GAINING DESIGN INSIGHT THROUGH INTERACTION PROTOTYPING TOOLS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Björn Hartmann
September 2009

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

(Scott R. Klemmer) Principal Adviser

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

(Terry Winograd)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

(Pat Hanrahan)

Approved for the University Committee on Graduate Studies:

ABSTRACT

Prototyping is the fundamental activity that structures innovation in design. While prototyping tools are now common for graphical user interfaces on personal computers, prototyping interactions for ubiquitous computing systems remains out of reach for designers. This dissertation contributes concepts and techniques, embodied in software and hardware artifacts, to answer two research questions:

1) How can design tools enable a wider range of designers to create functional prototypes of ubiquitous computing user interfaces?
2) How can design tools support the larger process of learning from these prototypes?

Fieldwork at professional design companies showed that design generalists lack the tools to fluently experiment with interactions for sensor-based interfaces and information appliances. The first contribution of this dissertation is a set of methods, embodied in authoring tools, that lower the expertise threshold required to author such novel interfaces. These tools enable more designers to author a wider range of interfaces, faster. Visual authoring of control flow diagrams and plug-and-play hardware linked to software abstractions for hardware components enable rapid authoring of interaction logic. This dissertation also introduces programming by demonstration techniques for sensor-based interactions to derive high-level events from continuous sensor data streams.

Enabling the construction of prototypes is an important function of design tools; however, it should not be the only goal. Prototypes are just a means to an end — they are built to elicit feedback about design choices. The second contribution of this thesis is a set of systems that explicitly support the design practices of exploration and iteration. Exploration is supported through enabling the creation of multiple, parallel user interface alternatives. The design-test-analysis loop of iterative design is supported through techniques for rapid review of user test data and techniques for revision of interaction diagrams. The presented work is informed by interviews and collaborations with professional interaction designers. The tools are evaluated through a combination of laboratory studies and deployments to interaction design students at Stanford and in industry.

ACKNOWLEDGMENTS

I would like to thank my advisor, Scott Klemmer, for the freedom and support he gave me to follow my interests and chart my own path over the last five years. Scott and I both arrived at Stanford in the Fall of 2004, and he took me on as his first newly-admitted student. I thank him for his trust, and the passion, commitment, and energy with which he led our research group. His door was always open and his advice for navigating graduate school and the research world prepared me well for my upcoming transition to faculty life. I also thank my other reading committee members, Terry Winograd and Pat Hanrahan, for their feedback and stimulating conversations over the years. I am grateful to Stu Card and John Haymaker, who served on my orals committee. Bill Verplank has been a source of inspiration throughout my time at Stanford. His course on building electronic musical instruments launched me in the direction of this dissertation.

Much of the work in this dissertation was undertaken with the help of a fantastic group of collaborators. I have had the privilege to supervise a group of talented and hard-working undergraduate summer interns through Stanford's CURIS program, who have contributed to every project presented here: Michael Bernstein, Loren Yu, Anthony Ricciardi, Timothy Cardenas, Sean Follmer, and Daniel MacDougall. Many other collaborators have worked with me on either the research presented in this dissertation, or the numerous other projects we pursued together. I am deeply indebted to them. At Stanford, I worked with Leith Abdulla, Abel Allison, Marcello Bastea-Forte, Joel Brandt, Jesse Cirimele, Kevin Collins, Scott Doorley, Wendy Ju, Michel Krieger, Dan Maynes-Aminzade, Nirav Mehta, Manas Mittal, Merrie Ringel Morris, Erica Robles, Leila Takayama, Leslie Wu, and Yeonsoo Yang. At Microsoft Research, Merrie Morris, Andy Wilson, and Hrvoje Benko were fabulous mentors.

My research would not have been possible without the tireless work of system administrator John Gerth and the lab administrative staff, Heather Gentner, Ada Glucksman, Melissa Rivera and Monica Niemic. I would also like to thank the professional designers, who shared their time and expertise with me; Arna Ionescu and Hans-Christoph Haenlein at IDEO were especially generous with their time.

Finally, I would like to express my gratitude to my parents, Volker and Lieselotte Hartmann, and my wife, Tania Treis. Your love and support has made this possible.
Thank you.

I was supported by an Agilent School of Engineering Fellowship and a SAP Stanford Graduate Fellowship during my five years at Stanford. The d.tools project was further supported by a grant from the Stanford Office of Technology Licensing. Dai Nippon Printing, through the Stanford MediaX organization, supported the development of Exemplar. Juxtapose was partially funded through NSF grant IIS-0745320. Nokia and Intel donated hardware for multiple projects.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

A decade and half after Weiser's call to integrate computation into the fabric of our lives [248], the design and evaluation of ubiquitous computing systems remains challenging. Difficulties arise partially because of a lack of appropriate design tools: the progress of any creative discipline changes significantly with the quality of the tools available [51,167,228,229]. As the creation of ubiquitous computing devices moves from research labs into design consultancies and product teams, better tools that support user experience professionals are needed. My fieldwork at professional design companies showed that design generalists currently lack the tools to fluently experiment with interactions for sensor-based interfaces and information appliances.

Prototyping is the pivotal activity that structures innovation, collaboration, and creativity in professional design. Design studios pride themselves on their prototype-driven culture; it is through the creation of prototypes that designers learn about the problem they are trying to solve. Effective prototyping tools aid and improve design space exploration, design team communication, and ultimately, lead to better products. The goal of this dissertation is to develop principles and authoring methods that guide the creation of more appropriate prototyping tools for interaction design. This dissertation contributes to the advancement of prototyping tools by considering two different research questions:

1) How can design tools enable a wider range of designers to create functional prototypes of ubiquitous computing user interfaces?
2) How can design tools support the larger process of learning from these prototypes?

Our exploration of the first question is concerned with improving prototyping methods—finding new ways to model and structure the authoring task. Two complementary methods for authoring sensor-based interactions are introduced: a control-flow-based visual authoring environment for interaction logic that is based on existing storyboard practices; and a programming-by-demonstration environment that helps designers extract useful high-level interaction events from continuous sensor data. These methods lower the expertise threshold required to author sensor-based interfaces. The tools enable more designers to author a wider range of interfaces, faster.

The second question concerns the function a prototype plays in the larger design process. Prototyping is not primarily about the artifacts that get built: it is about eliciting feedback (from the situation, from users, team members & clients). Prototypes embody design hypotheses and enable designers to test these hypotheses. Prototyping tools can thus become more valuable to designers when they explicitly offer support for eliciting and managing feedback. Today, software prototyping tools are mostly agnostic to this role. This dissertation contributes techniques for creating and managing multiple alternative design solutions, and for managing feedback from both external testers and design team members. A brief overview of the contributions and the contents of this dissertation follows.

1.1 THESIS CONTRIBUTIONS

This dissertation contributes principles and systems for prototyping user interfaces that span physical and digital interactions. The technical contributions are based on evidence collected through interviews with designers and online surveys. This evidence suggests that interaction designers lack tools to create interfaces that leverage sensor input; to explore alternative interface behaviors; to efficiently review interface test videos; and to effectively communicate interface revisions.

The dissertation makes the following technical contributions in three areas:

1) Techniques for authoring user interfaces with non-traditional input/output configurations.
    a. Rapid authoring of interaction logic through a *novel combination of storyboard diagrams* for information architecture *with procedural programming* for interactive behaviors.
    b. *Demonstration-based definition of discrete input events from continuous sensor data streams* enabled by a combination of pattern recognition with a direct manipulation interface for the generalization criteria of the recognition algorithms.
    c. *Management of input/output component configurations for interface prototypes* through an editable virtual representation of the physical device being built. This representation reduces cognitive friction by collapsing levels of abstraction; it is enabled by a custom hardware interface with a plug-and-play component architecture.

2) Principles and techniques for exploring multiple user interface alternatives.
    a. Techniques for *efficiently defining and managing multiple alternatives of user interfaces* in procedural source code and visual control flow diagrams.
    b. *User-directed generation of control interfaces* to modify relevant variables of user interfaces at runtime.
    c. *Support for sequential and parallel comparison of user interface alternatives* through parallel execution, selectively parallel user input, and management of parameter configurations across executions.
    d. *Implementations of the runtime techniques for three different platforms*: desktop PCs, mobile phones, and microcontrollers.
3) Techniques for capturing feedback from users and design team members on user interface prototypes, and for integrating that feedback into the design environment.
    a. *Timestamp correlation between live video, software states, and input events* generated during a usability test of a prototype to enable rapid semantic access of that video during later analysis.
    b. *Novel query techniques* to access such video recordings: *query by state selection* where users access video segments by selecting states in a visual storyboard; and *query by input demonstration* where sections of usability video are retrieved through demonstrating, on a physical device prototype, the kind of input that should occur in the video.
    c. *A visual notation and a stylus-controlled gestural command set for revising user interfaces* expressed as control flow diagrams.

This dissertation also provides evidence, through laboratory studies and class deployments, that the introduced techniques are successful. In particular, the dissertation contributes:

1) Evidence that the introduced authoring methods for sensor-based interaction are accessible and expressive through two laboratory evaluations and two class deployments.
2) Evidence from a laboratory study that the techniques for managing interface alternatives enable designers to explore a wider range of design options, faster.
3) Evidence from two laboratory studies that an interactive revision notation for interfaces leads to more concrete and actionable revisions.

1.2 DISSERTATION ROADMAP

This section presents a brief overview of the structure of this dissertation by chapters.

1.2.1 BACKGROUND: PROTOTYPES IN THE DESIGN PROCESS (CHAPTER 2)

The terms design, prototyping, and sketching have many meanings for different audiences. Drawing on literature in design research, this chapter stakes out a perspective on design practice and how prototyping activities occur throughout the design process. We briefly discuss the history of industrial design as a discipline grounded in the development of material goods and the later transfer of principles from industrial design to software [251].

While there are many different conceptions of the design process, models tend to agree on two core characteristics. First, design is exploratory and emergent — the structure of the design problem itself has to be uncovered and this uncovering happens through generating concrete design proposals and evaluating them. Designers think with, and communicate through artifacts and models [66]. These artifacts are prototypes. Second, generation and evaluation of solution proposals exemplifies a recurrent, fundamental interplay between divergent and convergent stages in design: first a range of different potential solutions is generated, then desirable solutions are selected from that set of alternatives. This cycle repeats as the focus shifts from design concepts to implementation strategies.

The chapter concludes with a survey of literature about the role that prototypes play in design and software engineering. This survey leads to a classification of prototypes according to three questions: What purpose do prototypes serve? What aspects of a design do they address? And what level of functionality should they offer?

1.2.2 RELATED WORK (CHAPTER 3)

How are existing tools supporting prototyping activity? What needs are still unmet? We describe the state of the art in professional practice and present an overview of research in prototyping tools. Our discussion of related research addresses the following concerns in separate sections:

USER INTERFACE PROTOTYPING TOOLS

Prior research has introduced environments for graphical user interfaces [155], web site information architecture design [171], and context-aware applications [219], among others. We review tools that focus on storyboard-based authoring, direct manipulation UI layout, and Wizard of Oz simulation [140] of interface functionality.

TOOLS SUPPORT FOR PHYSICAL COMPUTING

Physical computing combines physical and digital interactions. Consequently, tools in this area often focus on the interdependence of hardware and software. We review research into hardware toolkits and programming models for working with sensors and actuators [93,178,185].

VISUAL AUTHORING

Visual authoring or visual programming is thought to offer a lower threshold than textual programming. The reality is more nuanced. We provide an overview of visual formalisms and visual programming languages. We distinguish between control flow environments [87], data flow environments [209], augmented source editors [238], and hybrid environments [1].

PROGRAMMING BY DEMONSTRATION

Programming by demonstration promises to lower the barrier of specifying complex logic or behavior by demonstrating that behavior to a computer. The crucial step in the success of failure of programming by demonstration lies in the generalization step that transforms observed examples to general rules. We discuss how previous systems have addressed this challenge for programming [67] and computer vision applications [75].

DESIGNING MULTIPLES

If design indeed oscillates between generating multiple alternatives and then selecting between these alternatives, design tools should explicitly support working with sets of potential designs. We survey existing work in image processing [161,239], rendering [181] and information querying [177] that address this challenge.

CAPTURING & MANAGING FEEDBACK

How can design tools capture user test data or team feedback on prototypes? We review work in document annotation [198] and usability video structuring through event logs [39,179].

1.2.3 AUTHORING SENSOR-BASED INTERACTIONS (CHAPTER 4)

This chapter presents two novel prototyping methods that enable faster creation of functional interaction designs for sensor-based user interfaces. Myers et al. introduced the terms threshold and ceiling to describe use properties of a tool: the *threshold* is the difficulty of learning and using a system, while the *ceiling* captures the complexity of what can be built using the system [191]. d.tools and Exemplar help designers construct functional prototypes

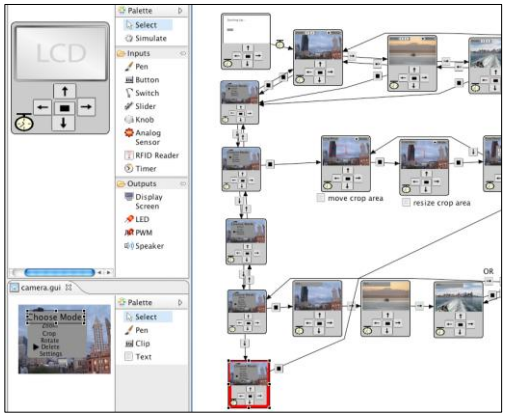

Figure 1.1: The d.tools visual authoring environment enables rapid construction of UI logic.

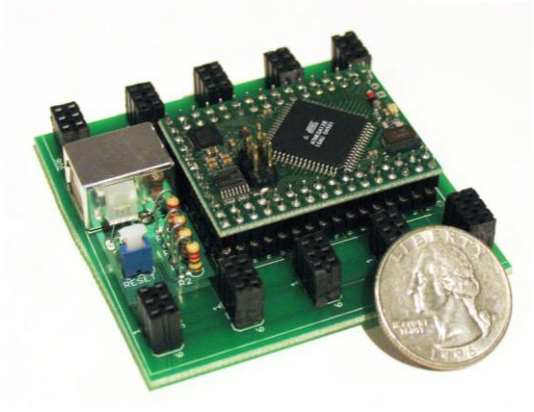

Figure 1.2: The d.tools hardware interface offers a plug-and-play architecture for interface components.

by lowering the threshold of required expertise. The goal of both systems is to enable users to focus on design thinking (how an interaction should work) rather than implementation tinkering (how hardware and sensor signal processing work).

d.tools is a software and hardware toolkit that embodies an iterative-design-centered approach to prototyping information appliances. d.tools enables non-programmers to work with the bits and the atoms of physical user interfaces in concert. Supporting early-stage prototyping through a visual, statechart-based approach, d.tools extends designers' existing storyboarding practices (Figure 1.1). As designers move from early-stage prototypes to higher fidelity prototypes, d.tools augments visual authoring with scripting. A hardware platform based on smart components that communicate on a shared bus offers plug-and-play use of sensors and actuators (Figure 1.2). The architecture exposes extension points for experts to grow the library of supported electronic components.

d.tools provides software abstractions for hardware and offers rapid authoring of interaction logic. An additional barrier for practitioners became apparent when we deployed d.tools to an HCI class: students often struggled to transform raw, noisy sensor data into useful high-level events for interaction design. Exemplar, an extension to d.tools, bridges the conceptual gap between conceiving of a sensor-based interaction and formally specifying that interaction through programming-by-demonstration. With Exemplar, a designer first demonstrates a sensor-based interaction to the system (e.g., she shakes an accelerometer — Figure 1.3). The system graphically displays the resulting sensor signals. The designer then marks up the part of the visualization that corresponds to the action — Exemplar learns

appropriate recognizers from these markups. The designer can review the learned actions through real-time visual feedback and modify recognition parameters through direct manipulation of the visualization.

Both d.tools and Exemplar have been evaluated through individual laboratory studies and deployment to interaction design courses and to industry. In a first-use evaluation of Exemplar, participants with little or no prior experience with sensing systems were able to design new motion-based controllers for games in less than 30 minutes (Figure 1.4). In our collaboration with educational toy company Leapfrog, we provided d.tools hardware schematics and software to Leapfrog's advanced development group. In return, Leapfrog manufactured a complete set of hardware toolkits for us to distribute to a second year of Stanford HCI students. In collaboration with Nokia, we also extended d.tools to author prototype interfaces for mobile devices.

1.2.4 CREATING ALTERNATIVE DESIGN SOLUTIONS (CHAPTER 5)

Creating multiple prototypes facilitates comparative reasoning, grounds team discussion, and enables situated exploration. However, current interface design tools focus on creating single artifacts. How might interaction design tools explicitly support creation and management of multiple user interface alternatives? This chapter discusses two approaches.

We first investigated how to support exploration in Juxtapose, a source code editor and runtime environment for designing multiple alternatives of interaction designs in parallel. Juxtapose offers a code editor for user interfaces authored in ActionScript in which

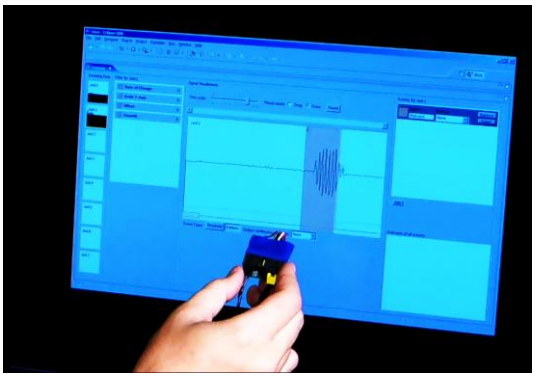

Figure 1.3: Exemplar combines programming-by-demonstration with direct manipulation to author sensor-based interactions.

Figure 1.4: This evaluation participant used Exemplar to control 2D aiming in a game with an accelerometer, and shooting with the flick of a bend sensor.

interaction designers can define multiple program alternatives through linked editing, a technique to selectively modify source files simultaneously. The set of source alternatives are then compiled into a set of programs that are executed in parallel (Figure 1.5). Optimizing user experience often requires trial-and-error search in the parameter space of application variables. To improve this tuning practice, Juxtapose generates a control interface for application parameters through source code analysis and language reflection (Figure 1.6). A summative study of Juxtapose with 18 participants demonstrated that parallel editing and execution are accessible to interaction designers and that designers can leverage these techniques to survey more options, faster. To show that general principles of working with alternatives carry over into other domains, we also developed Juxtapose runtime environments for mobile phones and microcontrollers.

We then discuss how ideas for exploring alternatives can be transferred from a textual programming environment such as Juxtapose to the visual authoring environment of d.tools. Visual control flow environments offer the opportunity to present alternative states side-by-side in the same canvas. They also present some challenges in managing the additional visual complexity resulting from capturing multiple behavior options.

Figure 1.5: Side-by-side execution in Juxtapose enables rapid comparison of alternatives.

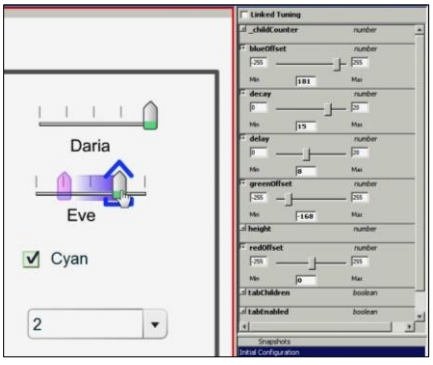

Figure 1.6: Juxtapose automatically generates control interfaces for program variables.

1.2.5 GAINING INSIGHT THROUGH FEEDBACK (CHAPTER 6)

If prototyping is about eliciting feedback, then tools that manage the feedback process explicitly can help designers gain insight, capture that insight, and act on it. We present two methods for integrating feedback capture and management directly into design tools.

Many prototypes go through team discussions and reviews before being tested. In word processing, revision management algorithms and interactions techniques effectively enable asynchronous collaboration over text documents. But no equivalent functionality exists yet for revising interaction designs. d.note introduces a revision notation for expressing tentative design proposals within d.tools. The tool comprises commands for insertion, deletion, modification and commenting on appearance and behavior of interface prototypes (Figure 1.7). d.note realizes three benefits: it visually distinguishes tentative changes to retain design history, allows for Wizard of Oz simulation of proposed functionality, and manages display of alternative design choices to facilitate comparison. In a laboratory evaluation, twelve design students critiqued existing d.tools prototypes with and without d.note. Participants reported that the ability to express and test functional changes was a clear benefit of d.note. In a follow-up study, eight design students interpreted the annotated diagrams produced in the first study, showing that d.note diagrams were less ambiguous to interpret, but that they lacked high-level justification when compared to free-form annotation.

When prototypes are tested with team mates or external users, test sessions are often recorded on video. Historically, the hours and days of work required for manual video analysis has limited the practical value of these recordings. The d.tools video suite provides integrated

Figure 1.7: d.note introduces stylus-driven revision of interaction diagrams.

Figure 1.8: The d.tools test & analysis functions link video clips of test sessions to event traces of the tested prototype.

		d.tools	d.note	Exemplar	Juxtapose
HOW	Visual Authoring Environment	✓	✓	✓	
(Techniques)	Programming by Demonstration			✓	
	Parallel Editing & Execution				✓
WHY	Rapid Construction & Iteration	✓		✓	
(Place in design)	User Feedback: Design-Test-Analyze	✓			
	Team Feedback: Design-Review-Annotate		✓		
	Exploring Design Alternatives	✓	✓		✓
WHAT	Custom, sensor-based interactions	✓	✓	✓	✓
(Artifacts built)	Mobile device interfaces	✓			✓
	Desktop GUIs				✓
	Tentative Sketches; Simulated Functionality		✓		

Table 1.1: An overview how research concerns map onto the concrete systems presented in this dissertation.

support for testing prototypes with users and rapidly analyzing test videos to inform subsequent iteration (Figure 1.8). d.tools logs all user interactions with a prototype and records an event-synchronized video stream of the user's interactions. The video is automatically structured through state transitions and input events. After a test, d.tools enables designers to view video and storyboard in parallel as a multiple view interface [41] into the test data. Two novel query techniques shorten the time to find relevant segments in the video recordings: query by state selection, where users access video segments by selecting states in the storyboard; and query by input demonstration, where designers demonstrate the kind of input that should occur in the video.

1.2.6 CONCLUSIONS & FUTURE WORK (CHAPTER 7)

The final chapter provides a review of the contributions, and offers an outlook to future work by reconsidering the fundamental assumptions made in this dissertation. The chapter discusses opportunities for different types of authoring tools that result if some of these assumptions are modified.

1.2.7 OVERVIEW: RESEARCH CONCERNS & PROJECTS

This dissertation explores the space of novel prototyping tools through multiple projects. To aid the reader, Table 1.1 shows how research concerns map onto the different concrete projects discussed in this dissertation.

1.3 STATEMENT ON MULTIPLE AUTHORSHIP AND PRIOR PUBLICATIONS

The research presented in this dissertation was not undertaken by me alone. While I initiated and led all of the projects described here, the contributions of a talented group of collaborators must be acknowledged — without their efforts, the research could not have been realized in its current scope. In particular, the d.tools project benefited from UI implementation contributions by Michael Bernstein, Leith Abdulla, and Jennifer Gee; and video editor programming and integration by Brandon Burr and Avi Robinson-Mosher. In the Exemplar project, Manas Mittal contributed to the signal processing routines; Leith Abdulla contributed to the Exemplar user interface implementation. Sean Follmer, Timothy Cardenas, and Anthony Ricciardi contributed to gesture recognition, revision management, and the graphical user interface editor in d.tools and d.note. Meredith Ringel Morris, Sean Follmer, Haiyan Zhang, and Jesse Cirimele collaborated on various demonstration applications for d.tools and Exemplar. Loren Yu worked with me on the source code editor and runtime user interface of Juxtapose; Abel Allison and Yeonsoo Yang helped with the implementation of Juxtapose functionality for microcontrollers.

This dissertation is partially based on papers published in previous ACM conference proceedings; I am primary author on all publications. In particular, the d.tools system was published at UIST 2006 [109]; Exemplar at CHI 2007 [107]; and Juxtapose at UIST 2008 [114]. A paper describing d.mix is still in submission at the time of publication of this dissertation.

CHAPTER 2 BACKGROUND: PROTOTYPES IN THE DESIGN PROCESS

This dissertation proposes novel tools for the prototyping of user interfaces as part of a larger user interface design process. Doing so successfully requires understanding underlying principles and practices of design. This chapter presents a brief review of different models of design and the role prototypes play in the process.

2.1 DESIGN, DEFINED

User interface design is informed and influenced by professional design disciplines such as product design on one side and by software engineering on the other side. This section provides a brief overview of the history of professional design and introduces some established models of the design process to motivate the development of design-specific tools.

2.1.1 WHAT DO WE MEAN BY DESIGN?

Herbert Simon provided a very broad definition of design as "devising courses of action aimed at changing current situations into preferred ones" [230]. Countless competing definitions exist. Common to many definitions is the focus on a specific *process*, with the goal of *creating plans or models* for the creation of new artifacts, which have to *fit* potentially conflicting sets of constraints, requirements, and preferences. To elaborate on these three core characteristics:

1) Design is a process and has structure — there is a set of core activities designers engage in, regardless of the domain of design.
2) Design is not manufacturing — for physical artifacts, the final realization is done by someone else. For software, the division between design and implementation may be less clear. In both domains, the end product of design is often a specification that will be interpreted and implemented by someone else.
3) Design has a client and users — it is accountable to external judgment. Different stakeholders may have conflicting expectations.

Design is thus distinguishable as a unique discipline from art (creation which is accountable to the vision of the artist); engineering ("the application of scientific and mathematical

principles to practical ends" [31]); and science (the development of generalizable knowledge through observation, experimentation and hypothesis testing).

A more pragmatic characterization would be that *design is what professional designers do*. The field of design research adopts this perspective and describes the practices of successful practitioners to analyze what makes these practices effective. Cross [66], a prominent design researcher, argues that design has a "unique way of knowing" and distills four core abilities exercised by professional practitioners:

1) resolving ill-defined problems
2) adopting solution-focused cognitive strategies
3) employing abductive or appositional thinking
4) using non-verbal modeling media

In *ill-defined* or "wicked" [213] problems, the problem formulation itself is not clear at the outset and remains to be defined. Because the problem statement itself is not fixed, it is not possible to enumerate all possible options or to find an optimal solution. Simon argued that design problems therefore cannot be solved by optimizing, they can only be *satisficed* [230] — one can tell an adequate solution from an inadequate one, and make relative judgments of fit, but no global optimum exists.

Designers adopt *solution-focused strategies* by generating possible solutions first, then checking to what extent the generated ideas are adequate for the problem. Cross, reporting on a study of designers, summarizes: "Instead of generating abstract relationships and attributes, then deriving the appropriate object to be considered, the [designers] always generated a design element and then determined its qualities." [66:100]. Creation comes before analysis, and only through the creation of prototypes and other representations is it possible to test to what extent a design idea fulfills the design goals.

This tendency to produce proposals first is an expression of a*bductive reasoning* which, in contrast to deductive or inductive thinking, starts with concrete observations and guesses, which only later lead to theories about a design space. Making the right guesses or creative leaps requires experience.

Finally, designers tend think with, and communicate through *artifacts* and *models* rather than written language – sketches, diagrams, models and prototypes are used both to work through problems as well as to anchor communication with design team members and other stakeholders [66:28].

2.1.2 A SHORT HISTORY OF PROFESSIONAL DESIGN

Architecture has the claim to being the oldest design discipline. Its focus is on the holistic creation of structures that simultaneously satisfy requirements of functionality, economy, and aesthetics. Notably the architect is not the one who creates the building itself: her role is to transform needs, requirements, and constraints into a suitable plan that can then executed by a builder. Professional *product design* as a discipline emerged as a result of the shift from one-off artifacts created by craftspersons to mass production after the industrial revolution. While craftsmen would iterate from project to project and slowly evolve a product over time, mass production yielded many identical copies [175,237]. Because making changes to the tooling for mass manufacturing became more expensive, while marginal cost of production decreased, more care and planning was needed before manufacturing commenced to ensure that the manufactured product was in fact functional and desirable to consumers. Notable pioneers of product design in the first half of the 20th century include Henry Dreyfuss, Raymond Loewy, Walter Dorwin Teague, and Norman Bel Geddes. Their autobiographies offer detailed accounts of the mid-century industrial design process in North America [73,175,237].

Product design as a methodology has since been assimilated by the software industry. One of the formative academic works that advocated for this transfer of process was Winograd's "Bringing Design to Software" [251]. As software is ultimately used by people, its user interfaces should be created with the same concern for utility, usability, and satisfaction as other artifacts of daily life.

2.1.3 HOW DO DESIGNERS WORK? MODELS OF THE DESIGN PROCESS

How do the underlying principles of designerly knowledge introduced in section 2.1.1 find expression in designers' work practices? The process that evolved from architecture and product design is characterized by four core strategies: *need finding* through user research methods to establish constraints, *ideation* to generate many possible ideas and subsequently select promising ideas, *prototyping* to create concrete models and approximations based on those ideas, and *iterative refinement* based on testing generated prototypes. A more detailed model of this iterative process, as described by Moggridge [189], is shown in Figure 2.1.

Need finding involves learning about the target users of a new product — what are their unmet needs and unresolved pains; what are their motivations? Needs, requirements, and constraints may be expressed in narrative form, *e.g.*, as personas and scenarios [64:p. 123]; or more formally, *e.g.*, as user and task models [99]. The data gathered from such user research is

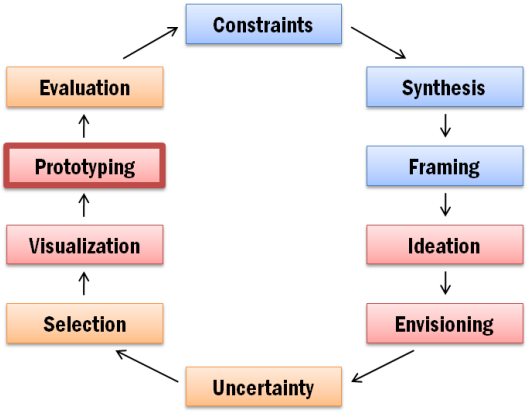

Figure 2.1: Design process stages according to Moggridge [189]. Diagram redrawn by the author.

then used to construct a concrete *point of view* or *framing* that encapsulates the goals a new design seeks to achieve. Given a framing, designers generate a multitude of concrete proposals. Initially, idea generation can take the shape of brainstorming or sketching of alternatives. To move from graphical envisioning towards concrete, testable artifacts, designers next generate concrete prototypes. These proposals are then compared an evaluated — against each other, or against user or stakeholder feedback. The gained knowledge is then used to drive the next iteration.

The process model described above is commonly observed, but by no means canonical. A wide-ranging overview of different design methodologies can be found in Jones [135]. Jones categorizes these methods and distills an important common thread: design is a sequence of divergent steps, where ideas are produced; and convergent steps, where ideas are eliminated. Buxton echoes this theme, writing that design alternates between concept generation and concept selection [55].

2.2 UNDERSTANDING PROTOTYPES

The prior section established that prototyping is a core activity in design across different domains. This section reviews some conceptions of prototypes in design and computer science and summarizes the literature on the purpose, role, and place of prototypes.

2.2.1 PROTOTYPES, DEFINED

The notion of a prototype is overloaded and there is no generally agreed upon definition. As a broad, inclusive definition, Moggridge regards a prototype as "a representation of a design, made before the final solution exists." [189]. Houde and Hill similarly point to the purpose of prototypes as indicators of a future reality, and distinguish between two functions — exploration and demonstration [126]. Buchenau and Suri add a third function of prototypes as tools for gaining empathy: "[A]n Experience Prototype is any kind of representation, in any medium, that is designed to understand, explore or communicate what it might be like to engage with the product, space or system we are designing" [52]. Lim and Stolterman foreground the role of prototypes as learning vehicles: "Prototypes are the means by which designers organically and evolutionarily learn, discover, generate, and refine designs." [170]

The above definitions place very little restrictions on the medium of the prototype or the attributes of a design it tries to represent. In the software engineering literature, prototypes are often defined more narrowly as working models, created in the same software medium as the final deliverable. Lichter writes that "Prototyping involves producing early working versions ('prototypes') of the future application system and experimenting with them" [166]. Connell and Schafer explicitly distinguish software prototypes from — in their view insufficient — other modeling media: "A software prototype is a dynamic visual model providing a communication tool for customer and developer that is far more effective than either narrative prose or static visual models for portraying functionality." (quoted in [208])

In contrast to the software engineering focus on producing functional software, Rettig [211] and Wong [255] advocate that user interface prototypes should not be constructed in software in early project stages. Both argue for low-fidelity paper-based prototypes. To better understand this multitude of viewpoints, this section summarizes prior publications about prototypes and prototyping in design in general, and within HCI and software engineering in particular.

2.2.2 BENEFITS OF PROTOTYPING

Are there concrete, measurable, defensible benefits of using a prototyping-driven design approach, as opposed to a more linear approach, *e.g.*, the waterfall model? This section reviews experimental and theoretical arguments for the benefit of prototyping.

2.2.2.1 Quantifying the Value of Prototyping

The ideal experimental result in favor of prototyping would be that prototyping leads to better design outcomes. However, operationalizing design quality in experimental settings is difficult and isolating the impact of prototyping has proven to be problematic for real-world design tasks. The most concrete result to date is reported by Dow et al. [72] who found that for a constrained experimental design task with a time limit and a concretely measurable outcome, participants who built early prototypes and iterated outperformed those who did not prototype. Dow's experimental task was the mechanical engineering egg-drop exercise — participants are asked to create a vessel that protects a raw egg from a vertical fall and subsequent impact, using a limited set of everyday materials. In a between-subjects design, the treatment group, which had to build a testable prototype early and was forced to iterate on that prototype, outperformed the control group, in which prototyping was not encouraged. In particular, novices unfamiliar with the task who prototyped performed as well as experts who did not prototype.

In the absence of other strong experimental results, a frequently cited benefit by proponents is that prototyping leads to earlier identification of problems and blind alleys, when it is still feasible to fix them. McConnell summarizes several studies that have shown that for software defects, the cost of finding an error increases by an order of magnitude for each product phase [183:29], and it appears reasonable to extrapolate similar costs to usability and user experience problems.

2.2.2.2 Cognitive Benefits of Prototyping

Research in Cognitive Science suggests that the construction of concrete artifacts — prototyping — can be an important cognitive strategy to successfully reason about a design problem and its solution space. This section presents some arguments for the cognitive benefits of prototyping.

ARGUMENT 1: WE KNOW MORE THAN WE CAN TELL.

Embodied cognition theory argues that thought (mind) and action (body) are deeply integrated and co-produce learning and reasoning [59,60,61]. In this view, "thinking through doing" — engaging with ideas on a tangible level — is a more successful strategy for design than thinking hard about the problem alone. Why might this be the case?

Polanyi argues that much of our expertise and skill are "action-centered" and as such not available to explicit, symbolic cognition. Polanyi introduced the term *tacit knowledge* to

describe such expertise. A well-known example is the problem of describing to someone else how to ride a bicycle. Riding a bike is an action-centered skill, one gained through repeated practice and one only accessible as an action in the context of sitting on a bike. Practicing designers such as Moggridge [189] argue that much knowledge in design is tacit and that designers therefore need to create concrete artifacts to express their tacit knowledge [207].

ARGUMENT 2: COGNITIVE ACTIVITY EXTENDS INTO OUR ENVIRONMENT.

Proponents of *distributed cognition* argue that what is cognitive extends beyond the individual and encompasses the environment, artifacts and other people [123,130]. Hutchins describes in detailed case studies how people solve hard problems by offloading tasks into appropriate artifacts in their environment. For example, medieval navigation was aided by the Astrolabe; airline navigation is a task distributed between pilot, co-pilot, and instruments.

In this view, designers need concrete artifacts such as prototypes to be more effective in their reasoning. Along the same lines, Hutchins also argues that "material anchors" help stabilize conceptual knowledge [131]: "Reasoning processes require stable representations of constraints. [...][T]he association of conceptual structure with material structure can stabilize conceptual representation."

ARGUMENT 3: ACTIONS IN THE WORLD CAN OUTPERFORM MENTAL OPERATIONS

Kirsh and Maglio introduced a distinction between pragmatic and epistemic actions [143]: pragmatic actions are those that advance us toward a known goal; epistemic actions in contrast uncover more information about the goal. Kirsh and Maglio showed, through a study of Tetris players, that external actions in the world can be faster or more efficient than mental operations. Their study measured the amount of piece rotations performed by novice and expert Tetris players, and found that experts rotated their pieces more frequently. Why? Because the cost of performing the rotation in the game and then visually comparing the shape of the piece with the shape of open gaps on the board was faster than mentally rotating and checking for fit. Similar results have been found for the game of Scrabble, where expert players rearrange their set of letters to help them reason about possible words that can be formed with that set. Constructing concrete prototypes could thus be faster than trying to reason about a design problem in the abstract.

2.2.2.3 Reflective Practice: The Value of Surprise

Schoen introduced the concept of *reflective practice* to describe designers' activity during visualization and prototyping [221]. Reflective practice is the repeated framing and evaluation

of a design challenge by working it through, rather than thinking it through. For Schoen, successful product and architectural designs result from a series of "conversations with materials." Here, the "conversations" are interactions between the designer and the design medium — sketching on paper, shaping clay, building with foam core. The production of concrete prototypes provides the crucial element of surprise, unexpected realizations that the designer could not have arrived at without producing a concrete manifestation of her ideas. Schoen terms this element of surprise "backtalk". The backtalk that artifacts provide helps uncover problems or generate suggestions for new designs.

2.2.2.4 Prototyping as a Teaching Technique

Prototyping has also been considered teaching technique that seeks to instill better design intuitions over time [136]. By continually forcing designers to be faced with the consequences of their actions through prototype testing, they are held accountable for their ideas. Designers thus develop a better sense for which ideas work and which do not.

2.2.3 THE PURPOSE OF PROTOTYPING — DESIGN PERSPECTIVES

What questions do prototypes answer? When and how should they be constructed? This section summarizes arguments from product design and human-computer interaction research. The subsequent section will present contrasting arguments from software engineering.

2.2.3.1 What Do Prototypes Prototype?

Houde and Hill [126] classified ways in which prototypes can be valuable to designers. Prototypes in their view include "any representation of a design idea, regardless of medium." Their model defines three types of questions a prototype can address: the *role* of a product in the larger use context; its *look and feel*; and its technical *implementation*. These questions are set up as end points in a triangular, barycentric coordinate design space into which prototypes are plotted (Figure 2.2).

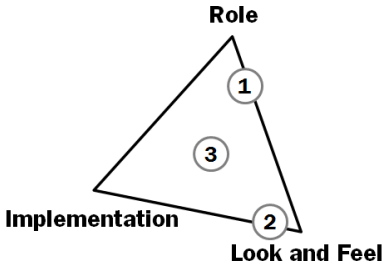

Figure 2.2: The Houde & Hill model distinguishes *Role*, *Implementation*, and *Look and Feel* functions of prototypes.

Role refers to questions about the function that an artifact serves in a user's life—the way in which it is useful to them. *Look and feel* is concerned with questions about the concrete sensory experience of using an artifact—what the user looks at, feels and hears while using it. Implementation refers to algorithms and engineering techniques used to realize functionality of a product — "the 'nuts and bolts' of how it actually works."

For reasons of economy, any given prototype will only address some of these aspects, or prioritize some over others. For example, a video clip that shows a "commercial" of an envisioned product in use would prioritize its role (Figure 2.2–1); a screen mockup of a new graphics application showing menus and toolboxes would prioritize look and feel (Figure 2.2–2); while a demonstration of algorithms required for that graphics application would prioritize implementation. Prototypes that strive to strike a balance and address all three questions are labeled "integration prototypes" (Figure 2.2–3). Such prototypes most closely approximate the final design and permit testing of the overall user experience, but are also most resource intensive to construct.

2.2.3.2 Experience Prototyping

Buchenau and Suri [52] introduced the term "Experience Prototyping" to refer to prototyping activity that enables stakeholders to gain first-hand experiential understanding of either design problems or of proposed solutions. An example of such a prototype given by the authors is wearing gloves while operating a consumer electronics device to experience the reduced dexterity of older adults. Experience prototypes focus on direct active bodily involvement of the designer or client in a constructed situation. Three uses for experience prototyping are described: understanding existing use; exploring future situations; and communicating designs to others.

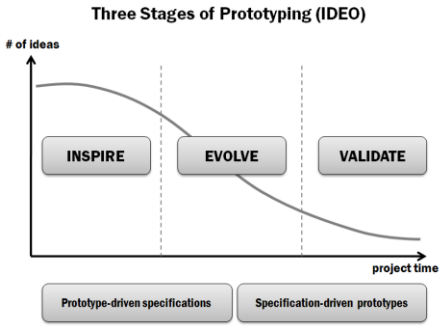

Figure 2.3: The IDEO three-stage model of prototyping: as a design project progresses, the number of entertained ideas decreases, and prototypes turn from inspiration tools to validation tools. Diagram redrawn by the author.

To understand existing situations that call for better design solutions, experience prototyping may involve role playing to gain empathy for target users. As an example, the authors cite the redesign of a remote control interface for an underwater camera vehicle. The existing experience was prototyped by one designer "playing" the vehicle with a shoulder mounted camera, and another designer yelling commands ("move up") and watching the video feed on a television monitor.

To explore future situations, Buchenau and Suri advocate creating multiple concrete artifacts or repurposing found artifacts and everyday objects. Designers on the project team have enough shared context to interpret these objects as stand-ins for future artifacts. For example, a pebble might be used to suggest a handheld wireless controller. However, if exploration requires input from external users, experience prototypes may have to be more specific and functional, as end users don't share the same background or conceptual framework with the team.

When communicating design solutions to clients and other external parties through prototypes, the intent is frequently to persuade. Such prototypes are often polished and complete and can take on the role of a "living specification." The authors caution that prototypes that succeed in conveying a complete experience can easily be mistaken to be a complete product.

2.2.3.3 Inspiration, Evolution, Validation

In personal communication, Hans-Christoph Haenlein, Director of Prototyping at IDEO, the prominent Bay Area design consultancy, described a company-internal three stage view of prototyping (Figure 2.3) [100]. In the beginning of a project, many parallel prototypes are

generated to get inspiration. Here, prototypes are often very dissimilar from each other to explore fundamentally different design options. Later on, a smaller number of ideas are iteratively evolved to resolve more focused design questions. Through both phases, project specifications are derived from the prototypes. Towards the end of a project, very complete prototypes are built to validate the design specification as a whole. Haenlein also makes an explicit distinction between prototypes used internally by the design team for exploration, and prototypes created for communicating design insights to external clients and other stakeholders.

Buxton [55] draws a distinction between *sketches* and *prototypes*. For him, sketches are "quick, timely, inexpensive, disposable, plentiful"; "they suggest and explore rather than confirm". Prototypes in contrast are "didactic, they describe refine, answer, test, resolve; they are specific and are depictions" [55:140]. While the distinction in nomenclature is unique to Buxton, the expressed difference between prototypes used for inspiration and those used for experimentation, evolution and validation matches the IDEO model.

2.2.3.4 Prototyping as Inquiry

Gedenryd stresses that prototypes are "inquiring materials", that is, materials with a cognitive purpose [84]. Many prototyping approaches all share the underlying goal to envision the future situation of the designed artifact in use — prototyping is thus a "situating strategy". Echoing distinctions drawn by Haenlein and Buxton, Gedenryd distinguishes between exploratory prototypes used to familiarize oneself with the problem, and experimental prototypes, which probe and test specific design hypotheses. He further distinguishes between horizontal relevance (breadth) and vertical relevance (depth) of the functionality explored in a prototype.

As a guideline, Gedenryd advocates that prototypes exhibit a minimalist approach: "A good prototype serves its purpose as a basis of inquiry and interactive cognition, while being simple to create. This means that it should have the properties required for its purpose, and as few other properties as possible. It also means that relevance is always relative to just what exactly a prototype will be used for; this determines what properties it will need to have." [84:165]

2.2.3.5 Low-Fidelity Prototypes Might Be Preferable

Rettig [211] and Wong [255] argue that the resolution or fidelity of a user interface prototype should match the level of detail of the questions asked of the prototype. In particular, Rettig

advocates against building functional software prototypes of user interfaces early on because their surface finish is too high at a time when the general resolution of the project is still low. According to Rettig, building functional UI prototypes ("high-fidelity prototypes") early on squanders design resources and yields the wrong kind of feedback. Particularly, Rettig cites four problems:

1) High-fidelity prototypes take too long to construct and modify.
2) Testers of the prototype are lead to comment on surface attributes such as typography and alignment, when those are not the attributes tested.
3) The act of constructing a high-fidelity prototype creates emotional investment by developers in that prototype, which results in resistance to act on feedback that asks for fundamental changes. Similarly, a high-fidelity prototype creates expectations by users exposed to the prototype that may be hard to change later.
4) High-fidelity prototypes are too brittle and have no graceful "repair strategies" if users run into bugs.

As an alternative, Rettig proposes paper prototyping of user interfaces, where interfaces are assembled out of different layers of cut out paper strips. A designer simulates the logic of the application by rearranging paper strips. Wong is also concerned with the fidelity of UI prototypes and suggests taking inspiration from graphic design by creating "rough" UI prototypes through sketching and omission of concrete details.

One fundamental shortcoming of paper-based UI prototyping is that the human "computer" who rearranges UI elements fundamentally changes the experience of interface dynamics. While useful for exploring questions of interface layout, content, and structure, paper prototypes are therefore less useful for exploring interactive behaviors in user interfaces.

2.2.4 THE PURPOSE OF PROTOTYPING — SOFTWARE ENGINEERING PERSPECTIVES

This section summarizes publications on prototyping from outside the field of human-computer interaction and product design. Not surprisingly, software engineering prototypes are more frequently concerned with testing implementation strategies than user experience. However, the software engineering literature also departs from human-computer interaction publications on prototyping in additional ways: prototypes are frequently seen as early version of the final software, rather than standalone artifacts to be discarded after testing. In

addition, more emphasis is placed on capturing and documenting what questions a prototype explored, and what was learned from it.

2.2.4.1 Exploration, Experimentation, Evolution

Floyd [79], in an early workshop on prototyping for complex software systems, describes two primary goals of prototypes: 1) functioning as "learning vehicles" and 2) enhancing communication between developers and users, as developer introspection of user needs often leads to inadequate products.

For Floyd, a software prototype must be functional enough to be demonstrated to users with "authentic, non-trivial tasks." That functionality may either be implemented, or simulated. In either case, Floyd assumes that for complex software projects, resource constraints only permit one such prototype to be built and tested at a time. Floyd also claims that by demonstrating a prototype to users, their expectations of the final system are "deeply influenced" so that the designer is committed to the overall outline of the prototype. This places the designer in a paradoxical situation: prototypes are constructed to learn, but their very construction constrains the extent to act on what was learned by modifying the design. This paradox may have been an artifact of the types of applications considered — custom software written for individual clients, so that the prototype testers and final users are identical.

