearby PC or smartphone. For batched pen data, the toolkit mirrored Anoto's existing model by providing page and region-indexed access to ink data when a pen is placed in its docking station. In the end, ten teams built mobile applications, but only four teams explored asynchronous pen interactions (see Figure 6, bottom row). An ideal paper+digital toolkit would provide a unified event-based programming model for both real-time and batched input.

Third, we observed that many teams created applications by mashing-up interactive paper with the pre-built functionality in web services and desktop applications (see Figure 6). These projects included paper-based web design, personal organizers, and sharing tools for news and blogs. Teams integrated web apps into their projects by scraping HTML or using established APIs (*e.g.*, Flickr and Google Calendar). One group created a Firefox plug-in. From this, we learned that modern platforms should facilitate data-transfer with web services and desktop applications. PaperToolkit supports this goal by exporting *Ink* objects to web-friendly formats (*e.g.*, XML and PNG). We have since also explored support for GUI feedback through Adobe Flash.

Next, we review the insights we achieved by using source-code analysis to inform toolkit design. Examining code produced by developers offers an empirical account of usage patterns and provides insight into usability of the architecture and API. We manually reviewed the students' 304 source files (~35K statements or ~51K lines of commented code) and also used automatic scripts to calculate statistics (see Figure 7). We recorded observations for each file, and grouped recurring themes. The following sections

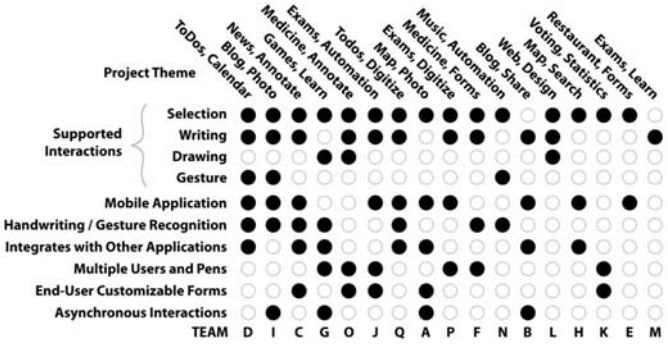

Figure 6. The 17 projects in this study (named A-Q) covered many themes. PaperToolkit supported *selection* (*e.g.*, check a box) and *writing* operations well. However, informal interactions requiring recognition were less common (*e.g.*, draw a musical note). We have since included better support for gesture recognition. Ten projects were mobile, and seven integrated with existing apps in mash-up fashion. Four supported batched input, where ink is processed after the user returns to his PC.

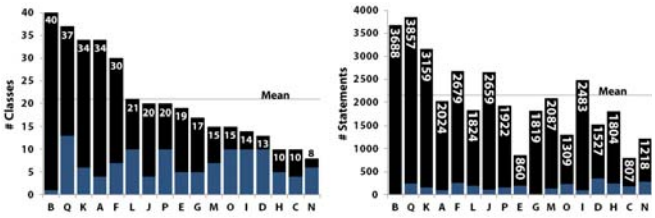

Figure 7. Student teams created substantial projects in six weeks, with the average project comprising 2102 statements and 21 Java classes. The size of the projects supports the external validity of our research insights. The light blue bars show the parts of projects directly interfacing with PaperToolkit.

present these themes, and introduce implications to inform the design of future pen-and-paper tools.

Composing Ink Operations for Gesture Recognition

The deployed toolkit provided handlers for directional marking gestures, but did not include a full recognition engine. Thus, teams added value to informal interactions by implementing their own gesture recognition. To understand how developers approached recognition, we gathered data on all *ink operations*. Figure 8 shows that only a handful of teams clustered or sorted ink strokes (to implement recognition via heuristics). This suggests that the value of recognition comes at a considerable cost of time and effort.

Four teams composed basic ink operations to recognize ink gestures. *Team D* detected when users crossed out handwritten text, and updated a web planner to reflect the completed task. *Team G* recognized paper-based games (*e.g.*, tic-tac-toe). *Team I* detected boxes users had drawn in a blog entry, and helped users import photos into those areas. *Team N* recognized handwritten music, including whole, half, quarter, and eighth notes, and translated the composition into MIDI. In the following snippets, we see that heuristics can be simple yet robust enough to rapidly prototype gesture recognition. In *Team I*'s project, the user inserts a picture into a blog entry by drawing a square with a single stroke. The team iterated through the page's strokes to find the one with the largest area:

```
getPointForPhotoInsertion() {
  for (ink : inkList) {
     for (strokes : ink.getStrokes()) {
        for (s : strokes) {
```

