Iterative Design of a Paper + Digital Toolkit: Supporting Designing, Developing, and Debugging

Ron B. Yeh, Scott R. Klemmer, Andreas Paepcke
Marcello Bastéa-Forte, Joel Brandt, Jonas Boli
Stanford University HCI Group, Computer Science Department
Stanford, CA 94305-9035, USA
[ronyeh, srk]@cs.stanford.edu

ABSTRACT

With advances in digital pens, there has been recent interest in supporting augmented paper in both research and commercial applications. This paper introduces the iterative design of a toolkit for event-driven programming of augmented paper applications. We evaluated the toolkit with 69 students (17 teams) in an external university class, gathering feedback through e-mail, in-person discussions, and analysis of 51,000 lines of source code produced by the teams. This paper describes successes and challenges we discovered in providing an event-driven architecture as the programming model for paper interaction. Informed by this evaluation, we extended the toolkit with visual tools for designing, developing, and debugging, thereby lowering the threshold for exploring paper UI designs, providing informal techniques for specifying UI layouts, and introducing visualizations for event handlers and programming interfaces. These results have implications beyond paper applications—R3 takes steps toward supporting programming by example modification, exploring APIs, and improved visualization of event flow.

ACM Classification Keywords

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

Keywords

Toolkits, augmented paper, design tools, device ensembles.

INTRODUCTION

This research addresses toolkit support for computationally-augmented pen-and-paper applications, as many domains—*e.g.*, biology [21, 42] and music (see Figure 1)—require a level of robustness, readability, and battery life not present in today's tablet computers. In recent years, the approach of integrating paper and digital interactions [13, 40] has received increasing attention due to advances in both commercial technology and interactions research. Commercial digital pens [2, 6] can now capture handwriting and send it to a computer wirelessly, in real-time. This synchronous interaction between pen, paper, and computer allows paper interfaces to control an application.

This toolkit research is motivated by the many projects that have demonstrated the potential of augmenting pen and paper; we highlight some here. For example, Audio Notebook is a paper interface that correlates handwritten notes with audio of lectures [36]. A-book enables biologists to augment their lab notebooks with a PDA that helps them create links to digital content [21]. PADD coordinates documents in the digital and physical worlds—handwritten physical annotations are overlaid on the source PDF [8]. PapierCraft enhances these digital documents, enabling users to edit them with pen gestures [20]. Print-n-Link detects citations in documents, and allows readers to retrieve them in a mobile setting [27]. Finally, ButterflyNet enables field biologists to find photos by navigating field notes [42]. In the commercial realm, the FLY pen can run applications from games to daily planners [19].

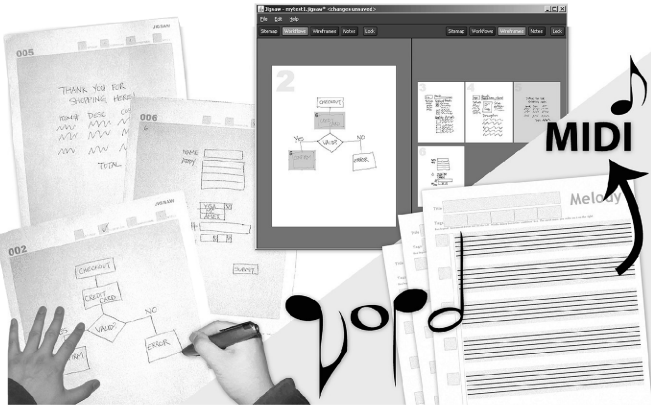

Figure 1. The R3 paper applications toolkit supports developers through an event-driven model, output to devices, and visualizations for design. With R3, developers have created tools for tasks such as web design (*left*) and music composition (*right*).

Currently, many applications leverage the Anoto pen platform [2], though other technologies are also available. For example, the EPOS pen works with unmodified paper, using ultrasound to determine the pen's location [6]. However, current infrastructure support for designing integrated paper and digital interactions requires considerable expertise. Anoto provides two tools: one for designing Forms (FDK) and another for processing pen data (SDK) [2]. The FDK augments paper with a tiny dot pattern, enabling digital pens to identify their location. The SDK provides access to the ink strokes after a user has docked his digital pen. It allows developers to render ink, but does not provide support for real-time interface events (*e.g.*, when a user crosses off a task on his paper calendar). Similarly, PADD integrates

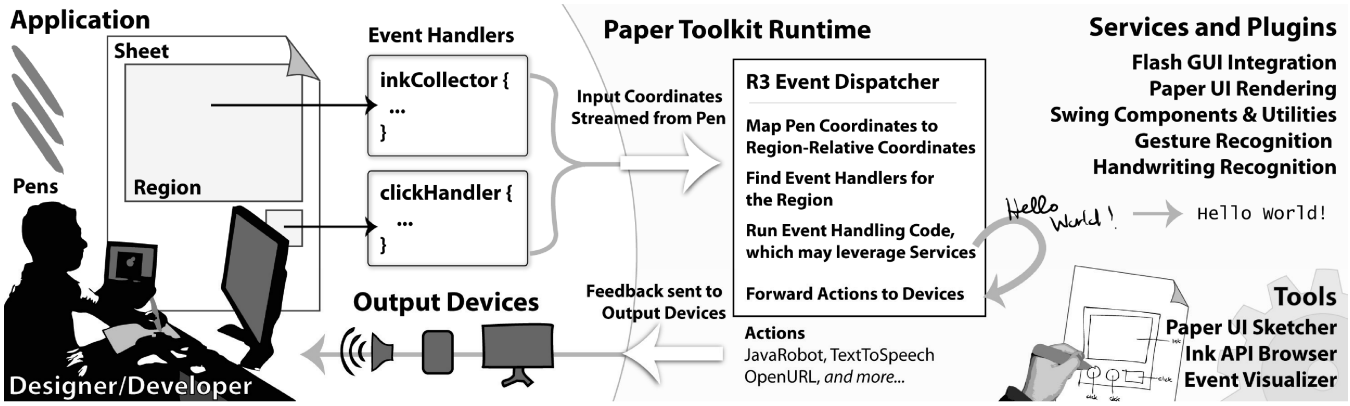

Figure 2. In this high-level overview of the R3 architecture, note how input arrives from pens and paper, and output is sent to devices. R3 lowers the threshold for processing real-time and batched pen input, and dispatches events to the handlers created by the developer. R3 services are flexibly coupled, so they can be used separately from the toolkit. On the lower right, we see the debugging tools that we implemented in response to user feedback and our analysis of the source code developers produced.