Three different purposes of prototyping are distinguished by Floyd (Table 2.1): *exploration* (clarifying requirements, discussing alternatives), *experimentation* (measuring how adequate a proposed solution is), and *evolution* (adapting an existing system to changing requirements). Floyd suggests that prototypes should be expanded into the target system or integrated into it — that is, the prototype is an earlier version of the final product. This implies using similar production tools for the prototype as for the final deliverable and thinking about modularity, both of which may require more time and expertise than the "quick and dirty" prototypes

Approach	Purpose	Topic of Investigation
Explorative	Elicit requirements, determine scope and different alternatives of computer support	Requirements
Experimental	Try out technical solutions to meet requirements	Particular solutions
Evolutionary	Continually adapt a system to a rapidly changing environment.	Evolving requirements

Table 2.1: Three purposes of prototypes according to Floyd [79] (table redrawn from Schneider's summary [220].)

advocated by designers, which are created with the expectation of being discarded.

2.2.4.2 Prototypes as Immature Products

Riddle [212] states that "prototyping is an approach to software development that emphasizes the preparation of immature versions that can be used as the basis for assessment of ideas and decisions." Riddle identifies two "dimensions of immaturity" along which a prototype may fall short of complete software: a prototype may offer less than a final, polished system in terms of *quality* (response time, maintainability, robustness), or in terms of *functionality*. While prototypes should be produced quickly, Riddle also stresses that a rational, controlled approach to prototype development is needed to preserve modifiability and understandability of the produced code, which suggests that the implementation of the prototype should be integrated into the main production codebase to some degree. Finally, since prototypes are constructed for assessment, Riddle argues that tools should also provide ways to instrument prototypes to gather pertinent usage data automatically.

2.2.4.3 Presentation Prototypes, Breadboards, and Pilot Systems

Lichter et al. [166] present case studies of prototype use in industrial software development and introduce a taxonomy that distinguishes kinds of prototypes, goals of prototypes, and prototype construction techniques. Four different kinds of prototypes are distinguished, based on the phase of software development they support:

1) A *presentation prototype* is used as a persuasive tool to convince a client of the feasibility of a project before starting major work on it. Other authors also describe prototypes as persuasive tools, but usually as the outcome of some design process, not its precursor.
2) A *prototype proper* is a "provisional operational software system" that is limited to specific parts of the user interface or implementation.
3) A *breadboard* is designed to clarify implementation problems for the development team and does not usually involve end-user feedback.
4) A *pilot system* is any software not constructed specifically for experimentation or communication, but part of the core project being developed (e.g., an alpha version).

The purposes of prototyping are adapted from Floyd (exploratory, experimental, and evolutionary). Construction techniques are distinguished based on whether functional coverage is horizontal across application layers (e.g., user interface only, database only) or vertical (e.g., implementing all aspects touched by the shopping cart in an ecommerce

system). Lichter et al.'s review of five real-world case studies showed little consistency in the selection of prototyping strategies in the surveyed companies.

2.2.4.4 Capturing and Sharing Knowledge Gained from Prototypes

Schneider [220], in investigating the role of prototypes in software engineering, lamented that frequently, no systematic effort is made to capture and share the knowledge gained from developing and testing prototypes. Because prototypes only examine particular details of a future product, they often cannot stand alone and require their developers' explanation to clarify context and scope: "The prototype itself is not well suited to indicate what it does well or poorly". Schneider therefore argues that the right level of analysis is the "developer-prototype system" since only the two together can fully capture intent and meaning. Documentation for each prototype should thus be systematically captured through design tools.

2.2.5 SYNTHESIS OF THE SURVEYED MATERIAL

Given the previous review of both human-computer interaction and software engineering literature on prototyping, we can now combine the various presented perspectives in to a single framework that addresses purpose, aspects, and functionality of user interface prototypes. For this dissertation, we will define a user interface prototype as *a concrete artifact that can be experienced by a user as if it possessed some or all of the interactive qualities of the envisioned interface, constructed for the purpose of generating feedback*.

Three high-level goals why designers prototype have been presented (Figure 2.4): First,

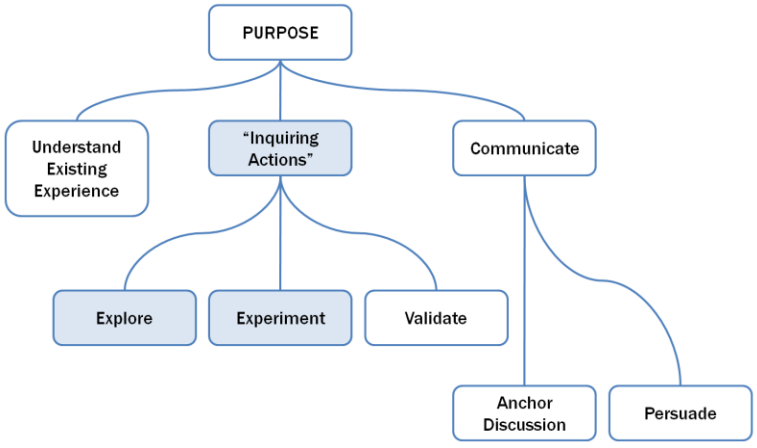

Figure 2.4: Why are prototypes constructed in design?

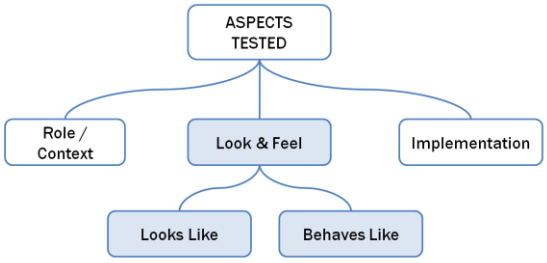

Figure 2.5: What aspects of a product can prototypes approximate?

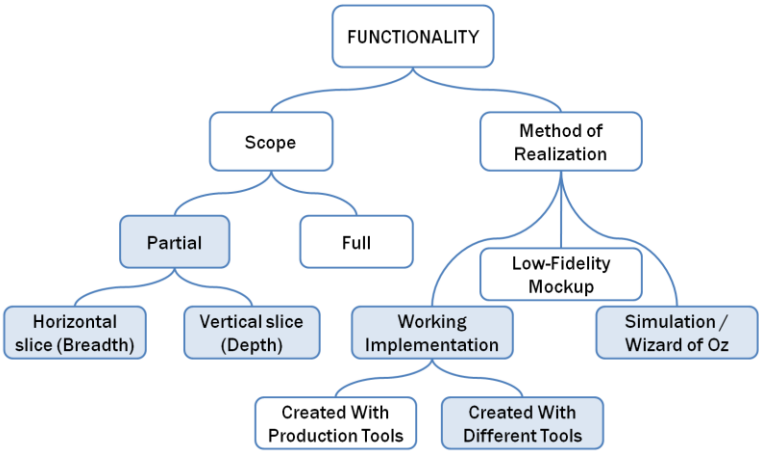

Figure 2.6: What kind of functionality can prototypes exhibit?

prototypes are built to give the designer experiential insight into some situation that already exists [52]. Second, prototyping is a technique to gain information about possible future situations [84]. As described by Floyd [79] and Haenlein [100], this stage of prototyping can have three different goals: to explore the space of alternatives, to conduct more focused experiments comparing two or more options, and to get real-world validation. Third, prototypes are used to aid communication between different project stakeholders with different "languages." Within a design team, experts with different realms of expertise use prototypes to serve as boundary objects [233] that can bridge language differences and serve as a common referent in discussion. For communication with clients, prototypes are frequently constructed to persuade the client.

Three different aspects of a final product can be tested in a prototype (Figure 2.5), as described in Houde & Hill [126]: The role a current or future product plays for a users; the look and feel of the product, and its implementation strategies. Within the category of look and feel, designers further distinguish between "looks like" prototypes that express the

aesthetic, visual, and material qualities of a product, and "works like" prototypes that exhibit interactive behaviors.

Works-like prototypes can either exhibit full functionality, or limit functionality by selecting a horizontal or vertical slice of behavior (Figure 2.6). The functionality in a works-like prototype may or may not share implementation strategies and tools with the final product. Thus, four different realization methods are possible: building a working implementation with the same toolset as the final product; building a working implementation with a different toolset specifically geared towards prototyping; creating a lower-fidelity approximation; or simulating the functionality.

The prototyping tools in this dissertation support the creation of a specific subset of prototypes (shown through shading in Figure 2.4–Figure 2.6). The introduced tools focus on prototypes created to explore design options or test specific ideas through experiments; these prototypes have working interactive behaviors, but are not necessarily comprehensive and are expressed in a new, prototype-specific tool, rather than in production-ready code.

With this particular point-of-view established, we next review related prior research into authoring techniques and systems.

CHAPTER 3 RELATED WORK

Authoring tools and techniques for creating user interfaces have a rich history in human-computer interaction. They have also been a commercial success — few graphical user interfaces are created without the help of UI authoring tools. This chapter first reviews the status quo of UI prototyping in industry, and then presents a survey of related research to answer three questions: What tools are interaction designers using today to prototype user interfaces? What additional tools have been introduced by prior research? What important gaps in tool support remain?

3.1 STATUS QUO: TOOLS & INDUSTRY PRACTICES TODAY

Before surveying related research, it is useful to understand which tools are used by interaction designers today. We will first review tools to build user interface prototypes, and subsequently survey tools to gain insight from those prototypes.

3.1.1 BUILDING PROTOTYPES

A wide variety of commercial applications are available for prototyping desktop user interfaces, and survey data reporting on the use of such tools by professionals is available. In contrast, few commercial applications support the creation of UIs that do not target desktop or mobile phone platforms, leading today's practitioners to appropriate other tools or build their own scaffolding for prototyping. This section reviews these two areas in turn.

3.1.1.1 Desktop-Based User Interfaces

Myers et al. [195] conducted a survey of 259 interaction designers of desktop- and web-based applications. Statistics for the most frequently used tools are reproduced in Figure 3.1. To make sense of these tool choices, consider the three different high-level tasks involved in creating a user interface prototype. Designers have to define a*ppearance* — the graphic design of static screens; i*nformation architecture* — how screens relate to each other; and *behavior* — animations and transitions. We can examine how each of the reported applications supports these three tasks:

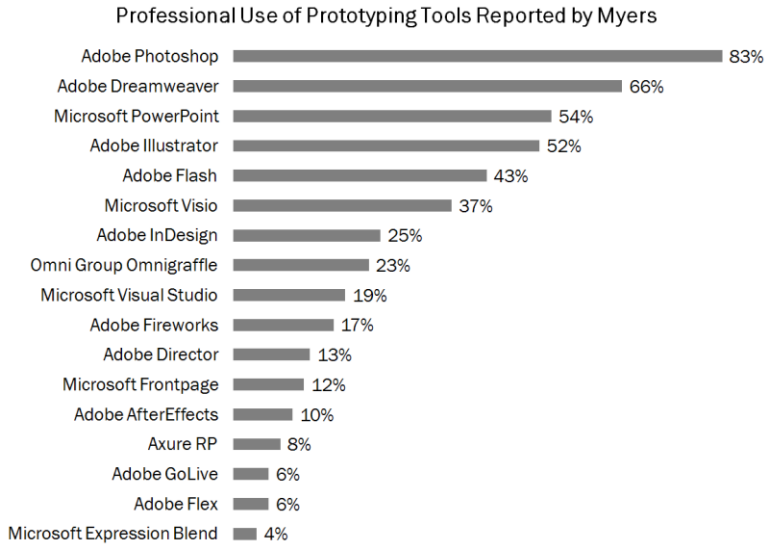

Figure 3.1: Common tools used for UI prototyping as reported in Myers' survey of interaction designers [195]. Figure redrawn by the author.

APPEARANCE

For static screen design, many designers rely either on complex graphics software for professionals (Adobe Photoshop and Illustrator) or they appropriate office productivity software with vector-graphics layout functions (Microsoft PowerPoint and Visio). It is not uncommon for interaction designers to have a background in graphic design, which gives them familiarity with professional tools. One factor favoring the use of office productivity tools may be their widespread availability on desktop computers, regardless of their suitability for the task.

INFORMATION ARCHITECTURE

To capture key interaction sequences, PowerPoint is used to create linear walkthroughs from screen to screen. Such walkthroughs can describe important paths through an interface, but they cannot capture the multiple options usually available to the user at any given point in an interface. For more complex structure, dedicated user interface construction tools such as Adobe Flash (for dynamic web applications), Adobe Director (for stand-alone applications), and Adobe Dreamweaver (for web pages) are used.

BEHAVIOR

The task of creating interactive behaviors was judged to be more difficult by Myers' respondents than creating appearance. The toolset behaviors is also more limited. Two

different kinds of dynamic behaviors are one-shot *animations* that are not dependent on user interaction once started, and *user-in-the-loop behaviors*, where continuous user input drives the behavior. One-shot animations can be prototyped using direct manipulation tools in presentation applications such as PowerPoint and in UI software such as Adobe Flash. User-in-the-loop behaviors mostly require textual programming to set up polling loops or event handlers.

It is notable that tools specifically created for the task of prototyping user interfaces, such as Axure RP, have relatively little mind- and market share with Myers' respondents. Whether this is due to a lack of perceived need for such software, or due to other factors such as pricing and marketing cannot be determined from the published data, but deserves additional attention.

3.1.1.2 Non-Traditional User Interfaces

Interfaces that target other devices than desktop PCs suffer from a relative paucity of tool support. Because smart phones are rapidly becoming the next standardized platform for software, it is useful to distinguish between interfaces for such commodity hardware, and interfaces for custom devices.

COMMODITY HARDWARE

Of the tools reported in the previous section, some support the creation of prototypes that can be tested on mobile devices: Adobe Flash can generate applications for mobile phones that run a special Flash player software; web pages and web applications can be used on a mobile phone if the target device has a suitable web browser. Mobile development platforms also often include a desktop PC-based emulator in which mobile applications can be tested without having to load the application onto a device. The downside to this approach is that the unique input affordances of the phone are lost and that it is not possible to test the prototype in realistic use contexts outside the lab. To our knowledge, no comprehensive survey about mobile prototyping techniques has been published to date.

CUSTOM HARDWARE

The commercial tools reported in Myers' survey all lack direct support for creating user interfaces with custom hardware. Creating functional prototypes of physical user interfaces involves the design of custom electronics or the creative repurposing of existing devices through "hardware hacking." In our own fieldwork with eleven designers and managers at three product design consultancies in the San Francisco Bay Area, we found that most

interaction designers do not have the technical expertise required for either approach. Where expertise exists, it is often restricted to a single individual within an organization. At two of the companies we visited, a single technology specialist would support prototyping activities by programming microcontroller platforms such as the Parallax Basic stamp [2] to the requirements of the design teams.

Two fundamentally different approaches to custom hardware are to create standalone devices that function on their own, or to create devices that are tethered in some way to a desktop PC, which can provide processing power and audio-visual output. While tethering constrains testing to the lab (or requires elaborate laptop-in-a-backpack configurations), it allows the design of prototypes without having to pre-maturely optimize for hardware limitations. Two examples of such tethered prototypes from the literature attest to their use in professional settings: Pering reports on prototyping applications for the Handspring PDA using custom hardware for input, and a PC screen for output [204] (Figure 3.2); Buchenau and Suri describe a prototype of an interactive digital camera driven by a desktop PC in [52] (Figure 3.3).

HARDWARE HACKING

In our own physical computing consulting work, we encountered requests from interaction designers to "glue" new hardware input into their existing authoring tools, for example by providing new input events to an Adobe Flash application. Such solutions are brittle since most authoring tools are built around the assumption that all input emanates from a single

Figure 3.2: Pering's "Buck" for testing PDA applications: PDA hardware is connected to a laptop using a custom hardware interface. Application output is shown on the laptop screen.

Figure 3.3: IDEO interaction prototype for a digital camera UI. The handheld prototype is driven by the desktop computer in the background.

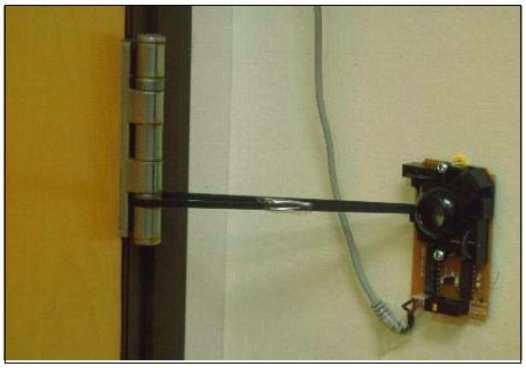

Figure 3.4: Buxton's Doormouse [56] is an example of a "hardware hack" that repurposes a standard mouse.

mouse and keyboard — the standard graphic GUI widgets are not written to interpret different kinds of input events. One approach to reusing standard GUI tools is to marshal hardware input events into mouse and keyboard events, e.g., by reusing standard keyboard and mouse electronics, but attaching different input mechanisms to them. Examples of hardware hacking for the purpose of prototyping include Zimmerman's augmented shopping cart, where a standard mouse was used to sense rotational motion of the cart (described in [108]); and Buxton's Doormouse [56], which sensed the state of an office door by means of a belt around the door hinge connected to the shaft encoder in a disassembled mouse (Figure 3.4). Hardware hacks might look appealing because they can control any existing application, but their reach is limited because of many assumptions made by operating systems and application about how input from standard devices is structured. For example, widget behavior for multiple simultaneous key presses is not well defined, and it is not easily possible to use more than one mouse in an application.

3.1.2 GAINING INSIGHT FROM PROTOTYPES

What tools are used in design practice to gain insight from prototypes? Three important aspects to consider are: support for expressing and comparing alternatives, capturing change suggestions through annotations and revisions, and capturing and analyzing feedback from user test sessions.

3.1.2.1 Considering Alternatives

Alternatives of static content such as UI layouts can be compared by showing them side-by-side on screen or by printing and pinning them to a wall. Different graphic alternatives can

also be generated using layer sets and other features in professional graphics programs. Terry reports that on a micro-level, designers use *undo* operations to explore A/B comparisons in such software [240]. While comparison of alternatives of UI appearance is feasible, Myers notes that tool support for comparing behaviors is still lacking:

"[D]esigners frequently wanted to have multiple designs side-by-side, either in their sketchbooks, on big displays, or on the wall. However, this is difficult to achieve for behaviors — there is no built-in way in today's implementation tools to have two versions of a behavior operating side-by-side." [195]

3.1.2.2 Annotating and Reviewing

To find out how interaction design teams currently communicate revisions of user interface designs, we contacted practitioners through professional mailing lists and industry contacts. Ten designers responded, and seven shared detailed reports. There was little consistency between the reported practices — techniques included printing out screens and sketching on them; assembling printouts on a wall; capturing digital screenshots and posting them to a wiki for comments; and using version control systems and bug tracking databases. We suggest that the high variance in approaches is due to a lack of specific tool support for user interface designers.

We also noted a pronounced divide between physical and digital processes [144] — one designer worked exclusively on printouts; four reported a mixture between working on paper and using digital tools; and two relied exclusively on digital tools. To make sense of this divide, it useful to distinguish between two functions: the recording of changes that should be applied to the current design (what should happen next?); and keeping track of multiple versions over time (what has happened before?). For expressing changes to a current design, five of the surveyed designers preferred sketching on static images because of its speed and flexibility. In contrast, designers preferred digital tools to capture history over time and to share changes with others. Designers would thus benefit from tools that bridge this divide and enable both fluid sketching of changes and tracking of revisions inside their digital authoring tools.

3.1.2.3 Feedback from User Tests

Evaluation strategies to assess the usability of user interface prototypes can be divided into *expert inspection* and *user testing*. In expert inspection techniques such as heuristic evaluation [199] and the cognitive dimensions of notation questionnaire [47], expert evaluators review

an interface and identify usability issues based on a pre-established rubric. The main benefit of expert inspection is its low cost.

Rubin's Handbook of usability testing [216] provides a blueprint for usability studies with non-expert users: Participants are asked to complete a set of given tasks with the device or software being tested, and are asked to vocalize their cognitive process using a think-aloud protocol [162:Chapter 5]. Sessions are video- and audio-recorded for later review and analysis.

What tools are used to record and analyze prototype evaluations in practice? A review of discussion threads on the Interaction Design Association mailing list [3] suggests that the use of specialized usability recording applications such as Silverback [4] and Morae [5] is common. Such tools record both screen output as the participant sees it, as well as video of the participant and audio of their utterances. These media streams are then composited or played back in synchrony for analysis. Some tools like Morae can also capture low-level input events, such as mouse clicks and key presses. However, the tested application is treated as a black box — no information about the application's internal state is recorded. Morae's observer software also makes it possible for the experimenter to add indices to the video as it is being recorded. These features echo functionality described in d.tools video suite and appeared roughly simultaneously. We became aware of them after our research was completed.

Usability recording tools are predominantly targeted at the evaluation of desktop UIs. Methods for testing mobile and custom device prototypes are less established. Video-recording the screen of a mobile device using either over-the-shoulder, head-mounted, or device-mounted cameras has been reported in mailing list discussions. Detailed interaction meta-data is usually not available for these approaches.

We next turn to a review of related research.

3.2 UI PROTOTYPING TOOLS

Prior research has introduced tools aimed at constructing and testing prototypes for particular types of user interfaces (e.g., desktop, mobile, speech) and for specific functionality exhibited by these interfaces (e.g., location awareness, animation). Research prototyping tools are often based on fieldwork with groups of target designers and seek to strike a balance between preserving successful elements of existing practice and introducing new functionality to aid or enhance the authoring process. Generally, these tools offer the following three benefits:

1) They decrease UI construction time.
2) They isolate designers from implementation details.
3) They enable designers to explore a new interface technology previously reserved to engineers or other technology experts.

While many prototyping tools target design professionals, the offered benefits also match the characteristics of successful end-user toolkits reported by von Hippel and Katz:

1) End-user toolkits enable complete cycles of trial-and-error testing.
2) The "solution space" of what can be built with the tools matches the user's needs.
3) Users are able to utilize the tools with their existing skills and conceptual knowledge.
4) The tools contain a library of commonly used elements that can be incorporated to avoid re-implementing standard functionality.

To facilitate comparison between the different systems reviewed in this section, Table 3.1 summarizes characteristic features and approaches, while Figure 3.5 provides a historical timeline.

	Target Platform					Authoring Techniques							UI Aspects Covered			
System	Desktop GUI	Web GUI	Mobile UI	Speech UI	Pen-based UI	Sketching	Widget Layout	Visual Programming / Sotryboards	Textual Programming	Wizard of Oz	Tangible Authoring	Programming by Demonstration	Appearance - Low Fidelity	Appearance - High Fidelity	Information Architecture	Dynamic Behavior
Hypercard	✓						✓		✓					✓	✓	✓
SILK	✓					✓		✓					✓		✓	
DENIM		✓				✓		✓					✓		✓	
Designers' Outpost		✓				✓		✓			✓				✓	
DEMAIS	✓		✓			✓		✓					✓		✓	✓
Topiary						✓		✓					✓			✓
Monet	✓					✓						✓				
K-Sketch	✓					✓						✓				
SUEDE				✓				✓		✓					✓	
Activity Designer			✓					✓				✓	✓			✓
Mixed-Fidelity Tool			✓					✓					✓	✓	✓	
Sketch Wizard					✓		✓			✓			✓			
Flash Catalyst	✓	✓					✓							✓	✓	✓

Table 3.1: Comparison of prior research in UI prototyping tools.

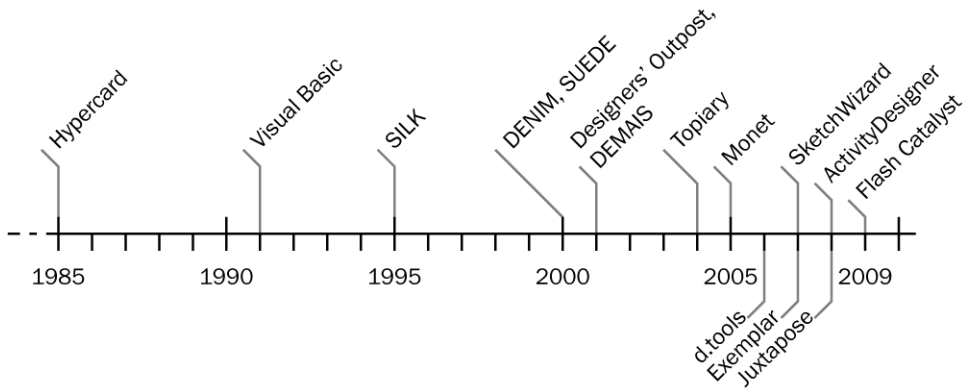

Figure 3.5: Timeline of prototyping tools for graphical user interfaces.

HYPERCARD & VISUAL BASIC

The first successful UI prototyping tool is probably Atkinson's HyperCard [36]. HyperCard enables rapid construction of graphical user interfaces on the Macintosh computer by introducing the notion of cards. A card contains both data and a user interface, composed of graphics, text, and standard GUI widgets. The user interface can be created through direct manipulation in a GUI editor. Different cards make up a stack; the links between cards in the stack is authored in HyperTalk, a scripting language. HyperTalk's legacy can be found in other applications that combine a direct manipulation GUI editor with a high-level scripting language, e.g., Visual Basic [63] and Adobe Director[6] and Flash[1]. One challenge such applications have faced is the constant pull to turn into a more complete, secure, robust development platform. Feature creep and software engineering abstractions progressively raise the threshold of expertise and time required to use the environment such that it becomes less and less suitable for rapid prototyping.

SILK

Landay's SILK system [155,156] introduced techniques for sketching graphical desktop user interfaces with a stylus on a tablet display. Stylus input preserves the expressivity and speed of paper-based sketching, while adding benefits of interactivity. A stroke recognizer matches ink strokes to recognize common widgets, which can then be interacted with during a test. To capture the information architecture of an interface, multiple screens can be assembled into a storyboard; transitions from one page to another can be initiated by the drawn widgets.

DENIM

Lin et al.'s DENIM [171] builds on the techniques introduced in SILK to enable stylus-based prototyping of (static) HTML web sites. Users draw pages and page elements on a 2D canvas

and establish hyperlinks by drawing connecting links between pages. DENIM adds semantic zooming — hiding and revealing information based on a global level-of-detail setting — to move between overview, sitemap, storyboard, sketch, and detailed drawing.

DESIGNERS' OUTPOST

The Designers' Output [147] also targets prototyping of web site information architectures, but focuses on design team collaboration, rather than individual work. Fieldwork uncovered that information architecture diagramming frequently takes place by attaching paper post-it notes to walls to facilitate team discussion. Respecting the physical aspects of this practice, Outpost introduces a large vertical display to which notes representing pages can be affixed. Notes are tracked and photographed using a computer vision system and a high-resolution still camera. Links between pages are authored using a digital pen, so the hierarchy model can be captured digitally.

DEMAIS

Bailey's DEMAIS [40] contributes a visual language to author dynamic behaviors through stylus marks (Figure 3.6). DEMAIS focuses on interaction with audio and video elements embedded in the user interface. It combines connections between different screens in a storyboard editor, similar to SILK and DENIM, with behaviors within a screen that can be authored with "behavioral ink strokes." The visual language for these ink strokes allows expression of source events (e.g., mouse click, mouse rollover, and elapsed time) and actions (navigational control of audio/video elements, show/hide elements).

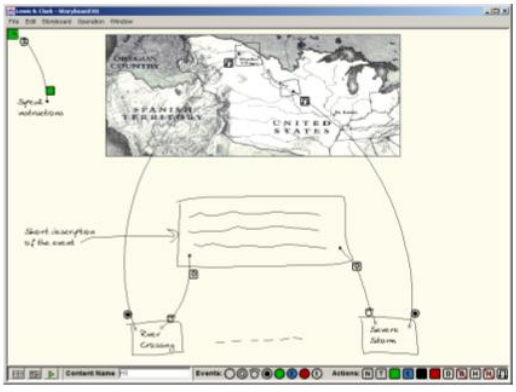

Figure 3.6: Bailey's DEMAIS system introduced a visual language for sketching multimedia applications [40].

Figure 3.7: Li's Topiary system for prototyping location-aware mobile applications [163].

TOPIARY & BRICKROAD

Topiary [163] contributes authoring techniques for prototyping location-based applications for handheld devices. In addition to the customary storyboard, it adds an "active map" view to the authoring tool, where users can model the location of places (regions on the map), objects, and people (Figure 3.7). Designers can then add actions that should be executed when the relation between places, objects and people change. Applications can be tested without requiring the designer to physically move using a Wizard of Oz interface [140], in which the designer can move people and objects on a map and see the resulting changes in the mobile interface a user would see. BrickRoad [174] expands on the role the Wizard can play by enabling real-time composition of mobile application output based on a visualization for the Wizard where the mobile device user is located at a given moment.

MONET & K-SKETCH

Where many of the other tools surveyed thus far are concerned with high-level interaction logic and information architecture, Monet [164] introduces techniques for prototyping continuous, user-in-the-loop graphical behaviors. Users sketch the interface appearance on a tablet PC and then demonstrate how the appearance should change during user interaction. K-Sketch [68] introduces interaction techniques for rapidly authoring animations. Its stylus controlled interface enables users to sketch graphics, and express animation of rotation, translation, and scale by demonstration. Motivated by the insight that too many features slow down the authoring process and raise the threshold for non-expert animators, K-Sketch introduces an optimization technique that seeks to find the minimum feature set in the authoring tool that satisfies the greatest number of possible use cases.

MAESTRO

Maestro [7] is a commercial design tool for prototyping mobile phone interactions. It provides a complex visual state language with code generators, software simulation of prototypes, and compatibility with Nokia's Jappla hardware platform. Maestro and Jappla together offer high ceiling, high fidelity mock-up development; however, the complexity of the tools make them too heavyweight for informal prototyping activities. The availability of such a commercial tool demonstrates the importance of prototyping mobile UIs to industry.

ACTIVITY DESIGNER

The ActivityDesigner tool [165] supports prototyping applications that respond to and support user activities, where an activity is defined as long-term transformation process towards a motivation (e.g., staying fit) that finds expression in various concrete actions.

ActivityDesigner distinguishes itself from other tools by not treating screens or UI states as the top-level abstraction. Instead, it introduces situations (location and social context), scenes (pairs of situations and actions), and themes (sets of related scenes). Prototyping applications thus involves both modeling of context through these abstractions as well as concrete authoring of application behavior.

MIXED-FIDELITY PROTOTYPING

De Sa's Mixed-Fidelity Prototyping tool [217] offers support for building prototypes of mobile applications at different levels of resolution. The most rapid way is to show single images of sketched user interfaces on the device; user interaction, e.g., stylus taps on the screen, are relayed back to a wizard, who can then select the next screen to show. Interaction logic can also be created using node-link diagrams and a library of widgets so wizard action is not required.

SKETCH WIZARD

Sketch Wizard [69] proposes to accelerate prototyping of pen-based user interfaces (those that rely on stylus input and handwriting) by providing a Wizard with a powerful control interface tailored to the pen-input domain. The end-user who interacts with a tablet application prototype can provide free-hand input on a drawing canvas. That drawing canvas is also shown to a Wizard on a desktop PC, who can modify the user's strokes, delete them, or add new content in response to the user's input. The main contribution of this work is the design of the control interface that enables designers to provide quick responses to stroke-based user input, which could be intended as text, drawing, or commands.

ADOBE FLASH CATALYST

Adobe Flash Catalyst [8], while not an academic research project, is worth including in this summary because it represents the latest commercial product specifically aimed at prototyping graphical user interfaces. Flash Catalyst uses states or frames as the top-level abstraction, as many of the other authoring environments reviewed in this section. However, different states are not laid out in a node-link diagram with transitions. Rather, transitions are listed in a table. Each transition can then be associated with animation commands for graphical elements in the source and destination states.

SUMMARY

A number of themes emerge from the review of related UI prototyping tools. Early UI design tools introduced a combination of direct manipulation UI layout editors with high-level

scripting languages for behavior programming. While successful in commercial tools such as HyperCard and Visual Basic, use of scripting languages has not been a focus in research prototyping tools. Many research tools use storyboards as an authoring abstraction. A frame or screen in such a storyboard corresponds to a unique screen in a user interface. To capture the UI architecture, relationships between storyboard frames are frequently expressed as node-link diagrams. Several tools rely on sketch-based input to define both UI contents as well as visual diagrams for UI logic. Finally, a recurring question across tools is to what extent functionality should be implemented (through script or diagrams) versus simulated (through Wizard of Oz techniques).

d.tools adopts successful choices from prior work such as the use of visual storyboards. It goes beyond purely visual authoring by enabling scripted augmentations to storyboard diagrams. All discussed tools assume some fixed hardware platform with standardized I/O components. d.tools and Exemplar move beyond commodity platforms by supporting flexible hardware configurations and definitions of new interaction events from sensor data. The Juxtapose project takes a different approach by building directly on top of ActionScript, the procedural language used by Adobe Flash; it investigates how to support the exploration of multiple interaction design alternatives expressed entirely in source code.

3.3 TOOL SUPPORT FOR PHYSICAL COMPUTING

The previous section reviewed prototyping tools for desktop, web, and mobile user interfaces. A separate set of research has enabled experimentation in physical computing with sensors and actuators. These systems have focused less on supporting professional designers, perhaps because the design of such user interfaces is not an established discipline yet. Greenberg argues that toolkits in established design areas, such as GUI design, play a different role from toolkits in emergent areas of technology, such as ubiquitous computing [91]: "Interface toolkits in ordinary application areas let average programmers rapidly develop software resembling other standard applications. In contrast, toolkits for novel and perhaps unfamiliar application areas enhance the creativity of these programmers." Table 3.2 provides a feature comparison of the physical computing systems reviewed in this section, while Figure 3.8 presents a historical timeline.

System	Tethering			I/O					Authoring		App Logic		Domain		Target Audience		
	Wired tether to PC	Wireless tether to PC	Device operates independently	Discrete Input	Continuous Input	Id Input	Camera Input	Actuation	Textual Programming	Visual Programming	Build custom app logic	Remote control existing apps	User Interfaces	Robotics	Designers	Programmers	Hobbyists / Students
Phidgets	✓			✓	✓	✓		✓	✓		✓		✓			✓	
Calder		✓		✓		✓			✓		✓		✓		✓	✓	
iStuff		✓		✓	✓	✓		✓	✓		✓		✓			✓	
iStuff Mobile		✓		✓	✓	✓			✓	✓	✓		✓			✓	
Lego Mindstorms			✓	✓	✓		✓	✓		✓	✓			✓			✓
Bug Labs BUG			✓	✓	✓		✓		✓		✓		✓			✓	
DART							✓		✓		✓		✓		✓	✓	
Arduino	✓	✓	✓	✓	✓	✓		✓	✓		✓		✓				✓
Thumbtacks	✓			✓						✓		✓	✓		✓		
Papier Mache	✓						✓		✓		✓		✓			✓	
EyePatch	✓						✓			✓	✓		✓			✓	✓

Table 3.2: Comparison of prior research in physical computing tools.

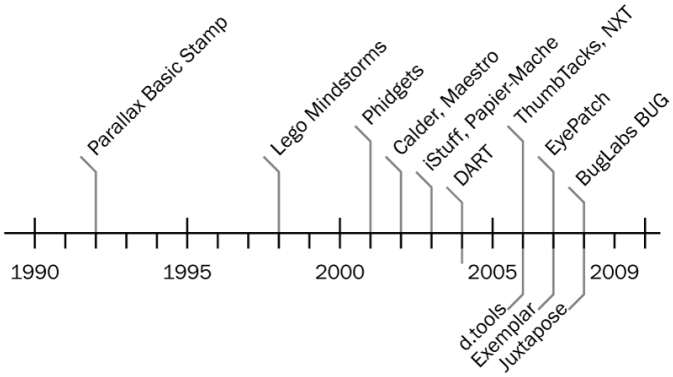

Figure 3.8: Timeline of selected physical computing toolkits.

BASIC STAMP

The Basic Stamp [2] represents an early attempt at making embedded development accessible to a broad range of users. Instead of writing firmware for a microcontroller in a low-level language like Assembly, Basic Stamp developers write programs in a high-level BASIC dialect, which is then interpreted on the Basic Stamp chip. The Stamp is successfully used to teach electronics and programming fundamentals in secondary schools, and has also found its way into product design studios, as reported by Moggridge [189]. The Basic Stamp is geared towards creating standalone devices and is not powerful enough to handle graphics, limiting its utility for modern interfaces that combine graphics output with novel input devices.

PHIDGETS

The Phidgets [93] system introduced physical widgets: programmable ActiveX controls that encapsulate communication with USB-attached physical devices, such as switches, pressure sensors, or servo motors. Phidgets abstracts electronics implementation into an API and thus allows programmers to leverage their existing skill set to interface with the physical world. In its commercial version, Phidgets provides a web service that marshals physical I/O into network packet data, and provides several APIs for accessing this web service from standard programming languages (e.g., Java and ActionScript). d.tools shares much of its library of physical components with Phidgets. In fact, Phidgets analog sensors can be connected to d.tools. Both Phidgets and d.tools store and execute interaction logic on the PC. However, d.tools differs from Phidgets in both hardware and software architecture. First, d.tools offers a hardware extensibility model not present in Phidgets. Second, on the software level, d.tools targets prototyping by designers, not development by programmers. The d.tools visual authoring environment contributes a lower threshold tool and provides stronger support for rapidly developing the "insides of applications" [191]. Finally, Phidgets only addresses the design part of the design-test-analyze cycle — it does not offer support for testing or analyzing user test data.

CALDER

Calder [37,159] makes RFID buttons and other wired and wireless devices accessible in C and the Macromedia Lingo language. The small form factor of Calder input components facilitate their placement on physical prototypes; Calder also describes desirable mechanical attachment mechanisms and electrical properties (e.g., battery-powered RF transceivers) of prototyping components. Like Phidgets, Calder's user interface is a textual API and only supports the creation of prototypes, not testing or exploration of alternatives.

ISTUFF & ISTUFF MOBILE

iStuff [43] contributes an architecture for loose coupling between input devices and application logic, and the ability to develop physical interactions that function across different devices in a ubiquitous computing environment. iStuff, in conjunction with the Patch Panel [44], enables standard UIs to be controlled by novel inputs. iStuff targets room-scale applications.

iStuff Mobile [42] introduces support for sensor-based input for mobile phone applications through a "sensor backpack," attached to the back of the mobile device. Since most mobile phones do not permit communication with custom hardware, iStuff mobile

interposes a desktop PC that receives sensor data using a wireless link. The events are processed using Quartz Composer [9], a visual data flow language and relayed to the mobile phone using a second wireless link. On the phone, a background application receives these messages and can inject input events to control existing phone applications.

LEGO MINDSTORMS

The Lego Mindstorms Robotic Invention System [10] offers plug-and-play hardware combined with a visual environment for authoring autonomous robotics applications. Mindstorms uses a visual flowchart language where language constructs are represented as puzzle pieces such that it is impossible to enter syntactically invalid programs. While a benchmark for low-threshold authoring, Lego Mindstorms targets autonomous robotics projects; the programming architecture and library are thus inappropriate for designing user interfaces. Mindstorms programs are downloaded and executed on the hardware platform without a communication connection back to the authoring environment, which prevents inspection of behaviors at runtime.

ARDUINO

The Arduino project [185] consists of a microcontroller board and a desktop IDE to write programs for that hardware platform in the C language. It is included in this review because it is one of the most popular platforms with students and artists today. Arduino wraps the open-source avr-gcc tool chain for developing and deploying applications on 8bit AVR RISC microcontrollers. The avr-gcc tool chain is commonly used by professional developers of embedded hardware. Unlike many other tools reviewed here, Arduino does not offer visual programming or high-level scripting. The success of the platform is probably attributable to hiding of configuration complexity where possible (removing "incidental pains" of programming); careful design of a small library of most commonly used functions; and a focus on growing a user community around the technology that contributes examples and documentation. The success of Arduino programming (and of HyperCard) suggests that it might not be necessary to eliminate all textual programming to build a rapid, accessible prototyping tool, if the tasks the designer wants to accomplish can be succinctly expressed using provided libraries.

THUMBTACKS

Hudson's Thumbtacks system [127] focuses on using novel hardware input to interact with existing applications. Only discrete input from capacitive switches in supported. Users capture screenshots of running existing applications, and draw regions onto those

screenshots corresponding to areas where mouse clicks should be injected when an external switch is pressed or released. Key presses can be similarly injected at the system event queue level. Exemplar also offers the ability to generate such events. Keyboard & mouse event injection has the benefit that any existing application can be targeted. It has serious drawbacks, too: the response to a mouse click or key press may depend on internal application state, which cannot be sensed or modeled in the Thumbtacks system. In addition, the rest of the computer is essentially inoperable while events are injected. One solution to this problem is to run the authoring environment on a different machine than the application that should be controlled, and relay events through network messages from one computer to the other.

DART

DART [178], The Designers' Augmented Reality Toolkit, supports rapid development of AR experiences, where computer-generated graphics (and audio) are overlaid on live video. DART was implemented as a set of extensions for Macromedia (now Adobe) Director [6], enabling designers familiar with that tool to leverage their existing skill set. d.tools shares DART's motivation of enabling interaction designers to build user experiences in a technical domain previously beyond their reach, but supports different types of interfaces and also introduces its own authoring environment instead of extending an existing software package.

PAPIER-MÂCHÉ & EYEPATCH

Papier-Mâché [146] focuses on supporting computer-vision based tangible applications. It introduces architectural abstractions that permit substitution of information-equivalent technologies (e.g., visual tag tracking and RFID). Papier-Mâché is a Java API — applications have to be programmed in Java — restricting its target audience to advanced programmers. Papier-Mâché contributes the methodology of user centered API design and a visual preview window, where internal state of recognition algorithms and live video input can be seen, enabling inspectability of running code. EyePatch [182] also targets vision-based applications. It offers a larger number of recognition approaches and outputs data in a format that a variety of other authoring applications can consume. EyePatch relies on programming by demonstration, and will thus be covered in more detail in that section.

BUG

The BugLabs BUG [11], a commercial product introduced after the publication of d.tools, is a modular hardware platform for developing standalone mobile devices. It consists of a base into which modular units (LCD display, GPS, general purpose IO, accelerometer) can be

plugged. Applications for the BUG are written in a subset of Java and execute on a virtual machine on the base unit. The development environment, which extends the Eclipse Java environment on a desktop PC, links to a shared online repository of applications that one can download and immediately execute on the BUG. The plug-and-play architecture resembles the d.tools hardware interface, although the embedded BUG Linux system is more powerful (and more complex to manage). Like Phidgets, the BUG system mainly targets accomplished programmers — while changing hardware configurations is trivial, the software abstractions of the BUG API are not suitable for non-expert developers.

3.4 VISUAL AUTHORING

Many existing prototyping tools have adopted some form of node-link diagram to express interface semantics. How do these particular authoring techniques fit into the larger space of visual programming? Why might they be a good fit or why might other techniques be more suitable? This section provides an overview of different approaches to use visual representations in the authoring process.

Visual means have been used both to describe programs, as well as to implement them. We will first review visual formalisms — systematic ways of diagramming or otherwise graphically describing computational processes. Visual programming proper uses graphics to implement software. The following section provides a summary of different visual programming systems. A third approach to leverage graphics in programming is to employ a textual programming language and offer rich visual editors that help with writing correct code (structured editors) or substitute graphic editing for some tasks, i.e., GUI layout, while also allowing textual programming (hybrid environments). The last section reviews important research in such structured editors and rich, hybrid IDEs.[1]

3.4.1 VISUAL FORMALISMS

Visual formalisms use graphical means to document or analyze computational processes. Some can be transformed automatically into executable code. Others may not be useful as a way to implement programs because they may be too abstract or purposefully ambiguous, or they may require too much effort to describe programs of useful complexity. Some of the main ways of graphically describing computation are state diagrams, statecharts, flowcharts, and

[1] The structure and some of the examples used in this section are inspired by Brad Myers' survey talk on the Past, Present and Future of Programming in HCI [189].

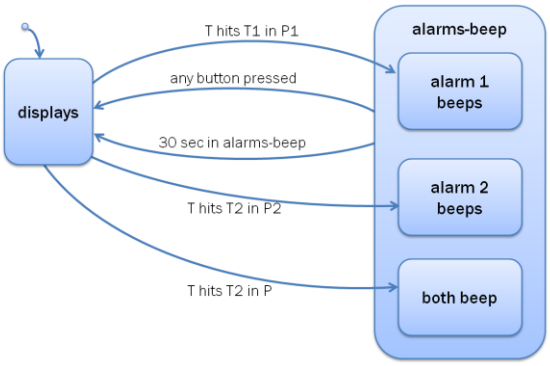

Figure 3.9: A partial, hierarchical statechart for a wrist watch with alarm function; redrawn from an example in Harel [105].

UML diagrams (which subsume the previous and add additional diagram types). A comprehensive review of additional visual specification techniques can be found in Wieringa's survey [250].

3.4.1.1 State Diagrams

State diagrams are graphical representations of finite state machines [125:Chapter 2], a computational model consisting of states, transitions, and actions. States capture the current point of computation; transitions change the active state based on conditional expressions over possible program input. Actions modify internal memory or generate program output. Actions can be defined for state entry, state exit, input received while in a state, and activation of a transition. State diagrams are easy to comprehend and to generate. However, they also have fundamental limitations: capturing concurrent, independent behaviors leads to a combinatorial explosion in the number of states. Adding behavior that should be accessible from a number of states requires authoring corresponding transitions independently for each state, which makes maintenance and editing cumbersome and results in a "rat's nest" of many transitions. As state and transition density increases, interpreting and maintaining state diagrams becomes problematic; state diagram thus suffer from multiple scaling limitations, a problem common to many visual formalisms and visual programming languages [53].

3.4.1.2 Statecharts

Harel's *statecharts* [105] find graphical solutions for some of the limitations of simple state machines. Statecharts introduce hierarchical clustering of states to cut down on transition density; introduce concurrency through multiple active states in independent sub-charts; and

offer a messaging system for communication between such sub-charts. Harel summarizes: "statecharts = state-diagrams + depth + orthogonality + broadcast communication." Harel uses the example of a programmable digital watch — an early ubiquitous computing device —and models its entire functionality with a statechart in his original paper on the topic (Figure 3.9). Both state diagrams and statecharts have been used extensively to describe *reactive systems*, i.e., those that are tightly coupled to, and dependent, on, external input. Both industrial process automation and user interfaces fit into this reactive paradigm. The added flexibility of the statechart notation makes generating correct charts and reasoning about them harder than working with simple finite state machines. Heidenberg et al. [117] studied defects in statecharts produced in an industrial setting and found that use of orthogonal components, one of the parts that make statecharts more powerful than state diagrams, also contributed to defect rate and advocated that its use should therefore be minimized.