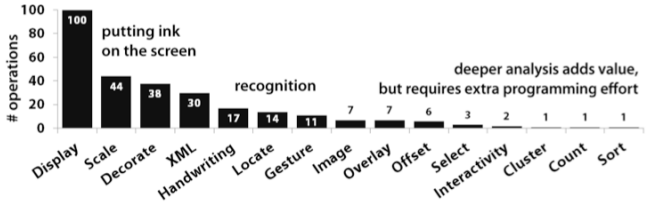

Figure 8. This shows the different ink operations we discovered (an *operation* may contain multiple statements). While *display* and *scale* were common, it took extra effort to recognize gestures (*e.g.*, by clustering strokes). Making it easier to explore these operations can raise the tail of this curve, adding extra value to informal pen-and-paper interactions.


```
if (s.getArea() > maxArea) {
   maxArea = s.getArea();
   boxInkStroke = s;
   boxInk = ink;
   ....
```

Later, they test the box against a minimum size threshold, and position the photo at its boundaries. Similarly, *Team N* applied heuristics to inkstrokes to determine note durations:

```
if (timeOfStroke <= shortTimeThreshold) {
   if (heightOfStroke <= shortHeightThreshold) {
      out.print("Whole note in Key of ");
   } else {
      out.print("1/2 note in Key of ");
   }
} else {
   if (firstY > lastY) { // if the note is going up
      if ((lastY - minY) > 5) { // detect flag
         out.print("1/8th note in Key of ");
      } else { // 1/4 note
      ....
```

This algorithm compares strokes to temporal and spatial thresholds, and detects their direction. An eighth note is recognized when the last ink samples are written in a direction opposite to the main stem (detecting the note's flag). These examples show how teams approached recognition by composing ink statistics. Heuristics are straightforward to specify in code and can be robust for simple interactions.

To make gesture recognition more approachable, we added a single-stroke gesture recognizer [41] and new methods to search and measure digital ink. PaperToolkit does not completely disregard heuristics-based recognition in favor of a full recognition engine. Heuristic can help developers rapidly prototype recognition for their applications. Heuristics can also be used to select a subset of ink input to be sent to a full recognizer, to enhance recognition rates. We plan to further improve support by providing techniques to *compose* ink operations, *select*/*cluster* strokes by time, size and location, and *visualize* recognition results.

Creating Interfaces by Example Modification

While examining the projects, we found evidence that developers would copy and paste code into their project, and then modify the skeletons to grow their interface around the working base. To find out *what* code developers copied, *how much* they would copy, and from *where*, we used a combination of methods. First, we used MOSS [34], a tool traditionally used to detect plagiarism in software, to detect similarities between projects and the toolkit. While MOSS worked for comparing projects fed to it, it was not suited for locating clones of code residing on the web. As a result, we manually identified clones in the corpus, noticing that unusual comments and identifiers were effective in signaling copied code. Once we found a clone candidate, we searched the web and the corpus to establish the source. This is the first work that uses static code analysis methods to study developer behavior and assess toolkit usability.

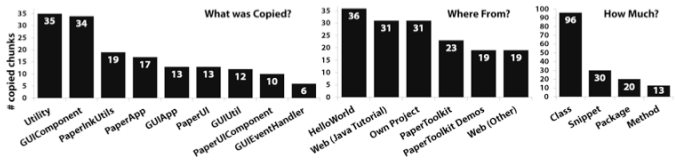

Figure 9. Teams used copy-and-paste to facilitate coding. Developers customized utility functions (*left*), and often copied code from their own projects (*middle*). Developers would copy an entire class file to get a program working, and then modify it (*right*). Programmers can benefit from tools that embrace this behavior (*e.g.*, by providing templates, facilitating refactoring of copied code, and tracking code lineage).

Developers copied code to reduce the boilerplate they need to write, and to solve problems they (or others) have solved before. Our data show that 41% of the 159 copied chunks directly supported GUIs, and 37% supported paper UIs (see Figure 9, *left*). The clones came from several sources, including a *Hello World* app they had worked through, the Web (*e.g.*, Java Swing tutorials), and PaperToolkit. In 96 of the 159 instances, developers copied an entire class file, and then modified the class to fit their application (*right*). These findings support earlier studies (*e.g.*, [18, 32]) that discovered that programmers copy blocks of method calls to save time. We find that developers copy boilerplate to construct UIs using unfamiliar APIs (to minimize errors). We suggest that tools embrace this approach of learning APIs by copy-and-paste. In this vein, we have integrated more example code into the documentation. We also made the entire toolkit available as open source, as opposed to sealed JAR files, to facilitate learning by example.