handwritten annotations back into digital documents [8], but does not support event handling. iPaper provides real-time retrieval of media associations [34], yet requires significant centralized infrastructure to handle requests.

We address this steep learning curve by taking a user-centered approach to designing a paper + digital toolkit. Specifically, this research demonstrates that providing tools to help designers explore different solutions, and iteratively improve upon them, can lower the *threshold* for creating applications [24] and provide *wider walls* for exploring the design space [33]. With effective tools, designers can better support mobile computing through digital paper.

This paper presents the R3 paper applications toolkit. Its goals are to *reduce* the time required for designers to create prototypes, to support the *reuse* of components through inspection and copying, and to help developers *recycle* and customize old solutions for new usage contexts. To support the large community of designers familiar with graphical interfaces, R3 introduces the GUI's event-driven and model-view-controller architectures [16] to augmented paper applications. Beyond lowering the learning threshold, R3 contributes methods to:

- Specify paper user interfaces and event handlers by sketching with an inking pen on physical paper.
- Integrate the development process with visualizations of the UI and event handlers to aid in debugging.
- Explore custom coding solutions through a direct manipulation API browser that generates source code.

The R3 toolkit enables designers to create rich paper-centric applications (see Figures 2). To do this, R3 builds upon the Anoto platform to support interactions with pen and paper. With Anoto, an inking pen reads a location-specifying dot pattern printed on a paper page. This enables R3 to receive the location, force, and time of each pen stroke either in *real time* (through Bluetooth) or in *batched mode* (after the user docks his pen).

To evaluate R3's architecture, we deployed the toolkit to an undergraduate HCI class of 69 students (17 teams). Through analysis of the students' source code, we found that R3 provided a low threshold for programmers to create paper user interfaces. Moreover, we found that developers:

- Used programming by example modification to speed their development with new frameworks (such as R3).
- Had cognitive difficulty in selecting and composing visual operations on digital ink.
- Depended on debugging output to iteratively grow their applications.

We also discovered limitations of the R3 approach, primarily concerning the speed at which a designer could explore and test designs. In response to these findings, we introduced visual design, development, and debugging tools to support rapid exploration. In the following section, we highlight R3's architectural features, and describe how a developer uses it to create pen-and-paper applications. We then describe our evaluation of the toolkit, through both internal use and external deployment. We detail observations from the deployment, and describe how we applied resulting design implications to a second iteration of R3. To conclude, we frame our contributions in related work and suggest future directions.

THE PAPER TOOLKIT ARCHITECTURE

In providing toolkit support for pen-and-paper interfaces, our goal was to *augment, and not replace*, developers' existing practices. For this reason, we modeled R3's architecture after event-driven GUI architectures, such as Windows Forms [22] and Java Swing [38]. R3 receives input from one or more digital pens, and invokes event handlers attached to *active regions* on a paper interface (see Figure 2). This approach, which draws on traditional GUI idioms, eases the development transition between graphical and augmented paper interaction by providing existing developers with a familiar programming model. On the whole, this model is effective for paper + digital interfaces.

However, the GUI architectural paradigm cannot be copied wholesale to this domain because paper cannot itself present real-time graphical feedback; we introduce the challenges to designing augmented paper interaction and describe how R3 addresses them.

Designing a Paper Interface

Consider this scenario: Karen would like to design a task management application that will allow users to jot notes on paper. After writing a note, the user will tap a dedicated rectangle at the bottom of the paper with his pen. Through a Bluetooth connection, a nearby computer will capture the strokes and add the note to the user's digital calendar.

R3's development process proceeds as follows: on a PC, Karen uses R3's interface layout tool to create a *Sheet* object (analogous to a Swing *JFrame*). Then, she creates one large *Region* to capture the user's handwriting, and a small *Region* to act as an upload button (see Figure 2, left). She adds two *Event Handlers*: an ink collector to the large region, and a click handler to the small region. Karen then prints the paper UI; R3 automatically renders Anoto pattern on the active regions. When active, the ink collector receives the user's strokes from the wireless connection. When the user taps on the paper button, Karen's code retrieves ink from the ink collector (optionally passing it through handwriting recognition), renders it as a JPEG image file, and uploads it to the user's web calendar.

However, suppose that Jim, a graphic designer who does not program, collaborates with Karen. In this case, Karen offloads the design of the paper UI to Jim. R3 allows Jim to use any graphical tool to design the interface's look-and-feel. Jim and Karen can thus work in parallel. Jim can design the art in Adobe Illustrator, export to PDF, and use R3's direct manipulation paper interface builder to *add* and *name* interactive regions. Meanwhile, Karen can create the back-end code. When Jim has finished, he provides Karen with his paper UI specification. Karen's program reads in the paper UI and attaches handlers to the named regions.

The R3 library provides many pen event handlers, including *click detectors*, *marking gesture interpreters*, and *handwriting recognizers* (which use the recognition service, seen in Figure 2). The flexibility of R3's event architecture allows developers to create their own handlers. When Karen and Jim print the paper interface, R3's print subsystem automatically instruments regions containing handlers with the Anoto dot pattern. Lastly, with the printed UI in hand, they can test their application immediately.