3.4.1.3 Flowcharts

Flowcharts [48] express algorithms as a directed graph where nodes are computational steps (evaluating statements that change variable values, I/O, conditionals) and arrows transfer control from one step to another (Figure 3.10). One important use of flowcharts is to document algorithms written in procedural programming languages. Nassi-Shneiderman structograms [197] are a more succinct graphical representation of control flow in procedural

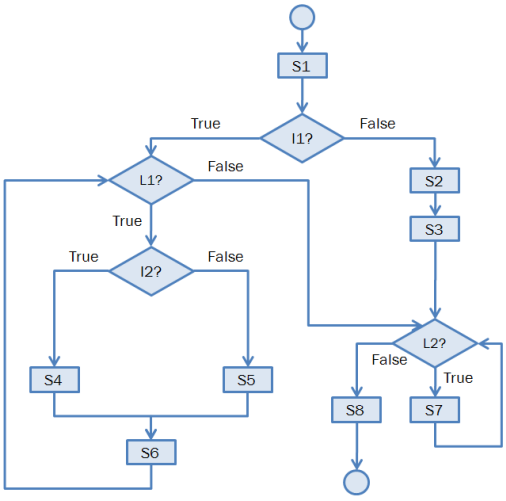

Figure 3.10: Example of a flowchart, adapted from Glinert [87].

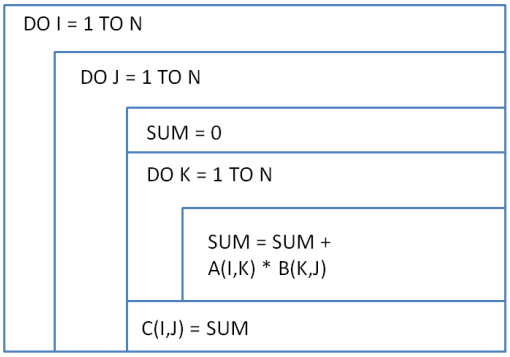

Figure 3.11: Example of a Nassi-Shneiderman structogram, adapted from Glinert [87].

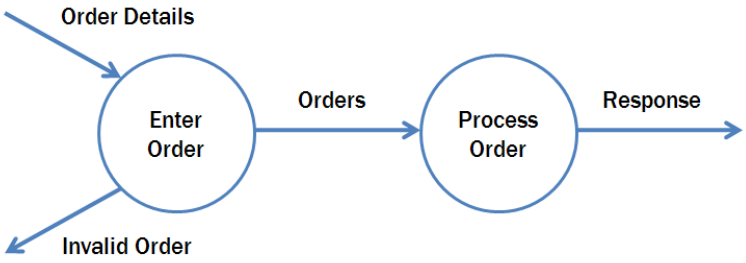

Figure 3.12: Example of a data flow diagram, redrawn by the author from Yourdon [259: p. 159]

languages (Figure 3.11). Graphical elements of structograms include process blocks, which contain program statements, branching blocks for conditionals, and testing loops, to express iteration with a stopping condition.

3.4.1.4 Data Flow Diagrams

State diagrams and flowcharts express the change of program *control* over time. In contrast, data flow diagrams (DFDs) focus on describing how *data* travels in complex, multi-component systems. Arrows in DFDs denote the *flow* of information from one component to another; nodes represent *processes* that operate on incoming data flows and emit outgoing flows (Figure 3.12). As a diagramming technique, data flow modeling is extensively used in Structured Systems Analysis [82,259]. Flow diagrams expose the type of data that is transmitted, its origin and destination. Flow diagrams do not capture any sequencing of computation. Reasoning about the order of execution or other temporal aspects of programs is therefore not well supported in DFDs.

3.4.1.5 Unified Modeling Language

The Unified Modeling Language (UML [12]) is an umbrella term used to characterize a set of 13 different diagramming techniques (in UML 2.0) that can be used to describe various aspects of a computer system. UML is closely linked to object oriented programming languages, while flowcharts arose at the same time as structured programming languages. UML distinguishes between structure diagrams that show the interrelation of different components, e.g., class diagrams, behavior diagrams, which subsume state machines; and interaction diagrams which model sequences of communication and control transfer between different components. Dobing and Parsons [71] report survey results on how UML is used in practice; their survey found that many diagram types were not well understood.

3.4.2 VISUAL PROGRAMMING PROPER

Visual programming constructs executable programs using graphical means. Many visual programming languages follow a node-and link diagram paradigm, but the meaning of nodes and links vary significantly. Two main approaches are control flow languages, where nodes express program state and links express transitions that move a program through those states; and data flow languages, where states are transformation operations to be performed on data, and links are pipes through which data flows from node to node. Some languages have been created by directly operationalizing the visual formalisms described in the previous section. State diagrams, statecharts, and all fall under the category of control flow; data flow diagrams, predictably, express data flow.

In general, purely visual programming languages are *not* widely used in practice to implement general programs or user interfaces. This is partially due to the relatively high viscosity (resistance to modification) of visual languages (see section 3.4.4 on cognitive dimensions of notations). The exceptions are applications in education where flowchart-based languages have had success with novice and hobbyist programmers; and digital signal processing (with electronic music being one application), where visual data flow languages are used.

Early research in visual programming languages has been reviewed by Glinert [87]. Rather than opting for breadth, this section discussed a small number of concrete examples chosen for historical interest or relevance to user interface design.

3.4.2.1 Control Flow Languages

FLOWCHARTS

Glinert's Pict [87] is an early example of a purely visual programming environment. Pict operationalizes program flowcharts and can be used to implement simple but non-trivial numerical algorithms such as the Fibonacci function. I/O is numerical only, so no user interfaces can be constructed. The design environment is entirely cursor controlled, without any text input. Pict introduced visual animation of execution by adding graphical decorators to the design diagram — such runtime feedback in the design environment is also used in d.tools. A user study with 60 computer science students revealed that novices reacted positively to Pict, while expert programmers were more critical and less likely to prefer it over textual approaches.

More recently, the Lego Mindstorms Robotics kit [10] includes a flowchart programming language. Programs can have parallel "tracks" to express concurrency, but the language does not have variables. Resnick's Scratch [13] is an environment for programming interactive animations and games aimed at young, novice programmers. The editor also offers a flowchart-inspired programming environment, with support for user-defined variables.

STATECHARTS

Wellner reports the development of an early user interface management system (UIMS) based on Statecharts [249]. Statecharts were drawn in a graphics package to capture event logic for UI dialogs. These graphics had to be manually transcribed into a text format to make them executable. Completely automatic systems that generate executable code from statecharts such as IBM Rational Rose RealTime [14] also exist, though they do not focus on integration with user interface development.

STORYBOARDS AS CONTROL FLOW LANGUAGES

Storyboards as used in UI prototyping systems in Section 3.2 are variants of finite state machines. Traditional, hand-drawn storyboards from film production present a linear sequence of key frames. When applied to user interface design, storyboard frames often consist of different unique user interface views. To capture the information architecture — how different screens relate to each other — storyboards are then enhanced with connecting arrows. The semantics of a canonical storyboard state diagram can be expressed as follows: the "enter state" action in each state corresponds to showing the interface of the particular storyboard frame. Transitions are conditionals that express that a different state or screen should be shown based on an appropriate input event. Because storyboard-driven authoring tools equate a state with a complete UI definition, no parallelism or encapsulation is offered.

3.4.2.2 Data Flow Languages

Data flow languages have found successful applications in domains such as digital signal processing (DSP) and electronic music. LabView [15] is a digital signal processing and instrumentation language widely used in electrical engineering. LabView programs are referred to as Virtual Instruments and are defined by graphically connecting different functional units, e.g., data providers such as sensors, Boolean logic, and mathematical functions. To support control flow constructs such as loops, LabView offers control flow blocks that are embedded into the data flow language. Other data flow languages used for measuring and instrumentation are MatLab Simulink [16], and Agilent Visual Engineering Environment [17]. In focusing on measurement and instrumentation, these applications support different user populations than d.tools and Exemplar, with different respective needs and expectations.

Max/MSP [18] and its open-source successor Pure Data (Pd) [209] are used by artists to program new electronic musical instruments, live video performances, or interactive art

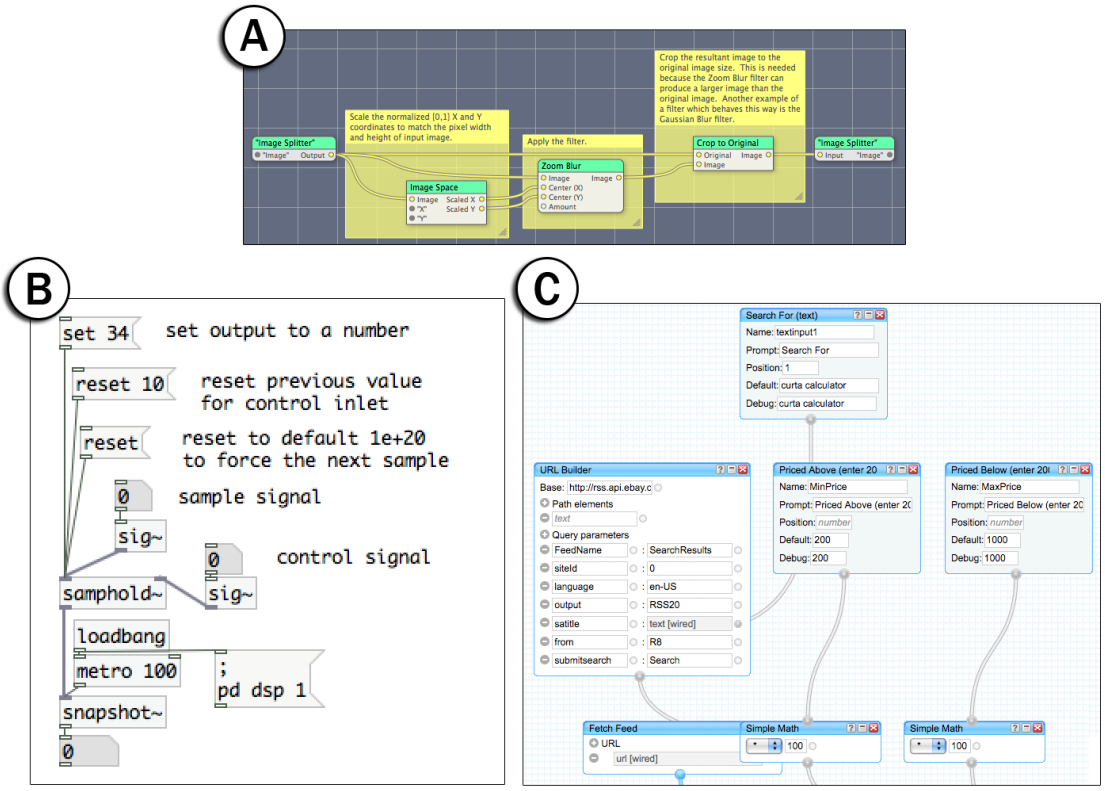

Figure 3.13: Examples of commercial or open source data flow languages. **A**: Quartz Composer; **B**: Pure Data; **C**: Yahoo Pipes

installations. The interface metaphor for these languages is that of an analog patch cord synthesizer, where different functional units (the nodes) are "patched together" with connections (Figure 3.13B). Output from one node thus flows into an input of another node. There is no visual notion of "state" and in fact, reasoning about the order in which operations are performed in these languages is subtle and non-trivial. Both environments make distinctions between nodes and transitions that operate on sound data, which has to be updated on a fixed audio rate, and those that operate on control data (e.g., human interaction), which is event-based and can be processed at much lower rates.

Data flow to process and transform input streams or filter signals has also been applied in other multimedia applications. Apple's Quartz Composer [9] employs a data flow paradigm to author image processing pipelines for graphics filters and animation (Figure 3.13A); it has been integrated into the iStuff Mobile toolkit [42] for sensor data processing. The MaggLite toolkit [129] uses a similar approach to provide a flexible interconnection layer between new kinds of input devices and suitably instrumented applications that expose abstract input event hooks.

A third application area for data flow languages has been the processing of online data streams queried via web service APIs and RSS feeds. Yahoo Pipes [19] is a recent example of a browser-based tool that allows the merging and filtering of multiple data streams (Figure 3.13C). Common examples programmed in Pipes are meta search engines that combine query results from multiple sources, and "mashups" which combine data from multiple web services in novel ways [108].

3.4.2.3 Control Flow and Data Flow in d.tools and Exemplar

In d.tools, states express output to screens or other output components (e.g., motors, LEDs). To enable continuous behaviors and animations, d.tools offers two extensions to pure visual control flow: First, d.tools supports a limited amount of data flow programming by drawing arrows from input components to output components within a state. This way, for example, an LED can be dimmed by a slider. However, there is no compositionality as in other data flow languages and d.tools' primary representation remains control flow, because it maps directly to interface states. Second, d.tools combines visual authoring with procedural scripting in each state. The next section will review different related approaches of combining visual and textual programming.

Exemplar is a direct manipulation visual environment that is organized according to a data flow model: raw sensor data arrives as input, and emerges transformed as high-level UI

output events. As such, Exemplar could be seen as one possible processing node in a data flow language, which would allow for further composition. At present, it is a standalone application that feeds event data to d.tools or controls existing GUI applications through mouse & keyboard event injection.

3.4.3 ENHANCED EDITING ENVIRONMENTS

Beyond purely visual programming, different ways of combining graphical and textual authoring exist. Three common combinations are: visual GUI editors that generate source code for textual programming languages; structured editors that use graphic techniques to facilitate code entry and prevent errors; and hybrid approaches where some computation is specified graphically, and other computation is specified in source code.

3.4.3.1 Visual Editors

GUI editors enable direct graphical layout of user interfaces. In general these editors are restricted to defining the appearance of UIs. Behavior and architecture have to be expressed separately in code. Two types of visual GUI editors exist: 1) editors that generate code in the source language, and 2) editors that generate code in some intermediate, often declarative language that is then interpreted later by a suitable library in the application.

Early GUI editors, e.g., for Java Swing, generate procedural code and accordingly read procedural code as well. A recurring issue for such systems is the *roundtrip problem*: If procedural code generated by these systems is later edited manually by a programmer, the visual editor may not be able to parse the modified text and re-created an editable graphical interface for it. The roundtrip problem exists for any environment that produces user-editable source, but it is exacerbated when the produced text is code for a full-fledged programming language, where arbitrary statement can be added.

Recent years have seen a shift towards GUI editors that generate declarative UI specifications, often in some UI-specific XML dialect (e.g., HTML, MXML, XAML). Such UI specifications may have more runtime overhead, but reduce the roundtrip problem. Declarative UI definitions are also thought to be easier to write and reason about, since layout of hierarchical UI elements on screen is expressed by the hierarchical structure of the source document. Examples of editors that produce declarative UI specifications are Adobe Flex Builder, Adobe Dreamweaver, and Microsoft Expression Blend.

Beyond visual editing of layout, some GUI editors also allow direct manipulation definition of dynamic behavior, such as path-following animations. Conversely, graphics

applications that are primarily direct manipulation editors may also offer programmability through scripting language APIs. 3D modeling applications such as Google SketchUp (programmable in Ruby) and Autodesk Maya (programmable in MEL, a C-like scripting language) are examples of such an approach.

3.4.3.2 Structured Source Editors

Structured editors add interaction techniques to source code editors that facilitate correct entry of source code by using knowledge about valid syntax constructs in the target language (e.g., by using the language grammar) or knowledge about the structure of language types and libraries. Where traditional syntax editors operate on individual characters in plain text files, structured editors operate on the abstract syntax tree (AST) that can be constructed by parsing the source text. Structured source editors can use this knowledge to enforce correct syntax, and other statically verifiable properties, by making it impossible to enter incorrect programs, or they can check these properties and inform the programmer of detected problems.

The Cornell Program Synthesizer [238] is an early example of a syntax-directed editor which enforced correct syntax through a template-instantiation system for programs written in PL/I. Common language constructs were encoded in templates — keywords and punctuation were immutable, while placeholders could be replaced by either inserting variables, immediate values, or other templates.

The CMU MacGnome project developed multiple structured editors to facilitate learning of programming by novices [188]. Alice2 [139], an environment for developing interactive 3D virtual worlds, features a structured editor where program statements can be composed through drag-and-drop. Suitable values (immediate or through variables) can be selected through drop-down lists. One challenge of structured editors is that they may increase the viscosity and hinder provisionality of expressed programs — by enforcing correctness, they may make it harder to experiment or make changes that require breaking the correctness of the program during intermediate steps (as noted by Miller [188]).

3.4.3.3 Hybrid Environments

A final way to combine visual and textual programming is to permit embedding of textual code into visual programming systems. One response to the criticism that visual programming either does not scale, or becomes hard to reason about and modify, is to move away from a purely visual system and permit expression of both visual and textual programs within the same environment. For example, the data flow language Pd permits procedural expressions within certain nodes (Figure 3.14). Conditional logic or mathematical formulas, which are cumbersome to express in pure data flow, can thus succinctly be captured in a single node. The Max/MSP language permits evaluation of JavaScript and use of Java classes in its language.

d.tools also opts for a hybrid approach by following a storyboard approach to capture high-level architecture of the designed interface, while relying on an imperative scripting language, BeanShell [200], for most continuous behaviors. This flexibility comes at a price: simultaneous presence of multiple different authoring paradigms raises the number of concepts a user has to learn to effectively use that environment.

3.4.4 ANALYZING VISUAL LANGUAGES WITH COGNITIVE DIMENSIONS OF NOTATION

How might one compare the relative merits and drawbacks of different visual programming environments, or of visual and textual programming languages? The most complete effort to date to develop a systematic evaluation instrument is Greene and Blackwell's Cognitive Dimensions of Notation framework (CDN) [89,90]. The CDN framework offers a high-level inspection method to evaluate the usability of information artifacts. In CDN, artifacts are analyzed as a combination of a notation they offer and an environment that allows certain manipulations of the notation. As an expert inspection method, it is most comparable to

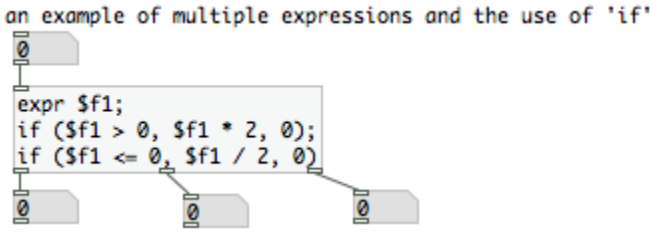

Figure 3.14: Example of hybrid authoring in Pure Data: a visual node contains an algebraic expression.

Dimension	Description
Abstraction	What are the types and availability of abstraction mechanisms?
Hidden Dependencies	Is every dependency overtly indicated in both directions?
Premature Commitment	Do programmers have to make decisions before they have the information they need?
Secondary Notation	Can programmers use layout, color, or other cues to convey extra meaning, above and beyond the 'official' semantics of the language?
Viscosity	How much effort is required to perform a single change?
Visibility	Is every part of the code simultaneously visible, or is it at least possible to juxtapose any two parts side-by-side at will?
Closeness of Mapping	Closeness of visual representation to problem domain. What 'programming games' need to be learned?
Consistency	When some of the language has been learnt, how much of the rest can be inferred?
Diffuseness	How many symbols or graphic entities are required to express a meaning?
Error-proneness	Does the design of the notation induce 'careless mistakes'?
Hard mental operations	Are there places where the user needs to resort to fingers or penciled annotation to keep track of what's happening?
Progressive evaluation	Can a partially-complete program be executed to obtain feedback?
Role-expressiveness	Can the reader see how each component of a program relates to the whole?

Table 3.3: The main dimensions of the Cognitive Dimensions of Notation inspection method (from [90: p.11]).

Nielsen's heuristic evaluation, with a different set of metrics. Blackwell and Green's Cognitive Dimensions Questionnaire [47] asks evaluators to first estimate of how time is spent within the authoring environment, and then analyze the software against the framework's 13 cognitive dimensions (Table 3.3).

Evaluation of the notation is always relative to some target activity. Greene distinguishes six major activities: incrementation, transcription, modification, exploratory design, searching, and exploratory understanding. Any given task will likely break down into a mixture of these cognitive activities. Similarly, any given notation and environment will support or impede these activities to different extents. A CDN analysis can therefore be seen as establishing the impedance match (or mismatch) of a particular programming task and a particular programming system.

3.5 PROGRAMMING BY DEMONSTRATION

Programming by demonstration (PBD) is the process of inferring general program logic from observation of examples of that logic. Given a small number of examples, PBD systems try to derive general rules that can be applied to new input. This generalization from examples to rules is the crucial step in the success or failure of PBD systems [168]. The inference step often leverages machine learning and pattern recognition techniques. Demonstration and generalization techniques are not enough to build a PBD system. In textual programming, more time is spent editing and modifying existing code than writing new code [194]. For PBD systems, where program logic is built "under the hood", this implies that separate techniques are needed to present what was learned back to the user, and allow her to edit this representation as well.

Because PBD builds functionality without requiring textual programming, it has been a strategy employed for end-user development [196]. Comprehensive surveys of PBD systems can be found in books edited by Cypher [67], Lieberman [168]; and Lieberman, Paterno and Wulf [169].

3.5.1 PBD ON THE DESKTOP

In many PBD systems, the examples or demonstrations are provided as mouse and keyboard actions in a direct manipulation graphical user interface. PBD has been employed in educational software to introduce children to programming concepts [231]; for the specification of functionality in GUI builders [192]; to author spreadsheet constraints [193]; and to author web automation scripts by demonstration [173]. In the realm of prototyping tools, demonstration has been used for authoring animations and other dynamic behavior in Monet [164]. Monet learns geometric transformations applied to widgets through continuous function approximation using radial basis functions centered on screen pixels.

3.5.2 PBD FOR UBIQUITOUS COMPUTING

Early examples of using demonstrations that take place in physical space or that have effects on physical space can be found in the robotics field. Andreae used demonstration to specify robot navigation [35]; Friedrich et al employed it to define grasp motions for robotic assembly arms [80]. Our research on Exemplar builds upon the idea of using actions performed in physical space as the example input.

The closest predecessor to Exemplar in approach and scope is a CAPella [70]. This system focused on authoring binary context recognizers by demonstration (e.g., is there a meeting going on in the conference room?), by combining data streams from discrete sensors, a vision algorithm, and microphone input. Exemplar shares inspiration with a CAPella, but it offers important architectural contributions beyond this work. First, a CAPella was not a real-time interactive authoring tool: the authors of a CAPella reported the targeted iteration cycle to be on the order of days, not minutes as with Exemplar. Also, a CAPella did not provide strong support for continuous data. More importantly, a CAPella did not offer designers control over how the generalization step of the PBD algorithm was performed beyond marking regions. We believe that this limitation was partially responsible for the low recognition rates reported (between 50% and 78.6% for binary decisions).

FlexiGesture is an electronic instrument that can learn gestures to trigger sample playback [186]. It embodies programming by demonstration in a fixed form factor. Users can program which movements should trigger which samples by demonstration, but they cannot change the set of inputs. Exemplar generalizes FlexiGesture's approach into a design tool for variable of input and output configurations. We share the use of the dynamic time warping algorithm [218] for pattern recognition with FlexiGesture.

We also drew inspiration for Exemplar from Fails and Olsen's Crayons technique for end-user training of computer vision recognizers [75]. Crayons enables users to sketch on training images, selecting image areas (e.g., hands or note cards) that they would like the vision system to recognize. Maynes-Aminzade's EyePatch [182], a visual tool to extract interaction events from live camera input data, expands on Crayons' interaction techniques. With EyePatch, users also directly operate on input images to indicate the kind of objects or events they would like to detect. While Crayons only supported a single recognition algorithm (induction of decision trees), EyePatch shows that different detection algorithms require different kinds of direct manipulation techniques. For example, training an object detector may require highlighting examples of objects in frames of multiple different video clips, while training a motion detector requires interaction techniques to select sequences of consecutive frames, and a visualization of the detected motion on top of the input video. Crayons and EyePatch complement our work well, offering a compelling solution to learning from images, where as Exemplar introduces an interface for learning from time-series data.

3.6 DESIGNING MULTIPLE ALTERNATIVES & RAPID EXPLORATION

To explore the space of possible solutions to a design problem, single point designs are insufficient. Two strategies for enabling broader design space exploration are to build tools that support working with multiple alternatives in parallel; and tools that minimize the cost of making and exploring changes sequentially. This section reviews prior art in both areas.

3.6.1 TOOLS FOR WORKING WITH ALTERNATIVES IN PARALLEL

The research on alternatives in this dissertation, embodied in Juxtapose, was directly motivated by Terry et al.'s prior work on tools for creating alternative solutions in image editing. Side Views [241] offer command previews, e.g., for text formatting, inside a tooltip. Parameter Spectrums [241] preview multiple parameter instances to help the user choose values. Similar techniques are now part of Microsoft Office 2007, attesting to the real-world impact of exploration-based tools. Parallel Pies [242] enable users to embed multiple image filters into a single canvas, by subdividing the canvas into regions with different transformations. Since Juxtapose targets the domain of textual programming of interaction designs, its contributions are largely complementary. Unlike creating static visual media, the artifacts designed with Juxtapose are interactive and stateful, which requires integration between source and run-time environments.

Terry also proposed Partials, an extension to Java syntax that delays assignment of values to variables until runtime [239:Appendix B]. Partial variables list a set of possible values in source code; at runtime, the developer can choose between these values through a generated interface. Juxtapose extends this work by contributing both authoring environment and runtime support for specifying and manipulating alternatives.

Automatic generation of alternatives was proposed in Design Galleries [181] a browsing interface for exploring parameter spaces of 3D rendered images. Given a formal description of a set of input parameters, an output vector of image attributes to assess, and a distance metric, the Design Galleries system computes a design-space-spanning set of variations, along with a UI for structured browsing of these images. Design Galleries require developers to manually specify a set of image features to steer a dispersion algorithm; options are then generated automatically. In Juxtapose, options are created by the designer. Juxtapose makes the assumption that the results of parameter changes can be viewed instantaneously, while rendering latency motivated Design Galleries. Table 3.4 shows a comparative overview of Design Galleries, Terry *et al.*'s work, and Juxtapose.

Subjunctive interfaces [177] introduced working with alternatives in information processing tasks. Multiple scenarios co-exist simultaneously and users are able to view and adjust scenarios in parallel. Clip, connect, clone [81] applies these interface principles to accessing web application data, e.g., for travel planning. There are no design tools for creating subjunctive interfaces; only applications that realize these principles in different information domains.

Spreadsheets also inherently support parallel exploration through their tabular layout. Prior research has applied the spreadsheet paradigm to image manipulation [161] and information visualization [58]. Such graphical spreadsheets offer a more complex model of defining and modifying alternatives than Juxtapose's local-or-global editing. Investigating how a spreadsheet approach could extend to interaction design is an interesting avenue for future work.

	Does evaluation of output require real-time input?	**How are parameter values created?**	**Who creates parameter-to-output mapping?**
Design Galleries	No — output is a static image or a sequence of images.	Generated by dispersion algorithm	Expert specifies for each DG instance
Side Views/ Parallel Pies	No — output is a static image	Mixed initiative: parameter spectrums are auto-generated; designers chooses values	Mixed: image processing library provides primitives; designers compose primitives in Side Views
Juxtapose	Yes, output is a user interface	Designer creates values in code alternatives or tunes at runtime	Developers specify mapping in their source code

Table 3.4: Differences between Design Galleries, set-based interaction, and Juxtapose are based on requirements of real-time input, method of alternative generation, and the source of input-output mapping.

TEAM STORM [101] addresses management of multiple sketches by a team of designers during collaborative ideation. The system, consisting of individual tablet devices and a shared display wall, allows design teams to manage and discuss multiple visual ideas. Like Terry's work, the system only addresses working with static visual media — interaction can be described in these sketches, but not implemented or tested.

3.6.2 RAPID SEQUENTIAL MODIFICATION

Rapid sequential changes, such as undo/redo actions are frequently used by graphic design professionals to explore alternatives [240]. In the realm of programmed user interfaces, research has explored several strategies to reduce the cost of making changes.

CREATING CONTROL INTERFACES

One strategy is to make data in a program modifiable at runtime. Many breakpoint debuggers for modern programming languages allow the inspection of runtime state when then program is suspended; some allow modification of the values as well. However, breakpoint debugging is not always feasible when testing interactions that require real-time user input. Furthermore, the user interface for parameter access has not been a focus of research in debuggers.

In Juxtapose, suitable control interfaces are automatically generated. Adobe's Pixel Bender Toolkit [20] also automatically creates control sliders for scalar parameters in image processing code. In this domain, the entire specified algorithm can be rerun whenever a parameter changes. Juxtapose offers a more general approach that enables developers to control what actions to take when a variable value is changed at runtime and to select which variables will be shown in the control interface.

Juxtapose furthermore enables settings of multiple parameters to be saved in "parameter snapshots." The notion of parameter snapshots exists in Isadora [62], a visual dataflow language for multimedia authoring. In Isadora, the parameter sets are predetermined by the library of data processing nodes. The notion of parameter snapshots is also commonly found in music synthesizers. Many early synthesizers offered a fixed hardware architecture, i.e., a certain number of oscillators and filters. The different presets or sounds shipped with the synthesizer were essentially different parameter snapshots for that given architecture. In Juxtapose, the programmer can define new variables for tuning in the source at any point.

Beyond changing parameter values, some tools offer "live" coding where source code can be modified while the program is executing. Interpreted languages such as Python may offer an interactive command line, which enables access to the internals of the running program. JPie [88] is an environment for Java education which permits real-time inspection and modification of all objects in a Java program. The Eclipse IDE [21] permits modifications of Java method contents in a running program. However, it is not always obvious when the modified class will be replaced in the virtual machine, and some modifications, e.g., to method signatures, require terminating and restarting the application. ChucK is a programming language expressly written for live music synthesis [246]. Juxtapose shares the goal of eliminating edit-compile-test cycles in favor of real-time adjustment. Juxtapose offers less flexibility than live coding languages for editing objects and logic. Conceptually, Juxtapose makes a distinction between a low-level source representation, and a higher-level set of "knobs" used for runtime manipulation. This higher-level abstraction allows for more controlled live improvisation.

3.7 FEEDBACK FROM USER TESTING

Prior research has investigated how to aid the analysis of user interface tests by making use of metadata generated either by the application being tested, by the system the application runs on, or by experimenters and users. Logged data is then either visualized directly, or it is used for structured access to other media streams, e.g., audio or video, also recorded during a test. Accelerating review and analysis of usability video data is especially valuable, as the high cost of working with video after its capture (in terms of person hours) restricts its use in professional settings today.

Two literature surveys are available that cover most existing techniques in automatically capturing and analyzing test data. Hilbert and Redmiles presented a comparative survey of systems that extract usability data from application event traces for remote evaluation, where experimenter and participant are geographically separated [119]. Ivory and Hearst present a survey of techniques to automate aspects of usability testing [133]. Their taxonomy distinguishes techniques for automatic capture of data (e.g., event traces) from techniques for automated analysis (e.g., statistics), and techniques for automated critique (e.g., suggestions for improvement). Most existing work has focused on WIMP applications on desktop PCs or

on web applications that run inside a browser. Test support for other types of user interfaces has received less attention.

3.7.1 IMPROVING WORK WITH USABILITY VIDEOS

Several techniques correlate time-stamped event data and video of GUI application tests. Mackay, in an early paper [179], described challenges that have inhibited the utility of video in usability studies, and outlined video functionality that would be useful to usability researchers and designers: capturing multiple, timestamp-correlated data streams, spatial viewing of temporal events, symbolic annotation, and non-destructive editing and reordering. Mackay introduced EVA, which offers researcher-initiated annotation at record time and later on during review. All annotations in this system were generated explicitly by the experimenter.

Hammontree et al. developed an early UI event logger that records low-level system events (key presses and mouse clicks). A programmable filter aggregates these observations into more meaningful, higher-level events such as command invocations. A "Multimedia Data Analyzer" then allows researchers to select elements in the log of UI events to locate the corresponding point in time in the video [103]. Hammontree speculated that video analysis tools would be particularly appropriate to compare different UI prototypes.

I-Observe by Badre et al. [38] enabled an evaluator to access synchronized UI event and video data of a user test by filtering event types through a regular expression language.

Akers et al. [34] showed that for applications that support creative authoring tools, e.g., image editing and 3D modeling, collecting undo and redo events then filtering usability video around these occurrences is a successful strategy to uncover many relevant usability problems at a fraction of the time required to survey full video.

While Weiler [247] suggests that solutions for event-structured video have been in place in large corporate usability labs for some time, their proprietary nature prevented us from learning about their specific functionality. Based on the data that is available, d.tools video analysis functions extend prior research and commercial work in three ways. First, they move off the desktop to physical UI design, where live video is especially relevant, since the designers' concern is with the interaction in physical space. Second, d.tools offers a bi-directional link between software model and video where video can also be used to access and replay flow of control in the model. Third, d.tools introduces comparative techniques for evaluating multiple user sessions.

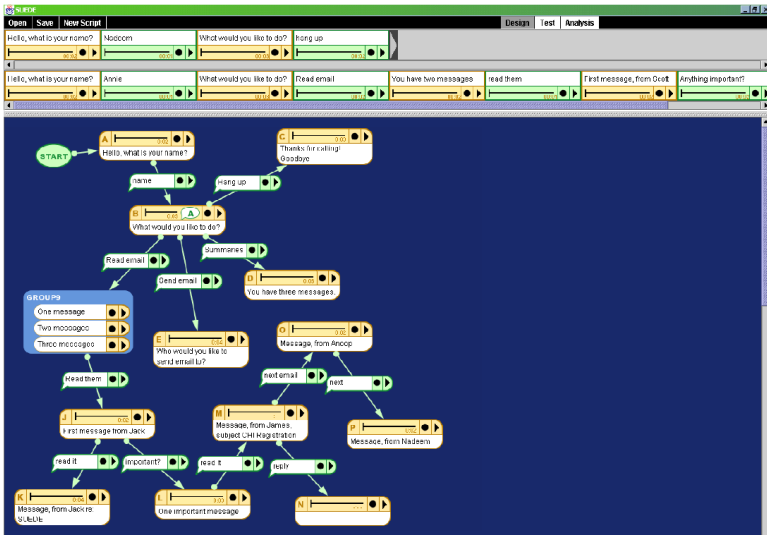

Figure 3.15: SUEDE introduced techniques to unite design, test, and analysis of speech user interfaces.

3.7.2 INTEGRATING DESIGN, TEST & ANALYSIS

Most closely related to the design methodology embodied in d.tools is SUEDE [148], a design tool for rapidly prototyping speech-user interfaces (Figure 3.15). SUEDE introduces explicit support for the design-test-analyze cycle through dedicated UI modes. It also offers a low-threshold visual authoring environment and Wizard of Oz support. At test time, SUEDE generates a wizard interface that allows the experimenter to guide the direction of a user test, by simulating speech recognition. SUEDE records a history of all user speech input and system speech output and makes that history available as a graphic transcript in analyze mode. SUEDE also computes statistics such as time taken to respond, and visualizes how many times a particular menu path was followed through varying link thickness. d.tools extends SUEDE's framework into a new application domain — physical user interfaces. It also adds integration of video analysis into the cycle. Like SUEDE, the d.tools system supports early-stage design activities. Aggregation and visualization of user sessions has also been applied to web site user tests in WebQuilt, where URL visitation patterns are logged using a proxy server [124].

3.8 TEAM FEEDBACK & UI REVISION

Gaining feedback on a UI prototype through user testing has high external validity, but it is resource intensive. Design team members can also provide valuable feedback in different roles

— as collaborators or as expert inspectors. Team members also have the design expertise to suggest changes. How can design tools aid this process of team-internal collaboration over prototypes and revision of prototypes?

Research in word processing and other office productivity applications has introduced annotation and change tracking tools that allow suggestion of changes along with tracking a history of modifications. But outside word processing and spreadsheets, such tools are still lacking. Research in version control and document differencing systems has introduced a complementary set of algorithms and techniques that compute and visualize differences between documents *after* they are made. We review both areas briefly.

3.8.1 ANNOTATION TOOLS

Fish et al.'s Quilt system [77] introduced annotation and messaging inside a word processor to support the social aspects of writing, noting that in some academic disciplines, the majority of publications are co-written by multiple authors. The combination of change tracking and commenting effectively enables asynchronous collaboration, where different members may have different functions, such as author, commenter, and reader [198]. In modern word processing tools, annotation and change tracking tools are now pervasive, attesting to the utility of asynchronous collaboration.

Sketching has also been used to capture and convey changes and comments. In Paper Augmented Digital Documents, annotations are written on printed documents with digital pens; a pen stroke interpreter then changes a digital document accordingly [96]. In ModelCraft [232] physical 3D artifacts, created from CAD models, can be annotated with sketched commands to express extrusions, cuts, and notes. These annotations are then converted into changes in the underlying CAD model for the next iteration. d.note applies this approach of selectively interpreting annotations as commands to the domain of interaction design.

3.8.2 DIFFERENCE VISUALIZATION TOOLS

Change tracking editors record modifications as they happen. Another approach is to compute and visualize differences of a set of documents after they were edited. The well-known diff algorithm computes a set of changes between two text two text files [128]. Offline comparison algorithms also exist for pairs of UML diagrams [86] and for multiple versions of slide presentations [74]. The d.note visual language for revising interaction design diagrams is

most closely related to the diagram differencing techniques introduced by Mehra et al. for CASE diagrams [184]. Difference visualization research contributes algorithms to identify and visualize changes. d.note contributes interaction techniques to create, test, and share such changes.

3.8.3 CAPTURING DESIGN HISTORY

Managing team feedback and design revisions is also related to research in capturing design histories, although the two fields have somewhat different goals. Design histories capture and visualize the sequence of actions that a designer or a design team took to get to a current point in their work. The visual explanations tend to focus on step-by-step transformations, e.g., for web site diagrams [149], illustrations [154,242], or information visualizations [116]. Revision tools such as d.note focus on a larger set of changes to a base document version, where the order of changes is not of primary concern. Design histories offer timeline-based browsing of changes in a view external to the design document; d.note offers a comprehensive view of a set of changes in situ, in the design document itself.

CHAPTER 4 AUTHORING SENSOR-BASED INTERACTIONS

Ubiquitous computing devices such as portable information appliances — mobile phones, digital cameras, and music players — are growing quickly in number and diversity. In addition, sensing technologies are becoming pervasive, for example in game controllers, and sensor hardware is increasingly diverse and economical. To arrive at usable designs for the user interfaces of such physical devices, product designers have to be able to prototype the experience of interacting with a novel hardware device. This chapter presents two systems that bring prototyping of interactions based on sensor data input within reach of interaction designers. The first section introduces d.tools, an authoring environment that combines visual authoring of application logic with a novel plug-and-play hardware platform (Figure 4.1). The second section introduces Exemplar, an extension to d.tools that enables designers to author sensor-based interaction events through programming by demonstration.

4.1 AUTHORING PHYSICAL USER INTERFACES WITH D.TOOLS

Fieldwork with professional interaction designers revealed that the creation of ubiquitous computing prototypes has remained largely out of their reach. d.tools lowers the expertise threshold and time commitment required for creating ubiquitous computing prototypes through two contributions. The first contribution is a set of interaction techniques and

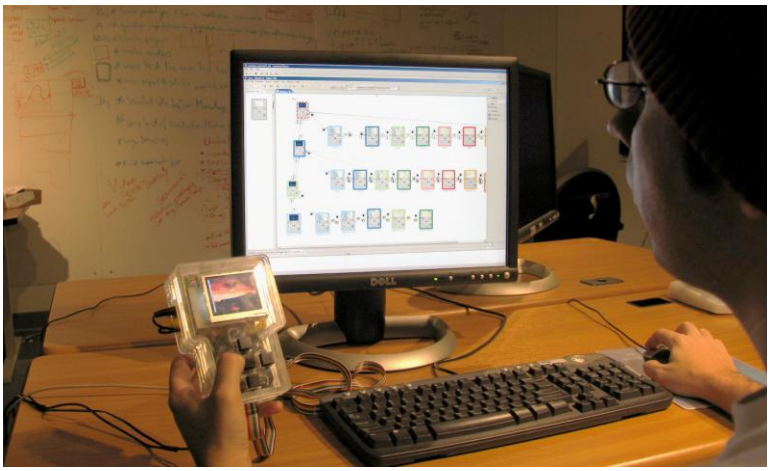

Figure 4.1: Overview of prototyping with d.tools: A designer interacts both with a hardware prototype (left) and the authoring environment (right).

architectural features that support rapid, early-stage prototyping. d.tools introduces a visual, control flow-based prototyping model that extends existing storyboard-driven design practice [126]. To provide a higher ceiling than is possible with visual programming alone, d.tools augments visual authoring with textual programming.

Second, d.tools offers an extensible architecture for physical interfaces. In this area, d.tools builds on prior work [37,43,92,93,159,185] that has shielded software developers from the intricacies of mechatronics through software encapsulation, and offers a similar set of library components. However, the d.tools hardware architecture is more flexible than prior systems by offering three extension points — at the hardware-to-PC interface, the intra-hardware communication level, and the circuit level — that enable experts to extend the library.

The rest of this section is organized as follows. We begin by outlining key findings of fieldwork that motivated our research. We then describe design principles, followed by the key interaction techniques for building, testing and analyzing prototypes that d.tools offers. We then outline implementation decisions and conclude with a report on three different strategies we have employed to evaluate d.tools.

4.1.1 FIELDWORK

To learn about opportunities for supporting iterative design of ubiquitous computing devices, we conducted individual and group interviews with eleven designers and managers at three product design consultancies in the San Francisco Bay Area, and three product design masters students. This fieldwork revealed that designing off-the-desktop interactions is not nearly as fluid as prototyping of either pure software applications or traditional physical products.

Most product designers have had at least some exposure to programming but few have fluency in programming. Design teams have access to programmers and engineers, but delegating to an intermediary slows the iterative design cycle and increases cost. Thus, while it is possible for interaction design teams to build functional physical prototypes, the cost-benefit ratio of "just getting it built" in terms of time and resources limits the use of comprehensive prototypes to late stages of their process. Comprehensive prototypes that integrate form factor (looks-like prototypes) and functions (works-like prototypes) are mostly created as expensive one-off presentation tools and milestones, but not as artifacts for reflective practice.

Interviewees reported using low-fidelity techniques to express UI flows, such as Photoshop layers, Excel spreadsheets, and sliding physical transparencies in and out of cases

(a glossy version of paper prototyping). However, they expressed their dissatisfaction with these methods since the methods often failed to convey the experience offered by the new design. In response, we designed d.tools to support rapid construction of concrete interaction sequences for experience prototyping [52] while leaving room to expand into higher-fidelity presentation models.

4.1.2 DESIGN PRINCIPLES

To guide the design of the d.tools authoring environment, we distilled the following design principles from our fieldwork observation and the general analysis of prototyping within the design process described in Chapter 2.

FAVOR CONCRETE, SPECIFIC INTERACTION SEQUENCES OVER GENERAL FUNCTIONALITY

The purpose of a UI prototype is to evoke the experience of using a future product, not to serve as an alpha version of the product. Exhibiting interactive behavior is a critical element for such prototypes, but only to the extent that it is needed to elicit the right feedback. Therefore, it is more important to rapidly build a concrete example of an interaction than to build general logic to handle different possible applications of the technique. Prototypes are concrete, narrow, and specific first; generalization and abstraction can be introduced at a later point. This guideline is a key differentiator between prototyping software and general programming tools.

MINIMIZE COGNITIVE FRICTION BETWEEN WORKING IN HARDWARE AND SOFTWARE BY BRIDGING ABSTRACTION LAYERS

When designing interactions for novel devices, more "moveable parts" exist than in traditional GUI design: the shape of the physical device, the type and layout of input and output components, and the mapping of input events to application logic have to be defined in addition to the standard concerns of interface appearance, information architecture, and behavior. To reduce some of the complexity of dealing with different levels of abstraction, d.tools introduces a device designer that serves as a virtual stand-in of the physical device being created. The goal of this device representation is to reduce the cognitive friction involved in switching between working with hardware and working with software.

OFFER IMMEDIATE, OBSERVABLE FEEDBACK ACROSS HARDWARE AND SOFTWARE

To allow the designer to experience their own design, the time between authoring a change and seeing that change, or between providing test input and observing the result, should be minimized. To this end, tight coupling between the software and hardware domains is used

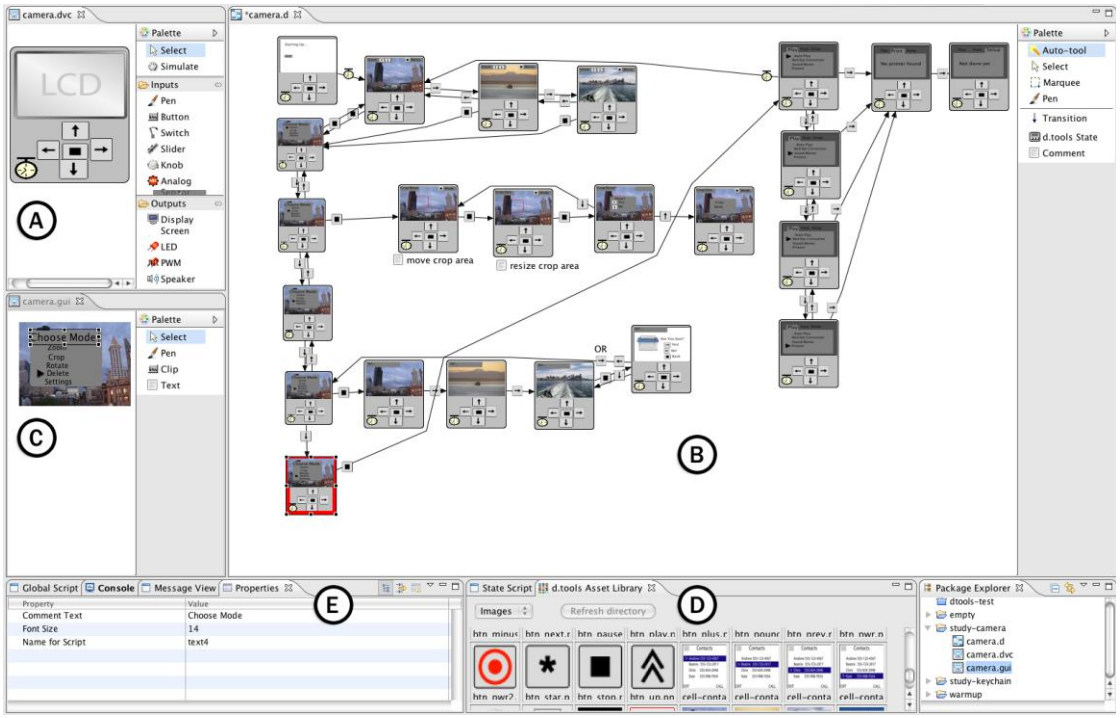

Figure 4.2: The d.tools authoring environment. **A**: device designer. **B**: storyboard editor. **C**: GUI editor. **D**: asset library. **E**: property sheet

when appropriate: an action in physical space (e.g., pressing a button) should have an immediate, observable result in the authoring environment. Vice versa, an action in the authoring environment (e.g., changing a screen graphic) should have an immediate observable result in hardware as well (e.g., show the changed graphic on an external display).

4.1.3 PROTOTYPING WITH D.TOOLS

In this section we discuss the most important interaction techniques that d.tools offers to enable the rapid design of interactive physical devices. d.tools' goal is to support design thinking rather than implementation tinkering. Using d.tools, designers place physical controllers (e.g., buttons, sliders), sensors (e.g., accelerometers, force sensitive resistors), and output devices (e.g., LEDs, LCD screens, and speakers) directly onto their physical prototypes. The d.tools library includes an extensible set of smart components that cover a wide range of input and output technologies. Software proxy objects of physical I/O components can be graphically arranged into a visual representation of the physical device (Figure 4.2A). On the PC, designers then author behavior using this representation in a visual language inspired by both storyboards and the statechart formalism [105] (Figure 4.2B). A graphical user interface editor enables composition of graphics for screen output (Figure

4.2C). Visual interaction models can be augmented by attaching code to individual states. d.tools employs a PC as a proxy for an embedded processor to prevent limitations of embedded hardware from impinging on design thinking. Designers can test their authored interactions with the device at any point in time, since their visual interaction model is always connected to the 'live' device.