Usage Frequency Reveals Opportunities for Design

Toolkit usage statistics can reveal opportunities for improving the API. Figure 10 shows how many instances of each class were declared. The most used classes (*Inches*, *Region*, *Sheet*) are for UI construction. Since developers spend so much time working with these classes, we might spend time making this part of the API more clear and concise, to improve code readability and minimize bugs.

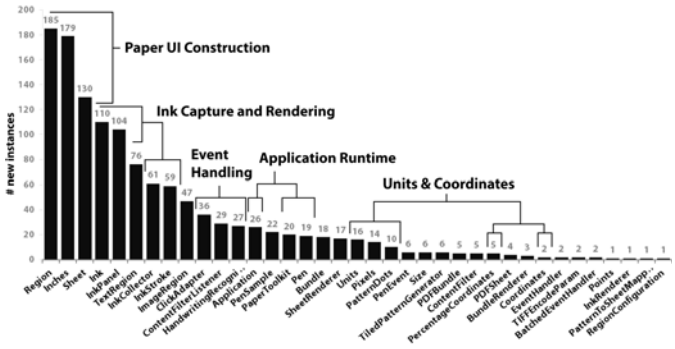

Figure 10. This shows how often teams used the toolkit classes. The heavily used classes are for constructing UIs (*e.g.*, *Region*). Classes for manipulating ink were popular (*e.g., InkPanel*). The seldom used classes were advanced features that were less well-documented. We can have impact if we improve the API of classes on the left. To raise the tail, we might add example code.

Examining the distribution's tail may reveal classes that were difficult to understand, hidden by lack of examples or documentation. For instance, *BatchedEventHandler* was only used twice, as the deployed toolkit touted real-time handling. We suggest that designers can improve toolkits by examining usage, and reflecting on why certain methods or classes are or are not used.

Using Debug Output to Understand Event Flow

We also noticed liberal use of debugging print statements. To discover where they helped most, we searched for all instances of debug output (as this study was conducted outside our lab, we did not log interactions with the IDE/debugger). The corpus contained 1232 statements containing "*println*." We examined them to see *where* the statements were located and *what* was printed in each one (see Figure 11). Overall, 39% of all debug statements were found inside event handlers: 333 were in GUI handlers, and 145 in PaperToolkit handlers. The top three printed values were *object instances* particular to the program, *error* messages, and *got here* messages serving to signal that a code block was reached. When coupled with the location of statements, we find that half of GUI handler *println*s and a third of PaperToolkit handler *println*s were *got here* statements. Sometimes, they tracked unimplemented handlers:

```
System.out.println("Zoom In");
// TODO Auto-generated Event stub actionPerformed()
```

As the handler was filled in, debug statements were then used to help developers visualize the program's (often hidden) behavior to help monitor for bugs. Print statements also helped programmers track how their application responded to pen input, so that they could iteratively debug their program "into existence" [32].

We suggest that tools can help developers understand *when* events are fired, and *what* the object values are at runtime. A *println* is lightweight compared to a breakpoint, as it can signal warnings at runtime without stopping execution. *Println*s are also manifest in the code, easily shared between teammates. If a tool can maintain this lightweight property and provide the benefits of a debugger, it can substantially speed up the development of applications.

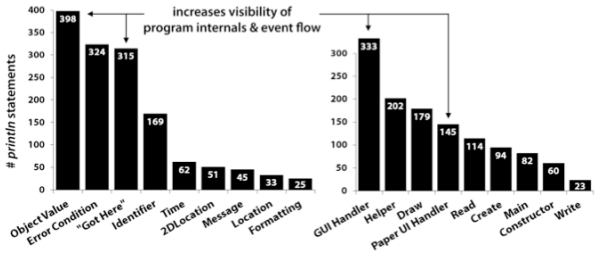

Figure 11. Code analysis revealed that most debugging statements (*println*s) helped programmers visualize *object value*s. Other statements reveal when the program encounters an error, or gets to a particular location. Developers place most of the *println*s in GUI and PaperToolkit handlers. This suggests a need for tools to help monitor objects and events at runtime.