Overall, R3's approach of separating interface design from implementation augments existing practices, as graphic artists can use familiar tools such as Adobe Illustrator to create the visual design of the paper UI. The paper UI development process is also flexible, as the developer can either 1) start from a PDF, 2) generate the UI directly through the R3 Java API, or 3) use R3's support for model-based paper UIs, which separate an XML interface specification from the Java-based application logic. This approach of using XML as an interface representation was seen in [1, 26], and is used in, *e.g.*, Mozilla's XUL [23].

Event Handling with Multiple Pens

The R3 architecture distinguishes itself from traditional GUI architectures in two ways. First, analogously to research on toolkits for multiple mice (*e.g.*, [10]), R3 handlers can receive input from multiple pens, determined through a *PenID* in the *PenEvent*. Applications that leverage this multiple-pen functionality include Diamond's Edge, a collaborative drawing environment, and the Twistr game, a two-player, bimanual interface where players use pens to tap photos from a set on a large paper print (see Figure 3). In Twistr, a single *pressed* event is invoked when any of the four pens is depressed. The handler requests the *PenID*, and rewards the appropriate player for finding his photo.

Providing Feedback through Output Devices

The second way that R3 differs from the traditional GUI architecture is that interaction happens within a device ensemble [31], where user actions are distributed across paper and computer (see *e.g.*, [42]). To support this interaction style, R3 provides feedback by invoking *Actions* on devices, such as a handheld display. These ensemble interactions are accomplished through a *mobile code* approach [39], passing Java as XML across a network. The programmer instantiates a *Device* with a remote IP address. The program can then ask the remote *Device* to invoke an *Action* (*e.g.*, *OpenURL*). Other than the computer's host name, R3 abstracts network details from the developer.

With this approach, a pen-and-paper program can provide real-time interaction across multiple devices. For example, in BuddySketch—an application we built to provide shared sketching in video conferencing—each computer is a *Device*, and in response to input on paper, one computer asks its remote peer to update ink or display photographs.

Debugging and Testing with Event Save and Replay

To assist developers in debugging applications, R3 logs

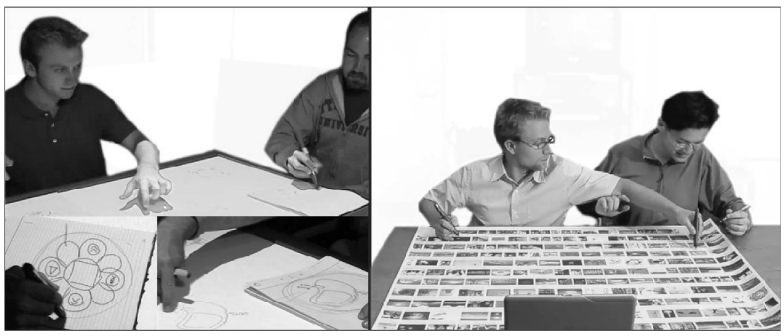

Figure 3. Left) Students used R3 to produce research such as Diamond's Edge [3], a collaborative drawing environment integrating sketching on paper with manipulations on a digital table. Right) The authors used R3 to explore large paper interfaces, including this Twistr game, which recognizes four pens.

every *PenEvent*. A developer can replay this logged input stream using the R3 GUI. Saving and replaying user input offers four benefits. First, logged input can be valuable in automated unit testing of graphical user interfaces, traditionally a weakness in testing frameworks. Second, debugging with logged input is preferable because working with the same input produces consistent results across trials. Third, debugging with logged input is more efficient, as it eliminates the need to physically reproduce the input on every occasion. Lastly, this architectural feature is useful for saving and later reviewing tests with end users.

Implementation

The R3 toolkit is primarily implemented with Java SE 6.0, with smaller components providing services to the main toolkit. Pen input from the Anoto SDK is handled through a Microsoft .NET 2.0 component, since the drivers are provided as Windows DLLs. As the "user interface" created by R3 is physical paper, there needs to be a printer-friendly format for these interfaces. To provide a system that is widespread, flexible, and low threshold, we chose PDF: R3's paper interface builder is implemented as an Adobe Acrobat plug-in; the interface can be augmented, and the dot patterns rendered, using the iText PDF library. Handwriting recognition is built on Microsoft's Tablet PC recognizer.

DETERMINING NEEDS THROUGH LONG-TERM USE

We employed a mixed-methods approach to designing and evaluating R3. Our evaluation comprised three methods: building applications ourselves, observing its use in a class, and analyzing the source code developers produced. Each approach highlights distinct considerations; for example, in-depth analysis of the code can help to improve the toolkit at the API and architecture levels, whereas anecdotes from long-term usage may inform the design process as a whole. Over the last ten months, the architectural features of R3 have evolved based on our desire to address three goals:

Learnability How low is the threshold for learning to create useful paper interfaces? Which aspects of R3 contributed to lowering this threshold, and which were bottlenecks to further lowering it?

Extensibility What is the ceiling on the complexity of applications that experts can create? Which R3 aspects contribute to this, and which prevent a higher ceiling?

Explorability Will designers create a large variety of applications, utilizing a large variety of input techniques? How can R3 better support the ability for designers to rapidly create and test ideas?

Feedback from Internal Use

Two students used an early version of R3 to develop Diamond's Edge (see Figure 3), a drawing environment that integrates paper with digital tables [3]. This project comprised only 20 source files, leveraging R3 for capturing input from multiple pens, rendering digital ink on a canvas, and sending drawings to printers. We also used R3 to support our own research on large paper surfaces (GIGAprints) [41]. Diamond's Edge and GIGAprints were presented as a poster and a video at Ubicomp 2006.