4.1.3.1 Designing Physical Interactions with 'Plug and Draw'

Designers begin by plugging physical components into the d.tools hardware interface (which connects to their PC through USB) and working within the device designer of the authoring environment. When physical components are plugged in, they announce themselves to the d.tools authoring environment, creating virtual duals the device designer (Figure 4.3). Alternatively — when the physical components are not at hand or when designing interactions for a control that will be fabricated later — designers can create visual-only input and output components by dragging and dropping them from the device editor's palette. A designer can later connect the corresponding physical control or, if preferred, manipulate the behavior via Wizard of Oz [140,148] at test time.

In the device designer (Figure 4.2A), designers create, arrange and resize input and output components, specifying their appearance by selecting images from an integrated image browser, the asset library (Figure 4.2D). Building a virtual, iconic representation of the physical device affords rapid matching of software widgets with physical I/O components and reduces the cognitive friction of switching between working with hardware and working with software. The device design can also be used to simulate interaction with a device: by selecting a simulation tool from the palette, clicking (for discrete inputs) and dragging (for

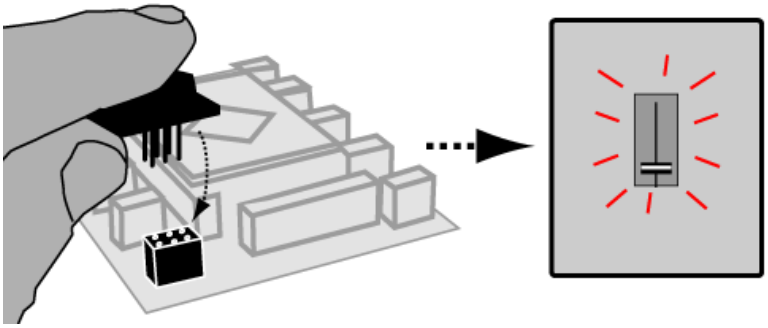

Figure 4.3: d.tools plug-and-play: inserting a physical component causes a corresponding virtual component to appear in the d.tools device designer.

continuous inputs) injects input events into the d.tools system as if the associated hardware input component had been pressed, moved, etc.

The component library available to designers comprises a diverse selection of inputs and outputs. Supported inputs include: discrete buttons and switches, rotary and linear potentiometers, rotary encoders, light sensors, accelerometers, infrared rangers, temperature sensors, force sensitive resistors, flex sensors, and RFID readers. Outputs include LCD screens, LEDs, DC motors, servo motors, and speakers. LCD and sound output are connected to the PC A/V subsystem, not our hardware interface. In addition, general purpose input and output circuit boards are available for designers who wish to build custom components. Physical and virtual components are linked through a hardware address that serves as a unique identifier of an input or output.

4.1.3.2 Authoring Interaction Models

Designers define their prototype's behavior by creating interaction diagrams in the storyboard editor (Figure 4.2B). States are graphical instances of the device design. They describe the content assigned to the outputs of the prototype at a particular point in the UI: screen images, sounds, and LED behaviors. States are created by dragging from the editor palette onto the storyboard canvas. As in the device editor, content can be assigned to output components of a state by dragging and dropping items from the asset library (Figure 4.2D) onto a component. All attributes of states, components and transitions (e.g., image filenames, event types, data ranges) can also be manipulated in text form via attribute sheets (editable tables that list attribute names and values – Figure 4.2E). To define graphic output, a graphical user interface editor provides common GUI design functionality: entering and positioning text, and loading, resizing and positioning graphical elements (Figure 4.2C). The designed graphical user interface is unique to each state.

Transitions represent the control flow of an application; they define rules for switching the currently active state in response to user input (hardware events). The currently active state is shown with a red outline. Transitions are represented graphically as arrows connecting two states. To create a transition, designers mouse over the input component which will trigger the transition and then drag onto the canvas. A target copy of the source state is created and source and target are connected. Transitions are labeled with an icon of the triggering input component (Figure 4.4A).

Conditions for state transitions can be composed using the Boolean AND/OR expressions (Figure 4.4B). A single Boolean connective is applied to all conditions on a

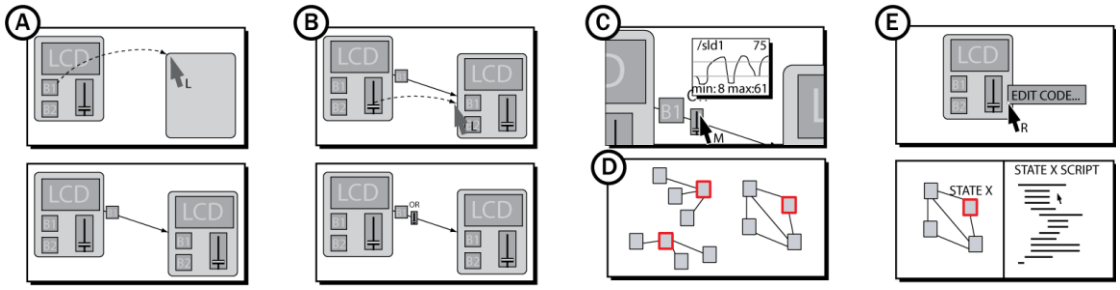

Figure 4.4: d.tools interaction techniques. **A**: creating new transitions through dragging. **B**: adding a new condition to an existing transition. **C**: Visualizing sensor signal input and thresholds in context. **D**: parallel active states. **E**: editing code attached to a state.

transition arrow, as complex Boolean expressions are error prone. Boolean combinations allow authoring conditionals such as 'transition if the accelerometer is tilted to the right, but only if the tilt-enable button is held down simultaneously.' More complex conditionals can be authored by introducing additional states.

To define discrete events for continuous sensors, designers define upper and lower thresholds for a sensor's value. Whenever the sensor value transitions into the threshold region, a transition event is generated. To help designers visualize such sensor thresholds, a graph showing both recent sensor history and threshold lines can be displayed on demand above the transition arrow utilizing the event (Figure 4.4C).

Within the visual editor, timers can be added as input components to a device to create automatic transitions or (connected with AND to a sensor input) to require a certain amount of time to pass before acting on input data. Automatic transitions are useful for sequencing output behaviors, and timeouts have proven valuable as a hysteresis mechanism to prevent noisy sensor input from inducing rapid oscillation between states.

While the storyboard aids a designer's understanding of the overall control flow of the prototype, complex designs still benefit from explanation. d.tools supports commenting with text notes that can be freely placed on the storyboard canvas.

4.1.3.3 Raising the Complexity Ceiling of Prototypes

The state-based visual programming model embodied in d.tools enables rapid design of the information architecture of prototypes, but the complexity of the control flow and interactive behavior that can be authored is limited. To support refining designs and permit higher-fidelity behaviors, d.tools provides two mechanisms that enable more complex interactions: parallel states and code extensions.

PARALLEL STATES

Expressing parallelism in single point-of-control state diagrams results in an exponentially growing number of states. Our first-use study also showed that expressing parallelism via cross products of states is not an intuitive authoring technique. To support authoring parallel, independent functionality, multiple states in d.tools can be active concurrently in independent sub-graphs (e.g., the power button can always be used to turn the device off, regardless of the other state of the model – Figure 4.4D). One limitation of parallel states in d.tools is that the system currently lacks an explicit mechanism to define what behavior should occur when two states try to assign output to the same component simultaneously.

ATTACHING CODE

To specify behaviors that are beyond the capability of the visual language (e.g., dynamically generating animations tied to user input), designers can attach textual code to visual states. The right-click context menu for states offers actions to edit and hook or unhook code for each state (Figure 4.7E). A d.tools API provides read and write access to hardware components, and allows procedural animation of graphics objects on screen. We implemented two different alternatives of d.tools code extensions — one with compiled Java classes, and one with interactively interpreted Java — to explore the tradeoffs of mixing visual and textual programming.

The compiled Java extension leverages the Eclipse programming environment's rich Java editing functionality. When users right-click on a visual state and choose the edit code command, d.tools generates a skeleton Java class file and launches the native Eclipse Java editor, which provides auto-completion, syntax highlighting, and integrated help. The primary benefit of this path is that is offers Eclipse's mature toolset. However, the toolset also brings with it a steep learning curve and a discontinuous authoring experience for two reasons. First, Eclipse targets professional software engineers and favors generality and completeness; many of the UI options offered are irrelevant to the more narrowly scoped task of writing state code in d.tools. Second, Java is a very verbose, strongly & statically typed, object-oriented language. The combination of these features requires designers to fully understand the object oriented development paradigm to make use of d.tools code extension — a barrier that proved too high in conversations with our target users.

In response to the identified complexity challenge, later versions of d.tools replaced the compiled Java architecture with an interactive interpreter which supports standard Java syntax but also offers "syntactic sugar" which results in much more concise code that focuses

on expressing the intended logic. The scripted Java extension trades off more concise and structurally simpler code against limited editor support for detecting syntax errors, suggesting corrections, and debugging.

EXECUTING INTERACTION MODELS AT DESIGN TIME

Designers can execute interaction models in three ways. First, they can manipulate the attached hardware. Second, they can imitate hardware events within the software workbench by using a simulation tool. Clicking on input components with the simulation tool then generate synthetic input events, e.g., button presses and release events, that are used to drive the interaction model as if real hardware input events had been received. Third, designers can employ a Wizard of Oz approach where they observe a user interacting with the prototype, and manually change the active state in the editor with their mouse.

4.1.4 ARCHITECTURE AND IMPLEMENTATION

Implementation choices for d.tools hardware and software emphasize both a low threshold for initial use and extensibility through modularity at architectural seams. In this section we describe how these design concerns and extensibility goals are reflected in the d.tools architecture.

4.1.4.1 Plug-and-Play Hardware

d.tools contributes a plug-and-play hardware platform that enables tracking identity and presence of smart hardware components for plug-and-play operation. I/O components for low-bandwidth data use a common physical connector format so designers do not have to worry about which plugs go where. Smart components each have a dedicated small microcontroller; an interface board coordinates communication between components and a

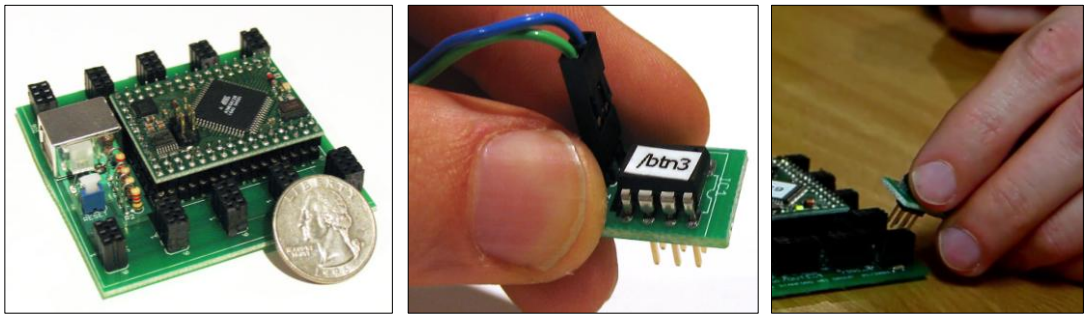

Figure 4.5: The d.tools hardware interface (left). Individual smart components (middle) are can be plugged into any bus connector (right).

PC (Figure 4.5). Components plug into the interface board to talk on a common I2C serial bus [32] (Figure 4.6). The I2C bus abstracts electrical characteristics of different kinds of components, affording the use of common connectors. The interface board acts as the bus master and components act as I2C slaves. A USB connection to the host computer provides power and the physical communication layer.

Atmel microcontrollers are used to implement this architecture because of their low cost, high performance, and programmability in C. The hardware platform is based around the Atmel ATmega128 microcontroller [22] on a Crumb128 development board from chip45 [172]. I/O components use Atmel ATtiny45 microcontrollers [23]. Programs for these chips were compiled using the open source WinAVR tool chain and the IAR Embedded Workbench compiler. Circuit boards were designed in CADsoft Eagle, manufactured by Advanced Circuits and hand-soldered.

d.tools distinguishes audio and video from lower-bandwidth components (buttons, sliders, LEDs, etc.). The modern PC A/V systems already provide plug-and-play support for audio and video; for these components d.tools uses the existing infrastructure. For graphics display on the small screens commonly found in information appliances, d.tools includes LCD displays which can be connected to a PC graphics card with video output. This screen is controlled by a secondary video card connected to a video signal converter. Displays that receive both graphics commands and power through a single USB connection are also becoming available and can be substituted.

4.1.4.2 Hardware Extensibility

Fixed libraries limit the complexity ceiling of what can be built with a tool by knowledgeable

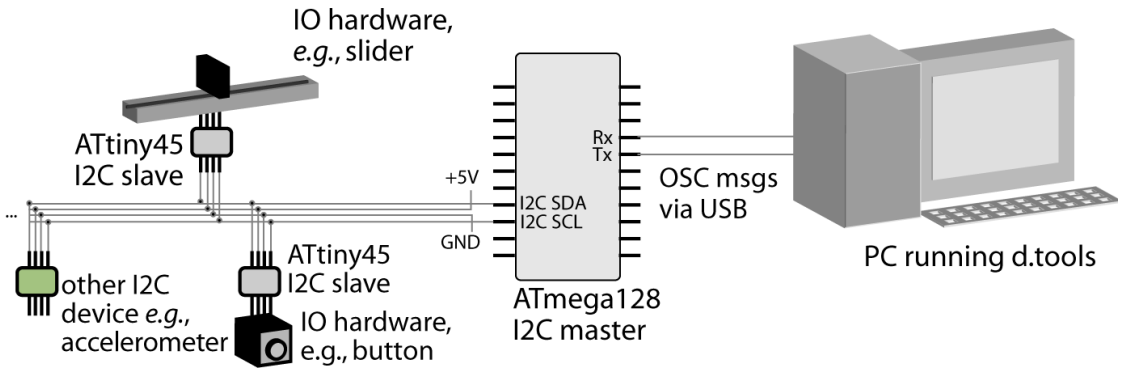

Figure 4.6: Schematic diagram of the d.tools hardware infrastructure. Smart components are networked on an I2C bus. A master microcontroller communicates over a serial-over-USB connection with the computer running the d.tools authoring environment.

users. While GUIs have converged on a small number of widgets that cover the design space, no such set exists for physical UIs because of the greater variety of possible interactions in the real world. Hence, extending the library beyond what 'comes with the box' is an important concern. In the d.tools software, extensibility is provided by its Java hooks. In the d.tools hardware architecture (Figure 4.6) extensibility is offered at three points: the hardware-to-PC interface, the hardware communication level, and the electronic circuit. This allows experts with sufficient interest and skill to modify d.tools to suit their needs.

d.tools hardware and a PC communicate by exchanging OpenSoundControl (OSC) messages [256] over a USB serial connection. OSC was chosen for its open source API, existing hardware and software support, and human readable addressing format (components have path-like addresses — e.g., buttons in d.tools are labeled `/btn1` or `/btn6`.) By substituting devices that can produce OSC messages or software that can consume them, d.tools components can be integrated into different workflows. For example, music synthesis programs such as Max/MSP [18] can receive sensor input from d.tools hardware. Connecting other physical UI toolkits to d.tools involves developing an OSC wrapper. As a proof of concept, we have written such a wrapper to connect Phidgets InterfaceKits [93].

Developers can extend the library of smart I/O components by adding components that are compatible with the industry standard I2C serial communication protocol. I2C offers a large base of existing compatible hardware. For example, the accelerometers used in d.tools projects are third party products that send orientation to d.tools via on-board analog-to-digital converters. Presently, adding new I2C devices requires editing of source code for the master microcontroller; this configuration step could also be pushed up to the d.tools authoring environment.

On the circuit level, d.tools can make use of inputs that vary in voltage or resistance and drive outputs with on/off control and pulse width modulation. This allows designers versed in circuit design to integrate new sensing and actuation technologies at the lowest level. This level of expansion is shared with other hardware platforms that offer direct pin access to digital I/O lines and A2D converters.

4.1.4.3 Software

To leverage the benefits of a modern IDE, d.tools was implemented in Sun's Java JDK 5 as a plug-in for the open-source Eclipse platform. Its visual editors are fully integrated into the Eclipse development environment [21]. d.tools uses the Eclipse Graphical Editing Framework (GEF) for graphics handling [24]. d.tools file I/O is done via serialization to XML, which enables source control of project files in ASCII format using version control tools. The design environment is platform independent except for "glue" code for USB port communication, and has been tested under Windows and Mac OS X.

CODE EXTENSIONS

The *compiled Java code extension* leverages the Eclipse programming environment's rich Java editing and compilation functionality. Eclipse automatically compiles the user's code into class files. d.tools, on entering a new state, uses a custom Java class loader to search for any new class files for the current state, and, if found, instantiates the class and calls its API methods. The initial d.tools Java API is reproduced in Table 4.1.

The subsequently developed *scripted code extension* model builds on BeanShell, a Java interpreter [200]. The interpreter's namespace is populated by d.tools with objects matching both hardware I/O components and graphics components defined in each state. For example, if a button with the name "upButton" exists in the device designer, then a variable corresponding to a Button object with name "upButton" will be present in the interpreter. Similarly, if a screen graphic object with the name "menuGraphic" is defined in a particular state, then a corresponding object with name "menuGraphic" will be accessible in the script

Function	Description
`enterState()`	Is called when the code's associated state receives focus in the statechart graph.
`update(String component, Object newValue)`	Is called when a new input event is received while the code's state has focus. The component's hardware address (*e.g.*, "/btn5" for a button) is passed in as an identifier along with the updated value (Booleans for discrete inputs, Floats for continuous inputs, and Strings for RFID tags).
`getInput(String component)`	Queries the current value of an input.
`setOutput(String component, Object newValue)`	Controls output components. LCD screens and speakers receive file URLs, and LEDs and general output components Booleans for on/off.
`println(String msg)`	Outputs a message to a dedicated debug view in our editor.
`keyPress(KeyEvent e)` `keyRelease(KeyEvent e)`	Inserts keyboard events into the system's input queue (using Java Robots) to remote control external applications.

Table 4.1: The d.tools Java API allows designers to extend visual states with source code. The listed functions serve as the interface between designers' code and the d.tools runtime system. Standard Java classes are also accessible.

of that state. The technique of creating objects based on name properties entered in a direct manipulation interface is also present in other GUI editors such as Adobe Flash.

For programming animations, the interpreter uses a polling method for calling user-defined graphics routines. If the user defines a function called loop(), that function is called repeatedly at 30Hz, a rate sufficient for generating animations. This polling technique is conceptually more straightforward than event callbacks and was inspired by its successful use in the end-user graphics programming environment Processing [210].

A challenge we noted in the compiled Java model was that persisting data for sharing between different states was cumbersome. To facilitate defining globally accessible variables and functions, the scripted Java extension therefore added the concept of a "global script" in addition to individual state scripts. Variables and functions declared in the global script are accessible to all state scripts. Global scripts are reloaded whenever the project files are saved. State-specific scripts are executed whenever the associated graphical state becomes active. The complete d.tools scripting API is reproduced in Table 4.2 and some examples of the API in use are given in Figure 4.7

```
//working with text objects
text1.setText("Hello");
text1.setText(text1.getText()+", World!");
text1.setFontSize(24);

//resize the image "clipImg" based on
//two button events
loop() {

  //scale down
  if(btnDown.getValue())
    clipImg.setScale(clipImg.getScale()-5);

  //scale up
  if(btnUp.getValue()) {
    clipImg.setScale(clipImg.getScale()+5);

  //center image on stage
  clipImg.setXY(stage.getWidth()/2-
                clipImg.getWidth()/2,
                stage.getHeight()/2-
                clipImg.getHeight()/2);
}
```

Figure 4.7: Code examples for the d.tools scripting API.

Global Functions for Drawing and Accessing Hardware	
Function	**Description**
`void loop()`	If the user's state script defines a loop function, the function will be called repeatedly at interactive rates while the given state is active.
`void print(String msg)`	Print a message to the debug console.
`boolean ditigalRead(String compName)` `void digitalWrite (String compName, boolean value)`	Read the last known state of a discrete input component such as a button or switch with identifier compName, Write a new value.
`float analogRead (String compName)` `void analogWrite (String compName, float value)`	Read the last known value of a continuous input. Write to a pulse-width modulated output component such as a PWM LED
Hardware Component Proxy Objects	
`component.setValue(boolean v)` `component.setValue(float v)`	Set the state of the component named "component"; overloaded based on component type (discrete or PWM output). Corresponds to digitalWrite() and analogWrite() above.

`boolean component.getValue()`
`float component.getValue()`
`int component.getValue()`
	Read the last known state of the component named "component"; overloaded based on component type (discrete, continuous or identity-reporting). Corresponds to digitalRead() and analogRead() above.

GUI Objects: Stage	
`int getWidth(), int getHeight()`	Return the width and height of stage, as defined in the device designer.
`void setColor (int r, int g, int b)` `int getColor(String which)`	Set/get the stage color.
GUI Objects: Graphic Clips	

`void setWidth(int w)`
`void setHeight(int h)`
`int getWidth(), int getHeight()`
	Set/get the width and height of the clip object.

`int getX(), int getY()` `void setX(int x), void setY(int y)`	Set/get the position of the clip object.
`setVisible(boolean v)` `boolean isVisible()`	Set/get the visibility of the clip object.
`setImage(String filename)` `String getImage()`	Set/get the image displayed by this clip object.
GUI Objects: Text	
`setX(int x), setY(int y),` `int getX(), int getY()`	Set/get the position of the text object.
`setVisible(Boolean v)` `boolean isVisible()`	Set/get the visibility of the text object.
`setText(String text)` `String getText()`	Set/get the string displayed by the text object.
`setFontSize(int size)` `int getFontSize()`	Set/get the text font size.

Table 4.2: The d.tools scripting API provides both global and object-oriented functions to interact with hardware, and a concise object-oriented set of function for manipulating GUI elements.

4.1.5 EVALUATION

In this section, we outline the methodological triangulation we employed to evaluate and iteratively refine d.tools. First, to ascertain the expertise threshold of d.tools, we conducted a first-use lab study with thirteen design students and professional designers. Second, the author and other members of our research group rebuilt prototypes of existing devices and used d.tools in two research projects. Third, we made d.tools hardware kits available to students in a project-centric interaction design course at our university.

4.1.5.1 Establishing Threshold with a First Use Study

We conducted a controlled laboratory study of d.tools to assess the ease of use of our tool; the study group comprised 13 participants (6 male, 7 female) who had general design experience. Three participants served as pilot testers to refine the testing protocol. Participants were given three design tasks of increasing scope to complete with d.tools within 90 minutes. Most participants were students or alumni of design-related graduate programs at our university.

Sessions started with a demonstration of the d.tools software editor and the hardware components by the experimenters. We then gave participants two narrowly defined tasks and one open-ended design project. For the first task, participants were asked to complete a menu navigation design that the experimenter had started during the demonstration. For the second task, participants were asked to build a functional physical prototype of a device with one button and one switch as inputs, and one LED and a speaker as outputs. Pressing the button should play a sound clip and toggling the switch should turn the LED on or off. The two components were to function independently of each other.

The third assignment was to begin prototyping a digital music player for children. Participants were given written guidelines such as "children prefer dedicated controls and like elements that move better than buttons." As the study allotted only 30 to 45 minutes for this part, participants were informed that they were not expected to produce a finished product. To sketch and build physical prototypes, we provided an 18" × 24" paper pad, sheets of foam core, pens, a selection of tools, glue and tape, and a label printer. As the final step of the study, participants were asked to complete a 26 question survey.

STUDY RESULTS

All participants successfully completed both close-ended tasks, regardless of prior experience in user interface design or physical computing. Task one took a mean of 9 minutes while task two took a mean of 24 minutes to complete (Figure 4.8).

For the music player design task, participants followed heterogeneous approaches: some started by exploring the ergonomics of different shapes to determine input component placement; others focused on requirements analysis on paper; yet others worked exclusively in software. d.tools was most frequently used for determining layout of interaction components in the device designer, and reasoning about the interaction model in the storyboard designer. Two participants with prior physical computing experience built functional physical prototypes with navigation and sound playback in less than 30 minutes.

SUCCESS OF A LOW THRESHOLD AND TIGHT COUPLING

Almost all users commented positively on the tight coupling of hardware components and their software counterparts, especially the automatic recognition of hardware connections. Authoring storyboards through link-and-create actions was immediately intuitive. Refining default behaviors through text properties and expressing functional independence in a

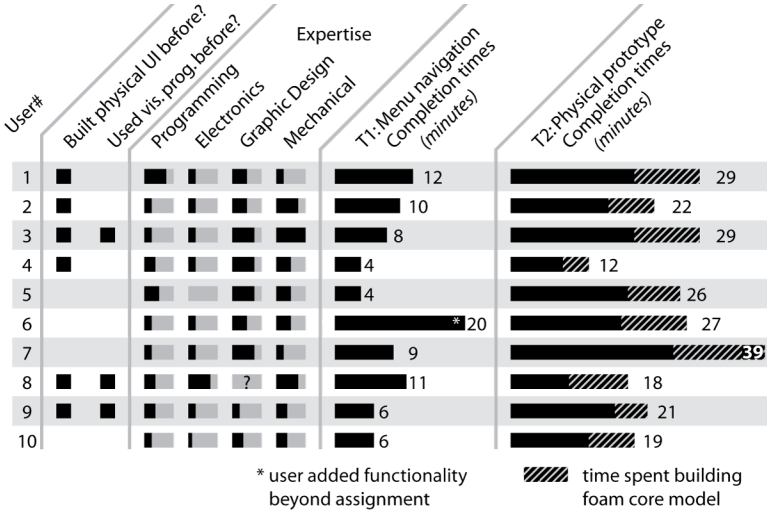

Figure 4.8: Task completion times, and prior experience and expertise of d.tools study participants. Participants completed task 1 in an average of 9 minutes, and task 2 in an average of 24 minutes. These times demonstrate that prototyping with d.tools is fast enough to be appropriate for early-stage design.

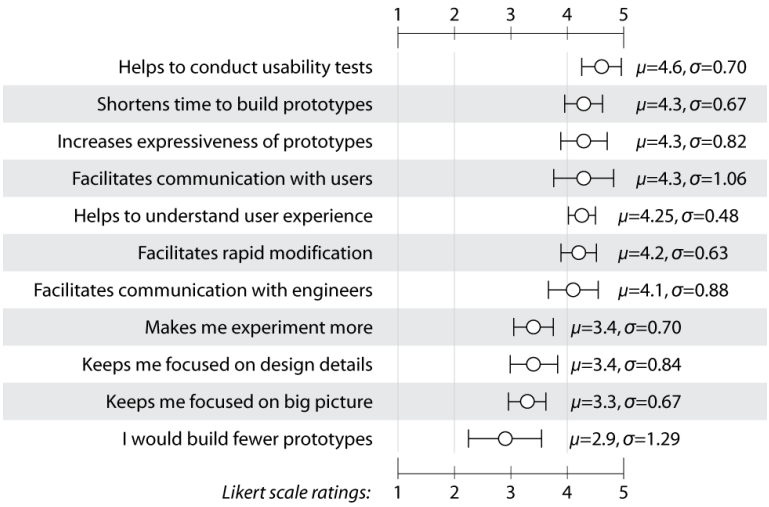

Figure 4.9: Post-test survey results from the d.tools user study. Participants provided responses on Likert scales.

storyboard took longer; nevertheless, participants mastered these strategies by the end of the session.

After an initial period of learning the d.tools interface, participants spent much of their time with *design thinking*—reasoning about how their interface should behave from the user's point of view instead of wondering about how to implement a particular behavior. This was especially true for authoring UI navigation flows.

The experimenter asked participants to hand over the devices built for the second task to test whether the required functionality had been achieved —while observing this on-the-spot user test, many subjects expressed the wish to iterate on their designs and produced another version two to ten minutes later. This suggests the advantage of the rapid iteration cycles that d.tools enables. In a post-test survey (Figure 4.9), participants consistently gave d.tools high marks for enabling usability testing (μ=4.6 on 5 point Likert scale), shortening the time required to build a prototype (μ=4.3), and helping to understand the user experience at design time (μ=4.25).

NEEDS: SOFTWARE SIMULATION, LARGER LIBRARY, RICHER FEEDBACK

One significant shortcoming discovered through the study was the lack of software simulation of an interaction model: the evaluated version did not provide a mechanism for stepping though an interaction without attached hardware. . This prompted the addition of our software simulation mode. Specifying sensor parameters textually worked well for subjects who had some comfort level with programming, but were judged disruptive of the

visual workflow by others. Interaction techniques for graphically specifying sensor ranges were added to address this issue. Users also wished for aggregate inputs that have become standard navigation elements for information appliances such as combined up/down buttons, five-way joysticks, and keypads.

4.1.5.2 Rebuilt Existing and Novel Devices

To evaluate the expressiveness of d.tools' visual language, we recreated prototypes for four existing devices — an Apple iPod Shuffle music player, the back panel of a Casio EX-Z40 digital camera (Figure 4.10A), Hinckley et al.'s Sensing PDA [121] (Figure 4.10B), and Partridge et al.'s TiltType [202] text entry device. The iPod Shuffle is a digital music player without a screen where all playback options are controlled through tactile switches. For the digital camera, we prototyped image review mode, where users can navigate through images, zoom, crop, and delete images. The Sensing PDA uses an accelerometer to detect device pose and adjust display orientation accordingly. An infrared distance sensor can detect whether the device is held close to a user's face; a force-sensitive touch sensor detects whether the device is held. The TiltType device uses a dual-axis accelerometer in conjunction with momentary switches under the user's index fingers to explore orientation-based text entry techniques. We distilled the central functionality of each device and prototyped these key interaction paths.

Additionally, two research projects in our group used d.tools to provide physical input for table and wall interfaces. The *tangible drawers* project explored physical drawers as a file access metaphor for a shared tabletop display [112]. The author built four drawer mechanisms mounted underneath the sides of a DiamondTouch interactive table (Figure 4.10C, Figure 4.10D). Opening and closing these drawers controlled display of personal data collections, and knobs on the drawers allowed users to scroll through their data. Ju et al. used d.tools to explore proxemics for interactive whiteboards through an array of infrared distance sensors mounted to the frame of a wall display [137] (Figure 4.10E).

From these exercises, we learned that interactive physical prototypes have two scaling concerns: the complexity of the software model, and the physical size of the prototype. d.tools diagrams of up to 50 states are visually understandable on a desktop display (1920 × 1200); this scale is sufficient for the primary interaction flows of current devices. Positioning and resizing affords effective visual clustering of subsections according to gestalt principles of proximity and similarity. However, increasing transition density makes maintaining and troubleshooting diagrams taxing, a limitation shared by other visual authoring environments.

Figure 4.10: Some applications built with d.tools in our research group. **A**: digital camera image navigation. **B**: sensor-enhanced smart PDA. **C** & **D**: tangible drawers for a multi-user interactive tabletop. **E**: proxemics-aware whiteboard. **F**: TiltType for orientation-based text entry.

In the tangible drawers project, the presence of multiple independent drawers prompted the need for multiple concurrently active states. This project as well as Range also required sensor data access from an existing Java application. d.tools can interact with existing applications in one of two ways: state change information and raw sensor data can be received by a 3rd party application using socket communication; or d.tools can inject mouse and keyboard events into the operating system event queue, a technique termed *screen poking* (similarly to Hudson's Thumbtacks project [127]). The first method was used to interface with the our other research project code bases; it raises the question which parts of the interaction should be authored in d.tools, and which parts in the external application's source code. Screen poking was used by the author to prototype accelerometer-based zoom and pan control for the Google Earth application in less than 30 minutes. However, screen poking is a brittle technique as d.tools is unaware of the internal state of the controlled application; it is therefore less useful for more complex prototypes.

The first author also served as a physical prototyping consultant to a prominent design firm. Because of a focus on client presentation, the design team was primarily concerned with the polish of their prototype — hence, they asked for integration with Flash. From a research standpoint, this suggests — for "shiny prototypes" — a tool integrating the visual richness of Flash with the computational representation and hardware abstractions of d.tools.

4.1.5.3 Teaching Experiences — HCI Design Studio

We deployed the d.tools hardware and software to student project teams in a master's level HCI design course at Stanford [150]. Students had the option of using d.tools (among other technologies) for their final project, the design of a tangible interface. Seven of twelve groups used d.tools. In the following year, d.tools was offered again to students in the class. In this real-world deployment, we provided technical assistance and tracked usability problems, bug reports, and feature requests. Figure 4.11 provides an overview of some projects built by students.

SUCCESSES

Students successfully built a range of innovative interfaces. Examples include a wearable watch that allows children to record and trade secret audio messages, a color mixing interface in which children can "pour" color from tangible buckets onto an LCD screen, and an augmented clothes rack that offers product comparisons and recommendations via hanger sensors and built-in lights.

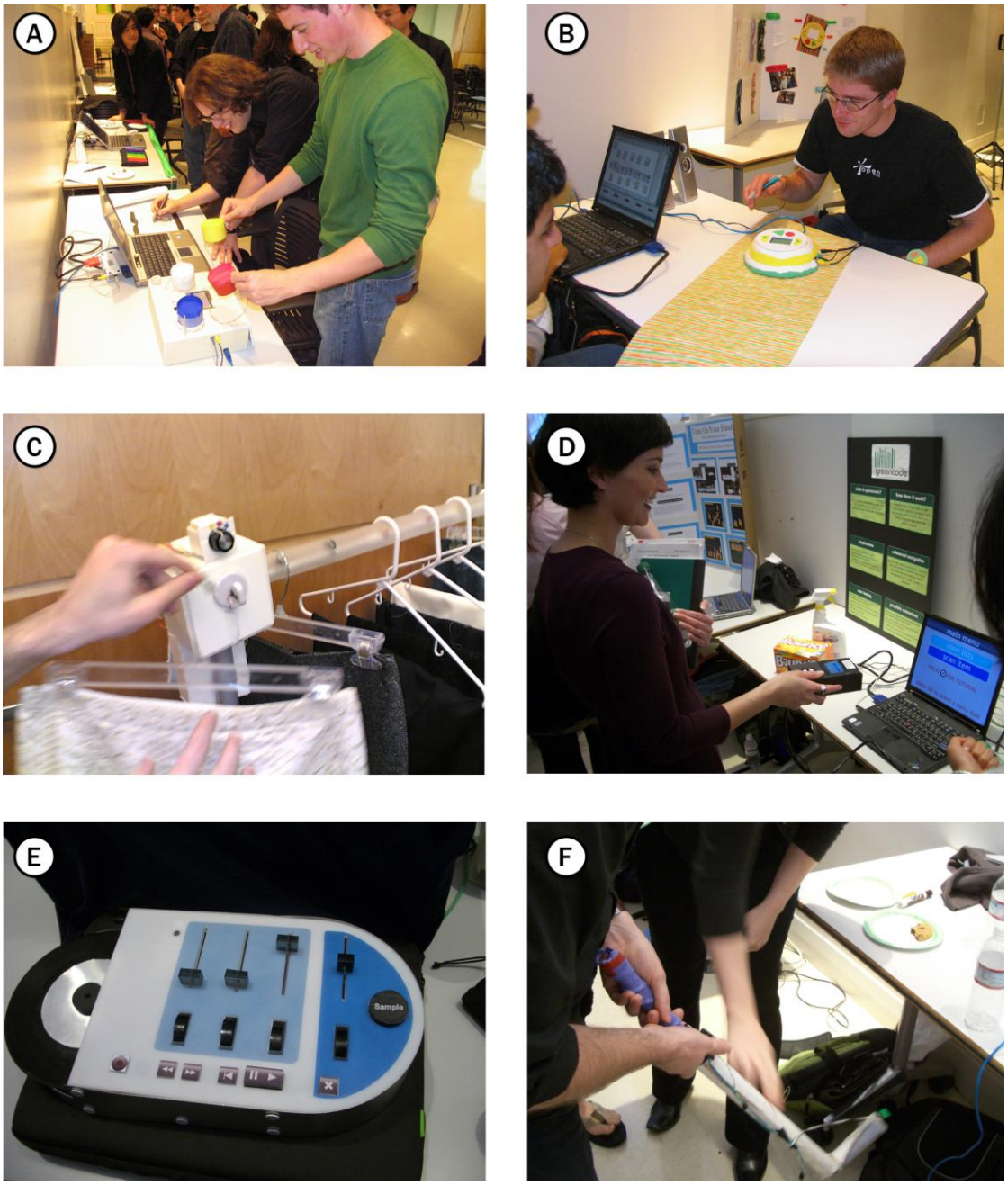

Figure 4.11: Some student projects built with d.tools. **A**: a tangible color mixing device where virtual color can be poured from physical paint buckets by tilting them over an LCD screen. **B**: a message recording system for children to exchange secrets. **C**: a smart clothes rack can detect which hangers are removed from the rack and display fashion advice on a nearby screen. **D**: a mobile shopping assistant can scan barcodes of grocery items and present sustainability information relating to the scanned item on its screen. **E**: a tangible audio mixer to produce cell phone ring tones. **F**: an accelerometer-equipped golf club used as a game controller.

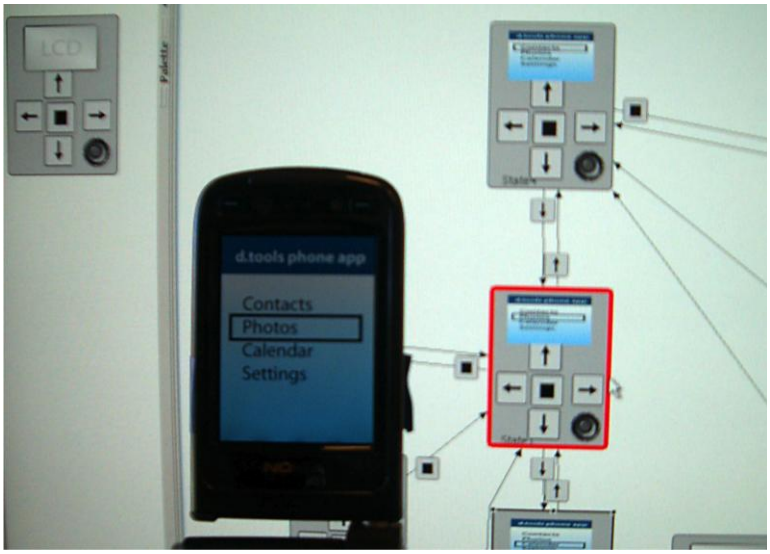

Figure 4.12: A d.tools mobile prototype on a Nokia N93 smart phone, with the storyboard logic of the prototype in the background.

Students were able to work with supplied components and extend d.tools with sensor input not in the included library. For example, the color mixing group integrated mechanical tilt switches and vibration motors into their project.

SHORTCOMINGS DISCOVERED

Remote control of third party applications (especially Flash) was a major concern for students — in fact, because d.tools did not have a graphical user interface editor in the supplied version, two student groups chose to develop their project with Phidgets [93], as it offers a Flash API. To address this need, we first released a Java API for the d.tools hardware with similar connectivity. We observed that student groups that used solely textual APIs ended up writing long-winded state machine representations using switch or nested conditional statements; the structure of their code could have been more concisely captured in our visual language. The need for direct control over GUI graphics also motivated the later addition of the d.tools graphical user interface editor.

4.1.6 D.TOOLS MOBILE

The d.tools architecture was designed to focus on prototypes that involve custom hardware. Might it also offer benefits for prototyping interfaces for commodity hardware, such as smart phones? To understand the utility of d.tools for mobile interaction design, we collaborated with Nokia to enable real-time input from and output to smart phones (Figure 4.12).

With the d.tools mobile system, designers author functionality in the standard visual environment. Designers do not need to create a device definition; they can load a pre-created model that matches layout of phone input keys and screen. For running and testing such prototypes, a custom d.tools client application is loaded onto a phone. This client intercepts all input events (i.e., key presses) and sends them over a wireless connection to the PC running d.tools, where they are used to trigger state transitions. Output commands resulting from state transitions are then sent to the phone to display graphics or play sounds (Figure 4.13). In essence, the phone is turned into a terminal, while all application logic executes in the d.tools authoring environment. Our current implementation was written in Python for Nokia S60 phones. We are using a Wi-Fi connection for message passing. Messages are sent as OpenSoundControl packets over UDP.

BENEFITS

We have tested the d.tools mobile approach informally in our lab and with collaborators at Nokia. A primary benefit of our approach is that it sidesteps many of the pain points of developing and deploying prototypes on phones, since development and execution both remain on the PC. In addition, the state of the phone application can be monitored in d.tools. It is also possible to change the interaction logic in the middle of a test. d.tools mobile is especially suited for quick exploration of applications with relatively static individual screens — a storyboard can be assembled in a few minutes and tested on the target device. Because of the reliance on a common messaging protocol, OSC, it is also possible to add external sensors connected to a d.tools hardware interface and explore interactions that rely on sensor input not provided by the phone itself.

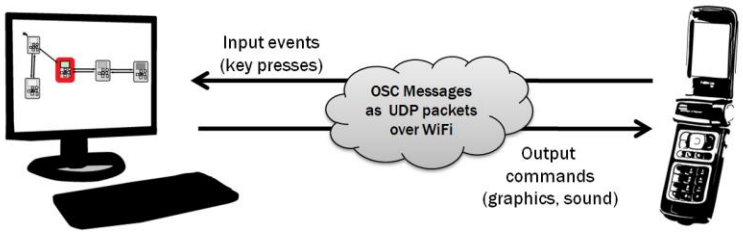

Figure 4.13: The d.tools mobile system architecture uses socket communication over a wireless connection to receive input events and send output commands to a smart phone.

LIMITATIONS

d.tools mobile also has multiple important limitations, some fundamental to its execution model, others merely due to its nature as research software. Fundamentally, interactivity is limited by the roundtrip latency of sending an event to d.tools, and receiving a message with output commands in return. Mobile devices have to trade off network latency and battery life, and as a result, we observed roundtrip latencies of 200-1000 milliseconds, with a large amount of jitter (the variation in latency). While fast enough for discrete control tasks such as navigating from screen to screen, d.tools mobile is not fast enough for continuous control tasks such as smooth panning or zooming. Latency and jitter will further increase if users try to take the device out of the lab and switch from a wireless Ethernet connection to a cellular data connection. This limits the applicability of d.tools mobile for testing outside the lab.

An important pragmatic limitation of our implementation is that the d.tools scripting language has not been ported to d.tools mobile yet. Thus dynamic behaviors cannot yet be implemented. Adding script execution is not trivial as it requires deciding which commands should be executed on the phone itself, and which commands should be executed on the PC. For example, graphics commands such as translation, rotation, and scaling are best executed on the phone itself so large graphics files don't have to be transmitted. Also, only keyboard input is currently supported. Although processing data from built-in phone sensors is certainly possible, the appropriate modules exposing such data to the Python programming language were not available to us.

To understand the relative benefits and limitations of d.tools mobile, we compare design decisions of d.tools mobile and important related work in Table 4.3. Ballagas' iStuff Mobile [42] is the most related system, as it also executes logic on a PC. iStuff mobile targets higher-fidelity development of mobile applications where phones are one among multiple devices in a ubiquitous computing ecology, while d.tools mobile targets lower-fidelity UI walkthroughs.

	d.tools mobile	iStuff mobile	Flash Lite	Juxtapose mobile
Authoring Environment	Visual (d.tools state diagrams)	Visual + Code (Quarts Composer + JavaScript)	Visual + Code (Adobe Flash IDE + ActionScript)	Code (ActionScript)
Where does computation happen?	PC	PC	Phone	Phone
Supported Input	Phone keyboard, external sensors	Phone keyboard, external sensors	Phone hardware only	Phone hardware only
Supported Output	Phone screen, sound	Phone screen, sound, external screens	Phone screen, sound	Phone screen, sound
Can application be inspected while running?	Yes	No?	No	No
Can application be modified while running?	Yes	No	No	Yes (Tuning + Alternatives)

Table 4.3: Comparison of d.tools mobile and related mobile prototyping tools.

4.1.7 LIMITATIONS & EXTENSIONS

To conclude our discussion of the d.tools system, this final section points out important limitations of the current architecture and implementation, and suggests paths for extensions.

4.1.7.1 Dynamic Graphics Require Scripting

One important limitation of the current d.tools authoring environment is that achieving dynamic graphic output, e.g., continuous animations, is only possible through the built-in scripting API; it cannot be authored visually. This is partially a side-effect of choosing states as the first-level abstraction. Consequently, information architecture can be rapidly prototyped, but more detailed work on temporal aspects of the user interface is not well supported.

Commercial tools [6,1] exist that focus on rapid creation of animated traditional, desktop-bound user interfaces. Flash Catalyst for instance also uses states as an abstraction principle, but lets designers specify explicitly how to animate transitions for individual graphical elements for each transition. Other research has looked into how to specify animations directly through stylus input [68,164]. However, it is likely insufficient to translate these techniques directly into the d.tools environment, as they do not offer support

for binding animation to the variety of possible input devices and input events in d.tools. Promising directions for this problem are to either use a visual dataflow paradigm [129] to link input events to graphical objects or to author constraints by demonstration [164].

4.1.7.2 Hierarchical Diagrams Not Supported

d.tools in its current version does not support hierarchical levels of abstraction for states. This limits the complexity of prototypes that can be built with d.tools. While we implemented parallel state machines (independent sub-graphs where one state is currently active in each sub-graph), we did not implement support for hierarchical abstraction. Abstraction has three primary benefits:

1. expressing multi-level logic, e.g., events that should apply to a set of states
2. enabling reuse of previously authored components
3. preserving screen real estate by collapsing the visual representation of clusters

Harel's original conception of statecharts [105] derives its visual economy from the notion of state clusters. Clustering also exists in dataflow languages such as Max/MSP [18]. One challenge with introducing a more powerful authoring abstraction is ensuring that this concept does not raise the expertise threshold required for novices to the tool. In informal testing, we found the notion of parallel states not well received by designers. One reason is that in parallel states, what will be shown to the user of a designed prototype is never completely visible in a single point in the diagram. Reasoning about program state now requires mentally combining the behavior of multiple active states. Similarly, reasoning about "what happens next" can be quite complex under multiple states active in parallel.

4.1.7.3 Screen Real Estate Not Used Efficiently

The current version of d.tools needlessly expends a large area of screen real estate by repeatedly displaying the device design, i.e., the set of input and output components arranged in a 2D layout, for every state in the state diagram. Having such a depiction of the device in each state enables the current authoring technique for creating transitions: clicking on an input component, then dragging out an arrow and releasing over a different state. But most of the pixels dedicated to this state display are only needed during this transition authoring. At other times, they clutter the diagram and reduce overall legibility. Hiding the device design would free up more pixels which could either be used to show more complex diagrams, or to devote more screen real estate to showing the graphical output of each state, by making the states bigger.