Multiple Coordinate Systems and Representations

Finally, we observed the common task of converting ink strokes between different coordinate systems and semantic representations. With paper UIs, developers often deal with coordinate conversions (in GUIs, most work is done in screen space). For example, two projects used maps, and needed methods such as:

```
getGoogleMapTileCoordinate(lat, lon, zoomlevel)
getStreetIntersection(penX, penY, pageNum)
```

With maps, we have paper coordinates (*e.g.*, inches), the device's screen coordinates (pixels) and world coordinates (GPS). To improve PaperToolkit, we included GPS support, and provided a *CoordinateConverter* for generic mappings.

Conversions do not always produce numerical results. For example, `getStreetIntersection()` invokes a database query that returns names of streets in San Francisco. Likewise, the music project interpreted ink coordinates as locations on a staff, converting them into musical tones. Even in non-mapping projects, developers transform coordinates to accomplish tasks such as displaying ink (*e.g.*, *Team M*'s flashcards handle when users write upside down on a card).

DESIGN IMPLICATIONS AND TOOLKIT ITERATION

In this section, we highlight the main design implications, and describe how we applied them in a second iteration of PaperToolkit (see Figure 12). We also touch on implications for designing paper applications in general.

Feedback in Mobile and Collaborative Environments

Because paper is tangible, lightweight, and robust, paper applications lend themselves to mobile and ubiquitous computing scenarios. These situations often require developers to handle feedback across multiple users and devices (*e.g.*, [4]). To integrate paper into this device ensemble [33], the second iteration of PaperToolkit provides abstractions to help developers cope with three issues:

First, unlike graphical UIs, paper does not provide real-time visual feedback. To provide feedback to end users, programmers can use Java Swing on the computer running the program. PaperToolkit differs from prior paper + digital platforms as it also integrates with Adobe Flash, to provide added flexibility in the look-and-feel of the feedback.

Second, programmers need to be able to distinguish multiple users, as in Twistr, a game we created where players use pens to pick photos from a printed poster (see Figure 4). To enable this, PaperToolkit captures input from multiple pens, and provides a *penID* in *PenEvent*s (similar to [8, 13]).

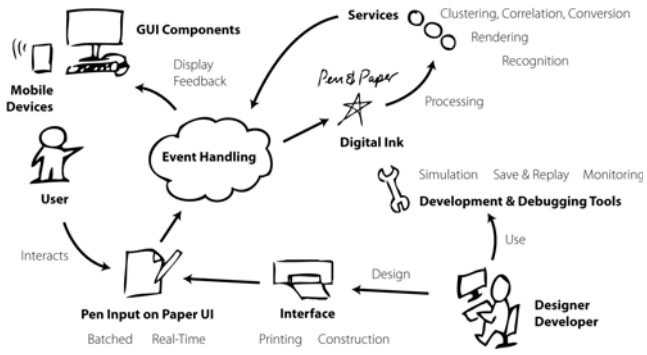

Figure 12. The concepts in the current iteration of PaperToolkit.

Third, mobile scenarios may involve multiple devices. For example, in BuddySketch (an application we built to provide shared paper sketching during video conferencing) the local computer asks its remote peer to refresh its ink display. This interaction is accomplished through mobile code [39], where each computer (a *Device*) invokes *Actions* (*e.g.*, *OpenURL*) on other *Devices*. For example, in Figure 3, lines 8 and 17 send feedback to a remote device. Network details are hidden. Behind the scenes, a call to the *Device* serializes an *Action* to XML and sends it to a remote device over TCP. The remote device (running a toolkit program which listens for *Actions*) reconstitutes the object and calls `invoke()`. PaperToolkit provides a set of common actions, and the option to pass arbitrary messages. This abstraction helps developers rapidly create ensemble interactions.

Unifying Real-time and Batched Event Handling

Paper applications can be used near or away from PCs, so input can arrive in real-time or in batch. Prior tool support allowed developers to access pen data *either* in real-time (via Bluetooth) [35], or in batched mode (when the user docks his pen) [11, 22]. Our experience shows that users can often benefit from real-time *and* batched interactions.

For example, a biologist using an augmented notebook [44] is frequently away from a PC. Here, a batched architecture enables the handwritten notes to be processed when the pen is docked at the field station. However, while working in the field, she may use a pen gesture to link a photograph she has just captured to a place in her notebook. When her camera recognizes this action, it provides *immediate* audio and visual feedback to acknowledge the linking gesture. In ButterflyNet, this was accomplished with two applications. The camera software recognized gestures in real-time, logging events to a file. The software that ran on the PC would take these timestamps and align them to the photos. While functional, this fractured implementation has negative impact on code readability and maintainability.