One genre of concern that these projects highlighted was the need to support flexible input. To accomplish this, we abstracted the input architecture, creating a *Pen* interface that enables simultaneous input from multiple physical devices, and enables developers to implement their own subclass for the input technology of their choice. We also found that in multiple projects, developers needed to package incoming pen samples and interpret them as higher-level user actions. While Anoto tools provide direct access to **x** and **y** coordinates, our toolkit collects these into abstractions like clicks, gestures, and freeform ink.

Observations of an External Deployment

While longitudinal use by experts (the authors and colleagues) offered insight on the ceiling of the platform's flexibility and extensibility, use by novices (students at another university) helped us understand R3's accessibility. To observe on-the-ground use of the principles manifest in the R3 toolkit, we provided it to an undergraduate HCI class at an external university (the authors were not part of the teaching staff). In this class, 69 students (17 teams) designed and built pen-and-paper projects. Students began using R3 in the eighth week of the fourteen-week class, after they had tested their early-stage paper prototypes [29]. We summarize these projects in Figure 4. Project topics were varied, including paper-based web design, personal organizers, and sharing tools for news and blogs. Of the 17 paper UIs created by these teams, 16 allowed the *selection* of buttons or areas on the page. Only three accepted pen gesture as input. Most applications were mobile (10 of 17), and four supported batched input.

During the deployment, the first author held two in-person sessions at the university to answer questions and receive feedback. He also responded to postings on the R3 newsgroup. In total, this comprised more than 20 hours of providing support and gathering feedback. After the semes-

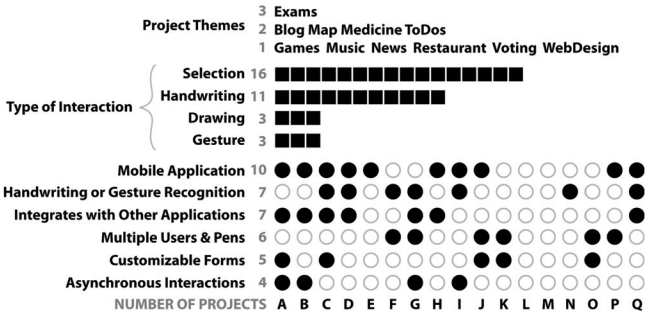

Figure 4. Summary of 17 class projects developed with R3. We see that R3 supported a variety of projects (though three dealt with university exams). Notice that while *selection* interactions were common (*e.g.*, check a box), advanced interactions such as gestures were rare (*e.g.*, draw a musical note). Notably, only four of the projects implemented asynchronous interactions, where ink and actions are batch processed once the user returns to their desktop.

ter, we analyzed the project materials, including reports and source code, to evaluate the successes and limitations of R3. This work demonstrates that in-depth analysis of the *products* of a toolkit can be used to inform the design of the toolkit itself.

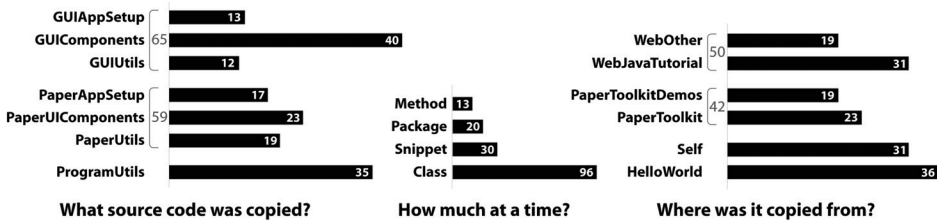

Figure 5. Teams used copy-and-paste to facilitate coding. Many times, developers would copy a *class file* needed to get a program working, and then customize the skeleton to address their new needs. Developers can benefit from tools that support this *coding-by-growing* behavior.

Out of our analysis of the team materials, our notes from providing support, and analysis of 50,962 lines of code, we identified three areas as opportunities for improved support: better *debugging* infrastructure, integrating *batched and real-time* interactions, and support for *web application* platforms. While streaming support worked well during development, operating the digital pens in *batched* mode—where data resides on the pen until it is synchronized through a cable—can ease deployment of mobile applications, as it eliminates the need for a PC within wireless range at runtime. Through the students' written reports, it became clear that tools should treat batched mode and streaming mode more *interchangeably*. This suggests that, as user interface tools support a broader spectrum of input technologies, the abstraction goals put forth through UIMS and model-based interface research [24] are likely to play an increasingly important role.

Consistent with current trends, six teams integrated web applications into their projects, from "scraping" HTML to working with established APIs (*e.g.*, Flickr and Google Calendar). Consequently, students asked that R3 provide stronger support for these kinds of applications. For example, one group wanted to integrate their application with the Apache Tomcat servlet container. A second group wanted to create a Firefox plug-in. As applications move online, toolkit support is most effective when it not only provides strong intra-application support, but support for integrating external services.

Overall, R3 was a big success. 17 teams with *no prior experience* in building paper interfaces (many without GUI programming experience) were able to build working projects using R3 in less than six weeks.

INFORMING DESIGN WITH SOURCE CODE ANALYSIS

We now describe how we used source code analysis as an evaluation method to help us assess R3's usability. Examining the source code produced by developers offers an empirical account of usage patterns and gives insight into usability successes and limitations of the API. We reviewed the 304 source files by hand; these files comprised ~35,000 statements and ~51,000 lines of code, including comments. We recorded observations for each file, with special emphasis on the paper related code. Throughout the code review, we noticed three recurring themes—*coding-by-example modification*, *customizing tool support*, and *iterative debugging*. In the next sections, we highlight the patterns, provide evidence, and introduce designs that we added to enhance these practices.

Programming by Example Modification

Our first observation was that developers would copy chunks of source code, paste it into their project, and then grow their application around this working base. This finding is consistent with earlier studies (*e.g.*, [14, 30]). In our analysis, we wanted to identify *what* types of code developers copied, *how much* they would copy at one time, and from *where* they would copy from the code.