One possible solution achieve space savings while keeping the current authoring technique would be a dynamic visualization where the device design is hidden by default and states only show output and transitions. On mouse-rollover or another explicit invocation mechanism, the full design is temporarily shown to allow easy transition authoring. The implementation of such a technique is straightforward for the standard GUI case, where there is only a single display. It is less clear how to automatically create a suitable state abstraction when multiple output components are defined. Two possible options are to automatically rearrange the component layout; or to give the designer explicit control over how this second representation should look in the device designer.

4.1.7.4 Lack of Support for Actuation

While d.tools supports output to LEDs, DC motors, and servo motors, most of our effort has been concentrated on how to support sensor data input. We have not yet sufficiently explored the space of more complex actuation. In particular, output is controlled at the single output component level — one has to author behavior for each LED in each state individually. Such limitations are analogous to programming screen output by only writing single pixels. Many interactive projects employ arrays of displays or mechanical actuators (e.g.,Hansen and Rubin's Listening Post [104], or Rozin's Wooden Mirror [215]). Tools should therefore support output abstractions that address collections of output devices. Also, hardware and power supply design becomes a consideration when dealing with multiple outputs. The current hardware interface would need redesign to support a wider variety of actuators.
Yet a different problem arises from the tethered nature of the d.tools kit: one can't currently explore interactions that rely on precise timing and low latency feedback loops, such as for haptic interactions. Haptic motor control routines require update rates near 1kHz. In the current d.tools architecture, control loops execute at less than 100Hz, because every message has to be relayed from the hardware interface to the PC and back.

4.1.7.5 Prototypes Have to be Tethered to PC by Wire

d.tools prototypes (other than d.tools for mobile phones) are currently restricted to be used inside the design studio because because of the required tether cable linking them to a PC. The tether has two functions: it is used for data exchange, since the interaction model itself lives on the PC, and it provides power for the hardware interface sensors (actuators may need additional, separate power). There are two general strategies for cutting this tether.

First, replacing the cable with a wireless data connection and operating the hardware platform with batteries. D.tools mobile follows this approach: mobile phones send input events to the PC over a WiFi connection, and receive output events in return. The advantage of this approach is that the designer can follow in real-time on the PC what state the prototype is in, and can make on-the-fly changes. The disadvantage is that one has to be within range of the wireless signal.

A second approach is to execute interaction models directly on embedded hardware. This could be achieved by either a) running the d.tools Java state machine code on an embedded processor that can execute Java or b) by generating code for the target embedded platform separately. We have done preliminary work in the first direction by connecting components to an embedded Intel XScale platform that can execute interaction models. Stepping beyond 8-bit microcontrollers also enables on-board graphics. The advantage of this approach is that the created devices are completely standalone and do not require a PC anymore. The disadvantage is that prototype behavior can no longer be tracked and visualized on the PC.

4.2 EXEMPLAR: PROGRAMMING SENSOR-BASED INTERACTIONS BY DEMONSTRATION

d.tools and other physical computing toolkits have lowered the threshold for connecting sensors and actuators to PCs [37,43,93,127,159], and for prototyping the application logic of systems that make use of sensors and actuators. Accessing sensor data from software has come within reach of designers and end users.

However, our experience of deploying d.tools in the classroom showed that specifying the relationship between sensor input and application logic remains problematic for designers and students alike for three reasons. First, most current tools, such as Arduino [185], require using textual programming to author sensor-based behaviors. Representations are most effective when the constraints embedded in the problem are visually manifest in the representation [201]. Thus, numbers alone are a poor choice for making sense of continuous signals as the relationship between performed action and reported values is not visually apparent. Second, existing visual tools (e.g., LabView [15]) were created with the intent of helping engineers and scientists perform signal analysis; as such, they do not support straightforward authoring of interactions. This leaves users with a significant gulf of execution, the gap between their goals and the actions needed to attain those goals with the system [132]. Third, the large time and cognitive commitment implied by a lack of tools inhibits rapid iterative exploration. Creating interactive systems is not simply the activity of translating a pre-existing specification into code; there is significant value in the epistemic experience of exploring alternatives [145]. One of the contributions of direct manipulation and WYSIWYG design tools for graphical interfaces is that they enable this 'thinking through doing' — the aim of our work is to provide a similarly beneficial experience for sensor-based interactions.

This section contributes techniques for enabling a wider audience of designers and application programmers to turn raw sensor data into useful events for interaction design through programming by demonstration. It introduces a rapid prototyping tool, Exemplar (Figure 4.14), which embodies these ideas. The goal of Exemplar is to enable users to focus on design thinking (how the interaction should work) rather than algorithm tinkering (how the sensor signal processing works). Exemplar frames the design of sensor-based interactions as the activity of performing the actions that the sensor should recognize — we suggest this approach yields a considerably smaller gulf of execution than existing systems. With Exemplar, a designer first demonstrates a sensor-based interaction to the system (e.g., she

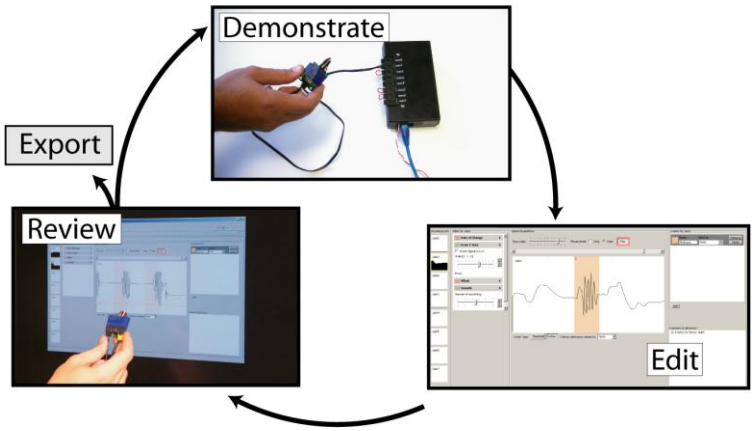

Figure 4.14: Iterative programming by demonstration for sensor-based interactions: A designer performs an action; annotates its recorded signal in Exemplar; tests the generated behavior; and exports it to d.tools.

shakes an accelerometer). The system graphically displays the resulting sensor signals. She then edits that visual representation by marking it up, and reviews the result by performing the action again. Through iteration based on real-time feedback, the designer can refine the recognized action and, when satisfied, use the sensing pattern in d.tools or other prototyping applications. The primary contributions of this work are:

1) A method of programming by demonstration for sensor-based interactions that emphasizes designer control of the generalization criteria for collected examples.
2) Integration of direct manipulation and pattern recognition through a common visual editing metaphor.
3) Support for rapid exploration of interaction techniques through the application of the design-test-analyze paradigm [109,148] on a much shorter timescale as the core operation of a design tool.

Programming by demonstration as a technique was introduced in Chapter 3.5. The rest of this section is organized as follows: we first describe relevant characteristics of sensors and sensor-based interactions to position our work. We provide an overview of the design principles embodied in Exemplar, then describe the research system, its interaction techniques and implementation. We finally report on two evaluation methods we have employed to measure Exemplar's utility and usability.

4.2.1 SENSOR-BASED INTERACTIONS

This section introduces an analysis of the space for sensor-based interactions from the designer's point of view. Prior work has successfully used design spaces as tools for thinking about task performance [57] and communicative aspects [46] of sensing systems. Here we apply this approach to describe the interaction designer's experience of working with sensors. This design space foregrounds three central concerns: the nature of the input signals, the types of transformations applied to continuous input, and techniques for specifying the correspondence between continuous signals and discrete application events.

4.2.1.1 Binary, Categorical, and Continuous Signals

As prior work points out [214], one principal distinction is whether sensing technologies report continuous or discrete data. Most technologies that directly sample physical phenomena (e.g., temperature, pressure, acceleration, magnetic field) output continuous data. For discrete sensors, because of different uses in interaction design, it is helpful to distinguish two sub-types: binary inputs such as buttons and switches are often used as general triggers; while categorical data inputs (multi-valued) such as RFID are principally used for identification. A similar division can be made for the outputs or actuators employed. Exemplar focuses on continuous input in one or more dimensions; it does not support working with categorical input data.

4.2.1.2 Working with Continuous Signals

Sensor input is nearly always transformed for use in an interactive application. Continuous transformation operations fall into three categories: signal conditioning, calibration, and mapping. Signal conditioning is about 'tuning the dials' so the signal provides a good representation of the phenomenon of interest, thus maximizing the visual signal-to-noise ratio. Common steps in conditioning are de-noising a signal and adjusting its range through scaling and offsetting. Calibration relates input units to real-world units. In scientific applications, the exact value in real-world units of a measured phenomenon is of importance. However, for the majority of sensor-based interfaces, the units of measurement are not of intrinsic value. Mapping refers to a transformation from one parameter range into another. Specifying how sensor values are mapped to application parameters is a creative process, one in which design intent is expressed. Exemplar offers support for both conditioning sensor signals and for mapping their values into binary, discrete, or continuous sets. When calibration is needed, experts can use Exemplar's extensible filter model.

4.2.1.3 Generating Discrete Events

A tool to derive discrete actions from sensor input has to choose both a detection algorithm and appropriate interaction techniques for controlling algorithm parameters. The computationally most straightforward approach is thresholding — comparing a single data point to fixed limits. However, without additional signal manipulations, e.g., smoothing and derivatives, thresholds are susceptible to noise and cannot characterize events that depend on change over time. Matching tasks such as gesture recognition require more complex pattern matching techniques. Exemplar offers both thresholding with filtering and pattern matching.

Equally important is the user interface technique employed to control how the computation happens. Threshold limits can be effectively visualized and manipulated as horizontal lines overlaid on a signal graph. The parameters of more complex algorithms are less well understood in our experience. Exemplar thus frames threshold manipulation as the principal mechanism for authoring discrete events. Exemplar contributes an interaction technique to cast parameterization of the pattern matching algorithm as a threshold operation on matching error. Through this technique, Exemplar creates a consistent user experience for authoring with both thresholding and pattern matching.

4.2.2 DESIGN PRINCIPLES

The following four design principles were derived from our analysis of sensor-based interactions.

FOCUS ON GENERATING DISCRETE EVENTS FROM CONTINUOUS SIGNALS

What kind of output should Exemplar produce? The previous section has argued that many of the most interesting potential input sources are continuous, and that discrete events are an important output category. Discrete events can be used to trigger transitions in d.tools, which provided the original motivation for this project, as well is an in other rule-based authoring systems. Signal mapping and parameter estimation (extracting not only a discrete category but also continuous parameters from sensor data) are separate problems left for future work.

LEVERAGE DEMONSTRATION TO PARTIALLY SPECIFY COMPUTATION; GIVE THE DESIGNER EXPLICIT CONTROL OF THE REMAINING STEPS

The crucial step in the success of any Programming by Demonstration system is the generalization from a small set of examples to general rules that can be applied to new input (see Section 3.5). When authoring recognizers for sensor data traces generated from human action, one has to contend with the ambiguity inherent in any recognition-based system:

there will be both misses and false positives. Giving the designer an understanding of the performance of the authored interaction and a handle on improving recognition accuracy requires showing a representation of what was learned. Exemplar uses data visualization for this task. Importantly, these visualizations are interactive — they can be manipulated to change parameters of the recognition algorithm.

PROVIDE REAL-TIME VISUAL FEEDBACK OF BOTH HARDWARE EVENTS AND APPLICATION-GENERATED EVENTS

One primary challenge for a designer of sensor-based interactions is trying to make sense of both the data streams from sensors, as well as the interaction events that are generated as a result. To aid this sensemaking task, Exemplar provides real-time visualizations of both incoming sensor data and outgoing event data in a unified graph window. Combining the two types of information in the same display helps designers reason about why a particular action did (or did not) happen.

PROVIDE ACCESS TO HISTORY OF COLLECTED SENSOR DATA

When tuning parameters of recognition algorithms, it is important to determine how those changes affect not only new performances of an action that should be recognized, but also past performances that have served as demonstrations or tests. Exemplar therefore records the entire history incoming sensor data and can visualize how any of these previous actions would have been recognized (or not recognized) given the latest recognition parameters. Reviewing this history can act as a lightweight regression test, to ensure that actions that were correctly recognized in the past are still recognized after a parameter change.

In the next section, we describe how the design principles outlined here are manifest in Exemplar's UI.

4.2.3 DESIGNING WITH EXEMPLAR

Designers begin by connecting sensors to a compatible hardware interface, which in turn is connected to a PC running Exemplar (Figure 4.15). As sensors are connected, their data streams are shown inside Exemplar. The Exemplar UI is organized according to a horizontal data-flow metaphor: hardware sensor data arrives on the left-hand side of the screen, undergoes user-specified transformations in the middle, and arrives on the right-hand side as discrete or continuous events (Figure 4.16). The success of data-flow authoring languages such as Max/MSP attests to the accessibility of this paradigm to non-programmers.

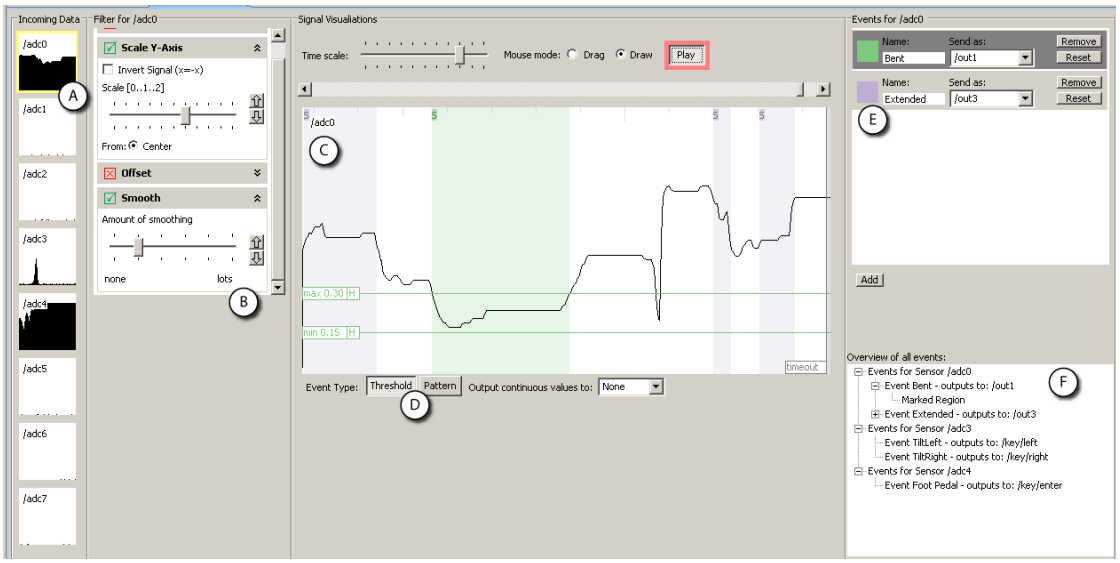

Figure 4.15: The Exemplar authoring environment offers visualization of live sensor data and direct manipulation techniques to interact with that data.

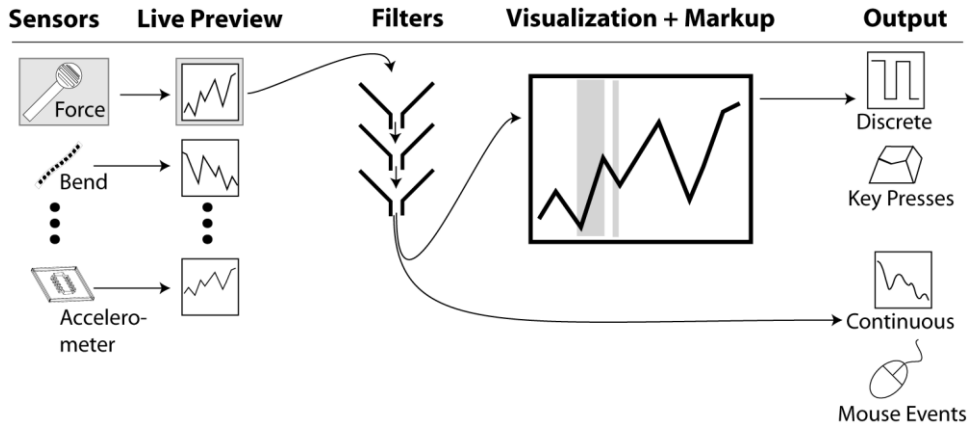

Figure 4.16: Sensor data flows from left to right in the Exemplar UI.

4.2.3.1 Peripheral Awareness

Live data from all connected sensors is shown in a small multiples configuration. Small multiples are side-by-side 'graphical depictions of variable information that share context, but not content' [245]. The small multiples configuration gives a one-glance overview of the current state of all sensors and enables visual comparison (Figure 4.15A). Whenever a signal is 'interesting,' its preview window briefly highlights in red to attract the designer's attention, then fades back to white. In the current implementation, this occurs when the derivative of a sensor signal exceeds a preset value. Together, small multiple visualization and highlighting afford peripheral awareness of sensor data and a visual means of associating sensors with their signals. This tight integration between physical state and software representation encourages discovery and narrows the gulf of evaluation, the difficulty of determining a system's state from its observable output [132]. For example, to find out which output of a multi-axis accelerometer responds to a specific tilt action, a designer can connect all axes, tilt the accelerometer in the desired plane, and look for the highlighted thumbnail to identify the correct input channel. Constant view of all signals is also helpful in identifying defective cables and connections.

4.2.3.2 Drilling Down and Filtering

Designers can bring a sensor's data into focus in the large central canvas by selecting its preview thumbnail (Figure 4.15C). The thumbnails and the central canvas form an overview + detail visualization [227]. Designers can bring multiple sensor data streams into focus at once by control-clicking on thumbnails. Between the thumbnail view and the central canvas, Exemplar interposes a filter stack (Figure 4.15B). Filters transform sensor data interactively: the visualization always reflects the current set of filters and their parameter values. Exemplar maintains an independent filter stack for each input sensor. When multiple filters are active, they are applied in sequence from top to bottom; filters can be reordered. Exemplar's filter stack library comprises four operations for conditioning and mapping:

1. Offset: adds a constant value
2. Y-axis scaling: multiplies the sensor value by a scalar, including signal inversion
3. Smoothing: convolves the signal with one-dimensional Gaussian kernel to suppress high frequency noise
4. Rate of change: takes the first derivative.

These four operations were chosen as the most important for gross signal conditioning and mapping; a later section addresses filter set extensibility.

Interaction with the filtered signal in the central canvas is analogous to a waveform editor of audio recording software. By default, the canvas shows the latest available data streaming in, with the newest value on the right side. Designers can pause this streaming visualization, scroll through the data, and change how many samples are shown per screen. When fully zoomed out, all the data collected since the beginning of the session is shown.

4.2.3.3 Demonstration and Mark-Up

To begin authoring, the designer performs the action she wants the system to recognize. As an example, to create an interface that activates a light upon firm pressure, the designer may connect a force sensitive resistor (FSR) and press on it with varying degrees of force. In Exemplar, she then marks the resulting signal curve with her mouse. The marked region is highlighted graphically and analyzed as a training example. The designer can manipulate this example region by moving it to a different location through mouse dragging, or by resizing the left and right boundaries. Multiple examples can be provided by adding more regions. Examples can be removed by right-clicking on a region.

In addition to post-demonstration markup, Exemplar also supports real-time annotation through a foot switch (chosen because it leaves the hands free for holding sensors). Using the switch, designers can mark regions at the same time they are working with sensors. Pressing the foot switch starts an example region; the region grows while the switch remains pressed, and concludes when the pedal is released. While this technique requires some amount of hand-foot coordination, it enables true real-time demonstration.

4.2.3.4 Recognition and Generalization

Recognition works as follows: interactively, as new data arrives for a given sensor, Exemplar analyzes if the data matches the set of given examples. When the system finds a match with a portion of the input signal, that portion is highlighted in the central canvas in a fainter shade of the color used to draw examples (Figure 4.15C). This region grows for the duration of the match, terminating when the signal diverges from the examples.

Exemplar provides two types of matching calculations — thresholds and patterns — selectable as modes for each event (Figure 4.15D). With thresholding, the minimum and maximum values of the example regions are calculated. The calculation is applied to filtered signals, e.g., it is possible to look for maxima in the smoothed derivative of the input.

Incoming data matches if its filtered value falls in between the extrema. Pattern matching compares incoming data against the entire example sequence and calculates a distance metric (to what extent incoming data resembles the example). Input matches when the distance metric is closer than a user-specified value.

Matching parameters can be graphically adjusted through direct manipulation. For threshold events, min and max values are shown as horizontal lines in the central canvas. These lines can be dragged with the mouse to change the threshold values (see Figure 2G). Parameters can be adjusted interactively: matched regions are automatically recalculated and repainted whenever parameters change. Thus, Exemplar always shows how the signal would have been classified. This affords rapid exploration of how changes affect the overall performance of the matching algorithm.

Sensor noise can lead to undesirable oscillation between matching and non-matching states. Exemplar provides three mechanisms for addressing this problem. First, a smoothing filter can be applied to the signal. Second, the designer can apply hysteresis, or double thresholding. In double thresholding, a boundary is represented by two values which must both be traversed for a state change. Dragging the hysteresis field of a graphical threshold manipulator (indicated by "H" in Figure 4.15G) splits a threshold into two boundary lines. The difference between boundary values is determined by the drag distance. Third, designers can drag a timeout bar from the right edge of the central canvas to indicate the minimum duration for a matching or non-matching state to be stable before an event is fired.

For pattern matching, Exemplar introduces a direct manipulation technique that offers a visual thresholding solution to the problem of parameterizing the matching algorithm (Figure

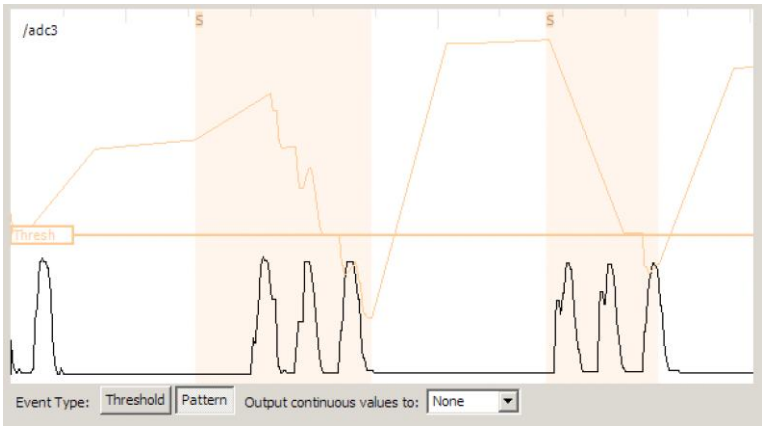

Figure 4.17: Exemplar shows output of the pattern matching algorithm on top of the sensor signal (in orange). When the graph falls below the threshold line, a match event is fired.

4.17). Exemplar overlays a graph plotting distance between the incoming data and the previously given example on the central signal canvas. The lower the distance, the better the match. Designers can then adjust a threshold line indicating the maximum distance for a positive match. When the distance graph falls below the threshold line, an event is fired. With this technique, the designer's authoring experience is consistent whether applying thresholds or pattern matching. In both cases, dragging horizontal threshold bars adjusts the specificity of the matching criteria.

4.2.3.5 Event Output

Exemplar supports the transformation from sensor-centric input into application-centric events. Exemplar generates two kinds of output events: continuous data streams that correspond to filtered input signals; and discrete events that are fired whenever a thresholding or pattern matching region is found. With these events in hand, the designer then needs to author some output, e.g., she needs to specify the application's response to the force sensor push. To integrate Exemplar with other design tools, events and data streams can be converted into operating system input events such as key clicks or mouse movements. Injecting OS events affords rapid control over third party applications (cf. [127]). However, injection is relatively brittle because it does not express association semantics (e.g., that the key 'P' pauses playback in a video application). For tighter integration with application logic, Exemplar can also be linked to d.tools. Exemplar events are then used to trigger transitions in d.tools' interaction models.

4.2.3.6 Many Sensors, Many Events

Exemplar scales to more complex applications by providing mechanisms to author multiple events for a single sensor; to run multiple independent events for different sensors simultaneously; and to author events that combine multiple sensors' data to create a single event.

To the right of the central canvas, Exemplar shows a list of event definitions for the currently active sensor(s) (Figure 4.15E). Designers can add new events and remove unwanted events in this view. Each event is given a unique color. A single event from this list is active for editing at a time, and regions drawn by the designer in the central canvas always apply to that active event.

The authored events for all sensors are always evaluated, and corresponding output is fired, regardless of which sensor is in focus in the central canvas — this allows designers to

author multiple interactions simultaneously. To keep this additional state visible, a tree widget shows authored events for all sensors along with their example regions in the lower right corner of the UI (Figure 4.15F).

Finally, Exemplar enables combining sensor data in Boolean AND fashion (e.g., 'scroll the map only if the accelerometer is tilted to the left and the center button is pressed'). When designers highlight multiple sensor thumbnails, their signals are shown stacked in the central canvas. Examples are now analyzed across all shown sensor signals and events are only generated when all involved sensors match their examples. Boolean OR between events is supported implicitly by creating multiple events. Together, AND/OR combinations enable flexibility in defining events. They reduce, but do not replace the need to author interaction logic separately.

4.2.3.7 Demonstrate-Edit-Review

The demonstrate-edit-review cycle embodied in Exemplar is an application of the design-test-think paradigm for tools introduced in prior work [109,148]. This paradigm suggests that integrating support for evaluation and analysis into a design tool enables designers to gain more insight about their project, faster. Exemplar is the first system to apply design-test-think to the domain of sensor data analysis. More importantly, Exemplar radically shortens the iteration times by an order of magnitude (from hours to minutes) by making demonstration, edit, and review actions the fundamental authoring operations in the user interface.

4.2.4 IMPLEMENTATION & ARCHITECTURE

Exemplar was written using the Java 5.0 SDK as a plug-in for the Eclipse IDE. Integration with Eclipse offers two important benefits: first, the ability to combine Exemplar with the d.tools prototyping tool to add visual authoring of interaction logic; second, extensibility for experts through an API that can be edited using Eclipse's Java tool chain. The graphical interface was implemented with the Eclipse Foundation's SWT toolkit [25].

4.2.4.1 Signal Input, Output, and Display

Consistent with the d.tools architecture, our hardware communicates with Exemplar using OpenSoundControl (OSC) [256]. This enables Exemplar to connect to any sensor hardware that supports OSC. At the present time, three hardware interfaces boards are supported: the d.tools I/O board, and the Wiring [45] and Arduino [185] boards with OSC firmware. OSC

messages are also used to send events to other applications, e.g., d.tools, Max/MSP, or Flash (with the help of a relay program). Translation of Exemplar events into system key presses and mouse movements and clicks is realized through the Java Robots package.

Exemplar visualizes up to eight inputs. This number is not an architectural limit; it was chosen based on availability of analog-to-digital ports on common hardware interfaces. Sensors are sampled at 50 Hz with 10-bit resolution and screen graphics are updated at 15-20 Hz. These sampling and display rates have been sufficient for human motion sensing and interactive operation. However, we note that other forms of input, e.g., microphones, require higher sampling rates (8-40 kHz). Support for such devices is not yet included in the current library.

4.2.4.2 Pattern Recognition

We implemented a Dynamic Time Warping (DTW) algorithm to match demonstrated complex patterns with incoming sensor data. DTW was first used as a spoken word recognition algorithm [218], and has recently been used in HCI for gesture recognition from sensor data [186]. DTW compares two time-series data sets and computes a metric of the distortion distance required to fit one to the other. It is this distance metric that we visualize and threshold against in pattern mode. DTW was chosen because, contrary to many machine learning techniques, only one training example is required. The DTW technique used in this work is sufficiently effective to enable the interaction techniques we have introduced. However, we point out that — like related work utilizing machine learning in UI tools [70,75] — we do not claim optimality of this algorithm in particular.

More broadly, this research — and that of related projects — suggests that significant user experience gains can be realized by integrating machine learning and pattern recognition with direct manipulation. From a developer's perspective, taking steps in this direction may be less daunting than it first appears. For example, Exemplar's DTW technique comprises only a small fraction of code size and development time. We have found that the primary challenge for HCI researchers is the design of appropriate interfaces for working with these techniques, so that users have sufficient control over their behavior without being overwhelmed by a large number of machine-centric 'knobs.'

4.2.4.3 Extensibility

While Exemplar's built-in filters are sufficient for a large number of applications, developers also have the option of writing their own filters, leveraging Eclipse's auto-compilation feature

for real-time integration. Developers derive from an abstract filter base class in their code and override functions for processing data. Users then specify a directory where Exemplar should search for compiled filter class files. Exemplar periodically scans that directory and adds successfully loaded extensions to the filter stack UI panel where they can be activated, deactivated and reordered like built-in filters. This architecture allows engineers on design teams to add to the filter arsenal and for users to download filters off the web. Exemplar's filter architecture was inspired by audio processing architectures such as Steinberg's VST [26], which defines a mechanism how plug-ins receive data from a host, process that stream, and return results. VST has dramatically expanded the utility of audio-editing programs by enabling third parties to extend the library of processing algorithms.

4.2.5 EVALUATION

Our evaluation followed a three-pronged approach. First, we applied the Cognitive Dimensions of Notation framework to Exemplar to evaluate the design tradeoffs of Exemplar as a visual authoring environment. Second, we conducted a first-use study in our lab to determine threshold and utility for novices, as well as to find usability problems. Third, we used Exemplar in public demonstrations and interactive installations to measure real-world performance with a larger group of participants.

4.2.5.1 Cognitive Dimensions Usability Inspection

The Cognitive Dimension of Notation (CDN) framework offers a high-level inspection method to evaluate the usability of information artifacts [89,90]. In CDN, artifacts are analyzed as a combination of a notation they offer and an environment that allows certain manipulations of the notation. CDN is particularly suitable for analysis of visual programming languages. We conducted a CDN evaluation of Exemplar following Blackwell and Green's Cognitive Dimensions Questionnaire [47] to allow the reader to revisit Exemplar according to categories independently identified as relevant, and to facilitate comparison with future research systems. This analysis begins with an estimate of how time is spent within the authoring environment, and then proceeds to evaluate the software against the framework's cognitive dimensions.

TIME SPENT

Exemplar's main notation is a visual representation of sensor data with user-generated mark-up. Lab use of Exemplar led us estimate that time is spend as follows:

30% Searching for information within the notation
(browsing the signal, visually analyzing the signal)

10% Translating amounts of information into the system (demonstration)

20% Adding bits of information to an existing description
(creating and editing mark up, filters)

10% Reorganizing and restructuring descriptions
(changing analysis types, redefining events)

30% Playing around with new ideas in notation without being sure what will result
(exploration)

This overview highlights the importance of search, and the function of Exemplar as an exploratory tool.

DIMENSIONS OF THE MAIN NOTATION

We present a discussion of the most relevant CDN dimensions here.

VISIBILITY AND JUXTAPOSABILITY (ABILITY TO VIEW COMPONENTS EASILY):

All current sensor inputs are always visible simultaneously as thumbnail views, enabling visual comparison of input data. Viewing multiple signals in close-up is also possible; however, since such a view is exclusively associated with 'AND' events combining the shown signals, it is not possible to view independent events at the same time.

VISCOSITY (EASE OR DIFFICULTY OF EDITING PREVIOUS WORK):

Event definitions and filter settings in Exemplar are straightforward to edit through direct manipulation. The hardest change to make is improving the pattern recognition if it does not work as expected. Thresholding matching error only allows users to adjust a post-match metric as the internals (the 'how' of the algorithm) are hidden.

DIFFUSENESS (SUCCINCTNESS OF LANGUAGE):

Exemplar's notation is brief, in that users only highlight relevant parts of a signal and define a small number of filter parameters through graphical interaction. The length of event descriptions is dependent on the Boolean complexity of the event expressed (how many ORs/ANDs of signal operations there are).

HARD MENTAL OPERATIONS:

Most mental effort is required to keep track of events that are defined and active, but not visible in the central canvas. To mitigate against this problem we introduced the overview list of all defined interactions (Figure 4.15F) which minimizes cost to switch between event

views. One important design goal was to make results of operations visible immediately in Exemplar.

ERROR-PRONENESS (SYNTAX PROVOKES SLIPS):

One slip occurred repeatedly in our use of Exemplar: resizing example regions by dragging their boundaries. This was problematic because no visual feedback was given on what the valid screen area was to initiate resizing. Lack of feedback resulted in duplicate regions being drawn, with an accompanying undesired recalculation of thresholds or patterns. Improved mouse manipulators on regions can alleviate this problem.

CLOSENESS OF MAPPING:

The sensor signals are the primitives users are operating on. This direct presentation of the signal facilitates consistency between the user's mental model and the system's internal representation.

ROLE-EXPRESSIVENESS (PURPOSE OF A COMPONENT IS READILY INFERRED):

Problems with role-expressiveness often arise when compatibility with legacy systems is required. Since Exemplar was designed from scratch for the express purpose of viewing, manipulating and marking up signals, this is not a problem. While the result of applying operations is always visible, the implementation "meaning" of filters and pattern recognition is hidden.

SECONDARY NOTATIONS:

Currently, Exemplar permits users to label events, but not filter settings or regions of the signal. If deemed important, this is an area for future work.

PROGRESSIVE EVALUATION:

Real-time visual feedback enables evaluation of the state of an interaction design at any point. Furthermore, Exemplar sessions can be saved and retrieved through serialization to disk.

In summary, Exemplar performs well with respect to visibility, closeness of mapping, and progressive evaluation. Many of the identified challenges stem from the difficulties of displaying multiple sensor visualizations simultaneously. These can be addressed through interface improvements — they are not inherent to the approach.

4.2.5.2 First-Use Study

We conducted a controlled study of Exemplar in our laboratory to assess the ease of use and felicity of our tool for design prototyping. The study group comprised twelve participants. Ten were graduate students or alumni of our university; two were undergraduates. While all participants had some prior HCI design experience, they came from a variety of educational backgrounds: four from Computer Science/HCI, four from other Engineering fields, two from Education, and two from the Humanities. Participants' ages ranged from 19 to 31; five were male, seven female. Two female participants served as pilot testers. Eight participants had had some prior exposure to sensor programming, but none reported to be experts (Figure 4.19).

STUDY PROTOCOL

Participants were seated at a dual-screen workstation with a d.tools hardware interface. The Exemplar software was shown on one screen, a help document on sensors was shown on the other. Participants were asked to author interactions that controlled graphics on a large projection display (Figure 4.18). We chose this large display to encourage participants to think beyond the desk(top) in their designs. We chose graphical instead of physical output since our study focused on authoring responses to sensor input only, not on actuation.

Individual study sessions lasted two hours. Sessions started with a demonstration of Exemplar. We also introduced the set of available sensors, which comprised buttons,

Figure 4.18: Exemplar study setup: participants were seated at a dual monitor workstation in front of a large wall display.

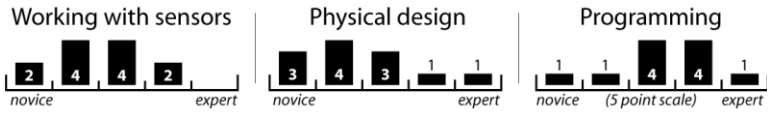

Figure 4.19: Self-reported prior experience of Exemplar study participants.

switches, capacitive touch sensors, light sensors, infrared distance rangers, resistive position sensors, force sensitive resistors (FSRs), load cells, bend sensors, 2D joysticks and 3D accelerometers. Participants were given three design tasks. For each task, we provided a mapping of triggers available in Exemplar to output behaviors in the instructions (e.g., sending an event called "hello" activated the display of the hello graphic in the first task).

The first task was a simple "Hello World" application. Subjects were asked to display a hello graphic (by issuing the "hello" event) when a FSR was pressed (through thresholding) while independently showing a world graphic when a second FSR was pressed three times in a row (through pattern recognition).

The second task required participants to augment a provided bicycle helmet with automatic blinkers such that tilting the helmet left or right causes the associated blinkers to signal. This task was inspired by Selker et al.'s Smart Helmet [225]. While blinking output was simulated on a "mirror" on the projection display, participants had to attach sensors to the real helmet.

Our last task was an open-ended design exercise to author new motion-based controls for at least one of two computer games. The first game was version of Lunar Lander in which the player has to keep a spaceship aloft, collect points and safely land using three discrete events to fire thrusters (up, left, and right). The second game was a shooting game with continuous x/y control used to aim and a discrete trigger to shoot moving targets.

STUDY RESULTS

In our post-test survey, participants ranked Exemplar highly for decreasing the time required to build prototypes compared to their prior practice (mean=4.8, median=5 on a 5-point Likert scale, σ=0.42); for facilitating rapid modification (mean=4.7, median=5, σ=0.48); for enabling them to experiment more (mean=4.7, median=5, σ=0.48); and for helping them understand user experience (mean=4.3, median=4; σ=0.48). Responses were less conclusive on how use of Exemplar would affect the number of prototypes built, and whether it helped focus or distracted from design details ($\sigma > 1.0$ in each case). Detailed results are shown in Figure 4.20.

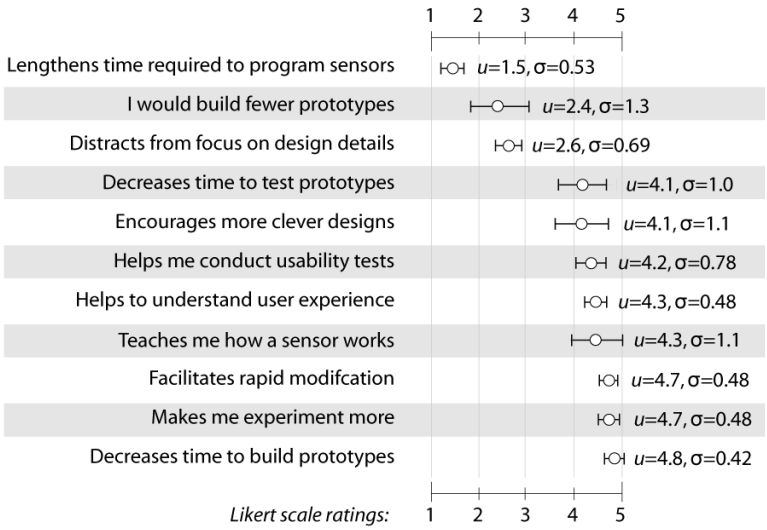

Figure 4.20: Exemplar post-experiment questionnaire results. Error bars indicate ½ standard deviation in each direction.

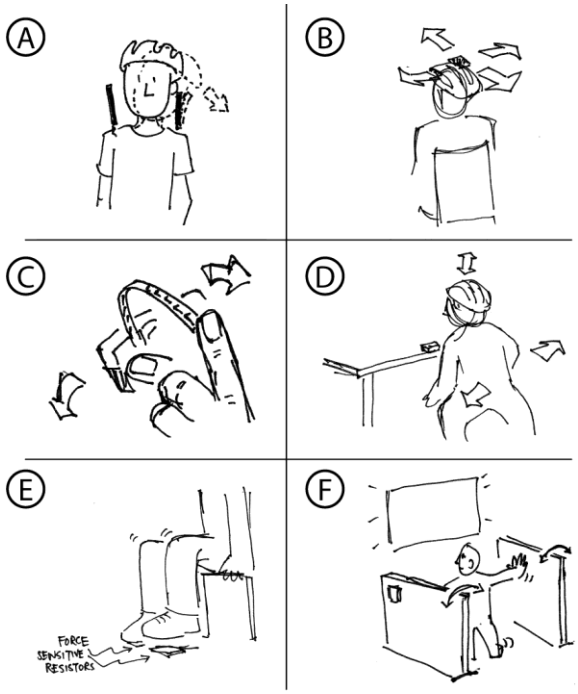

Figure 4.21: Interaction designs from the Exemplar user study. **A**: turning on blinkers by detecting head tilt with bend sensors; **B**: accelerometer used as continuous 2D head mouse; **C**: aiming and shooting with accelerometer and bend sensor; **D**: navigation through full body movement; **E**: bi-pedal navigation through force sensitive resistors; **F**: navigation by hitting the walls of a booth.

SUCCESSES

All participants successfully completed the two first two tasks and built at least one game controller. The game controller designs spanned a wide range of solutions (Figure 4.21). Once familiar with the basic authoring techniques, many participants spent the majority of their time sketching and brainstorming design solutions, and testing and refining their design. This rapid iteration cycle allowed participants to try out up to four different control schemes for a game (Figure 4.22). We see this as a success of enabling epistemic activity: participants spent their time design thinking rather than implementation tinkering.

Exemplar was used for exploration: given an unfamiliar sensor, participants were able to figure out how to employ it for their purposes. For example, real-time feedback enabled participants to find out which axes of a multi-axis accelerometer were pertinent for their design. Participants also tried multiple sensors for a given interaction idea to explore the fit between design intent and available technologies.

Interestingly, performance of beginners and experts under Exemplar was comparable in terms of task completion time and breadth of ideation. Two possible explanations for this situation are that either Exemplar was successful in lowering the threshold to entry for the types of scenarios tested; or that it encumbered experts from expressing their knowledge. The

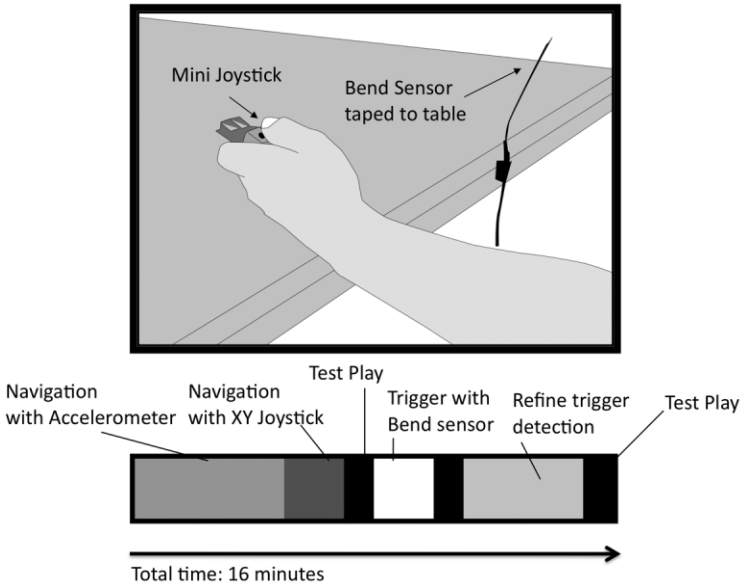

Figure 4.22: Example of one study participant's exploration: the participant created two different navigation schemes and two iterations on a trigger control; he tested his design on a target game three times within 16 minutes.

absence of complaints by experts in the post-test surveys provides some support for the first hypothesis.

SHORTCOMINGS DISCOVERED

Participants identified two key areas for improvement. One recurring theme in our feedback was the need for visualization of Exemplar's hidden state. At the time of the study, participants could only see events authored for the sensor in focus. While other events were still active, there was no comprehensive way of listing them. Also, highlighted regions corresponding to training examples were hard to retrieve after more data was collected, as the regions were pushed farther into the history of the signal. To address these difficulties, Exemplar now displays a full list of active events, along with the corresponding example regions. Selecting those regions jumps to the time of their definition in the central canvas.

Expert users expressed a need for finer control over hysteresis parameters for thresholding and a visualization of time and value units on the axes of the signal display. In response to these requests, we added direct manipulation of hysteresis and timeout parameters to threshold events.

The importance of displaying quantitative data in addition to visualization to aid the designer's mental model of events deserves further study. Participants also requested ways to provide negative examples, techniques for displaying multiple large sensor visualizations simultaneously, and finer control over the timing for pattern matching (both in terms of latency and duration).

4.2.5.3 Using Exemplar to Create Game Controllers

To gain real-world experience with a larger number of users we exhibited Exemplar at the 2007 San Mateo Maker Faire, and created a motion-controlled game for the Interactivity exhibit at the 2007 CHI conference (Figure 4.23).

The Maker Faire is a large annual gathering of amateurs interested in electronics, crafts and do-it-yourself projects. We exhibited Exemplar under the theme "Build your own game controller." We supplied the set of sensors and games used in the Exemplar lab study, as well as a collection of household items such as garden gloves, staplers, and frying pans to attach sensors to. Interested visitors were invited to come up with their own game control scheme and implement it in Exemplar with the help of one of the researchers. Several hundred visitors took part over the course of two days. Preparing for this installation sensitized us to the limitations of the Java Robot event injection technique to control closed-source 3rd party

applications: because generated keyboard and mouse events cannot be targeted to a specific application, it is easy for novices to inadvertently direct keyboard and mouse input back into Exemplar itself, which is certainly not intended. A workable but expensive solution is to use two computers: one to run Exemplar, and another to run the game. The game computer then also requires a helper application that receives socket messages from Exemplar and translates them into system keyboard and mouse events.

For the CHI conference exhibition, we used Exemplar as the back end for a wireless gaming system [261]. The game, based on Zhang's Control Freaks concept [260], featured a portable, wireless 3D accelerometer mounted to a clamp (disguised as a plush cartoon character) that could be attached to clothing or other objects to turn those objects into game controllers (Figure 4.23, right). For example, people could attach the clamp to their shoes to detect running and jumping, or to a chair to detect swiveling the chair left and right. Using Exemplar for this installation sensitized us to the limits of pattern recognition for fast-paced game play — pattern recognition incurs a compulsory latency cost a pattern can only be detected *after it has happened*. Thresholds can detect the onset of an action but may require additional application logic to suppress spurious matches beyond timeouts and hysteresis.

4.2.6 LIMITATIONS & EXTENSIONS

Exemplar currently focuses on recognizing discrete actions in low-frequency continuous sensor signals. This assumption limits the applicability of Exemplar in the following ways.

4.2.6.1 Lack of Support for Other Time Series Data

Much human motion can be adequately sampled at 50-100Hz (Winter for example reports

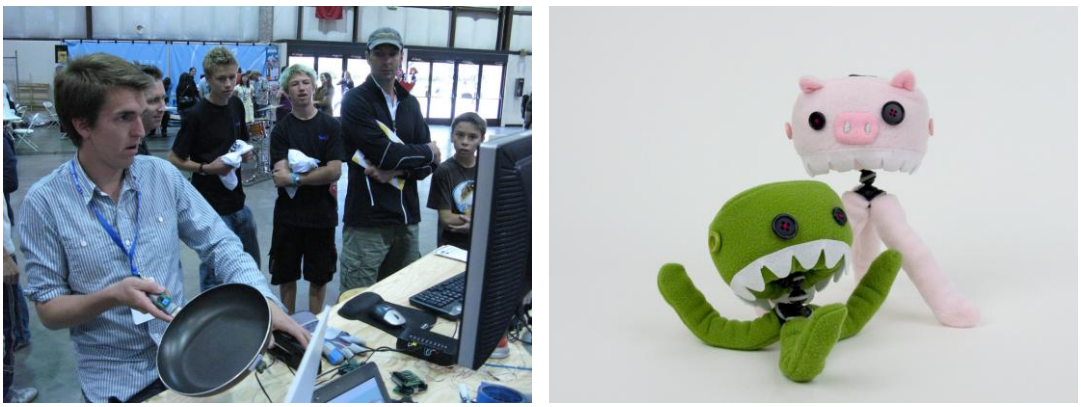

Figure 4.23: Exemplar was used for public gaming installations at the San Mateo Maker Faire and at CHI 2007. For the CHI installation, wireless accelerometers were disguised as plush characters; the characters could be attached to clothing or objects in the environment. Characters and game concept were developed by Haiyan Zhang.

that many gait analyses can be performed at 24Hz [252:Ch. 2]). However, there are applications and types of sensors for which this rate is insufficient. Audio input, for example for recognizing the scratching of fingernails on a surface [106], is commonly sampled at rates of 10-96kHz. Such higher frequency signals need different real-time visualization algorithms (which we could borrow from audio editing). We have not yet investigated to what extent dynamic time warping can be run in realtime on many parallel audio signals, or if it would offer comparative recognition performance.