PaperToolkit helps developers co-locate event handling for batched and real-time events. To do this, it provides a flag in *PenEvent*s so that data received through the wireless connection can be distinguished from data received via the dock. In event handlers, developers check the `isRealTime` flag to provide appropriate feedback. To support this, when a pen is docked, PaperToolkit reads the data and injects events into the application's event stream. Besides the creation timestamp and the `isRealTime` flag, batched and real-time events behave equivalently. This architectural feature enables developers to present feedback appropriate to each situation, facilitates clear code organization, and provides developers a way to test in real time applications that actually target batched usage.

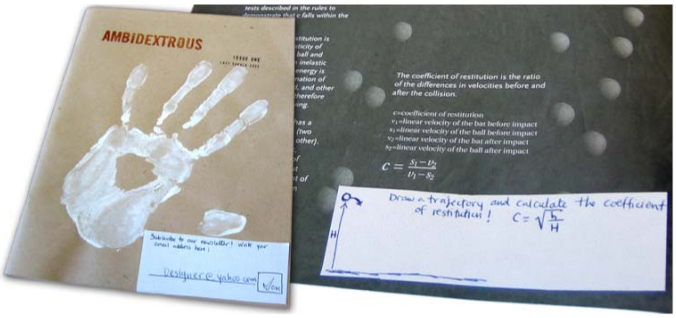

Figure 13. Designers can create functional prototypes by binding patterned pieces of paper to *Regions* containing event handlers, and then attaching them to physical objects such as this magazine. On the left, a reader can write his email to subscribe to a mailing list. On the right, he can draw a path to calculate a ball's elasticity. With simulation, designers can prototype interesting interactions in very little time.

Debugging with Save and Replay

When iterating designs, a developer is limited by the speed at which she can test. PaperToolkit introduces several techniques to help developers test applications. For example, since repeated testing is laborious, we introduced a toolkit feature to allow developers to save and replay user input. PaperToolkit accomplishes save and replay by logging all input at runtime. The developer can then select a log file and replay the saved input stream. When she selects a file to replay, the replay manager injects events into the *EventDispatcher*. The dispatcher relays the events to handlers, as if they had come from a physical pen. Save and replay minimizes the need to reproduce input sequences, and produces consistent results across trials. In our experience, it has become an indispensable feature.

Save and replay also enables developers to write code to handle pre-defined test input, similar to test-driven development. As an example, we return to Karen's blogging application. She first runs a skeleton version, where none of the handlers are defined. She uses the paper UI as if it were functional, composing a blog article, performing a gesture to insert a photo, and tapping a button to upload the entry. When she later fills in the event handlers, she can repeatedly replay her input to see the program gradually develop.

Simulating Printed UIs with Off-the-Shelf Notebooks

Our experience in building paper interfaces (*e.g.*, [43, 44]) taught us that the time required to physically print the interface became a bottleneck to testing. Additionally, students in our study reported that they seldom iterated their paper UIs, since printing slows the workflow. The toolkit now addresses this by delaying the printing step until it is absolutely necessary, such as when deploying to an end user. To minimize the need for printing, PaperToolkit provides a technique to test event handlers using a graphical simulator or pre-printed Anoto notebook.

The graphical simulator is accessed through the tray icon of the running application, and can be used with a mouse or Tablet PC stylus. Input is translated into *PenEvents* and sent to handlers. To test with an actual digital pen, the designer uses pre-printed digital paper. Through the tray menu, she selects a *Region* to test. The system prompts her to define that region by drawing a rectangle on a patterned page; this binding is saved for future test runs.

Simulation can speed up prototyping. For example, a designer can cut out the defined *Regions* and affix them to physical prototypes, such as a "smart magazine" (see Figure 13). This also provides a method for user testing early prototypes. Since input is saved, developers can gather user data before implementing any event handlers.

Enhancing Debugging with Visualizations

Visual tools can help programmers debug programs by exposing hidden aspects of the application. Previously, visualizations have helped people understand algorithms and data structures (*e.g.*, [36]). In our case, visualizations might help programmers understand event flow at runtime. To explore this, we created a working prototype that provides a visual log of user input and application output (see Figure 14). The top half of the interface displays a visual log of application input (ink strokes) and output (GUI feedback). The bottom half lists events that were fired, and objects sent to *println*. We hypothesize that this design can improve upon today's text-based logs; we leave a deeper exploration of this debugging approach to future work.

Interaction Language of Paper Interfaces

To add value to their applications, developers recognize informal input and gestures. However, this presents problems when a user wants to learn the UI. GUIs have built up a language of interaction through widgets, but paper UIs have not. Some patterns translate well (*e.g.*, paper buttons respond to pen taps). To specify a map location, however, a user might tap, circle, or draw an ×.