We used a combination of static analysis methods to detect the code clones. First, we used MOSS, a tool traditionally used to detect plagiarism in software [32], to detect similarities between the student projects and the toolkit. Since MOSS could not work with code residing on the web, we also reviewed the corpus by hand to identify potential clones. We found that looking for unusual comments and method names was effective in identifying copied code. Once we identified a candidate, we would perform a text search over the entire corpus, and a web search, to discover the source of the copied code. To our knowledge, this paper presents the first work that uses static code analysis to study developers' copy-and-paste behavior for the purpose of assessing the usability of a toolkit.

We found that the frequency of copying was independent of whether the code was supporting the *paper* or the *GUI* parts of the interaction (see Figure 5). The data shows that 41% of the 159 copied pieces of functionality supported the GUI, and 37% supported the paper. Of the instances we discovered, developers most often (96 of 159) copied one class file (rather than whole packages, single methods, or snippets) and then modified the class to fit their application. Developers copied from several sources, including their own "Hello World" assignment, the R3 toolkit, and the Web (*e.g.*, the Java Swing tutorials).

In addition to studying *what* was copied, and *how much* was copied at one time, we also analyzed *where* code was copied from. We found that the single most common source of copied code was the paper application template provided for use in the students' first assignment. Prior research (*e.g.*, [14, 30]) found that developers use copy-and-paste to save *time* during development. In addition to efficiency, we find that developers use this technique to cope with learning APIs (to reduce errors using the unfamiliar framework).

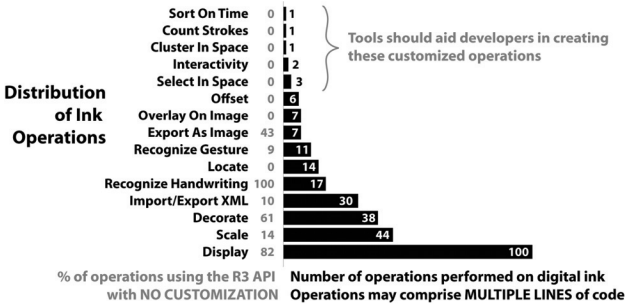

Figure 6. While the *display* and *scale* operations were commonly customized, it took more effort to create application-specific solutions for calculating metrics for and recognizing strokes. Making it easier to explore these opportunities may lower the threshold and *pull up* the tail of this curve.

Since the developers only had a few weeks to learn the R3 architecture, copy-and-paste was a natural strategy. Copying provided a working base functionality upon which developers could grow their project. This suggests that tools can embrace the development practice of growing code through tools to generate these "Hello World's" and support the copying-and-pasting of working code segments. Today, development environments provide ways to generate common code templates that can be customized. For example, the Eclipse IDE expands the word "try" into a full Java *try-catch* exception handling block. These templates are intended to help developers avoid the mundane details, reducing errors and speeding up the programming process. There are two alternate approaches. First, R3 can provide ways to generate working examples from documentation. Second, R3 can support the rapid generation of working code from high-level specifications of the paper application (such as drawings). We address these issues later.

Customizing Tool Support: Extending Ink Operations

In their projects, students not only used the provided R3 library elements, they also created their own (*e.g.*, a custom paper UI PDF renderer). This behavior was most pronounced in the area of manipulating digital ink, where developers who needed custom features would either subclass, or copy-and-modify existing toolkit components. To gather this data, we searched for and categorized all instances of ink operations in the class's source code corpus. In Figure 6**,** we see that while groups directly used R3 to decorate and display ink objects, only a handful felt comfortable enough to implement customized interactions for their applications. The developers who needed custom solutions extended the library to include operations to recognize inked gestures, select ink in space and time, and cluster strokes for calculating location and size. Since some developers extended R3's ink operations library, one might conclude that R3 omitted elements that should have been included; however, we expect that even with a large set of available operations developers will still find the need to composite or create their own custom solutions.

Iterative Debugging of Event Handlers

One technique that we used to understand "trouble spots" in the API—where developers struggled—was to search the source code for debug output (*e.g.*, *System.out.println()*). As debugging statements are generally used to display state, the values of variables, or signal error conditions, they may reveal which parts of R3 were more difficult to work with. The source files contained 1232 debugging statements (containing *println*). We examined and annotated each one (see Figure 7). From this data, we see *where* these debugging statements are located. Our code analysis found that 39% *of all console output functions were inside event handlers*: 333 debugging statements were located in GUI event handlers, and 145 were located in R3 event handlers.

We also examined *what* was printed in each debug statement. While many of the values were objects particular to each project, we found that a large portion of statements were of the "got here" type (statements that serve no purpose other than to tell the developer that a code block was reached). In fact, when coupled with the data on where the debug statements were located, we find that more than half of GUI event handler *println*s and almost a third of R3 event handler *println*s were "got here" statements. This suggests that we can help developers better understand when their event handlers are being reached.

Early on, these statements can serve as working stubs, helping developers keep track of which event handlers have not been implemented. We observed this when *println*s were placed next to tool-generated comments. For example:

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

Later, these debug statements can help developers visualize what their program is doing in response to pen input. This code evolution is referred to as "debugging into existence" [30]. The *println*s in event handlers suggest that we can improve developers' understanding of event handlers, and provide better support for existing debugging practices.