4.2.6.2 Matching Performance Degrades for Multi-Dimensional Data

The employed dynamic time warping algorithm was created to compare one-dimensional time series data. The sequence alignment algorithm does not extend in a straightforward manner to matching in multiple dimensions. Exemplar uses the following generalization to make matching in multiple dimensions possible: An example for an event spanning multiple input dimensions for a given time interval is defined as individual examples $ex_1, ex_2, ..., ex_n$ in each input dimension. For new input data $in_1, in_2, ..., in_n$, a set of n DTW algorithms is then run in parallel, one for each dimension. Each outputs a binary match/no-match decision, based on individual thresholds on matching distance. Only if all dimensions independently report a match is the multi-dimensional event fired:

$$dtw\_match\big((ex_1, \dots, ex_n), (in_1, \dots, in_n)\big) = \bigcap_{k=1}^{n} dtw\_match(ex_k, in_k)$$

This approach ignores the fact that the data dimensions are interdependent and may distort different dimensions differently.

4.2.6.3 Lack of Visualization Support for Multi-Dimensional Data

Exemplar relies on the designer to make decisions about recognition algorithms and parameters based on a visualization of live sensor data. It is therefore important that the designer can interpret the visualization and make sense of it. While we found straightforward timeline visualization to be sufficient for one-dimensional sensors, this is not true for more complex sensors that return multi-dimensional data. For example, a resistive touch screen will return an *(x,y)* position; a three-dimensional accelerometer will return *(x,y,z)* acceleration data. The inherent structure of such signal spaces cannot currently be shown in Exemplar. Future work should investigate to what extent different visualizations can be used to give a

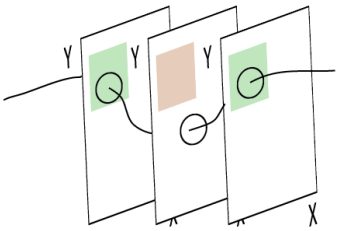

Figure 4.24: A possible visualization for 2D thresholding in Exemplar.

designer greater leverage. One challenge will be how to visualize time in higher-dimensional time-series data.

As an example, take the (x,y) position task: instead of two independent timelines, it may be advantageous to enable the designer to see a 2D space and to define thresholds as regions within that space (Figure 4.24). The 2D space could be shown as stacked, rotated slices through which the signal then describes a 3D trajectory. Events would be fired whenever the signal moves within the threshold region. A recent survey of possible visualization techniques that can inform future development can be found in [33].

4.2.6.4 Lack of Support for Parameter Estimation

Exemplar's recognizers only make binary decisions, e.g., they recognize that a tennis swing has occurred from accelerometer data. They do not yet offer parameter estimation, e.g., detecting how fast the racket was swung. A new demonstration technique would be needed that elicits examples for different parameter values from a designer. In addition, new algorithms would be needed that, given multiple examples with parameter values and new input data, can output parameter estimates. Note that it is already possible to author categorical recognizers by defining multiple events on a given signal dimension — the recognizers are then run in parallel and the single best match wins. But generalizing to the continuous case is not possible.

4.2.6.5 Difficult to Interpret Sensor Data History

Whenever the parameters of an event recognizer are changed by the designer, e.g., by moving a threshold line in the user interface, Exemplar recomputes how past data would have been classified given the new definition and updates its event visualization accordingly. However, it is hard to match the highlighted sensor signal traces back to the specific actions that produced these traces. A promising way to give the designer a better handle on understanding

how actions affect past demonstrations would be to also record live video of the demonstration and replay it inside the authoring environment as the designer reviews the history of collected data. The technique resembles interaction with the d.tools video editor (see Section 6.1), but with a much more focused role: instead of reviewing the usability of an entire prototype, the video is used to review examples used to define interaction events that are used within that prototype.

CHAPTER 5 CREATING ALTERNATIVE DESIGN SOLUTIONS

5.1 ALTERNATIVES IN JUXTAPOSE

Design frequently alternates between divergent stages, where multiple different options are explored, and convergent stages, where ideas are selected and refined [55,66,135] (Figure 5.1). When designers create multiple distinct prototypes prior to committing to a final direction, several important benefits arise. First, alternatives provide designers with a more complete understanding of a design space [83]. Second, developing different "what if" scenarios enables more effective, efficient decision making within organizations [222]. Third, discussing multiple prototypes helps project stakeholders better communicate their requirements [157]. Finally, presenting multiple alternatives in user studies facilitates participants' ability to understand design tradeoffs and offer critical feedback [243].

Placing "enlightened trial and error" at the core of design raises the research question, *how might authoring environments support designers in creating and managing design options?* Traditionally, design tools have focused on creating single artifacts [240]. Research in subjunctive interfaces [177] pioneered techniques for parallel exploration of multiple scenarios during information exploration. Set-based interaction techniques have also been introduced for graphic design [241,242] and 3D rendering [181]. Providing alternative-aware tools for interaction design adds the challenge of working with two distinct representations:

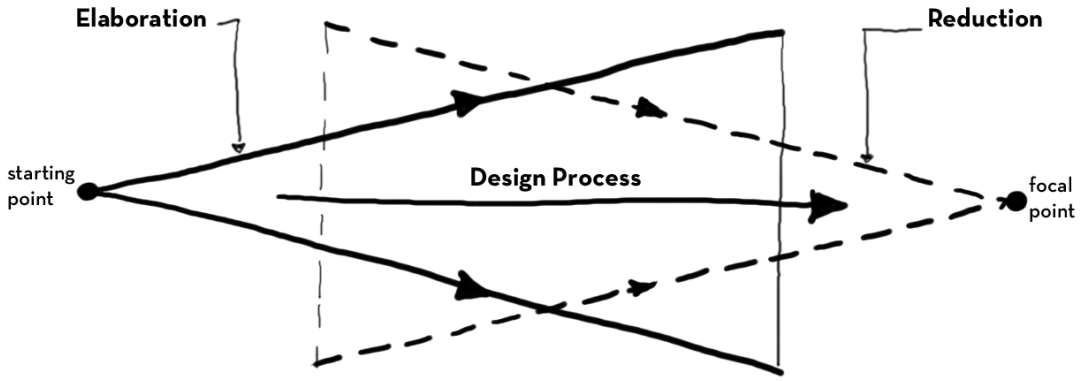

Figure 5.1: Design alternates between divergent and convergent stages. Diagram due to Buxton [55], redrawn by the author.

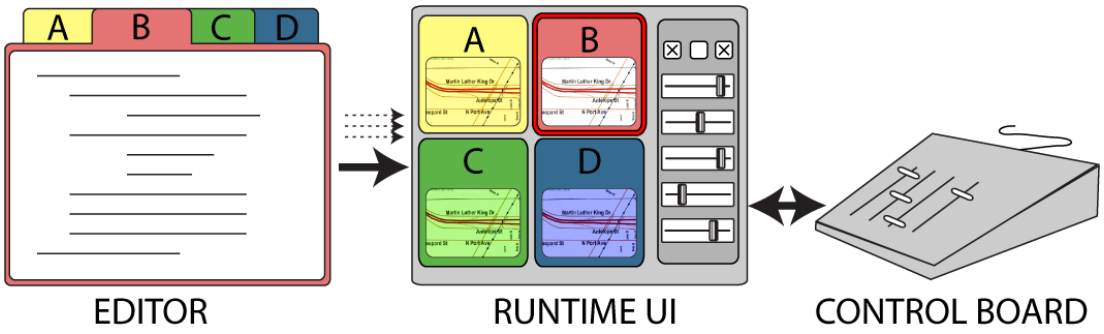

Figure 5.2: Interaction designers explore options in Juxtapose through a source code editor that supports alternative code documents (left), a runtime interface that offers parallel execution and tuning of application parameters (center), and an external controller for spatially multiplexed input (right).

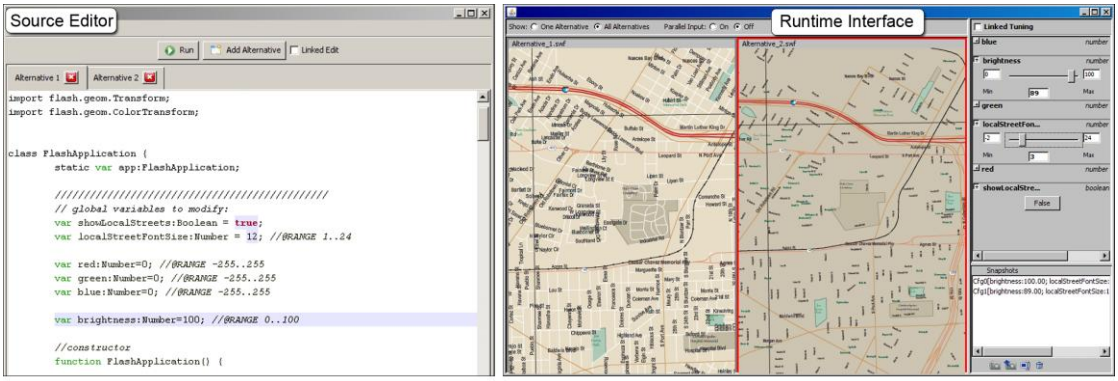

Figure 5.3: In the Juxtapose source editor (left), users work with code alternatives in tabs. Users control whether modifications affect all alternatives or just the presently active alternative through linked editing. In the runtime interface (right), alternatives are executed in parallel. Designers tune application parameters with automatically generated control widgets.

source code, where changes are authored; and the running program, where changes are observed.

This chapter suggests that interaction design tools can successfully scaffold exploration by managing alternatives across source and execution environments, and introduces Juxtapose, an authoring tool manifesting this idea (Figure 5.2). Juxtapose makes two fundamental contributions to design tool research.

First, it introduces a programming environment in which interaction designers create and run multiple program alternatives in parallel (Figure 5.3 left). Juxtapose extends linked editing [244], a technique to selectively modify source duplicates simultaneously, by turning source alternatives into a set of programs that are executed in parallel. The Juxtapose runtime environment enables interacting with these parallel alternatives.

Second, Juxtapose introduces "tuning" of interface parameters at runtime by automatically generating a control interface for application parameters through source code analysis and language reflection (Figure 5.3 right). We hypothesize that runtime controls encourage real-time improvisation and exploration of the application's parameter space. Designers can save parameter settings in presets that Juxtapose maintains across alternatives and executions. To facilitate simultaneous control over multiple tuning parameters, a physical, spatially-multiplexed control surface is supported.

This chapter first introduces findings from formative interviews that motivate our work. We then describe the key interaction techniques for creating, executing, and modifying alternatives with Juxtapose. We describe implementations for desktop, mobile, and tangible applications. Next, we present evaluation results and conclude by discussing tradeoffs and limitations of our approach.

5.2 FORMATIVE INTERVIEWS

To augment our intuitions from our own teaching and practice, we conducted three interviews with interaction designers. Here, we briefly summarize the insights gained.

First, arriving at a satisfying user experience requires *simultaneous adjustment of multiple interrelated parameters*. For example, a museum installation developer shared that getting an interactive simulation to "feel right" required time-intensive experimentation with parameter settings. Similarly, an instructor for a course on computer-vision input in HCI reported that students found adjusting recognition algorithm parameters to be a lengthy trial-and-error process.

Second, *creating alternatives of program logic* is a complementary practice to parameter tuning. In one participant's code, we saw multiple alternative code strategies living side-by-side inside a single function (Figure 5.4). To try out these different approaches in succession, this interviewee would change which alternative was uncommented (i.e., active), recompile, and execute.

Lastly, all interviewees reported writing custom control interfaces for internal program variables when they were unsure how to find good values. These tuning interfaces are not actually part of the functionality of the application — they function exclusively as exploratory development tools.

Across the three concerns, interviewees resorted to ad-hoc practices that allowed for some degree of exploration despite a lack of tool support. The following scenario illustrates

```
int calculateNextSize(int [][]currentSizes, int i, int j) {
  float denominator = 0;
  int sumOfNeighbors = 0;
  int maxOfNeighbors = 0;
  if(i != 0) {
    sumOfNeighbors += currentSizes[i - 1][j]; denominator += 1;
    maxOfNeighbors = currentSizes[i - 1][j];
//    if(j != 0) sumOfNeighbors += currentSizes[i - 1][j - 1]/2; denominator += .5;
//    if(j != currentSizes[0].length - 1) sumOfNeighbors += currentSizes[i - 1][j + 1]/2; denominator += .5;
  }
  if(i != currentSizes.length - 1) {
    sumOfNeighbors += currentSizes[i + 1][j]; denominator += 1;
    if(currentSizes[i + 1][j] > maxOfNeighbors) maxOfNeighbors = currentSizes[i + 1][j];
//    if(j != 0) sumOfNeighbors += currentSizes[i + 1][j - 1]/2; denominator += .5;
//    if(j != currentSizes[0].length - 1) sumOfNeighbors += currentSizes[i + 1][j + 1]/2; denominator += .5;
  }
```

Figure 5.4: Example code from our inquiry: two behaviors co-exist in the same function body. The participant would switch between alternatives by changing which lines were commented.

how Juxtapose can improve such exploration by explicitly addressing parameter variation, alternative creation and control interface generation.

5.3 EXPLORING OPTIONS WITH JUXTAPOSE

Tina is designing the graphical interface for a new handheld GPS device that both pedestrians and bicyclists will use. She imagines pedestrians will pan the map by tilting the device, and use buttons for zooming. Bicyclists mount the device in a fixed position on their handlebars, so they will need buttons to pan and zoom.

To try out navigation options, Tina loads her existing map prototype and clicks the Add Alternative button (Figure 5.5A); this duplicates her code in a new tab. With the Linked Edit box checked, she adds a function to respond to button input. This code change propagates to both alternatives. She clears the Linked Edit checkbox so that she can write distinct input handlers in the function body of each alternative (Figure 5.5B). In unlinked mode, edits only apply to the active tab. A colored background highlights code that differs between alternatives (Figure 5.5C).

Tina executes her designs. Juxtapose's runtime interface shows the application output of each code alternative side-by-side (Figure 5.5D). One alternative is active, indicated by a red outline. Global Number and Boolean-typed variables of this alternative are displayed in a variable panel to the right of the running applications. Tina expands the entries for layer visibility, panning speed and zoom step size to reveal tuning widgets that allow her to change values of each variable interactively (Figure 5.5E). Tina uses the tuning widgets to arrive at fluid pan and zoom animations.

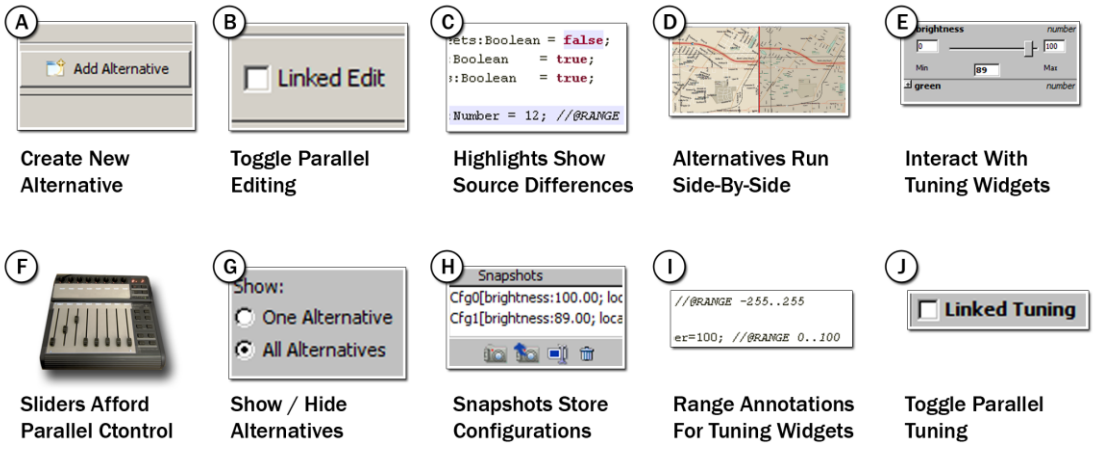

Figure 5.5: UI vignettes for the Juxtapose Scenario.

Tina also hypothesizes that bicyclists will value velocity-contingent visual and typographic levels of detail. To adjust the text sizes of multiple road types simultaneously, she moves her non-dominant hand to an external physical control board (Figure 5.5F). She places one finger on each slider, and quickly moves multiple sliders simultaneously to visually understand the gestalt design tradeoffs, such as legibility and clutter. To focus in on the details of one alternative, she toggles between viewing alternatives side-by-side, and viewing just one alternative (Figure 5.5G).

Tina finds several promising parameter combinations for showing levels of detail and uses the snapshot panel to save them (Figure 5.5H). Back in the code editor, she introduces a speed variable to simulate sensed traveling velocity, and adds code to load different snapshots from the Juxtapose environment when the speed variable changes. To constrain tuning to useful values, she adds range annotation comments, e.g., indicating that speed should vary between 1 and 30 mph (Figure 5.5I). She runs her design again and selects speed for tuning. Moving the associated slider now switches between the snapshot values she previously saved. She checks the Linked Tuning box to propagate changes in simulated speed to all alternatives in parallel (Figure 5.5J).

5.4 ARCHITECTURE FOR ALTERNATIVE DESIGN

This section outlines fundamental requirements for parallel editing, execution, and tuning, and describes how the Juxtapose implementation supports these techniques.

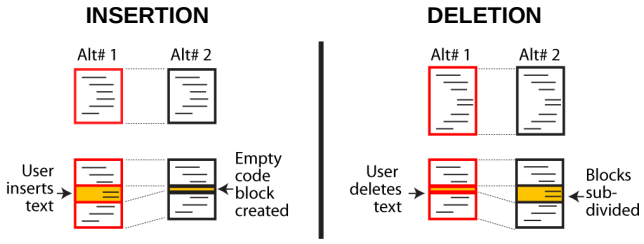

Figure 5.6: Juxtapose's implementation of linked editing is based on maintaining block correspondences between alternatives across document modifications.

5.4.1 PARALLEL EDITING

To make working with multiple code alternatives feasible, an authoring environment must keep track of code differences across alternatives, make this structure visually apparent to the user, and offer efficient interaction techniques for manipulating content across alternatives. To support these three requirements, Juxtapose extends Toomim et al.'s linked editing technique [244]: alternatives are accessible through document tabs; source differences between tabs are highlighted with a shaded background; and edits can be either local to one alternative or global to all alternatives. Toomim's work focused on sharing code snippets across different locations within a project. Juxtapose instead targets creation of sets of applications based on a core of shared code. To enable interactive editing across multiple documents, Juxtapose replaces Toomim's algorithm with incremental correspondence tracking during editing and slower content differencing during compilation. The efficiency gains thus realized enable Juxtapose to run comparisons after each key press. Average times for single character replacement operations were under 1 ms with up to 5 alternatives on a 2 GHz PC running Windows Vista.

Juxtapose tracks correspondences between alternatives by partitioning all source alternatives into corresponding blocks. In linked editing, the block structure stays fixed and block content is modified in all alternatives. In unlinked editing, code blocks are subdivided and alternatives store different content in their sub-blocks (Figure 5.6). When inserting text while unlinked, Juxtapose's data structure splits the code into pre- and post-insertion blocks and creates a new code block for the inserted text. Juxtapose splits all alternatives, inserting an empty element into the unmodified alternatives. Deletions also split code blocks. Here, the active document represents the deletion with an empty element; the corresponding elements in the other alternatives contain the deleted text. Code modifications are expressed as deletions followed by insertions. Blocks are never merged during editing.

Incremental structure tracking performs differently than content-based matching if a user types identical code into corresponding locations in two distinct documents: content-based approaches will mark this as a match; structure-based approaches will not. To obtain both interactive performance and content matching, Juxtapose optimizes global block structure with a slower longest common subsequence algorithm at convenient times (i.e., when compilation is started).

5.4.2 PARALLEL EXECUTION AND TUNING

Executing a set of related interaction designs raises two principal questions: Should alternatives be presented in series or in parallel? And should users interact with these alternatives one-at-a-time or simultaneously? To investigate how different target devices offer unique opportunities for parallel input and output, we implemented versions of the Juxtapose environment for three domains: desktop interactions written in ActionScript for Adobe Flash; mobile phone interactions for Flash Lite; and physical interactions based on the Arduino microcontroller platform. The three implementations share a common editor but differ in their runtime environment. We discuss each in turn.

DESKTOP

Desktop PCs offer sufficient screen resolution to run alternative interactions side-by-side, analogous to application windows. In our implementation, alternatives are authored in ActionScript 2, from which Juxtapose generates a set of Flash movie files using the MTASC compiler [27]. The generated files are then embedded into the Juxtapose Java runtime interface using a Windows-native wrapper library [28]. For consistency with the temporally multiplexed input of windowed operating systems, only one active alternative receives keyboard and mouse input events by default. However, Juxtapose offers the option to replicate user input across alternatives through event echoing [176]. By using a provided custom mouse class, mouse events can be intercepted in the active alternative and injected into all other alternatives, which then show a ghost cursor. This parallelism only operates at the low level of mouse move and click events, which is useful when both application logic and visual layout are similar across alternatives. However, in the absence of a model that translates abstract events in one application into equivalent events in another, users cannot usefully interact with different application logic simultaneously. While development of an abstract input model that provides such a mapping is certainly possible, it is unlikely to occur during prototyping, when the application specification is still largely in flux.

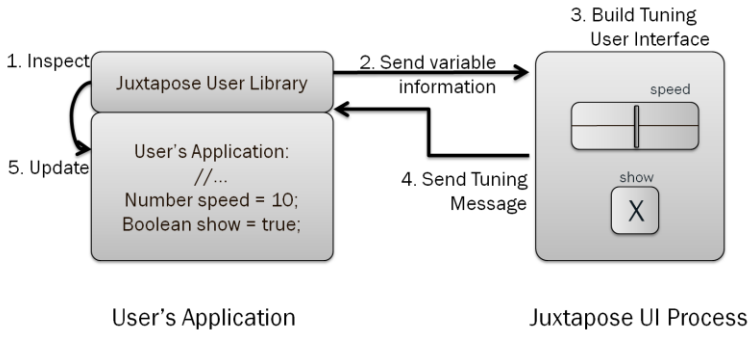

Figure 5.7: Runtime tuning is achieved through bi-directional communication between a library added to the user's application and the Juxtapose runtime user interface.

To accomplish runtime variable tuning, bi-directional data exchange between the user's application and the tuning interface is required. On startup, the application transmits variable names, types, and values to Juxtapose (Figure 5.7). The tuning interface in turn sends value updates for variables to the application whenever its widgets are used. Loading snapshots defined in the tuning interface from code is initiated by a request from the user application, followed by a response from Juxtapose. To accomplish this communication, the user adds a Juxtapose library module to their code. In our implementation, communication between the Flash application and the hosting Java environment takes place through a message-passing protocol and synchronous remote procedure call interface built on top of the Flash Player API.

MOBILE PHONE

For smart phones, the most useful unit of abstraction for parallel execution might not be an application window on a handset, but rather the entire handset itself. The small form factor and comparatively lower cost make it attractive to leverage multiple physical devices in parallel (Figure 5.8). In Juxtapose mobile, developers still compose and compile applications on a PC. At runtime, the tuning interface resides on the PC, and the alternatives run on different handsets. A designer can rapidly switch between alternatives by putting one phone down and picking another one up. To target tuning events to an application running on a particular phone, Juxtapose offers alternative selection buttons in the runtime interface.

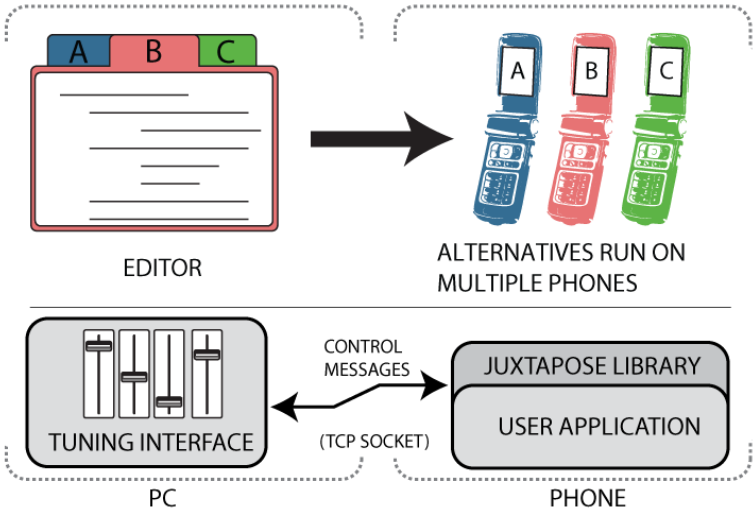

Figure 5.8: When using Juxtapose mobile, code alternatives are executed on different phones in parallel. Variable tuning is accomplished through wireless communication.

Our Juxtapose mobile prototype generates binaries which run on the Flash Lite 2.0 player on Nokia N93 smart phones. The desktop tuning interface and the smart phone communicate through network sockets. When designers run an application on the mobile phone, it opens a persistent TCP socket connection to the Juxtapose runtime interface on the PC. Our prototype uses Wi-Fi for simplicity. Informally, we found that the phone receives variable updates at approximately 5 Hz, much slower than on the PC, but still sufficient for interactive tuning. Response rates are slower because mobile devices trade off increased battery life for slower network throughput and increased latency. A limitation of the current

Figure 5.9: Two prototypes built with Juxtapose mobile. Left: A map navigation application explored use of variable tuning. Right: Two alternatives of a fisheye menu navigation technique running on two separate phones.

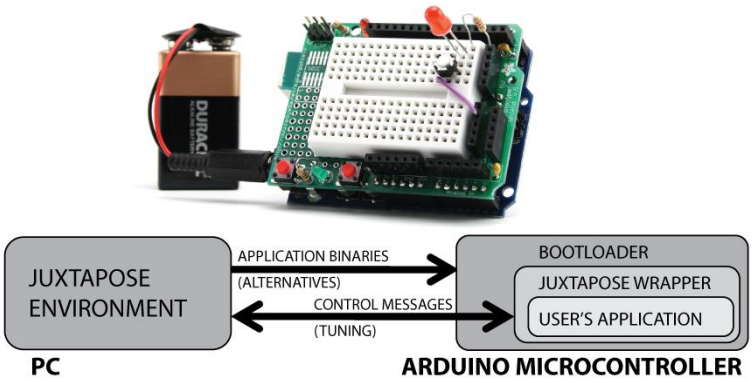

Figure 5.10: For microcontroller applications, Juxtapose transparently swaps out binary alternatives using a bootloader. Tuning is accomplished through code wrapping.

Juxtapose mobile implementation is that users must manually upload compiled files to the phones and launch them within the Flash Lite player. This is due to restrictions of the phone's security architecture. We have explored the utility of Juxtapose mobile with several UI prototypes, including map navigation and fisheye menus (Figure 5.9). While the latency of tuning messages made the external MIDI controller less useful in our tests (it generates too many events which queue up over time), the ability to modify the application running on the phone while another user is interacting with that phone appeared to be especially useful.

PHYSICAL INTERACTIONS

Many interaction designers work with microcontrollers when developing new physical interfaces because they offer access to sensors and actuators. The primary difference to both desktop and mobile development is that novel physical interaction design involves building custom hardware, which is resource intensive. Consequently, designers are likely to embed multiple different opportunities for interaction into the same physical prototype.

Juxtapose supports developing for the Arduino [185] platform and language, a combination popular with interaction designers and artists. Code for all alternatives is cross-compiled with the AVR-GCC compiler suite. Juxtapose for Arduino uploads and runs only one code alternative on one attached Arduino board at a time. When the designer switches between alternatives, Juxtapose transparently replaces the binary running on the microcontroller through a bootloader (Figure 5.10).

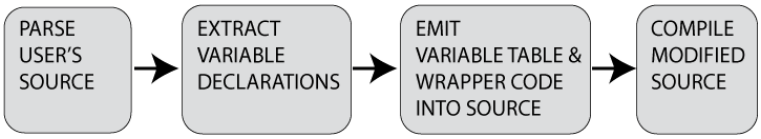

Figure 5.11: The pre-compilation processing step extracts variable declarations and emits them back into source code as a symbol table.

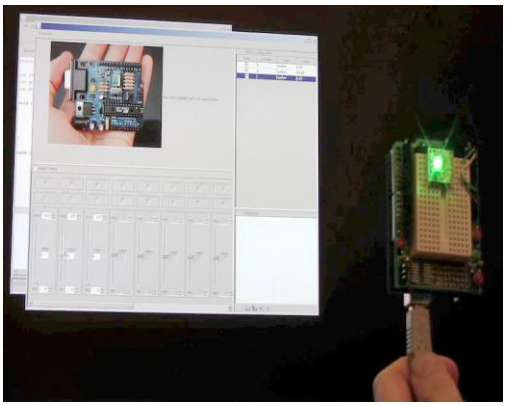

Figure 5.12: Example application demonstrating live tuning of color parameters of a smart multicolor LED through the Juxtapose runtime user interface.

Real-time tuning of variables requires a mapping from variable names to types and storage locations, which is not available in the C language that Arduino uses. Juxtapose constructs this map using a preprocessing step that transforms a user's program before compilation (Figure 5.11). The user's source code is parsed to build a table of global variable names, types, and pointers to their memory locations. The source is then wrapped in Juxtapose-specific initialization code, into which the variable table is emitted as C code. When a variable is tuned (Figure 5.12), the embedded wrapper code uses this table to find a pointer to the correct runtime variable from its name and changes the value of the memory location. The wrapper code also contains communication functions to exchange information between microcontroller and PC through a serial port. Some price must be paid for this added flexibility. The developer has to relinquish control of a hardware serial port, and application state is lost whenever alternatives are switched. Snapshots provide a way to save and restore values across such changes.

5.4.3 WRITING TUNABLE CODE

Ideally, programmers should be able to leverage tuning and alternatives in their project without changing their source. In practice, tuning is invisible unless modified parameter values have some observable effect on program execution. In other words, the changed variable has to be read again and some action has to be taken based on its value after it was modified at runtime. Thus programmers may have to write additional code that is solely concerned with making their application tunable.

To help programmers express the logic for runtime updates, callback functions provide a lightweight harness: whenever a variable is tuned at runtime, the application is notified of the parameter name and its updated value. In ActionScript, this callback facility is already provided on the language level by the `Object.watch()` method. The following example calls a redraw routine whenever the variable tunable is updated by the Juxtapose tuning UI:

```
01  var tunable = 5; //@RANGE 0..100
02  var counter; //@IGNORE
03  var callback= function(varName,oldVal,newVal){
04    redraw();
05    return newVal;
06  }
07  this.watch('tunable',callback);
```

Beyond callbacks, protocols to communicate information from the source code to the runtime interface enable designers to initialize the runtime UI programmatically. Programmers can specify minimum and maximum values for Number variables through comment annotations (line 1). They can also hide variables for which tuning is not useful, e.g., counters, from the variable list (line 2). Code annotations have been used in other projects as a source of meta-information, e.g., for labeling different experimental conditions for user testing [180]. Juxtapose currently uses code comments to capture annotations; this functionality could become part of the language definition in an alternative-aware programming language.

5.4.3.1 Hardware Support

Three important benefits can be realized by using a dedicated external controller instead of mouse and keyboard input for parameter control. First, spatially multiplexed input enables users to modify multiple parameters simultaneously. Second, with mouse control, tuning is mainly a hand-eye coordination task — with a dedicated control board, it turns into a motor task that leaves the eyes free to focus on the application being tuned. Third, moving the tuning UI to a dedicated controller allows for tuning of interactions that require mouse and

Figure 5.13: An external controller enables rapid surveying of multidimensional spaces. Variables names are projected on top of assigned controls to facilitate mapping.

keyboard input, e.g., adjusting the rate at which mouse wheel movement magnifies a document.

Our implementation supports a commercially available USB MIDI device [29] with 16 buttons with LED status indicators, 8 rotary encoders (presently not used) and 8 motorized faders (Figure 5.13). The controller transmits input events as MIDI control change messages and receives similar control change messages to actuate sliders and toggle LED feedback. Actuation of the hardware controller is essential for saving and restoring parameter snapshots — without actuation it is impossible to recall saved parameter values and edit them incrementally. To facilitate locating a particular variable's control, the mixer was augmented with a small top-mounted projector which displays parameter names next to the appropriate controls, a technique inspired by Crider et al. [65]. While a projector setup is unwieldy in practice, controllers with embedded text LCDs that can offer the same functionality are commercially available.

5.5 USER EXPERIENCES WITH JUXTAPOSE

To evaluate the authoring approach embodied in Juxtapose, we built example prototypes using the tool and conducted a summary usability study of Juxtapose for desktop applications. We recruited 18 participants, twelve male, six female. Participants were undergraduate and graduate students with HCI experience. Their ages ranged from 20 to 32

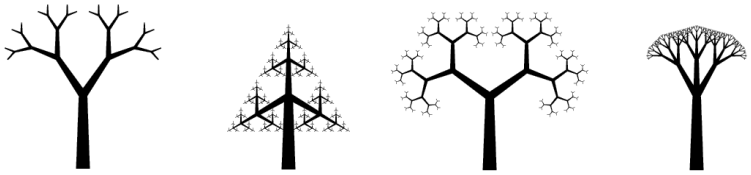

Figure 5.14: Study participants were given a code example that generates images of trees. They were asked to then match the four tree images shown above.

years. All but one participant had at least working knowledge of procedural programming and all had at least some expertise in interaction design.

5.5.1 METHOD

Evaluation sessions lasted approximately 75 minutes. Participants were seated at a workstation with mouse, keyboard and MIDI controller. After a demonstration of Juxtapose, participants were given three tasks. The first task was a warm-up exercise to modify a grid animation reacting to mouse movement, adapted from the book Flash Math Creativity [206]. Participants were asked to make changes that required both code alternatives and tuning.

The second task was a within-subject comparison that asked participants to adjust four parameters of a recursive tree-drawing routine to match four specific tree shapes (Figure 5.14). The provided code was also adapted from Flash Math. For two trees, this was accomplished using the full Juxtapose interface. For the other two, participants were given the same editor without the possibility of creating alternatives or tuning. Order of assignment between Juxtapose and control conditions was counterbalanced and a random tree order was generated for each participant.

The third task asked participants to work on the mapping scenario introduced earlier. They were provided with a working ActionScript program that loaded a map containing 28 different layers of information (e.g., land areas, parks, local streets, local street names, highways). Participants were given 30 minutes to create two map navigation alternatives. They were then asked to present their maps to a researcher. Documentation contained examples for how to programmatically change visibility of layers, color and brightness, text size and formatting, and mouse interactions. Participants had to modify and add to these examples to either hardcode design decisions or to set up tunable parameters through callback functions in the source code.

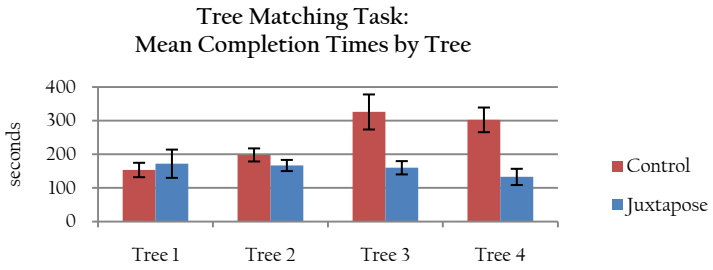

Figure 5.15: Study participants were faster in completing the tree matching task with Juxtapose than without.

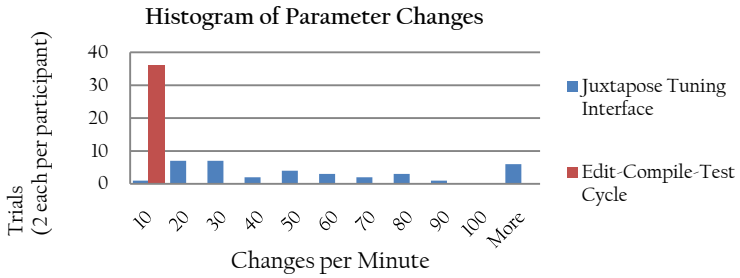

Figure 5.16: Study participants performed many more design parameter changes per minute with Juxtapose than without.

5.5.2 RESULTS

In all tasks, all participants properly applied linked and unlinked editing and tuning, with no apparent confusion. Participants commented positively on the ease of adjusting numerical parameters through tuning and the reduced iteration time this permitted. One participant commented that the explicit management of alternative documents improved on their existing practice of "half-hearted attempts to name saved [configurations] with memorable names." Today, designers commonly use layer sets as a technique for composing alternatives in graphics. A participant commented that Juxtapose brings this pattern to interaction design.

TUNING ENABLES MORE PARAMETER EXPERIMENTATION, FASTER

In the tree matching task, participants took an average of 258 seconds (σ: 133 s) to complete the matching in the control condition, and an average of 161 seconds (σ: 82 s) to complete the task with Juxtapose. This difference was significant (one-tailed, paired Student's t-test; p ‹

0.01). When looking at completion times by tree (Figure 5.15), a large discrepancy for trees three and four becomes apparent. For these trees, participants quickly narrowed in on the approximate shape but frequently had trouble minimizing the remaining visual disparity when they could no longer reason about how to proceed toward the goal. Participants then often broadened their search in parameter space and diverged from the solution while looking for the right parameters to adjust. We believe that Juxtapose outperformed the control condition here because the penalty for an uncertain, diverging move was much smaller — the result could immediately be observed and corrected.

To quantify the cost of making a change, we investigated how many parameter combinations participants explored. In the control condition, on average, participants tested 2.60 parameter combinations per minute to arrive at matches (σ : 0.93; we counted each execution after changing source as one combination). In contrast, using Juxtapose, participants executed the Flash file only once, and generated parameter changes through the tuning interface. Here participants explored 64 combinations on average (σ : 80; we counted each variable change sent to Flash as a tuning event). The external MIDI controller generated many input events and one might contend that our definition of parameter change over-estimates the number of perceptually different states explored by users. We note that participants adopted a wide range of tuning strategies — some exclusively typing in numbers in the tuning interface, others using multiple sliders simultaneously. This resulted in a wide spread of parameter changes per minute for Juxtapose (Figure 5.16), but even participants at the lower end of the histogram explored an order of magnitude more states than participants in the control condition.

ALTERNATIVES & TUNING PROVIDE VALUE, AT A PRICE

In our mapping task, many participants began by adding instrumentation code to the provided framework to make map attributes tunable at runtime. While hard-coding design choices into source code would have been easier from a programming perspective, participants spent extra effort to make variables tunable so they could experiment at runtime. Two participants mixed strategies, making some parameters tunable while setting others in code in different alternatives when they were sure about their desired values. For example, one participant hard-coded a higher initial magnification factor in the pedestrian map interface.

Most participants preferred to set the ranges for Number variables in source code, not in the runtime interface. Only one participant used the runtime interface for this purpose. A

possible explanation is that reasoning about ranges has to do with how a variable is used in the source so participants were more inclined to express ranges there.

SUGGESTIONS FOR IMPROVEMENT

The map task also uncovered a number of usability shortcomings. In multiple instances, participants closed the runtime window to change a line of code and recompile, discovering that their runtime parameter settings from the last execution were gone. To address this, Juxtapose could automatically save the last parameter values in a snapshot when the runtime window is closed.

Participants also wished for a larger range of variables to access — for the study, only variables declared in the main application class and variables of the root object of the visual hierarchy were accessible for tuning. Participants thus had to introduce intermediate variables to influence other graphical objects. It would be preferable to have a "tuning mode" for direct manipulation of all graphical objects, extending ideas introduced in SUIT [203].

Many participants expressed frustration at the lack of search and undo in the source editor. Both could clearly be added. Multiple participants also felt that it was overly onerous to properly write the application callbacks that make a design tunable. This can be addressed in two ways. Directly modifying object fields can be handled by making all fields tunable, not just global variables. More complex parameter mappings however will still require callbacks: producing these callbacks can be supported through a code generation wizard.

5.6 LIMITATIONS & EXTENSIONS

Juxtapose focused on exploring alternatives of user interfaces that were programmatically defined within a single file of source code. The design choices made during the development of Juxtapose represent one particular point in a larger space of tools for explorative programming. In this section, we discuss assumptions made in our current design and highlight limitations of our implementation. Following Fitzmaurice's design space for graspable interfaces [78], we summarize the most salient design decisions in Figure 5.17. This design space is not meant to be exhaustive — it covers the decision points encountered during prototyping and development. Nevertheless, the table suggests additional techniques, such as automatic generation of alternatives, which may be a fruitful area for future work.

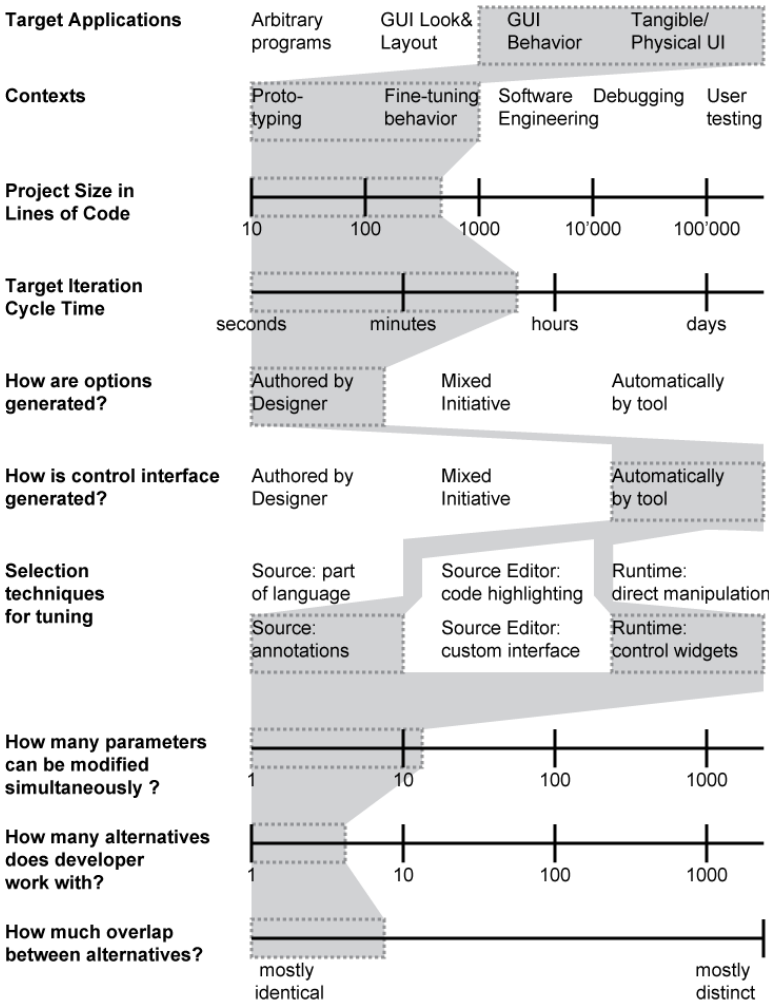

Figure 5.17: A design space for exploring program alternatives. Choices implemented by Juxtapose are shown with a shaded background.

5.6.1 WILL DESIGNERS REALLY BENEFIT FROM LINKED SOURCES?

The efficacy of linked editing in Juxtapose rests on the assumption that interaction designers create multiple alternatives of a common code document, where individual alternatives only differ in parameter settings and small sections of code. Experimenting with code in this manner only covers part of the solution space for a given problem. Different solution approaches may be based on distinct implementations. Alternatives as discussed in this paper explore options within one particular solution strategy. Are alternative designs related enough in practice to benefit from linked editing and tuning?

Beyond evidence from our formative interviews, the book Flash Math Creativity [206] provides detailed examples of source code experimentation by professionals: 15 Flash designers share how they create computational designs in 56 projects. Each project starts from a single idea, e.g., animating geometric grid structures. The designers then show how they modified the initial source to explore the design space. 12 of 15 designers showed multiple alternatives for their projects (mean: 10.2 alternatives per project; range: 3 to 23). The difference between these alternatives is usually small: a change to a line of code to load different graphics, alterations to parameter values, or substitutions of function calls.

5.6.2 IS TUNING OF NUMBERS AND BOOLEANS SUFFICIENT?

Juxtapose's runtime tuning focuses on direct manipulation of Boolean and Number types. Would designers benefit from more expressive abstractions and additional functionality in the tuning interface?

An underlying assumption in this work is that developers both produce the application and tune it. If they desire a more complex mapping, e.g., a logarithmic parameter scale, they may express this mapping in the source. Locating additional functionality in the source itself may be more useful since logic expressed in the tuning UI is not available when the application is run outside Juxtapose. This assessment changes if alternatives and tuning options are used by a third party, e.g., during participatory design sessions. In this case it would make sense to imbue the runtime interface with more flexibility to let users express a more complete set of modifications without editing the program source, e.g., by providing rich widgets for commonly used complex data types such as colors or coordinates.

5.6.3 ARE CODE ALTERNATIVES ENOUGH?

Perhaps the most important limitation is that Juxtapose does not offer support for managing multiple alternatives of graphical assets. Interface design is concerned with both look and feel — graphics and behavior. Many popular user interface authoring tools today follow a hybrid authoring approach, where graphical appearance is edited through visual direct manipulation, while behavior is specified in source code (e.g., Flash [1], Director [6]). We believe Juxtapose is a first step towards an integrated authoring environment that offers management of alternatives across graphics and code. Future research should investigate to what extent it is possible to offer a coherent method of exploring alternatives for both, in a single tool. The most relevant prior work for exploring graphical alternatives is Terry's work on embedding alternatives for graphics manipulations into a single canvas [242], and research on editable

graphical histories [153,236]. However, a naive crossproduct of Juxtapose's linked editing and graphical alternative or history techniques is unlikely to work, because it would likely overburden the user with too many inconsistent methods of making choices. The goal of future research should be to find a single, "simple-enough" mental model.

5.6.4 ALTERNATIVES FOR COMPLEX CODE BASES

Another open question is how an alternative-aware editor could be extended to handle large software projects. Juxtapose targeted UI prototypes, for which interaction logic is frequently authored in a single source file today. If the goal is not the design of a new UI, but the augmentation of an existing program, designers may have to contend with large existing code bases. For example, a software engineer at Adobe reported that to try alternatives for a new feature in a large authoring tool, he would have to check out several thousand files into independent workspaces, and manage any changes between alternatives manually [94].

As an interaction technique, we have envisioned the use of hierarchical tabs where the top level identifies the alternative, and a lower level identifies the file within the alternative.The primary challenge will be to reduce the potential complexity stemming from dealing with multiple alternatives in the authoring interface. As an implementation strategy, it would be interesting to consider to what extent virtualization technology can be harnessed to quickly create independent copies of complex applications and system configurations that are adequately isolated from each other.