A related issue lies in enforcing constraints. In a GUI, developers can gray out a component when it is not appropriate, directing users toward an ordering of interactions. On paper, one can never prevent a user from checking a box or turning the page. Developers must account for this by embedding tutorials in the paper UI and by dealing with incomplete input.

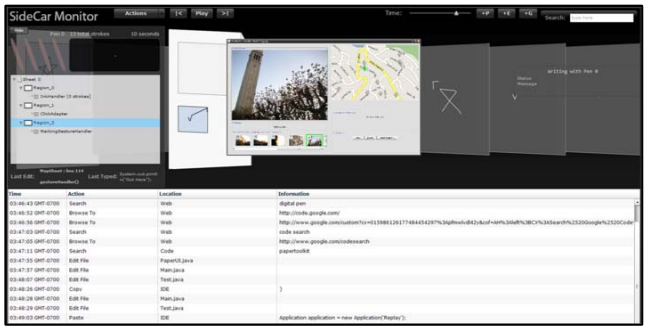

Figure 14. We built a visual monitor for debugging. During testing, the monitor logs all user input, system events, and application output along an interactive timeline. Unlike current practices of using *print* statements and debuggers, this approach does not require the developer to instrument her code.

RELATED WORK

This research builds on prior studies of software engineers' development and debugging practices, and earlier work in user interface architectures and design tools.

Supporting Existing Programming Practices

The literature on development practices shows that copy-and-paste and example modification are common techniques among software engineers. Rosson and Carroll studied four programmers and found that they benefited from having working examples they could modify to include in their own project [32]. Other studies indicate that programmers use copy-and-paste to reduce typing, and ensure that the easy-to-forget details (*e.g.*, method ordering) are correct. For example, Kim *et al.* studied expert programmers and found that copy-and-paste was used to save time when creating or calling similar methods [18]. Later, LaToza *et al.* found that modifying example code was one of *several* types of code duplication, which can cause problems when fixing bugs and refactoring [20]. Our results support these earlier findings, contributing from a much larger corpus. We also find that developers tend to copy from *their own* code (earlier projects, or places in the same project), probably because these snippets have a high probability of working, and are easy for them to understand. We also find that programmers copy when they need to learn a new API, such as PaperToolkit's.

The earlier studies have inspired a number of programming environments that map to the way programmers (and non-programmers) think about software [29]. For example, WhyLine [19] provided a timeline view for inspecting how events produced behaviors in an animation. PaperToolkit's replay was inspired by this timeline, which allowed programmers to scroll back to visualize hidden dependencies.

User Interface Software Architectures and Tools

Overall, we learned that the GUI model for programming largely works for programming paper UIs. PaperToolkit borrowed the ideas of components, layout, event handling, and extensibility from toolkits like Java Swing [38], Windows Forms [27], and SubArctic [14]. This work has also explored XML representations of the paper UI, inspired by the movement to better separate the view from event handling, as seen in [1, 30]. PaperToolkit extends beyond GUIs by including ensemble interactions across paper and digital devices and by supporting asynchronous event handling.

PaperToolkit also distinguishes itself from existing tools for *paper* interfaces. Anoto's SDK [3] enables developers to access pen data, but does not support event handling. Cohen *et al.*'s work [2, 7] combines input with speech commands. PADD integrates annotations on a physical document back into the digital one [11]. iPaper maps pen input to objects stored on the remote iServer [31, 35], providing media retrieval and event handling through its active components. PaperToolkit contributes by combining real-time and batched event handling into a single programming model, by foregrounding abstractions for mobility and recognition, and by supporting debugging with event replay. Compared with iPaper, PaperToolkit also differs by not requiring a database for media retrieval, and by providing *Actions* to simplify communication between devices. Our methodological contribution beyond the prior platforms is iteration of the abstractions—informed by a toolkit deployment, in-depth evaluation, and code analysis.

CONCLUSIONS AND FUTURE WORK

Through the deployment, evaluation, and improvement of the toolkit, we have learned that an event-driven approach provides a solid platform for building paper applications. Added support for multi-device communication, unifying batched and real-time event handling, ink processing, and rapid debugging helps to provide a low barrier for entering this space. Our results can have impact on tools outside of this domain. For example, we found it valuable to combine evidence from long-term use with static analysis of source code to inform toolkit design, and thus suggest that tool designers adopt these techniques. GUI platforms can also benefit from better abstractions for integrating web services and mobile devices, as consistent with today's trends.