Successes and Shortcomings

R3 main success centers on its familiar programming model, which presented a low threshold for students. For instance, one team wrote in their final report for the course: "We have a very good impression of the R3 toolkit, and we

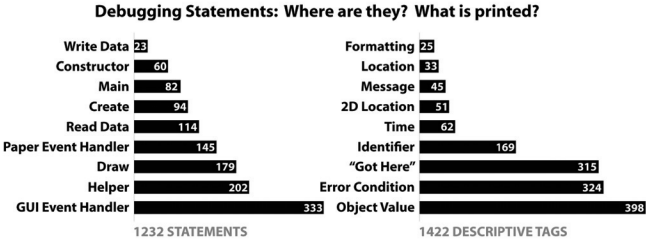

Figure 7. Analysis of the source code of 17 projects revealed that people place most of their debugging statements (println) in event handlers (GUI and R3). Many of the only statements tell the developer when the code "Got Here." However, most are of object values specific to the particular application.

believe that it presents an acceptable threshold of entrance for a novice to moderately skilled Java programmer." Because R3 extended established GUI conventions, students in the semester-long class could use their experience while working on their paper interfaces. Notably, *many students learned GUI programming as a part of this introductory course* (one group reported that "none of us had developed event-driven programs prior to this project."). The fact that Swing and R3 are architecturally similar meant that students did not have to learn two different programming models.

Through its extensible architecture, R3 provides a high ceiling of application complexity—four teams leveraged this to create their own ink handling. *Team D* recognized 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, and supported import of photos into those areas. *Team N* recognized handwritten musical symbols, including whole, half, quarter, and eighth notes, and translated the composition into MIDI files.

This field study also exposed shortcomings in R3. *First*, designing with R3 had a bottleneck—users reported that printing paper UIs inhibited rapid design, development, and testing, as printing a paper interface is much slower than rendering a GUI. We later eliminated the need to print during design and testing by providing a graphical preview of the paper UI, and a means to use preprinted notebooks to simulate the UI.

Second, R3 developers could not debug paper UIs without physical pens. One developer noted that "the fact that we had only one pen to share made it extremely difficult for everyone to write individual pieces of code….only one person at a time could perform any debugging." While R3's support for recording pen events and replaying them provides a mechanism that addresses this issue, *save & replay* was not advertised as a tool for distributed debugging.

Third, R3 did not provide a transparent way to swap batched and real-time pen interactions. Batched pen data would appear to the program as ink (*like in* PADD), and would not explicitly invoke event handlers (R3) or media associations (iPaper). The deployment revealed that while developers prefer using real-time pen input during testing, many expect their users to operate in a disconnected environment (10 of 17 projects were mobile apps). Providing easy ways to interchange batched and real-time interaction would address the desire to use synchronous systems for *debugging* and asynchronous ones for *deployment*. *Finally*, because R3 provides many features, it is difficult for newcomers to quickly grasp the extent of the toolkit's architecture. Looking at the developer experience, we find that R3 should provide visual aids for the exploration, development and testing of paper UIs.

```
<sheet width="8.5" height="8.867">
  <region name="Region1" x="0.69" y="0.56" width="6.994" height="5.216">
   <eventHandler type="ink"/>
  </region>
  <region name="Region2" x="4.849" y="6.567" width="2.412" height="1.328">
   <eventHandler type="click"/>
  </region>
  ...
</sheet>
```

```
public SketchedPaperUI() {
  super("SketchedPaperUI");
  sheet = new XMLSheet(new File("SketchedPaperUI.xml"));
  addSheet(sheet);
  initializePaperUI();
}

private void initializePaperUI() {
  Region regionRegion1 = sheet.getRegion("Region1");
  setupRegion1(regionRegion1);

  Region regionRegion2 = sheet.getRegion("Region2");
```

Figure 8. R3 translates low-fidelity paper sketches to working paper interface specifications (XML & Java), or an equivalent GUI that can be used for simulation.

INTEGRATING CODE WITH VISUAL DESIGN AND TEST

In R3's second iteration, we introduced tool support for *exploration*, augmenting the coding practices we observed:

- coding-by-growing from "hello world" programs
- debugging events through "got here" statements
- customizing and composing operations of ink strokes

Supporting these practices enhances R3's explorability, as designers can prototype, customize, and test their programs more quickly. The insight here is that currently, designers of paper UIs must maintain a mental mapping between the code that they write and the 2D visual representation of the input surface. That is, there exists both a large gulf of execution, the gap between the designer's goals and the toolkit actions he needs to attain those goals, and a large gulf of evaluation, the difficulty in determining whether the paper UI is working based on the toolkit's output [12]. We suggest that visual tools can narrow these gaps. The ideas we present in this section bridge the visual task of designing and testing paper UIs to the less visual task of writing back-end code to make the application work.

From Paper Prototypes to Working Interfaces

To support the *coding-by-example modification* observed in our code analysis, we included a feature to allow a designer to export a drawn-on-paper sketch of an interface to code that will generate the UI (see Figure 8). This reduces the initial effort of learning to lay out paper UI components and attach event handlers to them, and enables designers without programming experience to create paper UIs. Currently, we support a simple visual language: the outermost box becomes a *Sheet*; internal components become *Regions*; lines that exit the *Sheet* become *Event Handlers*. Text written next to a handler is recognized and matched to a particular handler in R3's library. Finally, the sketch is exported to an XML representation, which is read in at runtime to generate the paper user interface.

Visual Browser for Ink Operations

Our source code analysis revealed that people would extend the existing ink operations by copying and modifying toolkit code (*e.g.*, an ink rendering snippet). We also found that occasionally, developers would rewrite code even when it

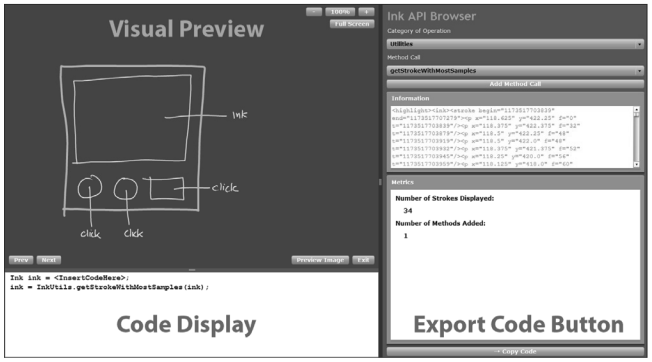

Figure 9. R3 supports rapid exploration of the Ink API by providing a browser that shows the effect of method calls visually and immediately, and allowing developers to export code to their IDE.

already existed in the toolkit (*e.g.*, exporting ink to a JPEG). These observations on the *customization of ink operations* suggest that we can provide a more effective way for developers to understand what is available in the R3 Ink API, how each operation would affect the digital ink, and how one might customize them.