5.6.5 SUPPORT EXPLORATION AT THE LANGUAGE LEVEL

Juxtapose chose to implement support for runtime tuning at the library level — the source language, ActionScript in the case of Juxtapose, remained unchanged. Juxtapose shares this approach with prior work like Amulet [190]. Operating as a library has the advantage that Juxtapose can target a widely used language; it has the drawback that the program has to be explicitly changed to include library support. More importantly, the library has limited control over program execution at runtime. For example, when running multiple alternatives side by side, it is not possible to pause execution of one application as it loses focus — all applications run in parallel, even if interaction with them is sequential.There are two possible ways for future research to extend the reach of runtime exploration:

1) augment an existing programming language with additional language constructs
2) develop a new language to provide explicit developer control over alternatives and variable parameter spaces.

Terry's Partials project [239: Appendix B] was an exploration of the first option. He augmented the Java language with the keywoard "partial" which could be used to decorate variable definitions to gain runtime control over those variable values. It is worthwile to explore what benefits an entirely new language targeted at exploration could provide.

5.6.6 INTEGRATE WITH TESTING

A final direction worth pursuing in future work is to extend parallel editing and tuning to support user testing of alternatives. A particularly promising application domain would be the authoring of user interfaces for web applications, since online deployment could provide a way to rapidly gather empirical data on user preferences for different alternatives. Large web sites already routinely test alternatives of new features by running controlled bucket experiments: a small percentage of site visitors are exposed to a new proposed feature or layout, and results (time spent on site, purchases made) are compared with the control condition [16]. An interesting an as-of-yet unexplored research question is to what extent such comparative testing with remote users is possible during earlier prototyping stages.

5.7 SUPPORTING ALTERNATIVES IN VISUAL PROGRAMS

How might support for alternative behavior transfer from the textual programming domain of Juxtapose into visual authoring environments such as d.tools? Following our implementation of Juxtapose, we examined to what extent the advantages of defining and editing multiple alternatives can be realized within d.tools. We have not yet investigated how to transfer variable tuning; partially because variables play a less prominent role within d.tools projects. Because d.tools focuses on user interfaces with custom hardware, parallel execution of alternatives is less likely to be useful. We therefore focused on expressing and managing alternatives in the editor, but only support executing one alternative at a time.

What level of abstraction should alternatives operate on? Juxtapose manages alternatives at the file level. For visual diagrams, this choice is also possible, but less compelling. A prototype implementation of file alternatives in d.tools suggested that making sense of the differences between alternative files is harder for visual programs than for textual ones. Specifically, changes in the visual gestalt of the diagram are not necessarily related to changes in the functionality expressed by the diagram. Rearranging states in a d.tools diagram changes appearance but not logic. We therefore sought ways to express alternatives within a single diagram, at the state level.

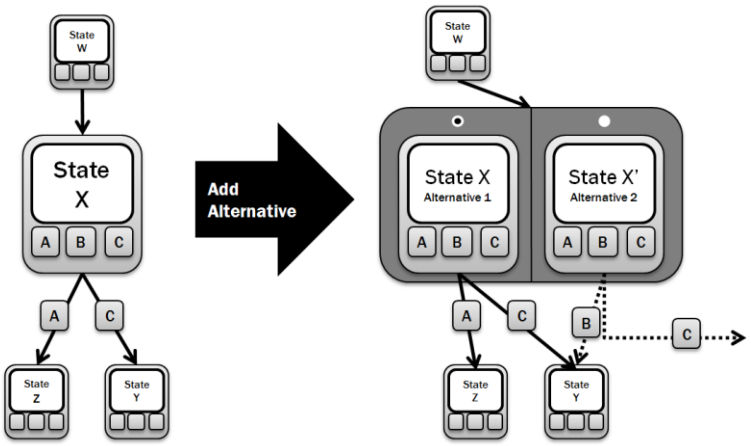

Figure 5.18: Schematic of state alternatives in d.tools: alternatives are encapsulated in a common container. One alternative is active at a time. Alternatives have different output and different outgoing transitions.

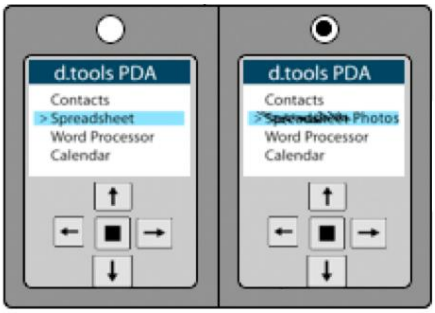

Figure 5.19: Screenshot of a d.tools container with two state alternatives. In the right alternative, screen graphics have been revised.

Designers can introduce *state alternatives* in d.tools to define both appearance and application logic. An alternative container (Figure 5.18, Figure 5.19) encapsulates two or more states. State alternatives are created in a manner analogous to the Juxtapose editor: designers select a state and choose "Add Alternative" from its right-click context menu. The original state (with all defined output such as screen graphics) is duplicated and both states are placed into an alternative contained. To express that the incoming transitions remain the same, regardless of which alternative is active, the original state's incoming connections are rerouted to point to the encapsulating container. To define which of the alternative states should become active when control transfers to an alternative container, the container shows

radio buttons, one above each contained state. Outgoing transitions are not shared between alternatives: each state can thus define its own set of target states and transition events. To reduce visual clutter, only outgoing transitions of the active alternative are shown; other outgoing transitions are hidden until that state is activated.

State alternatives support more localized changes than Juxtapose's code alternatives. If alternatives are defined for more than one state, managing correspondences between the different alternatives is currently cumbersome. Support to combine different alternatives into coherent alternative sets is needed and should be addressed in future work. State alternatives have been evaluated in laboratory studies as part of the d.note project on revising d.tools diagrams, which will be described in the next chapter.

CHAPTER 6 GAINING INSIGHT THROUGH FEEDBACK

Iterative design proceeds in cycles of creating prototypes, testing what was created, and analyzing the obtained feedback to drive the next design. The ultimate purpose of a prototype is thus to elicit feedback that can inform future designs. If iteration based on feedback is a central activity of design, then tools should include functionality to explicitly support capturing, organizing, and analyzing feedback obtained from a particular prototype. This chapter presents two approaches to make prototyping tools feedback-aware: capturing and organizing video data from prototype test sessions, and managing revisions and change suggestions in visual storyboard diagrams.

6.1 FEEDBACK IN USER TESTING: SUPPORTING DESING-TEST-ANALYZE CYCLES

Video recordings of prototypes in use can provide critical usability insights and aid in communicating these insights to other team members, but working with usability video can be prohibitively time consuming [179]. Our fieldwork indicated that, even though video recording of user sessions is common in design studios, resource limits often preclude later analysis of this data. Video is recorded, but rarely used after the fact. This section introduces techniques that radically shorten the time required to review usability test data of physical prototypes to unlock some of the latent value of usability videos for design teams. The d.tools

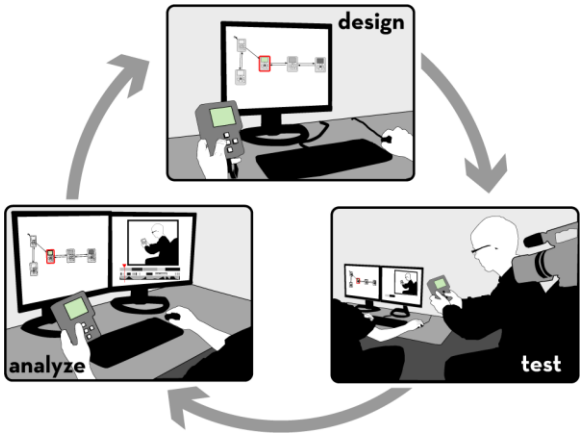

Figure 6.1: d.tools supports design, test & analysis stages through integration with a video editor.

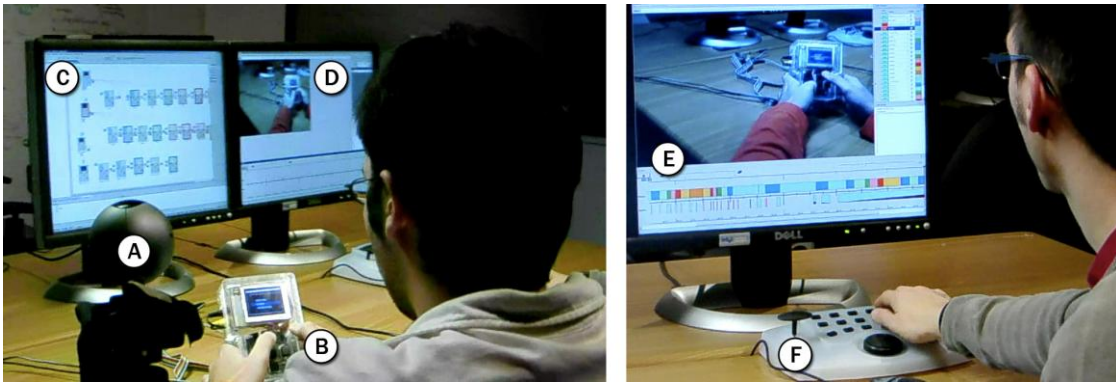

Figure 6.2: Testing a prototype built with d.tools: A camera (A) is aimed at the tester and the physical prototype (B), which is driven by a storyboard (C) in d.tools. Live video of the test is recorded in the video editor (D) and annotated with events and state changes (E). Designers can add additional events to the record with a control console (F).

video suite integrates support for design, test, and analysis of prototypes in a single tool (Figure 6.1). The guiding insight is that *timestamp correlation between recorded video and execution event traces of the prototype can provide access from the video to the model, and vice versa*.

The d.tools video suite adds two usage modes to the d.tools environment: test mode, which records live video and an event trace of the test; and analysis mode, which provides access to the recorded data from one or more test sessions and introduces interaction and visualization techniques that enable rapid video querying. The following sections describe each mode in turn.

6.1.1 TESTING PROTOTYPES

After completing construction of a prototype, when seeking to gather feedback from others, designers switch to test mode. In test mode, d.tools records live video and audio of user interactions with the prototype — important for understanding ergonomics, capturing user quotes, and finding usability problems (Figure 6.2). During a test, a video camera (Figure 6.2A) is aimed at the tester and the prototype (Figure 6.2B). The interaction logic of the prototype is defined by a particular storyboard (Figure 6.2C). As in design mode, input from the prototype causes state transitions in the storyboard and corresponding output defined in states is shown on the prototype. In addition to this normal functionality, all device events and state transitions are saved in a time stamped log for video synchronization. Live video from the camera is recorded in a video editor (Figure 6.2D & E). The live video stream is augmented with event and state transition metadata in real-time. As events and transitions

Figure 6.3: The video recording interface in test mode. **A:** Active states at any point in time are encoded in a timeline view. **B:** Discrete input events show up as instantaneous events or press/release pairs. **C:** Continuous input data is visualized in-situ as a small graph in the timeline.

occur during a test, they are visualized in several annotation tracks in a timeline display (Figure 6.3).

One row of the timeline corresponds to the active state of the storyboard at any given point in time (Figure 6.3A). To clarify correspondence between storyboard states and video segments, state outlines in the editor are color coded, and the same color scheme is used for timeline segments. A second row in the timeline displays hardware input events. Three types of hardware events are displayed. Instantaneous events, such as a switch changing from on to off, appear as short slices on the timeline. Events with duration, such as the press and release of a button, show up as block segments (Figure 6.3B). Lastly, continuous events, such as slider movements, are drawn as small line graphs of that event's value over time (Figure 6.3C).

In addition to automatically generated timeline events, the designer can also explicitly add markers during a test session on an attached video control console (Figure 6.2F). The console enables designers to quickly mark sections for later review (e.g., interesting quotes or usability problems). The experimenter's annotations are displayed in the video view as a separate row on the timeline.

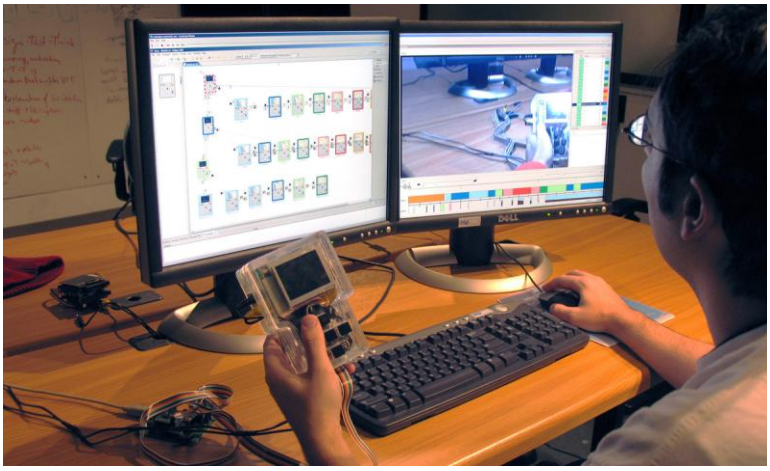

Figure 6.4: In analysis mode, a dual-screen workstation enables simultaneous view of state model and video editor.

6.1.2 ANALYZING TEST SESSIONS

Analyze mode allows the designer to review the data from one or more user test sessions. The video view and storyboard editor function in tandem as a multiple view interface [41] into the test data to aid understanding of the relationship between the user experience and the underlying interaction model (Figure 6.4). d.tools supports both single user analysis and group analysis, which enables designers to compare data across multiple users.

6.1.2.1 Single User Analysis

Single user mode provides playback control of a single test session video. The timeline visualization of collected metadata shows the flow of UI state and data throughout that session. d.tools speeds up video analysis by enabling designers to access interaction models through the corresponding video segments and to access video segments from the interaction model, facilitating analysis within the original design context. In addition to this dynamic search and exploration, the storyboard also shows an aggregation of all user interactions that occurred during the test: the line thicknesses of state transitions are modified to indicate how often they were traversed (Figure 6.5). This macro-level visualization shows which transitions were most heavily used and which ones were never reached.

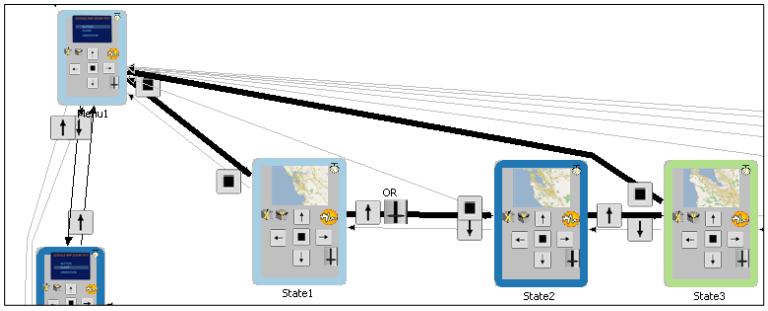

Figure 6.5: Line thickness in analysis mode shows how many times a given transition was taken.

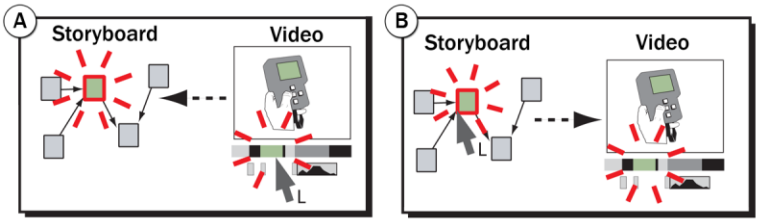

Figure 6.6: Two query techniques link storyboard and video. **A**: Selecting a video segment highlights the state that was active at that time. **B**: Selecting a state in analyze mode highlights the corresponding video segment(s).

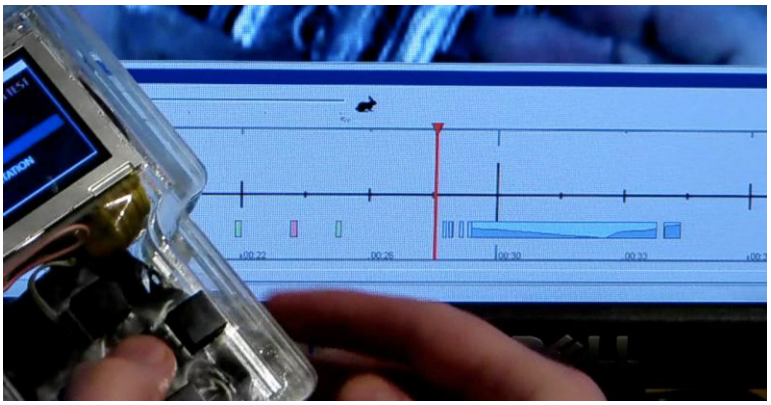

Figure 6.7: Designers can *query by demonstration*: Generating input events in analyze mode filters recorded video so that only those sections where similar events were received are shown.

VIDEO TO STORYBOARD

During video playback, a dynamic visualization of transition history is displayed in the d.tools storyboard. The state that was active at the current point in time of the video is highlighted. d.tools also animates moving trail along the state transitions, indicating which state a user

was coming from, and which state will become active next. This window into the chronology of interactions provides a visual reminder of context. Selecting any state segment in the timeline moves the play head to that segment and, as a result, highlights the corresponding state in the storyboard (Figure 6.6A)

STORYBOARD TO VIDEO

To query video using the interaction model, the designer can select a state in the storyboard — the recorded video is then filtered such that only segments where the user was in the corresponding state are highlighted in the timeline view and played (Figure 6.6B). In addition to querying by selecting states, designers can query for video segments in which certain input components were used through a *query-by-demonstration* technique: manipulating a hardware component on the physical prototype (e.g., pushing a button or moving a slider) causes the corresponding input event category to be selected in the video view (Figure 6.7). Designers can also select multiple categories by manipulating multiple hardware components within a small time window. Thus, the designer can effectively search for a particular interaction pattern within the video data by reenacting the interaction on the prototype itself.

6.1.2.2 Group Analysis

Group mode collects data from multiple test sessions of a given storyboard (Figure 6.8). The timeline now aggregates flows for each user. The video window displays an $n \times m$ table of

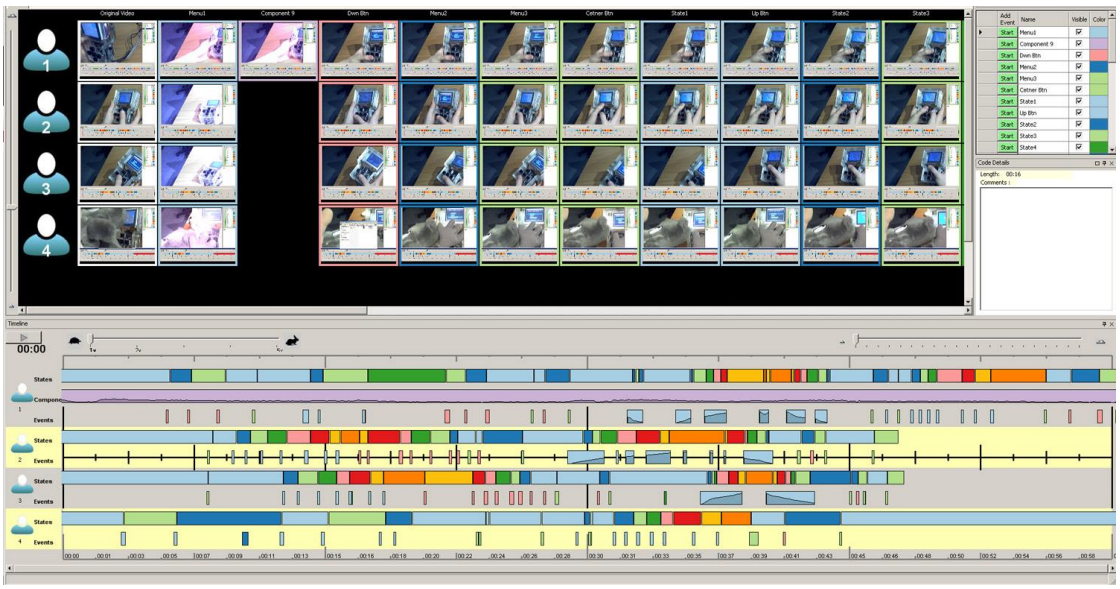

Figure 6.8: Group analysis mode aggregates video and event data of multiple user sessions into one view.

videos, with the rows corresponding to the *n* users, and the columns corresponding to *m* categories (comprised of states, hardware events, and annotations). Thus, a cell in the table contains the set of clips in a given category for a given user. Any set of these clips may be selected and played concurrently. Selecting an entire row plays all clips for a particular user; selecting an entire column plays all clips of a particular category. As each clip is played, an indicator tracks its progress on the corresponding timeline.

6.1.3 IMPLEMENTATION

The d.tools video editor is implemented as an extension to the VACA video analysis tool [54]. The video viewer is implemented in C# and uses Microsoft DirectShow technology for video recording and playback. Synchronization between the storyboard and video views is accomplished by passing XML fragments over UDP sockets between the two applications. As video recording and playback is CPU-intensive, this separation also allows authoring environment and video editor to run on different machines. DirectShow was chosen because it allows synchronized playback of multiple video streams, which is needed for group analysis mode. The use of Microsoft APIs for video processing limits use of d.tools testing and analysis modes to Windows PCs.

6.1.4 LIMITATIONS & EXTENSIONS

The d.tools video suite introduces ways to integrate the prototype authoring environment and a video review tool for analyzing user test sessions. The focus on rapid review of test sessions of a single prototype limits the utility of d.tools video functions in some important areas, which we review in this section.

6.1.4.1 No Support for Quantitative Analysis

Analysis in formal user testing often involves quantifying observations and computing statistics for those observations. In d.tools, video analysis so far is restricted to accessing and reviewing video segments. The introduced interaction techniques shorten the time required to access the right parts of the video. We hypothesize that tools could further aid designers by extracting relevant statistics for the test automatically.

Two complementary strategies to support more quantitative analysis suggest themselves: the first is to automatically extract data from the recorded test (e.g., state dwell statistics — how long did users spend in each state, which states were never reached). The second is to provide the reviewer with better tools to conduct such analyses manually. An

example of a more fine-grained analysis approach is Experiscope [97], a tool for analyzing user tests of mouse- or stylus-based interaction techniques. Experiscope can both visualize input event data well as produce aggregate reports of event frequency and duration. As an initial step into this direction, d.tools visualizes how many times a transition was taken by changing transition line thickness in the diagram. However, it is not currently possible to extract the precise number of times the transition was taken, or to derive a similar figure for the number of times a state was active during a test.

6.1.4.2 Limited Visibility of Application Behavior During Test

d.tools video records a single stream of live video from a digital camera. Recording how a device was handled is especially important for devices with new form factors, as ergonomics and questions about device control layout may be part of the test. This focus on embodied use of a device during a test comes at a price: it is not always possible to see what happened on the screen(s) of the tested prototypes in live video. Linking the video to the state diagram enables the tester to see which state the device was in at any given time. However, states present only a static view of the application. Dynamic animations scripted in d.tools are not visible — reviewing these may be important as well. One possible solution suggested by commercial GUI testing applications such as Silverback [4] is to record multiple video streams of both live video and screen output and to then composite those streams into a single video feed.

6.1.4.3 Cannot Compare Multiple Prototypes in Analysis Mode

The video spreadsheet view enables comparison of multiple test sessions by multiple users, but only for a single prototype. As the previous chapter has argued, exploration of design alternatives is an important practice and should therefore be supported in analysis tools as well. We see two separate opportunities for further research: 1) enabling comparative testing of multiple, simultaneously developed alternatives; 2) supporting comparison of prototypes across different design iterations.

Tohidi and Buxton [243] note that testing multiple prototypes is preferable to testing a single prototype, since users will feel less pressured to be "nice" to experimenters and can draw comparisons between prototypes. In addition, if prototypes are more refined and the designer has concrete hypotheses in mind, formal comparative testing is required to support or reject these hypotheses. For traditional GUI interactions, tools that support such comparative analysis of alternatives exist. Experiscope [97] enables testers to visually

compare event traces of multiple treatment conditions side-by-side. However, existing tools such as Experiscope do not link the recorded trace back to the source of the application being tested. It is an open question how tools can show video, event traces, and software models for multiple alternative designs simultaneously without overwhelming the designer with complexity.

A separate question is how one might support the comparison of different iterations of a given project over time. In the iterative design-test-analyze paradigm, subsequent iterations are informed by what was learned before. Testing tools should offer support for checking whether the feedback collected during prior iterations was properly acted on in later iterations and if identified issues were in fact resolved.

6.1.4.4 Limited Query Language

An additional limitation of d.tools video analysis is that the query language over states and events is rather primitive at the present time. The queries that can be executed select segments from single video files based on states or input events. A natural extension would be to enable testers to specify more complex, and thus more useful, queries. Badre suggests using regular expressions to filter user events [39]. We are skeptical whether regular expressions are accessible to our target audience. An alternative approach would be to use a textual query language, such as SQL, and then building GUI tools for specifying queries in that language. Interactive query builders are common for expressing SQL queries in database applications.

6.1.4.5 Interaction Techniques Have Not Been Formally Evaluated

The introduced interactions have not been evaluated in a formal user study. Their efficacy in real design contexts has not been established, although the rapid video query techniques have received positive comments from professional designers in informal conversations and at presentations to professional design conference attendees.

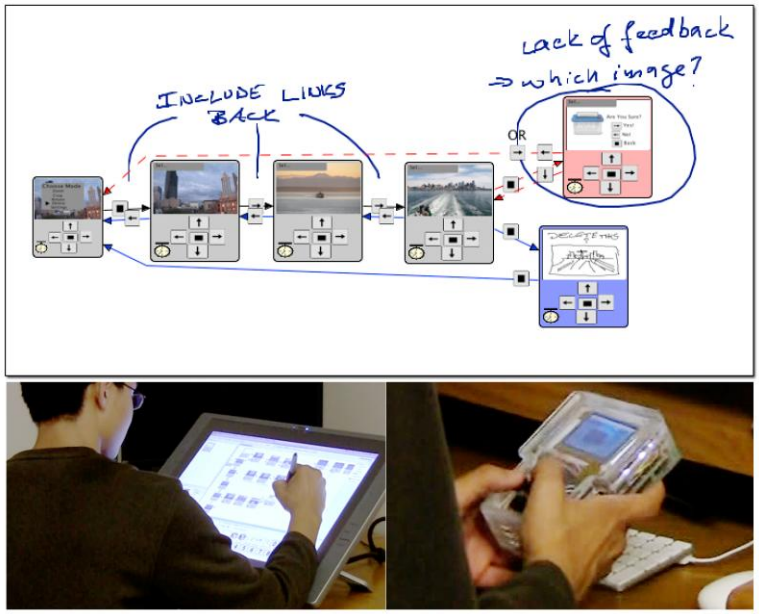

Figure 6.9: d.note enables interaction designers to revise and test functional prototypes of information appliances using a stylus-driven interface to d.tools.

6.2 CAPTURING FEEDBACK FROM OTHER DESIGNERS: D.NOTE

Interaction design in teams oscillates between individual work and team reviews and discussions. Team reviews of user interface prototypes provide valuable critique and suggest avenues forward [189:pp. 374-5]. However, changes proposed by others can rarely be realized immediately: often the proposer lacks the implementation knowledge, the changes are too complex, or the ideas are not sufficiently resolved.

In many areas of design, annotations layered on top of existing drawings and images, or "sketches on top of sketches" [55], are the preferred way of capturing proposed changes. They are rapid to construct, they enable designers to handle different levels of abstraction and ambiguity simultaneously [66], and they serve as common ground for members with different expertise and toolsets [205]. Individual designers later incorporate the proposed changes into the next prototype. This annotate-review-incorporate cycle is similar to revising and commenting on drafts of written documents [198]. While word processors offer specialized revision tools for these tasks, such tools don't yet exist for the domain of interaction design.

This section demonstrates how three primary text revision techniques can be applied to interaction design: commenting, tracking changes, and visualizing those changes. It also

introduces revision tools unique to interaction design: immediate testing of revisions and proposing alternatives. The novel revision techniques are embodied in d.note (Figure 6.9), an extension to d.tools. The d.note notation supports modification, commenting, and proposal of alternatives (see Section 5.7, p. 140) for both appearance and behavior of information appliance prototypes. Concrete modifications to behavior can be tested while a prototype is running. Such modifications can exist alongside more abstract, high-level comments and annotations.

This section also contributes a characterization of the benefits and tradeoffs of digital revision tools such as d.note through two user studies. We show that the choice of revision tool affects both what kind of revisions are *expressed*, as well as the ability of others to *interpret* those revisions later on. Participants who used d.note to express revisions focused more on the interaction architecture of the design, marked more elements for deletion, and wrote fewer text comments than participants without d.note. Participants who interpreted d.note diagrams asked for fewer clarifications than participants that interpreted freeform annotations, but had more trouble discerning the reviser's intent.

In the remainder of this section, we first describe revision principles from related domains. Current practices of UI designers were described in Section 3.1.2.2. We then introduce d.note and its implementation. We present results from two studies of revision expression and interpretation, and conclude by discussing the design space of revision tools.

6.2.1 REVISION PRACTICES IN OTHER DOMAINS

Interaction designers are concerned with both look and feel of applications [189]. Absent a current, complete solution for both aspects, we can draw on important insights from revising textual documents, source code, and movie production.

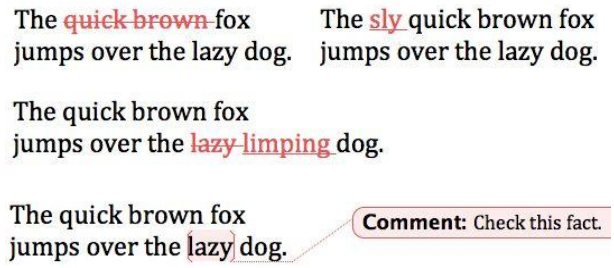

Figure 6.10: Interlinear revision tracking and comment visualization in word processing.

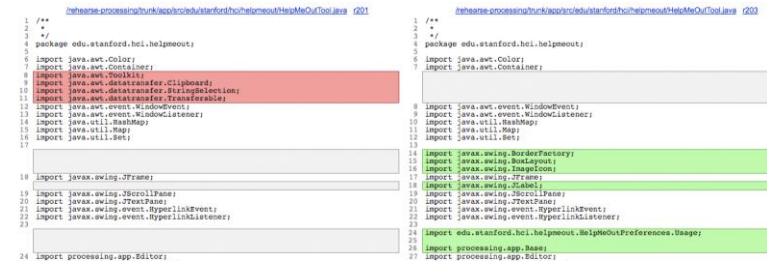

Figure 6.11: Source code comparison tools show two versions of a file side-by-side.

Figure 6.12: Video game designers draw annotations directly on rendered still images (from [55:p. 179]).

TEXT DOCUMENTS

The fundamental actions in written document revision are history-preserving modification (insertion, deletion) and commenting. Each operation has two components: visual syntax and semantics. For example, in word processing, a common interlinear syntax to express deletion is striking through the deleted text (Figure 6.10); the semantics are to remove the stricken text from the next version of the document, should the revision be accepted. Original and modification are visible simultaneously, to communicate the nature of a change. Furthermore, edits are visually distinguished from the base version so the recipient can rapidly identify them. When editing documents collaboratively, different social roles of co-author, commenter, and reader exist [198]. Offering ways to modify the underlying text as well as adding meta-content that suggests further modification serves these different roles.

SOURCE CODE DOCUMENTS

Source code revision tools, such as visual difference editors, enable users to compare two versions of source files side-by-side [115] (Figure 6.11). In contrast to document revision tools, changes are generally not tracked incrementally, but computed and visualized after the fact. Comments in source code differ from comments in text documents as they are part of the

source document itself. Meta comments (comments about changes) are generally only available for an entire set of changes.

VISUAL MEDIA

WYSIWYG document editors do not distinguish between source and final document; authors revise a single, shared representation. For program source code, there is no way to comment directly on the output of the program, only the source. In contrast, movie producers and video game developers convey revisions by drawing directly on output, i.e., rendered video frames (Figure 6.12). Because the revisions address changes in appearance, sketching is the preferred method of expression. Working in the output domain is a compelling approach, but has thus far been limited to static content [55].

DESIGN PRINCIPLES

Comparing these three existing domains leads to the formulation of four design principles. UI revision tools should support the following workflows:

1) History-preserving incremental modification of the source representation
2) Commenting outside the underlying source language
3) Sketching as an input modality for graphical content
4) Revising the output, i.e., the resulting user interface screens, not just the source.

6.2.2 A VISUAL LANGUAGE FOR REVISING INTERACTIONS

Guided by our assessment of current practice and tools available in other domains, we developed d.note, a revision notation for user interface prototypes. d.note extends the d.tools authoring environment. In text, the atomic unit of modification is a character. Because visual program diagrams have a larger set of primitives, the set of possible revision actions is more complex as well. In d.tools, the primitives are states, transitions, the device definition, and graphical screens. With each primitive, d.note defines both syntax and semantics of modification. This section will provide an overview of each modification operation. Concrete examples of these operations in d.note are provided in Figure 6.13 – Figure 6.17.

6.2.2.1 Revising Behavior

d.note uses color to distinguish base content from elements added and removed during revision. In d.note and in the following diagrams, states and transitions rendered with a black outline are elements existing in the base version; added elements are shown with a blue outline; deleted elements in red.

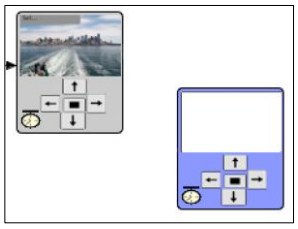

Figure 6.13: States added during revision are rendered in blue.

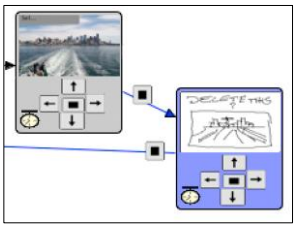

Figure 6.14: New screen graphics can be sketched in states.

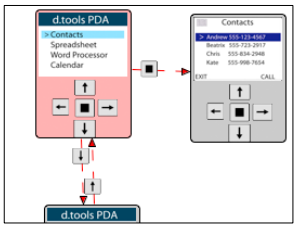

Figure 6.15: State deletions are rendered in red. Connections are marked as inactive.

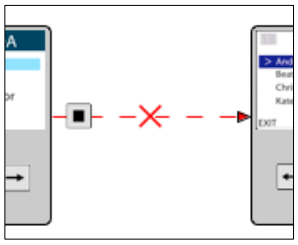

Figure 6.16: Transition deletions are marked with a red cross and dashed red lines.

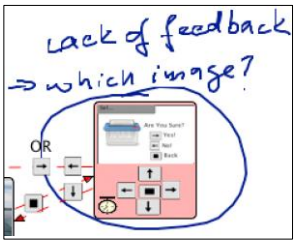

Figure 6.17: Comments can be attached to any state.

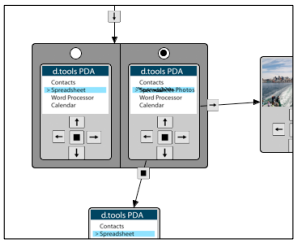

Figure 6.18: Alternative containers express different options for a state.

In revision mode, users can add states and transitions as they normally would; these states and transitions are rendered in blue to indicate their addition (Figure 6.13, Figure 6.14). Semantically, these states and transitions behave like their regular counterparts.

When users remove states from the base version, the state is rendered as inactive in red. To visually communicate that a state can no longer be entered or exited, all incoming and outgoing transitions are rendered as inactive with dashed lines (Figure 6.15). At runtime, incoming transitions to such states are not taken, making the states unreachable. Individual transitions can also be directly selected and deleted. Deleted transitions are shown with a dashed red line as well as a red cross, to distinguish them from transitions that are inactive as a result of a state deletion (Figure 6.16). As with many source code and word processing tools, deleting states or transitions that were added in revision mode completely removes the objects from the diagram.

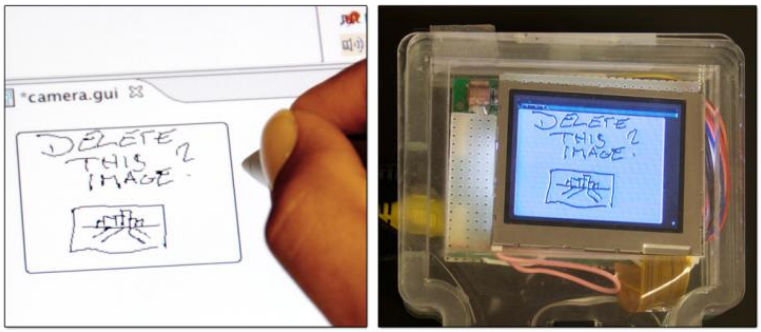

Figure 6.19: Sketched updates to screen content are immediately visible on attached hardware.

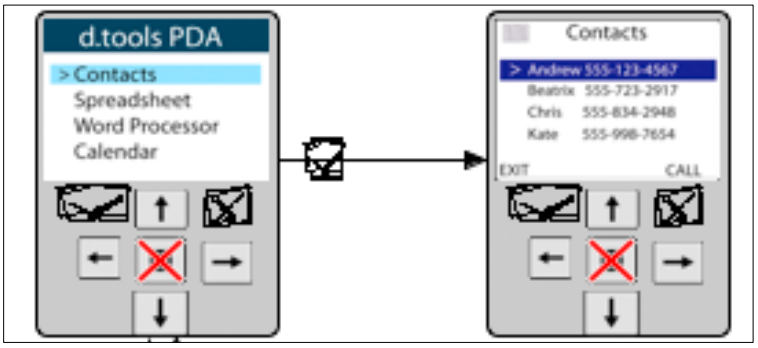

Figure 6.20: Changes to the device configuration are propagated to all states. Here, one button was deleted while two others were sketched in.

6.2.2.2 Revising Appearance

Designers can modify graphics by sketching directly on top of them with a pen tool within the d.tools graphics editor (Figure 6.19). Sketched changes are then rendered on top of the existing graphics in a state at runtime. In addition to sketching changes to appearance, users may also rearrange or otherwise modify the different graphical components that make up the screen output of a state. d.note indicates the presence of such changes by rendering the screen outline in the state editor in a different color, as showing modification side-by-side with the original graphics would interfere with the intended layout. The changes are thus not visualized on the level of an individual graphical widget, but in aggregate.

6.2.2.3 Revising Device Definition

Thus far, we have described changes to the information architecture and graphic output of prototypes. When prototyping products with custom form factors such as medical devices,

the set of I/O components used on the device may also be subject to change and discussion. When revising designs in d.note, users can introduce new input elements by sketching them in the device editor (Figure 6.20). Prior to binding the new component to an actual piece of hardware, designers can simulate its input during testing using the d.tools simulation tool (see Section 4.1.3). Currently, the d.note implementation does not support adding output devices through sketching; we believe adding output within this paradigm would be fairly straightforward.

6.2.2.4 Commenting

In addition to functional revision commands, users can sketch comments on the canvas of device, graphics, and storyboard editors (Figure 6.17). Any stroke that is not recognized as a revision command is rendered as ink. This allows tentative or ambiguous change proposals to coexist with concrete changes. Inked comments are bound to the closest state so they automatically move with that state when the user rearranges the diagram.

6.2.2.5 Proposing Alternatives

As covered in Section 5.7 (p. 140), users can introduce alternatives for appearance and application logic. We summarize the functionality of alternative containers again briefly: d.tools represents the alternative by duplicating the original state and visually encapsulating both original and alternative (Figure 6.18). The original state's incoming connections are rerouted to point to the encapsulating container. Each state maintains its own set of outgoing transitions. To define which of the alternative states should become active when control transfers to an alternative set, the set container shows radio buttons, one above each contained state. To reduce visual clutter, only outgoing transitions of the active alternative are shown; other outgoing transitions are hidden until that alternative is activated.

6.2.3 SCENARIO

The following scenario summarizes the benefits d.note provides to interaction design teams. Adam is designing a user interface for a new digital camera with on-camera image editing functions. To get feedback, he drops his latest prototype off in Betty's office. Betty picks up the camera prototype, and tries to crop, pan and color balance one of the pictures that Adam preloaded on the prototype. She notices that exiting to the top level menu is handled inconsistently in different screens. She opens up the d.tools diagram for the prototype and, with d.note enabled, changes the transitions from those screens to the menu state. She next

notices that the image delete functionality is lacking a confirmation screen – images are deleted right away. To highlight this omission, Betty creates a new state and sketches a rudimentary confirmation dialog, which she connects to the rest of the diagram with new transitions so she can immediately test the new control flow. Betty is not convinced that the mapping of available buttons to crop an image region is optimal. She selects the crop state and creates an alternative for it. In the alternative, she redirects button input and adds a comment for Adam to compare the two implementations. She also thinks that the current interface for balancing colors via RGB sliders is cumbersome. Since she does not have time to change the implementation, she circles the corresponding states and leaves a note to consider using an alternative color space instead.

6.2.4 THE D.NOTE JAVA IMPLEMENTATION

d.note was implemented as an extension to d.tools. As such, it was written in Java 5 and makes use of the Eclipse platform, specifically the Graphical Editing Framework (GEF) [24]. d.note runs on both Windows and Mac OS X operating systems.

6.2.4.1 Specifying Actions Through Stylus Input

Because much of early design relies on sketches as a visual communication medium [55], d.note's revision interface can be either operated through mouse and keyboard commands, or it can be entirely stylus-driven. Stylus input allows for free mixing of commands and non-command sketches. When using the stylus, strokes are sent through a recognizer (the Paper Toolkit [258] implementation of Wobbrock et al.'s $1 recognizer [253]) to check if they represent a command. Command gestures to create states and alternatives use a pigtail delimiter [120], to reduce the chance of misinterpretation of other rectangular strokes (Figure 6.21). Gesture recognition takes into account what existing diagram element (if any) a gesture was executed above. The gesture set contains commands to delete the graphical element

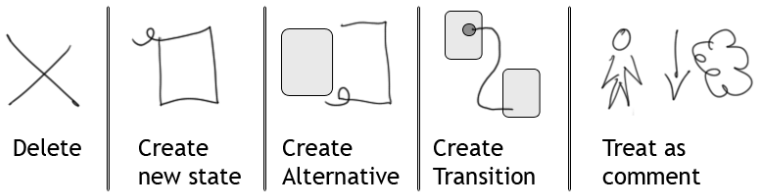

Figure 6.21: The d.note gesture set for stylus operation. Any stroke not interpreted as one of the first four actions is treated as a comment.

underneath the gesture, and to create new states, transitions and alternatives. All other strokes are interpreted as comments. In addition to providing drawing and gesture recognition, d.note extends the d.tools runtime system to correctly handle the interaction logic semantics of its notation, e.g., ignore states marked for deletion.

6.2.5 EVALUATION: COMPARING INTERACTIVE & STATIC REVISIONS

To understand the user experience of the interactive revision techniques manifest in d.note, we conducted two studies: the first compared *authoring* of revisions with and without d.note; the second compared *interpretation* of revisions with and without d.note. We recruited product design and HCI students at our university. Because the required expertise in creating UIs limited recruitment, we opted for a within-subjects design, with counterbalancing and randomization where appropriate.

6.2.5.1 Study 1: Authoring Revisions

In the domain of word processing, Wojahn [254] found that the functionality provided by a revision interface influenced the number and type of problems discussed. Do users revise interaction designs differently with a structured, interactive tool than by making freeform, static annotations on a diagram?

METHOD

We recruited twelve participants. Participants each completed two revision tasks: one without d.note and one with. The non-d.note condition was always assigned first to prevent exposure to d.note notation from influencing freeform annotation patterns. Each revision task asked participants to critique one of two information appliance prototypes, one for a keychain photo viewer, and one for the navigation and management of images on a digital still camera (Figure 6.22). The tasks were inspired by student exercises in Sharp et al.'s interaction design textbook [226]. We counterbalanced task assignment to the conditions.

Participants were seated in front of a Mac OS X workstation with an interactive 21", 1600×1200 pixel tablet display (Figure 6.23). Participants could control this workstation with stylus as well as keyboard and mouse. We first demonstrated d.tools to participants and had them complete a warm-up menu navigation design (taken from the d.tools evaluation in Section 4.1.5.1) to become familiar with the visual authoring language. In the condition with d.note, students were given a demonstration of its revision features, and five minutes to become familiar with the commands using the warm-up project they completed earlier.

Figure 6.22: Participants were given a prototype device with a color display and button input. They were asked to revise designs for a keychain display and a digital camera, both running on the provided device.

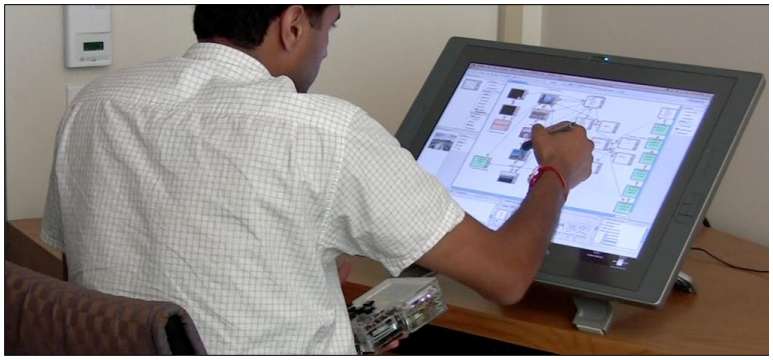

Figure 6.23: Participants in study 1 revised d.tools designs on a large tablet display.

Participants were then given a working prototype, run by d.tools and d.note, and were asked to take 15 minutes to revise the prototype directly in the application using d.note's commenting and revision features.

In the non-d.note condition, participants were given a working prototype along with a static image of the d.tools state diagram for the prototype. The image was loaded in Alias Sketchbook Pro [30], a tablet PC drawing application, and participants were given 15 minutes to draw modifications and comments on top of that image.

The caveat of our design is that ordering of conditions may have affected usage. For example, participants may have become more comfortable, or more fatigued, for the second condition. However, we judged this risk to be lower than the potential learning effect of becoming familiar with the d.note annotation language and then applying it in the non-d.note

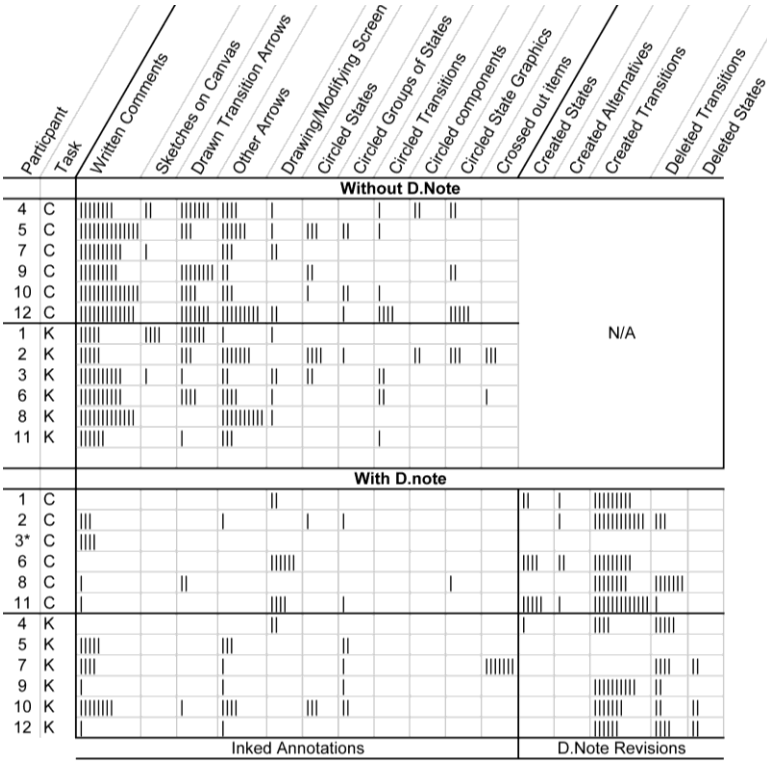

Table 6.1: Content analysis of d.tools diagrams reveals different revision patterns: with d.note, participants wrote less and deleted more.