However, there remain opportunities for research. First, we would like to involve non-programmers (*e.g.*, designers). One line of future work would be to provide tools to specify interactions by example. Second, since programmers frequently need to learn new libraries and toolkits, we will examine how visualizations can help developers understand program internals to speed development. Finally, in today's paper applications, if the user needs to update his UI, he must print out a new copy. In the future, we will support the scheduling of *automatic updates*, and treat the paper UI as a view (from MVC) with a very slow refresh rate.

PaperToolkit is open-source. The code and documentation are at http://hci.stanford.edu/paper. For more details, see [42]. Hopefully, our iterative approach for toolkit design can inspire and inform future design tools.

ACKNOWLEDGMENTS

We thank the CS160 students, Maneesh Agrawala, David Sun, and Gerald Yu for using PaperToolkit. Joel Brandt, Marcello Bastéa-Forte, and Jonas Boli contributed toolkit code; others provided feedback as users. Anonymous reviewers provided much insight. The work was supported by NSF IIS-0534662, and by hardware from Intel and Nokia.

REFERENCES

1 Abrams, M., C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster. UIML: An Appliance-Independent XML User Interface Language. In Proceedings of *The Eighth International World Wide Web Conference*, 1999.

2 Adapx, *Mapx and Penx*, 2007. http://www.adapx.com

3 Anoto AB, *Anoto Technology*, 2007. http://www.anoto.com

4 Ballagas, R., M. Ringel, M. Stone, and J. Borchers. iStuff: a physical user interface toolkit for ubiquitous computing environments. *CHI: ACM Conference on Human Factors in Computing Systems*. pp. 537–44, 2003.

5 Bastéa-Forte, M., R. B. Yeh, and S. R. Klemmer. Pointer: Multiple Collocated Display Inputs Suggests New Models for Program Design and Debugging. *UIST Extended Abstracts (Posters)*, 2007.

6 Bernstein, M., A. Robinson-Mosher, R. B. Yeh, and S. R. Klemmer. Diamond's Edge: From Notebook to Table and Back Again. *Ubicomp Extended Abstracts (Posters)*, 2006.

7 Cohen, P. R. and D. R. McGee. Tangible Multimodal Interfaces for Safety Critical Applications, *Communications of the ACM*, vol. 47(1): pp. 41–46, 2004.

8 Dietz, P. and D. Leigh. DiamondTouch: A Multi-User Touch Technology. *UIST: ACM Symposium on User Interface Software and Technology*. pp. 219–26, 2001.

9 EPOS, *EPOS Digital Pen*, 2007. http://www.epos-ps.com

10 Foley, J. D. and V. L. Wallace. The Art of Natural Graphic Man-Machine Conversation. *IEEE* 62(4). pp. 462–71, 1974.

11 Guimbretière, F. Paper Augmented Digital Documents. *UIST: ACM Symposium on User Interface Software and Technology*. pp. 51–60, 2003.

12 Heiner, J. M., S. E. Hudson, and K. Tanaka. Linking and Messaging from Real Paper in the Paper PDA. *UIST: ACM Symposium on User Interface Software and Technology*. pp. 179–86, 1999.

13 Hourcade, J. P. and B. B. Bederson, *Architecture and Implementation of a Java Package for Multiple Input Devices (MID)*. Technical Report, University of Maryland 1999. http://www.cs.umd.edu/hcil/mid

14 Hudson, S. E., J. Mankoff, and I. Smith. Extensible Input Handling in the subArctic Toolkit. *CHI: ACM Conference on Human Factors in Computing Systems*. pp. 381–90, 2005.

15 IBM Pen Technologies, *CrossPad and TransNote*, 2007. http://researchweb.watson.ibm.com/electricInk

16 Jiang, H., R. B. Yeh, T. Winograd, and Y. Shi. DigiPost: Writing on Post-its with Digital Pens to Support Collaborative Editing Tasks on Tabletop Displays. *UIST Posters*, 2007.

17 Johnson, W., H. Jellinek, L. K. Jr., R. Rao, and S. Card. Bridging the Paper and Electronic Worlds: The Paper User Interface. *CHI: ACM Conference on Human Factors in Computing Systems*. pp. 507–12, 1993.

18 Kim, M., L. Bergman, T. Lau, and D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. *International Symposium on Empirical Software Engineering*. pp. 83–92, 2004.

19 Ko, A. J. and B. A. Myers. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. *CHI: ACM Conference on Human Factors in Computing Systems*. pp. 151–58, 2004.