Our working prototype presents source code alongside *visual previews* of ink strokes (see Figure 9). Suppose a developer wants to find the longest stroke in a list of *InkStrokes*. She browses the API through drop-down menus, and selects *getStrokeWithMostSamples*(). Upon adding this method, the longest stroke is highlighted in red. She copies the resulting code into her IDE, where she can grow the code if necessary. A complementary approach would be to integrate API finding into Web searches [37].

Visual Debugging of Event Handlers

The challenge with using console output for debugging events is that event handling code does not run until the developer provides input to bring the program to the desired state. Having a rapid way to debug events would save considerable work. Our source code analysis of *debugging output* reveals that we need to help developers understand what happens when event handlers are called.

The second iteration of R3 provides techniques to support debugging. We now provide visual representations of the UI during testing, and visualize the source code reached by each event handler as the program runs. To help developers understand which event handlers are called, R3 provides a visualization of the event handlers laid out on the 2D paper UI (see Figure 10). The visualization tracks statistics, such as how many times an event was triggered. To improve existing practice, R3 provides a new debug-to-console technique; the developer invokes R3's *showMe()* method to send values to *both* the console and a 2D visualization of the paper UI. This helps a developer see which event handlers were called, and evaluate the object values in context of that event handler. When the debugging tool is hidden, *showMe()* maintains current practice by outputting to the console, behaving exactly like *println()*.

Paper and Tablet-based Simulation

Finally, to *eliminate* the need to print while debugging, we added two ways to simulate the paper UI. First, we allowed simulation (through a tablet or mouse). Second, we now allow developers to bind any patterned paper to regions at application runtime. Therefore, developers can use pre-printed Anoto notebooks to simulate their paper UIs.

RELATED WORK

This research builds upon earlier work in user interface software architectures, design tools, and studies of existing development and debugging practices.

User Interface Software Architectures and Design Tools

The R3 approach was inspired mainly by architectures for *graphical* user interfaces. The first iteration of the paper toolkit borrowed the basic ideas of components, layout, event handling, and extensibility from GUI toolkits like Java Swing [38], Windows Forms [22], and SubArctic [11]. R3 extends this model to device ensembles composed of both augmented paper and digital systems. The second iteration of R3 supports an XML representation of the paper UI, generated by the designer's hand-sketched prototypes. This representation was inspired by the movement to better separate the view from event handling, seen in earlier work [1, 26] and now on commercial platforms (*e.g.*, XUL [23]).

For *paper* interfaces, there exist several authoring tools. Anoto's SDK [2] enables developers to access pen samples, but provides no explicit support for event handling or output to devices. Several frameworks build on Anoto. Cohen *et al.*'s work [4] integrates pen input with speech commands. PADD supports the integration of annotations on a physical document back into the digital one [8]. iPaper is a data-centric approach that maps pen input to remote data and code stored on the iServer [27, 34]. One limitation with this approach is that a database server containing these resources must be accessible to the pen's host. This generalized approach works in a production system, but it limits the speed at which a designer can explore prototypes on her local machine. iPaper is the platform most related to our

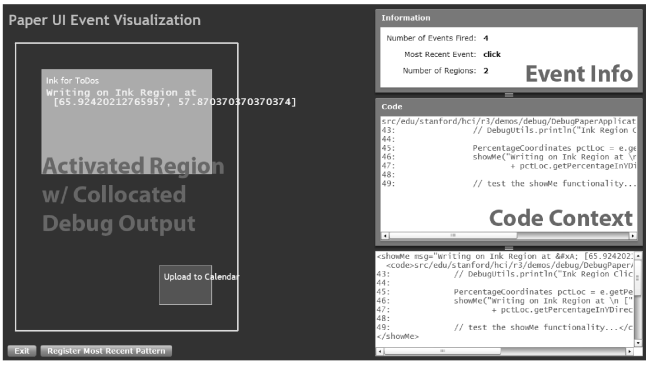

Figure 10. Building on debugging practices, R3 presents output on a 2D visualization of the paper UI. Here, the developer has interacted with the large paper region; its digital counterpart is highlighted. A panel displays the event code from the context of the region. Finally, *showMe()* displays output next to the region.

own, and is largely complementary to our interaction-centric approach. However, R3's *Actions* hides the network complexity that iPaper exposes (HTTP requests between client and server); instead, R3 devices act as peers. R3's main contribution beyond the prior work is the depth in the evaluation of extensive use of the toolkit, and the subsequent iteration of the architectural abstractions.

Supporting Existing Coding Practices

In designing the R3 study and understanding the results, we drew on prior research on the development practices of software engineers, specifically in how software is created and modified. Rosson and Carroll studied the code reuse practices of four programmers and found that they benefited from having working examples (*usage contexts*) that they could modify to include in their own project [30]. In addition to providing running demonstrations, R3 supports rapid generation of working code through sketching, allowing novice developers to specify a working base and incrementally grow their application. This sketching approach was introduced in SILK [17], and is used by recent systems (*e.g.*, [25]). R3's differs in that developers can sketch with *pen and paper* instead of a digital tablet (providing a more mobile alternative), and then specify event handlers by writing their name. R3 also exports the interface to integrate with final working code.

Besides [30], at least two other studies note that programmers use copy-and-paste to reduce typing, and ensure that the fine details (*e.g.*, method names) are correct. First, Kim *et al.* studied expert programmers and found that copy-and-paste was used to save time when creating or calling similar methods [14]. Later, LaToza *et al.* found that modifying usage contexts was one of *several* types of code duplication, which causes problems when fixing bugs or refactoring [18]. However, these studies did not concentrate on user interface development. Our own observations support the existence of the copy-and-paste and code-by-growing behaviors, and suggest that users rely on copying when they need to *learn* a new API.

R3's visualizations for the paper UI and handlers extend ideas developed in software visualization research. DeLine *et al.* introduced designs to help developers visualize common code paths [5]. The R3 debugger applies this real-time highlighting to event handlers. However, most of the work in this community seeks to understand class relationships, algorithms, and data structures [35]. We extend this effort by helping developers understand the relationships between GUIs, event handlers, and debug output. In digital arts, Fry has visualized call graphs of code bases (*e.g.*, [7]). In interface research, Hands demonstrated an accessible, playing-card visualization for objects, where properties were shown in a tabular format [28]. However, event handlers were represented only in natural language, or implicitly defined by textual properties. Papier-Mâché's monitoring window demonstrated that visuals of objects and events can enhance the debugging process [15]. R3 also supports the display of event activation, but increases visibility by overlaying debug output on the paper UI.

FUTURE WORK

Looking forward, we see the results of the R3 study as suggesting three valuable directions for further research:

Ordering Constraints—it is difficult to enforce interaction constraints on a paper UI. In a display with graphical feedback, a developer can gray out a component when it is not appropriate. On paper, one cannot stop a user from arbitrarily checking a box or turning to the next page. Today, developers must provide textual directions to the user, and handle input that is incomplete or out-of-order.

Synchronous vs. Asynchronous—R3 supports real-time pen interactions through *event handlers*; batched data requires separate *ink handlers*, to import saved ink into an application. We have since found this to be a limitation. Future R3 iterations will process batched input through event handlers, and allow developers to provide *hints* (*e.g.*, "if real-time feedback is unavailable, disregard this event").

Slow Refresh—today, if the user needs to update his paper interface, he must print out a new copy with the new information. Future toolkits should provide explicit support for scheduling updates to this paper *view* (we think of paper as the view in MVC, but with an extremely low refresh rate).

CONCLUSION

Through an iterative design of the R3 paper applications toolkit, we learned that a traditional event-driven approach can provide an approachable platform for programmers to build pen-and-paper applications. Additionally, support for visual development and debugging can make the process much more efficient. However, there remain toolkit challenges in this space (*e.g.*, support for integrating non-programmers into the process).

Our results also have implications for the design of *graphical* applications. For example, one might allow a designer to import a paper sketch into a GUI builder. We also found it valuable to use both long-term deployment and the static analysis of source code to inform the design process. We suggest that future tool design should be informed by such techniques. From this study, we conclude that toolkits should explicitly support programming by example modification, provide efficient exploration of APIs, and present good visualizations of program event flow.

The R3 toolkit and code analysis tools are open source; they, and a video demonstrating this research, can be found at http://hci.stanford.edu/paper.

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

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

4 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.

5 DeLine, R., A. Khella, M. Czerwinski, and G. Robertson. Towards Understanding Programs through Wear-based Filtering. *SoftVis: ACM Symposium on Software Visualization*. pp. 183–92, 2005.

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

7 Fry, B., *distellamap*, 2007. http://benfry.com/distellamap

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

9 Hong, J. I. and J. A. Landay. SATIN: a Toolkit for Informal Ink-based Applications. *UIST: ACM Symposium on User Interface Software and Technology*. pp. 63–72, 2000.

10 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

11 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.

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

13 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.

14 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.

15 Klemmer, S. R., J. Li, J. Lin, and J. A. Landay. Papier-Mâché: Toolkit Support for Tangible Input. *CHI: ACM Conference on Human Factors in Computing Systems*. pp. 399–406, 2004.

16 Krasner, G. E. and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Object Oriented Programming* **1**(3). pp. 26–49, 1988.

17 Landay, J. and B. A. Myers. Interactive sketching for the early stages of user interface design. *CHI: ACM Conference on Human Factors in Computing Systems*. pp. 43–50, 1995.

18 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.

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

20 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.

21 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.

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

23 Mozilla, *XUL*, 2007. http://www.mozilla.org/projects/xul

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

25 Newman, M. W., J. Lin, J. I. Hong, and J. A. Landay. DENIM: An Informal Web Site Design Tool Inspired by Observations of Practice. *Human-Computer Interaction* **18**(3). pp. 259–324, 2003.

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

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

28 Pane, J., *A Programming System for Children that is Designed for Usability.*, Unpublished PhD, Carnegie Mellon University, Computer Science, Pittsburgh, PA, 2002. www.cs.cmu.edu/~pane/thesis

29 Rettig, M. Prototyping for tiny fingers, *Communications of the ACM*, vol. 37(4): pp. 21-27, 1994.

30 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.

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

32 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.

33 Shneiderman, B., G. Fischer, M. Czerwinski, B. Myers, and M. Resnick, *Creativity Support Tools*. Washington, DC: National Science Foundation. 83 pp. 2005.

34 Signer, B., *Fundamental Concepts for Interactive Paper and Cross-Media Information Spaces*, Unpublished PhD, ETH Zurich, Computer Science, Zurich, 2006. http://www.globis.ethz.ch/script/publication/download?docid=411

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

36 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.

37 Stylos, J. and B. A. Myers. Mica: A Web-Search Tool for Finding API Components and Examples. *VLHCC: Visual Languages and Human-Centric Computing*. pp. 195–202, 2006.

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

42 Yeh, R. B., *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.