20 LaToza, T. D., G. Venolia, and R. DeLine. Maintaining Mental Models: A Study of Developer Work Habits. *International Conference on Software Engineering*. pp. 492–501, 2006.

21 LeapFrog Enterprises, *FLY Pentop Computer*, 2007. http://www.flypentop.com

22 Liao, C., F. Guimbretière, and K. Hinckley. PapierCraft: A Command System for Interactive Paper. *UIST: ACM Symposium on User Interface Software and Technology*. pp. 241–44, 2005.

23 Liao, C., F. Guimbretière, and C. E. Loeckenhoff. Pen-top Feedback for Paper-based Interfaces. *UIST: ACM Symposium on User Interface Software and Technology*. pp. 201–10, 2006.

24 Livescribe Inc., *Livescribe*, 2007. http://www.livescribe.com

25 Lowagie, B. and P. Soares, *iText Java-PDF Library*, 2007. http://www.lowagie.com/iText

26 Mackay, W. E., G. Pothier, C. Letondal, K. Bøegh, and H. E. Sørensen. The Missing Link: Augmenting Biology Laboratory Notebooks. *UIST: ACM Symposium on User Interface Software and Technology*. pp. 41–50, 2002.

27 Microsoft, *Windows Forms*, 2007. http://www.windowsforms.net

28 Mutton, P., *Java EPS Graphics2D*, 2007. http://www.jibble.org/epsgraphics

29 Myers, B. A., J. F. Pane, and A. Ko. Natural Programming Languages and Environments, *Communications of the ACM*, vol. 47(9): pp. 47–52, 2004.

30 Nichols, J., B. A. Myers, *et al.* Generating Remote Control Interfaces for Complex Appliances. *UIST: ACM Symposium on User Interface Software and Technology*. pp. 161–70, 2002.

31 Norrie, M. C., B. Signer, and N. Weibel. Print-n-Link: Weaving the Paper Web. *DocEng: ACM Symposium on Document Engineering*, 2006.

32 Rosson, M. B. and J. M. Carroll. The Reuse of Uses in Smalltalk Programming. *ACM Transactions on Computer-Human Interaction* 3(3). pp. 219–53, 1996.

33 Schilit, B. N. and U. Sengupta. Device Ensembles. *Computer* 37(12). pp. 56–64, 2004.

34 Schleimer, S., D. S. Wilkerson, and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. *SIGMOD: ACM International Conference on Management of Data*. pp. 76–85, 2003.

35 Signer, B., *Fundamental Concepts for Interactive Paper and Cross-Media Information Spaces*, Unpublished PhD, ETH Zurich, Computer Science, Zurich, 2006. http://people.inf.ethz.ch/signerb/publications/signer16218.pdf

36 Stasko, J., J. Domingue, M. H. Brown, and B. A. Price, *Software Visualization: Programming as a Multimedia Experience*: MIT Press. 550 pp. 1998.

37 Stifelman, L., B. Arons, and C. Schmandt. The Audio Notebook: Paper and Pen Interaction with Structured Speech. *CHI: ACM Conference on Human Factors in Computing Systems*. pp. 182–89, 2001.

38 Sun Microsystems, *Swing*, 2007. http://java.sun.com/javase/6/docs

39 Thorn, T. Programming Languages for Mobile Code, *ACM Computing Surveys (CSUR)*, vol. 29(3): pp. 213–39, 1997.

40 Wellner, P. Interacting With Paper on the DigitalDesk, *Communications of the ACM*, vol. 36(7): pp. 87–96, 1993.

41 Wobbrock, J. O., A. D. Wilson, and Y. Li. Gestures without Libraries, Toolkits or Training: A $1 Recognizer for User Interface Prototypes. *UIST: ACM Symposium on User Interface Software and Technology*, 2007.

42 Yeh, R. B., *Designing Interactions that Combine Pen, Paper, and Computer*, Unpublished PhD Thesis, Stanford University, Computer Science, Stanford, CA, 2008. http://hci.stanford.edu/publications/dissertations/yeh.pdf

43 Yeh, R. B., J. Brandt, J. Boli, and S. R. Klemmer. Interactive Gigapixel Prints: Large, Paper-based Interfaces for Visual Context and Collaboration. *Ubicomp Videos*, 2006.

44 Yeh, R. B., C. Liao, *et al.* ButterflyNet: A Mobile Capture and Access System for Field Biology Research. *CHI: ACM Conference on Human Factors in Computing Systems*. pp. 571–80, 2006.