Perceived advantages of d.note for expressing revisions		Perceived disadvantages of d.note for expressing revisions	
Can test proposed changes	|||||||	Commenting is more difficult	|||
Can make functional changes	||||||	Steeper learning curve	||
Less cluttered than drawing	||	Danger of getting stuck on details	||
Notation easier to interpret	||	Lack of rich drawing tools		
Can express alternatives			Diagramms become too cluttered		

Table 6.2: Most frequently mentioned advantages and disadvantages of using d.note to *express* revisions.

condition. After the design reviews, participants completed a survey that elicited high-level summative feedback in free response format.

RESULTS

We categorized all marks participants made; Table 6.1 summarizes the results. Figure 6.24 shows four examples of diagrams; two for each condition. Most notably, participants wrote significantly more text comments without d.note than with it. In contrast, deletions were rare without d.note (4 occurrences); but common with d.note (34 occurrences; 8 out of 12 participants). Finally, revisions with d.note focused on changes to the information

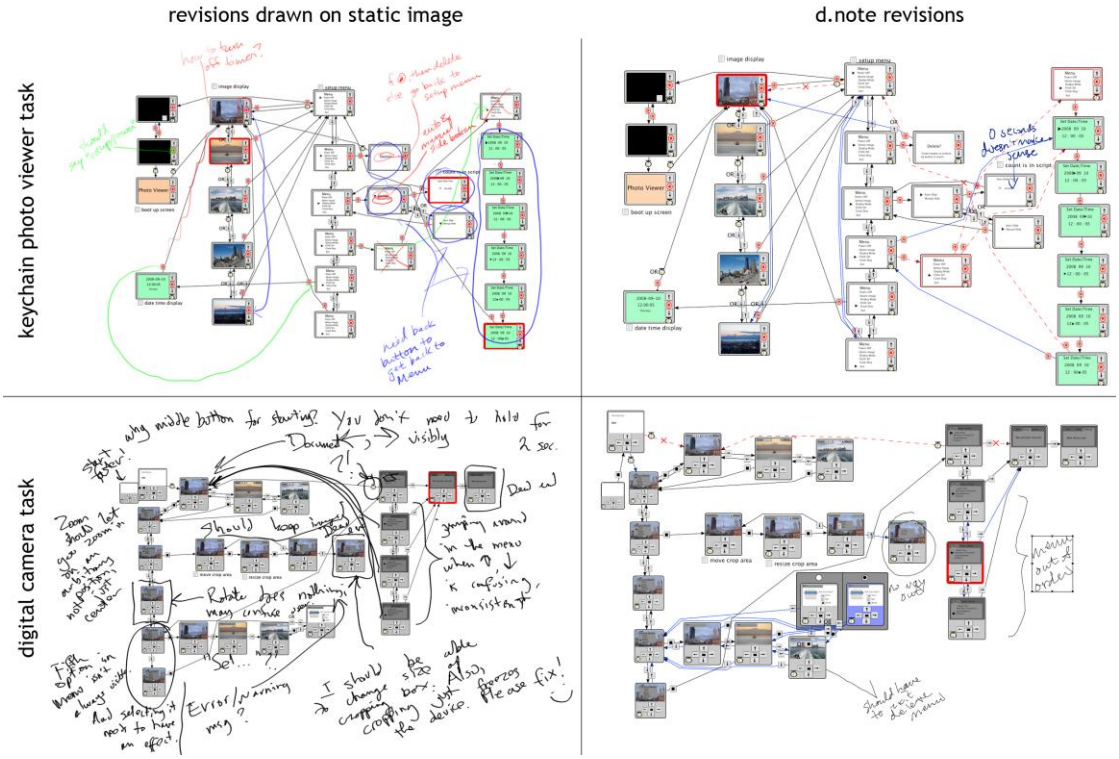

Figure 6.24: Two pairs of revision diagrams produced by our study participants. Diagrams produced with Sketchbook Pro in the control condition are shown on the left; diagrams produced with d.note are shown on the right.

architecture, while freeform revisions often critiqued the prototype on a more abstract level. Our results thus corroborate Wojahn's finding that the choice of revision tool affects the number and type of revision actions [254].

The post-test survey asked participants to compare the relative merits of Sketchbook and d.note. We categorized their freeform written answers (Table 6.2). The two most frequently cited advantages of d.note were the ability to make functional changes (6 of 12 participants), and to then test proposed changes right away (7 of 12 participants). Three participants suggested that commenting was more difficult with d.note; two wrote that the tool had a steeper learning curve. Two participants with a product design background wrote that using d.note led them to focus too much on the details of the design. In their view, the lack of functionality in the Sketchbook condition encouraged more holistic thinking.

DISCUSSION

Why did participants write less with d.note? One possibility is that that users wrote more with Sketchbook because it was easier to do so (Sketchbook is a polished product, d.note a research prototype). To the extent this is true, it provides impetus to refine the d.note

implementation, but tells us little about the relative efficacy of a static and dynamic approach to design revision.

More fundamentally, d.note may enable users to capture intended changes in a more succinct form than text comments. Four participants explicitly wrote that d.note reduced the need for long, explanatory text comments in their survey responses: "[with d.note] making a new state is a lot shorter than writing a comment explaining a new state"; "[without d.note] I felt I had to explain my sketches." d.note's rich semantics enable a user's input to be more economical: an added or deleted transition is unambiguously visualized as such. In d.note, users can implement concrete changes interactively; only abstract or complex changes require comments. Without d.note, both these functions have to be performed through the same notation (drawing), and participants explained their graphic marks with additional text because of the ambiguity. In our data, inked transition arrows drawn without d.note (44 drawn transitions) were replaced with functional transitions with d.note (78 functional transitions added; only 3 drawn as comments).

Though participants could have disregarded the revision tools and only commented with ink, the mere option of having functional revision tools available had an effect on their activity. This tendency has been noted in other work [55,156] as well.

Why did participants delete more with d.note? While participants created new states and transitions in both conditions, deletions were rare without d.note. Deletions may have been implied, e.g., drawing a new transition to replace a previously existing one, but these substitutions were rarely noted explicitly. We suggest that deletions with d.note were encouraged by the ability to immediately test concrete changes. Quick revise-test cycles exposed areas in which diagrams had ambiguous control structure (more than one transition exiting a state on the same event).

Why were more changes to information architecture made with d.note? The majority of revision actions with d.note concerned the flow of control: adding and deleting transitions and states. In the Sketchbook condition, participants also revised the information architecture, but frequently focused on more abstract changes (Example comment: "Make [feedback] messages more apparent"). The scarcity of such comments with d.note is somewhat surprising, as freeform commenting was equally available. One possible explanation is that participants focused on revising information architecture because more powerful techniques were at hand to do so. Each tool embodies a preferred method of use; even if other styles of work remain possible, users are driven to favor the style for which the tool offers the most leverage.

6.2.5.2 Study 2: Interpreting Revisions

The first study uncovered differences in expressing revisions. Are there similar characteristic differences in interpreting revisions created with the two tools?

METHOD

Eight (different) participants interpreted the revisions created by participants of the first study. After a demonstration and warm-up task (as in study 1), participants were shown the two working prototypes (camera and key chain) and given time to explore. Next, participants were shown screenshots of annotated diagrams from the first study (Figure 6.24) on a second display. Participants were asked to prepare two lists in a word processor: one that enumerated all revision suggestions that were clear and understandable to them; and a second list with questions for clarification about suggestions they did not understand. Participants completed this task four times: one d.note and one freeform diagram were chosen at random for each of the two prototypes.

RESULTS

The cumulative count of clear and unclear revision suggestions for all participants are shown in Table 6.3. Participants, on average, requested 1.3 fewer clarifications on revisions when using d.note than when sketching on static images (two-sample $t(29)=1.90$, $p=0.03$).

The post-test survey asked participants to compare the relative merits of interpreting diagrams revised with d.note and Sketchbook. The most frequently mentioned benefits arose from having a notation with specified semantics (Table 6.4): revisions were more concrete, specific, and actionable. Frequently mentioned drawbacks were visual complexity and problems discerning high-level motivation in d.note diagrams.

DISCUSSION

Why did participants ask for fewer clarifications with d.note? When interpreting revised diagrams, participants are faced with three questions: First, what is the proposed change? Second, why was this change proposed? Third, how would I realize that change? The structure of the second user study asked participants to explicitly answer the first question by transcribing all proposed changes. We suggest that the formal notation in d.note decreased the need for clarification for two reasons. First, the presence of a formal notation resulted in a smaller number of handwritten comments, and hence fewer problems with legibility (Example without d.note: "Change 6 — unreadable"). Second, because of the ad-hoc nature of handwritten annotation schemes in absence of a formal system, even if comments were

S2 Participant	Task	S1 Participant	Clear Annotations	Unclear Annotations	S1 Participant	Clear Annotations	Unclear
		With D.Note			Without D.Note		
1	C	6	IIIIII	I	7	IIIIIII	
	K	4	IIII		3	IIIII	IIIIII
2	C	8	IIII	III	9	IIIIIIII	I
	K	12	IIIIII		2	IIIIII	IIII
3	C	11	IIIIIIIII	IIIIII	4	IIIIIIII	IIIII
	K	9	IIIII	I	8	IIIIIIIII	IIIIII
4	C	2	IIIIIIIIIIIIIII	I	10	IIIIIIIIII	II
	K	10	IIIIIIIII		11	IIII	II
5	C	1	IIII	I	5	IIIIIII	III
	K	4	III	I	6	IIIII	I
6	C	11	IIIIII	III	12	IIIIIIII	IIII
	K	10	IIII	IIIII	5	IIIII	II
7	C	6	IIIIII	I	7	IIIIIIII	
	K	12	IIIIIII		3	IIIII	IIIIII
8	C	2	IIIIIIIIIIIIII	II	4	IIIIII	IIII
	K	9	IIII	I	8	IIIIIIIII	I

Table 6.3: How well could study 2 participants interpret the revisions created by others? Each vertical bar is one instance.

Perceived advantages of d.note for interpreting revisions	
Changes are concrete and specific	IIII
Contains proposed solutions	III
Can automatically apply changes	I

Perceived disadvantages of d.note for interpreting revisions	
Visual clutter in regions of dense changes	IIIII
Hard to glean motivation for changes	IIII
Hard to keep track which changes were already examined	II

Table 6.4: Perceived advantages and disadvantages of using d.note to *interpret* revisions as reported by study participants.

legible, participants frequently had trouble tying the comments to concrete items in the interface (Example: "I have no idea what it means to 'make it clear that there is a manual mode from the hierarchy'. What particular hierarchy are we talking about?")

In the survey, participants commented on the remaining questions of why changes were proposed and how one might implement those changes. We next discuss mitigation strategies for managing visual complexity and the reported problems discerning high-level motivation in d.note diagrams.

Visual complexity of annotated diagrams: Visual programs become harder to read as the node & link density increases. Showing added and deleted states and transitions simultaneously in the diagram sometimes yielded "visual spaghetti": a high density of transition lines made

distinguishing and following individual lines hard. The connection density problem becomes worse when state alternatives are introduced because each alternative for a state has an independent set of outbound transitions.

In response, we already modified the drawing algorithm for state alternatives to only show outgoing connections for the currently active alternative within an alternative container. Additional simplification techniques are needed though. One option to selectively lower transition density in the diagram while preserving relevant context would be to only render direct incoming and outgoing transitions for a highlighted state and hide all other transitions on demand.

Capturing the motivation for changes: While many handwritten comments focused on high-level goals without specifying implementations, tracked changes make the opposite tradeoff: the implementation is obvious since it is already specified, but the motivation behind the change can remain opaque. We see two possible avenues to address this challenge. First, when using change tracking, multiple individual changes may be semantically related. For example, deleting one state and adding a new state in its stead are two actions that express a desired single intent of replacement. The authoring tool should detect such related actions automatically or at least enable users to specify groups of related changes manually. Second, even though freeform commenting was available in d.note, it was not used frequently. Therefore, techniques that proactively encourage users to capture the rationale for changes may be useful.

6.2.6 LIMITATIONS & EXTENSIONS

The d.note project introduced a notation and interaction techniques for managing revisions of user interface designs expressed as state diagrams. Diagrams can be modified and annotated. The particular implementation of revision techniques in d.note represents only one point solution in a larger design space of possible user interface revision tools. The main salient dimensions we considered during our work are summarized in Figure 6.25. This table reveals limitations and additional areas of exploration we have not touched upon so far.

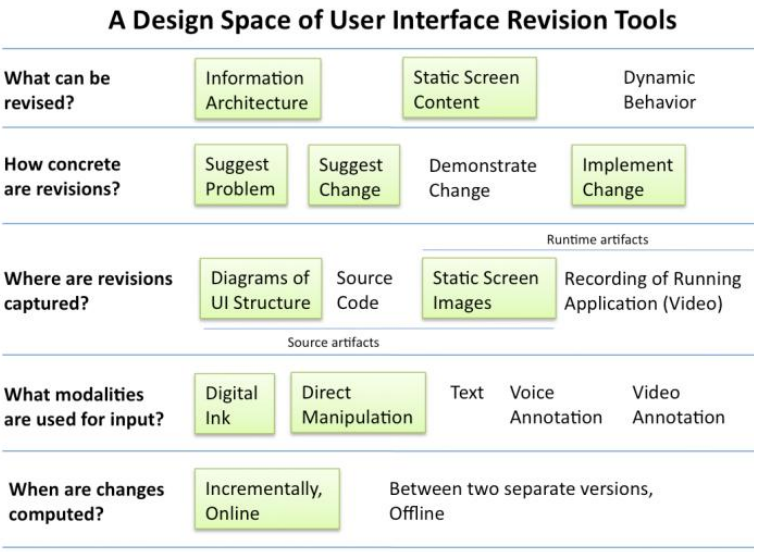

Figure 6.25: A design space of user interface revision tools. The sub-space d.note explored is highlighted in green.

6.2.6.1 Cannot Comment on Dynamic Behavior

The stylus-driven annotation makes it easy to add comments to both layout and information architecture. It is not feasible to efficiently comment on dynamic behaviors, as there is no visual record of these behaviors in the interaction diagram. Recording and annotating video of an application's runtime output is one promising avenue to enable comments on behavior. d.tools can already record live video of interaction with a built prototype. If this video capture were augmented with a second stream of screen captures, then designers could sketch directly onto those video frames. To make such sketches useful for others, they have to be retrievable from the editing environment. Future work should examine how to associate such video annotations with the state diagrams and other static source views.

6.2.6.2 Cannot Revise Dynamic Behavior

d.note currently enables designers to express functional changes to the information architecture of the user interface, and to the screen content of a given state within that larger architecture. However, changes to scripts are not well supported in that there are no visualizations to show in detail what has changed, and no interaction techniques to accept or undo such changes.

6.2.6.3 How To Support Identified Revision Principles for Source Code?

The presented design space finally raises the question how one might offer the benefits of a revision tool such as d.note for user interfaces specified entirely in source code. The particular revision techniques of d.note are based on a visual language that shows both user interface content and information architecture in the same environment. The techniques should therefore transfer to other visual control-flow tools such as DENIM [171] or SUEDE [148]. But what about user interfaces that are not programmed visually? Existing source revision techniques for non-visual programs do not permit designers to comment or revise the output of their application. Future research should investigate if sketch-based input and annotation in the output domain of a program can be transferred to such applications.

CHAPTER 7 CONCLUSIONS AND FUTURE WORK

This dissertation has shown how to support creation, exploration, and iteration of user interface prototypes for ubiquitous computing applications. This final chapter recapitulates the contributions made by the presented systems, and concludes with an outlook on future work.

7.1 RESTATEMENT OF CONTRIBUTIONS

We introduced principles and systems for prototyping user interfaces that span physical and digital interactions. Three areas of technical contributions can be distinguished:

1) Techniques for authoring user interfaces with non-traditional input/output configurations. This dissertation contributed:
    a. Rapid authoring of interaction logic through a *novel combination of storyboard diagrams* for information architecture *with imperative programming* for interactive behaviors.
    b. *Demonstration-based definition of discrete input events from continuous sensor data streams* enabled by a combination of pattern recognition with a direct manipulation interface for the generalization criteria of the recognition algorithms.
    c. *Management of input/output component configurations for interface prototypes* through an editable virtual representation of the physical device being built. This representation reduces cognitive friction by collapsing levels of abstraction; it is enabled by a custom hardware interface with a plug-and-play component architecture.
2) Principles and techniques for exploring multiple user interface alternatives. The dissertation contributed:
    a. *Techniques for efficiently defining and managing multiple alternatives of user interfaces* in procedural source code and visual control flow diagrams.
    b. *User-directed generation of control interfaces* to modify relevant variables of user interfaces at runtime.

c. *Support for sequential and parallel comparison of user interface alternatives* through parallel execution, selectively parallel user input, and management of parameter configurations across executions.

d. *Implementations of the runtime techniques for three different platforms*: desktop PCs, mobile phones, and microcontrollers.

3) Techniques for capturing feedback from users and design team members on user interface prototypes, and integrating that feedback into the design environment. The dissertation contributed:

   a. *Timestamp correlation between live video, software states, and input events* during a usability test of a prototype to enable rapid semantic access of video during later analysis.

   b. *Novel video query techniques*: *query by state selection* where users access video segments by selecting states in a visual storyboard; and *query by input demonstration* where sections of usability video are retrieved through demonstrating, on a physical device prototype, the kind of input that should occur in the video.

   c. *A visual notation and stylus-controlled gestural command set for revising user interfaces* expressed as control flow diagrams.

The dissertation also provided evidence, through laboratory studies and class deployments, that the introduced techniques are successful. In particular, the dissertation contributed:

1) Evidence that the introduced authoring methods for sensor-based interaction are accessible and expressive through two laboratory evaluations and two class deployments.
2) Evidence from a laboratory study that the techniques for managing interface alternatives enable designers to explore a wider range of design options, faster.
3) Evidence from two laboratory studies that an interactive revision notation for interfaces leads to more concrete and actionable revisions.

7.2 FUTURE WORK

Future work in the space of design tools outlined by this dissertation falls into two general categories. First, additional research can extend the introduced systems and techniques, to overcome present limitations or to take logical next steps that enhance expressivity and utility. Second, reconsidering the assumptions underlying the systems described in this thesis

yields additional opportunities for different types of tools that can support a broader range of authoring tasks. Important limitations and possible extensions were discussed at the conclusion of each preceding chapter, in Sections 4.1.7 (d.tools, p. 92), 4.2.6 (Exemplar, p. 116), 5.6 (Juxtapose, p. 136), 6.1.4 (d.tools video analysis, p. 149), and 6.2.6 (d.note, p. 167). This chapter briefly discusses some larger future research directions.

In retrospect, most of the work presented in this dissertation implicitly shares a set of assumptions: that *an individual designer* creates one or more alternative designs for *a single device*, starting *from scratch*, through a *desktop-based graphical user interface tool*. Changing any of these four core assumptions yields areas of future work that suggest different types of design tools. We review each of these four areas in turn.

7.2.1 DESIGN TOOLS THAT SUPPORT COLLABORATION

Most existing authoring tools for user interfaces, the ones proposed in this dissertation included, focus on the work of a single creative individual. Future research should broaden this scope to integrate support for collaboration and sharing directly into authoring environments. Two reasons for making such a shift are the predominance of team-based design in industry, and the rise of open, amateur design communities online.

PROFESSIONAL DESIGN TAKES PLACE IN TEAMS

Professional work on complex user interfaces takes place in design teams; and an increasing number of such teams are geographically distributed. Office suite applications such as word processors and spreadsheets now routinely offer support for asynchronous review and annotation; some web-based applications also support synchronous collaborative editing. Outside the realm of office applications, support for distributed work is still lacking. In this dissertation, the d.note project for revising interaction design diagrams considered the importance of asynchronous communication about such diagrams between team members. But the presented work has not yet addressed synchronous collaboration. How can technology help teams jointly construct, discuss, and test user interface prototypes? In this chapter, Section 7.2.3.1 proposes a concrete project to redesign the interaction design studio itself to better support team activities.

SUPPORTING AMATEUR DESIGN COMMUNITIES

Beyond the professional, corporate context, social production of both information and software is becoming increasingly important. Successful online environments for collaborative information production (e.g., Wikipedia, 'view source' on Web 1.0 HTML

pages) are built around open access to modify, copy, and reuse content. For interaction design beyond HTML pages, and programming in general, most social exchanges today happen outside the authoring environments, through plain text in online forums and blogs. We believe that there is significant additional latent value in integrating collaborative aspects of design and development directly into our authoring tools, where richer ways for collecting, presenting, and interacting with authored media are available.

As a first step, some programming IDEs have begun to integrate support for publishing projects online. Scratch [13], the multimedia programming environment for children developed at the MIT Media Lab, has a function to share one's program on the Scratch website. Resnick recently reported that 30% of projects on the Scratch website are based on other projects; and that some projects have been "remixed" (copied, modified, and shared again) up to 29 different times [76]. We believe that sharing the authoring process in addition to the end result can significantly aid designers and developers in gaining expertise, integrating pre-existing pieces of functionality into their project, and understanding and correcting problems. The following section on authoring by example modification introduces some concrete research projects along these lines.

7.2.2 AUTHORING BY EXAMPLE MODIFICATION

Most existing authoring tools implicitly assume that creators start with a clean slate, and then create their design, e.g., a user interface, a layout of a brochure, or a personal website, from scratch. However, less design happens *tabula rasa* than one might surmise. In practice, much creative work starts with finding relevant existing examples and modifying those to fit a new context.

Examples play at least two fundamental roles in the design and programming of user interfaces: they can provide *inspiration* by providing anchors for analogical thinking [85], and they can provide *concrete functionality* that can shortcut the time required for implementation. For inspiration, designers like to immerse themselves in the domain of their current project by collecting a large and diverse set of examples [118]. These examples can be competing products, swatches of materials, color schemes (e.g., 'mood boards'), or clever mechanisms (e.g., the IDEO Tech Box [141:pp. 143-145]). In fact, "shopping for functionality" was reported as an important early design activity in our study of interaction, web, and hardware designers [108]. Examples provide an experiential feel for the space of existing solutions and allow identification of desirable traits, both concrete ("knob should be self-centering with detents")

and abstract ("product should feel warm and welcoming"). These traits are then transferred to the product being designed by analogy.

Many designers and programmers also rely on examples to provide working implementations of desired functionality. Integrating existing examples may be faster, more economical, or may enable designers to leverage functionality they could not create themselves. In the software domain, *programming by example modification* [196] is especially useful for learning how to integrate existing libraries into one's own project. Brandt et al. found programming by example modification to be pervasive [50]. In a lab study where subjects had to implement a chat room application, all participants extensively copied code found on web sites: 1/3 of the final code in participants' projects came from pre-existing examples..

If use of examples is pervasive in design and programming, what are the implications for future design tools? We see four aspects deserving of future work: New tools can help users *find* relevant examples, *synthesize* new examples if none exist, *extract* examples from larger projects, and *facilitate integration* of found examples into projects. The following sections review three of these areas in some additional detail.

7.2.2.1 Finding Examples

For programmers, code search engines like Assieme [122] and Mica [235] provide support for finding relevant source examples. Brandt's Blueprint system integrates search for example code snippets directly into the Adobe Flex development environment [49]. Going beyond source code, it is not immediately clear how searches for examples should be specified. For visual material, hierarchical browsing interfaces [181], faceted metadata browsing [257], and image search by sketching [223] have been proposed, but we are not aware of studies about the efficacy of such techniques for design. It is even less clear how designers might search for interactive behaviors.

7.2.2.2 Synthesizing Examples

Our d.mix project [113] explored how to automatically synthesize new examples of web service API calls by enabling developers to point to elements on web pages that they would like to access programmatically. The Design Galleries system [181] generates a space-spanning set of examples based on algorithms evaluating alternatives. For any system that automatically generates examples, designers somehow have to steer and control the synthesis

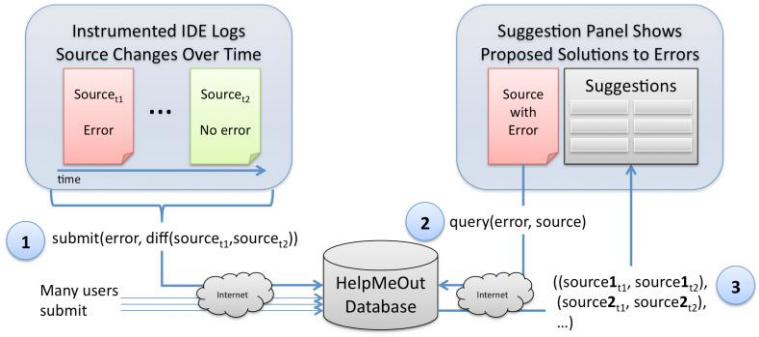

Figure 7.1: HelpMeOut offers asynchronous collaboration to suggest corrections to programming errors. 1: IDE instrumentation extracts bug fixes from programming sessions to a remote database. 2: Other programmers query the database when they encounter errors. 3: Suggested fixes are shown inside their IDE.

process. Whether the right interfaces for doing so can be general or have to be domain-specific remains to be determined.

7.2.2.3 Extracting Examples

Useful examples for programmers are short, minimal, self-sufficient, and have explanatory documentation. These attributes are entirely different from the characteristic of source code found in open repositories, where projects are large, complex, and rife with interdependencies. One possible area of future research is therefore how to give developers the right tools to make it easier (or automatic) to publish relevant, small examples from their larger codebases.

Our recently started HelpMeOut project [110] suggests that IDE instrumentation can be used to automatically collect histories of problems and problem fixes during programming sessions. A database of such fixes can then be used as a source of examples for other programmers who are experiencing similar problems (Figure 7.1).

7.2.2.4 Integrating Examples

Once relevant examples have been found, how can designers integrate parts of those examples into their projects? The Adaptive iDEAS project [158] introduced limited support for copying font and color attributes of web pages these exemplars into one's own page designs. Kumar and Kim [152] are expanding on the motivation of this work by enabling designers to reuse the layout structure of existing web pages, but substitute one's own content.

Kelleher's Looking Glass project [138] aims to enable users of the Alice virtual world authoring system to "steal" behaviors from other users. Since these behaviors are represented in code in a structured editor, key challenges are how to find the right scope of code to copy, and how to adapt the found code to fit, e.g., by remapping object identifiers. How to aid similar integration for arbitrary code remains an open question. It would also be valuable to have a more concrete understanding which kind of examples are most frequently consulted and appropriated for different kinds of design projects by studying example use in real-world scenarios.

7.2.3 AUTHORING OFF THE DESKTOP

The tools introduced in this dissertation focused predominantly on prototyping user interfaces that aim beyond the standard desktop paradigm. However, the tools proposed for doing so were desktop applications themselves. What benefits can be realized by moving the authoring environment off the desktop? We propose two possible research directions: *going large* to create new design studio spaces, and *going small* by researching authoring tools for mobile computing devices.

7.2.3.1 Going Large: New Studio Spaces for Interaction Design

As noted in section 7.2.1, professional design is a team activity. Creative work alternates between phases of individual production and team discussion, ideation, and review. Based on insight into design team work patterns, what should the computing infrastructure in the interaction design studio of the future look like? To what extent can designers benefit from interactive spaces that are tailored to their design process, as opposed to generic meeting support? Three different "form factors" have been proposed in prior work to support team collaboration: large interactive wall displays, interactive tables, and entire augmented rooms that combine interactive walls, tables, and other computing devices.

WALLS

A number of prior systems have focused on supporting design teams with interactive display walls. Notable systems include PostBrainstorm [98], a large high-resolution projected mural for enhancing and capturing brainstorming sessions, TEAM STORM [101] a brainstorm support system that uses individual tablet displays and a shared wall display; and the Designers' Outpost [147], a wall display that integrates digital capture and projection, and physical post-it notes to create information architecture diagrams for web sites. While wall

Figure 7.2: The Pictionaire table supports co-located design team work through multi-touch, multi-device input and overhead image capture.

displays offer the benefit of a shared focal point, arm fatigue limits their use for extended work sessions [95:p. 1322].

TABLES

Interactive tables have unique benefits that may make them suitable for interaction design and product design. Discussion in these domains is almost always tied to concrete artifacts: designers use sketches, photographs, physical prototypes, and other products to structure conversation and creativity. As a result, design meetings, whether they focus on planning, brainstorming, or reviewing, draw on a wide variety of "stuff." Creative thought moves freely across digital and physical boundaries. We hypothesize that interactive tables are particularly suited to support and enhance group design work when they enable co-habitation of digital and physical artifacts on the table surface. In our own recent work, we have developed Pictionaire (Figure 7.2), a large, multi-user, standing height interactive table that supports physical to digital transition techniques through overhead image capture [111]. Pictionaire was expressly created for team meetings of user interface designers; its software supports the creation of linear interface walkthroughs from sketches and photographs. The next logical step is to move beyond sketching straightforward walkthroughs into higher-fidelity prototyping of interfaces on the table.

There are additional reasons for moving away from desktop UIs, even for individual design work: in the domain of 3D modeling and animation, repetitive strain injuries (RSI) are a serious problem for professional artists. Research on leveraging multi-touch authoring techniques for animation professionals, e.g., at Pixar, is ongoing [142]. Large interactive tables

that offer high-resolution pen-input for digital drawing are also an active area of research [102].

To truly gauge the potential of table form factors and to find the right fit with professional practice, longer deployments outside research labs are needed. It would therefore be valuable to study use of a large interactive table such as our Pictionaire system with a local professional design company.

ROOMWARE

Streitz' iLand [234] and the Stanford iRoom [134] investigated how collections of many different computing form factors can support team work in a single room. The results, at least for the Stanford iRoom, have been mixed. Distinct benefits of a room-scale infrastructure include the ability to migrate applications between multiple displays and retarget interaction based on the best available input device at the time. However, the complexity of room-scale systems also creates maintenance and challenges that may outweigh the offered benefits. It is telling that one particular interactive wall display was replaced with a traditional, non-interactive whiteboard after it fell into disuse. The experience with roomware then should serve as reminder not to blindly accept a vision of an all-digital future. More realistically, future research will have to find solutions that tread a careful line between keeping appropriate physical processes physical while adding digital flexibility where it is beneficial.

7.2.3.2 Going Small: Authoring on Handheld Devices

As a counterpoint to large, complex team design environments, we may also ask what kind of authoring is possible on very small devices such as smart phones or PDAs. This question is reasonable to consider because of two trends:

1) At the cutting edge of technology, smart phones today offer the processing power found on desktop computers only a few years ago. Video and still image capture, location sensing, and 3D graphics acceleration are becoming common place. The latest version for Apple's iPhone now includes an application for video cutting and editing on the phone.
2) On the other end of the spectrum, for the majority of the world's population, access to computation happens through cheap, low-powered cell phones. The mobile phone may be the only computing device millions of people will ever have access to.

These two trends raise the following research questions: Fundamentally, what kind of content will users *want* to author on mobile devices in the future? What kind of content *can* be authored on such devices? The technical challenges are plentiful. The limited input/output

affordances of mobile devices are an immediate, obvious hurdle. While mobile authoring is unlikely to replace the desktop paradigm, these questions are deserving of future study.

7.2.4 DESIGNING DEVICE ECOLOGIES

d.tools, Exemplar, and Juxtapose all assumed that a single, standalone device or software interface was being designed. Increasingly, this assumption is no longer sufficient, as a growing number of smart products offer their value through device or application ecologies with multiple, connected components. An important, if overused, example of such an ecology is the Apple iPod + iTunes system. The iPod is a portable digital music player; iTunes is an application to play and manage one's digital media library on a desktop computer, linked to an online store for browsing and purchasing new music. The overall user experience arises out of the tight integration between the components. As another example, personal fitness devices such as heart rate monitors are starting to include web interfaces for analyzing and sharing the collected data [224].

Sensor networks — collections of small, programmable, self-powered computing nodes that communicate with each other over ad-hoc wireless networks, are another area where behavior for multiple interconnected components has to be authored. While early sensor networks were used for unattended data collection, for example in conflict areas or for environmental monitoring, future applications, e.g., controlling energy usage in smart buildings, will likely require end-user interfaces. Merrill's Siftables project [187] explicitly realizes the potential of sensor networks as user interfaces. Each node has a small color display and can sense neighboring nodes as well as acceleration. While existing research has introduced hardware and software tools for programming sensor network applications (e.g., tinyOS [160]), and multi-display applications (e.g., Vigo [151]), such tools are aimed at researchers and technology experts. Support for prototyping and end-user authoring of multi-display or multi-device applications is still lacking and worthy of future research.

7.3 CLOSING REMARKS

The desktop computing paradigm has largely ossified around a common set of input devices and interaction techniques. With the rise of mobile and ubiquitous computing, it has also already eclipsed its zenith. While desktop computing still has an important role to play, a wider variety of different computing devices are quickly populating our lives. Beyond bringing new technologies for novel interfaces within the reach of interaction designers, this dissertation advocated that tools should also explicitly support fundamental design process steps. By encouraging exploration of alternatives, informed by feedback, design tools can help designers create interfaces that truly fit their intended users, contexts, and tasks, while being delightful to use. The research presented in this dissertation empowers designers to better envision and realize a broader range of such alternative futures for the post-desktop computing age.

REFERENCES

1. *Flash*. Adobe Systems. http://www.adobe.com/products/flash/
2. *BASIC Stamp*. Parallax. http://www.parallax.com/
3. IxDA Discussion Listserv. http://www.ixda.org/discuss.php
4. *Silverback Usability Testing Software*. Clearleft Ltd. http://silverbackapp.com/
5. *Morae User Testing Software*. TechSmith Corporation. http://www.techsmith.com/morae.asp
6. *Director*. Adobe Systems. http://www.adobe.com/products/director/
7. *Cybelius Maestro*. Nickom Ltd. http://cybelius.com/
8. *Flash Catalyst*. Adobe Systems. http://labs.adobe.com/technologies/flashcatalyst/
9. *Quartz Composer*. Apple Computers. http://developer.apple.com/graphicsimaging/quartz/quartzcomposer.html
10. *Mindstorms*. The LEGO Group. http://mindstorms.lego.com/
11. *BUG: modular, open source hardware*. Bug Labs. http://www.buglabs.net/
12. *UML - Unified Modeling Language*. Object Management Group. http://www.uml.org/
13. *Scratch*. MIT Media Lab Lifelong Kindergarten Group. http://scratch.mit.edu/
14. *Rational Rose*. IBM. http://www-01.ibm.com/software/awdtools/developer/technical/
15. *LabVIEW*. National Instruments. http://www.ni.com/labview/
16. *Simulink*. The MathWorks. http://www.mathworks.com/products/simulink/
17. *VEE Pro*. Agilent Technologies. http://agilent.com/find/vee
18. *Max/MSP*. Cycling74. http://www.cycling74.com/products/max5
19. *Pipes*. Yahoo. http://pipes.yahoo.com/pipes/
20. Pixel Bender. *Adobe Systems*. http://labs.adobe.com/technologies/pixelbender/
21. *Eclipse IDE*. Eclipse Foundation. http://www.eclipse.org/
22. *ATmega128 8-bit Microcontroller*. Atmel Corporation. http://www.atmel.com/dyn/products/product_card.asp?PN=ATmega128
23. *ATtiny45 8-bit Microrontroller*. Atmel Corporation. http://www.atmel.com/dyn/products/product_card.asp?PN=ATtiny45
24. *Eclipse Graphical Editing Framework (GEF)*. Eclipse Foundation. http://eclipse.org/gef/
25. *SWT: The Standard Widget Toolkit*. Eclipse Foundation. http://www.eclipse.org/swt/
26. *VST - Virtual Studio Technology*. Steinberg Media Technologies GmbH. http://ygrabit.steinberg.de/~ygrabit/public_html/index.html
27. *MTASC Actionscript 2 Compiler*. Motion-Twin Technologies. http://mtasc.org/
28. *EMFlash*. Markelsoft. http://www.markelsoft.com/products/emflash/
29. *BCF2000 MIDI Controller*. Behringer. http://www.behringer.com/EN/Products/BCF2000.aspx
30. *SketchBook Pro*. Autodesk. http://www.autodesk.com/sketchbookpro

31. *The American Heritage Dictionary of the English Language*. Houghton Mifflin Company, 2000.
32. *I2C-bus specification and user manual Rev 03*. NXP Semiconductors, 2007. http://www.nxp.com/acrobat/usermanuals/UM10204_3.pdf
33. Aigner, W., Miksch, S., Muller, W., Schumann, H., and Tominski, C. Visual Methods for Analyzing Time-Oriented Data. *Visualization and Computer Graphics, IEEE Transactions on 14*, 1 (2008), 47-60.
34. Akers, D., Simpson, M., Jeffries, R., and Winograd, T. Undo and erase events as indicators of usability problems. *Proceedings of the 27th international conference on Human factors in computing systems*, ACM (2009), 659-668.
35. Andreae, P.M. Justified Generalization: Acquiring Procedures from Examples. 1985. http://dspace.mit.edu/handle/1721.1/6950
36. Atkinson, B. *Hypercard*. Apple Computers.
37. Avrahami, D. and Hudson, S.E. Forming interactivity: a tool for rapid prototyping of physical interactive products. *Proceedings of the 4th conference on Designing interactive systems: processes, practices, methods, and techniques*, ACM (2002), 141-146.
38. Badre, A.N., Guzdial, M., Hudson, S.E., and Santos, P.J. A user interface evaluation environment using synchronized video, visualizations and event trace data. *Software Quality Journal 4*, 2 (1995), 101-113.
39. Badre, A.N., Hudson, S.E., and Santos, P.J. Sychronizing video and event logs for usability studies. *Proceedings of the workshop on Advanced visual interfaces*, ACM (1994), 222-224.
40. Bailey, B.P., Konstan, J.A., and Carlis, J.V. DEMAIS: designing multimedia applications with interactive storyboards. *Proceedings of the ninth ACM international conference on Multimedia*, ACM (2001), 241-250.
41. Baldonado, M.Q.W., Woodruff, A., and Kuchinsky, A. Guidelines for using multiple views in information visualization. *Proceedings of the working conference on Advanced visual interfaces*, ACM (2000), 110-119.
42. Ballagas, R., Memon, F., Reiners, R., and Borchers, J. iStuff mobile: rapidly prototyping new mobile phone interfaces for ubiquitous computing. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2007), 1107-1116.
43. Ballagas, R., Ringel, M., Stone, M., and Borchers, J. iStuff: a physical user interface toolkit for ubiquitous computing environments. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2003), 537-544.
44. Ballagas, R., Szybalski, A., and Fox, A. Patch Panel: Enabling Control-Flow Interoperability in Ubicomp Environments. *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, IEEE Computer Society (2004), 241.
45. Barragán, H. *Wiring: Prototyping Physical Interaction Design. Unpublished Master's Thesis.* Interaction Design Institute Ivrea, 2004.
46. Bellotti, V., Back, M., Edwards, W.K., Grinter, R.E., Henderson, A., and Lopes, C. Making sense of sensing systems: five questions for designers and researchers. *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, ACM (2002), 415-422.

47. Blackwell, A. and Green, T.R.G. A Cognitive Dimensions Questionnaire. 2000. http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDquestionnaire.pdf
48. Böhm, C. and Jacopini, G. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM 9*, 5 (1966), 366-371.
49. Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S.R. *Example-Centric Programming: Integrating Web Search into the Development Environment. Technical Report CSTR 2009-01.* Stanford University Computer Science Department, 2009. http://hci.stanford.edu/cstr/
50. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., and Klemmer, S.R. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. *Proceedings of the 27th international conference on Human factors in computing systems*, ACM (2009), 1589-1598.
51. Brooks, F.P. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley Professional, 1995.
52. Buchenau, M. and Suri, J.F. Experience prototyping. *Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques*, ACM (2000), 424-433.
53. Burnett, M.M., Baker, M.J., Bohus, C., Carlson, P., Yang, S., and Zee, P.V. Scaling Up Visual Programming Languages. *Computer 28*, 3 (1995), 45-54.
54. Burr, B. VACA: a tool for qualitative video analysis. *CHI '06 extended abstracts on Human factors in computing systems*, ACM (2006), 622-627.
55. Buxton, B. *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann, 2007.
56. Buxton, W. Living in Augmented Reality: Ubiquitous Media and Reactive Environments. In K. Finn, A. Sellen and S. Wilber, eds., *Video Mediated Communication*. Erlbaum, Hillsdale, NJ, 1997, 363-384.
57. Card, S.K., Mackinlay, J.D., and Robertson, G.G. A morphological analysis of the design space of input devices. *ACM Transactions on Information Systems (TOIS) 9*, 2 (1991), 99-122.
58. Chi, E.H., Riedl, J., Barry, P., and Konstan, J. Principles for Information Visualization Spreadsheets. *IEEE Computer Graphics and Applications 18*, 4 (1998), 30-38.
59. Clark, A. *Being There*. MIT Press, 1997.
60. Clark, A. *Supersizing the Mind*. Oxford University Press US, 2008.
61. Clark, A. and Chalmers, D. The Extended Mind. *Analysis 58*, 1 (1998), 7-19.
62. Coniglio, M. *Isadora*. TroikaTronix. http://www.troikatronix.com/isadora.html
63. Cooper, A. *Visual Basic*. Microsoft Corporation.
64. Cooper, A. *The Inmates Are Running the Asylum*. Sams Publishing, 1999.
65. Crider, M., Bergner, S., Smyth, T.N., et al. A mixing board interface for graphics and visualization applications. *Proceedings of Graphics Interface 2007*, ACM (2007), 87-94.
66. Cross, N. *Designerly Ways of Knowing*. Birkhäuser Basel, 2007.
67. Cypher, A., ed. *Watch what I do: Programming by Demonstration*. MIT Press, 1993.
68. Davis, R.C., Colwell, B., and Landay, J.A. K-sketch: a 'kinetic' sketch pad for novice animators. *Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, ACM (2008), 413-422.

69. Davis, R.C., Saponas, T.S., Shilman, M., and Landay, J.A. SketchWizard: Wizard of Oz prototyping of pen-based user interfaces. *Proceedings of the 20th annual ACM symposium on User interface software and technology*, ACM (2007), 119-128.
70. Dey, A.K., Hamid, R., Beckmann, C., Li, I., and Hsu, D. a CAPpella: programming by demonstration of context-aware applications. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2004), 33-40.
71. Dobing, B. and Parsons, J. How UML is used. *Communications of the ACM 49*, 5 (2006), 109-113.
72. Dow, S., Heddleston, K., and Klemmer, S.R. The Efficacy of Prototyping Under Time Constraints. *Proceedings of Creativity & Cognition 2009*, ACM (2009).
73. Dreyfuss, H. *Designing for people*. Simon and Schuster, 1955.
74. Drucker, S.M., Petschnigg, G., and Agrawala, M. Comparing and managing multiple versions of slide presentations. ACM (2006), 47-56.
75. Fails, J. and Olsen, D. A design tool for camera-based interaction. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2003), 449-456.
76. Fischer, G., Jennings, P., Maher, M.L., Resnick, M., and Shneiderman, B. Creativity challenges and opportunities in social computing. *Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, ACM (2009), 3283-3286.
77. Fish, R.S., Kraut, R.E., and Leland, M.D.P. Quilt: a collaborative tool for cooperative writing. *Proceedings of the ACM SIGOIS and IEEECS TC-OA 1988 conference on Office information systems*, ACM (1988), 30-37.
78. Fitzmaurice, G.W., Ishii, H., and Buxton, W.A.S. Bricks: laying the foundations for graspable user interfaces. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press/Addison-Wesley Publishing Co. (1995), 442-449.
79. Floyd, C. A Systematic Look at Prototyping. In Budde, ed., *Approaches to Prototyping*. Springer Verlag, 1984, 105-122.
80. Friedrich, H., Münch, S., Dillmann, R., Bocionek, S., and Sassin, M. Robot programming by demonstration (RPD): supporting the induction by human interaction. *Machine Learning 23*, 2-3 (1996), 163-189.
81. Fujima, J., Lunzer, A., Hornbæk, K., and Tanaka, Y. Clip, connect, clone: combining application elements to build custom interfaces for information access. *Proceedings of the 17th annual ACM symposium on User interface software and technology*, ACM (2004), 175-184.
82. Gane, C. and Sarson, T. *Structured Systems Analysis: Tools and Techniques*. McDonnell Douglas Systems Integration Company, 1977.
83. Gaver, B. and Martin, H. Alternatives: exploring information appliances through conceptual design proposals. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2000), 209-216.
84. Gedenryd, H. *How Designers Work*. PhD Dissertation, Lund University, 1998.
85. Gick, M.L. and Holyoak, K.J. Analogical Problem Solving. *Cognitive Psychology 12*, 3 (1980), 306-55.
86. Girschick, M. *Difference Detection and Visualization in UML Class Diagrams. Technical Report TUD-CS-2006-5*. University of Darmstadt, 2006.
87. Glinert, E.P. *Visual Programming Environments*. IEEE Computer Society Press, 1990.

88. Goldman, K.J. An interactive environment for beginning Java programmers. *Science of Computer Programming 53*, 1 (2004), 3-24.
89. Green, T.R.G. Cognitive Dimensions of Notations. In *People and Computers V*. 1989, 443-460.
90. Green, T.R.G. and Petre, M. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages & Computing 7*, 2 (1996), 131-174.
91. Greenberg, S. Toolkits and interface creativity. *Multimedia Tools and Applications 32*, 2 (2007), 139-159.
92. Greenberg, S. and Boyle, M. Customizable physical interfaces for interacting with conventional applications. *Proceedings of the 15th annual ACM symposium on User interface software and technology*, ACM (2002), 31-40.
93. Greenberg, S. and Fitchett, C. Phidgets: easy development of physical interfaces through physical widgets. *Proceedings of the 14th annual ACM symposium on User interface software and technology*, ACM (2001), 209-218.
94. Greenfield, E. Presentation at the Adobe/Stanford Workshop on Tools for Exploration, Creativity, and Play. 2008.
95. Greenstein, J.S., Helander, M., Landauer, T.K., and Prabhu, P. Pointing Devices. In *Handbook of Human-Computer Interaction*. Elsevier Science BV, 1977, 1317-1348.
96. Guimbretière, F. Paper augmented digital documents. *Proceedings of the 16th annual ACM symposium on User interface software and technology*, ACM (2003), 51-60.
97. Guimbretière, F., Dixon, M., and Hinckley, K. ExperiScope: an analysis tool for interaction data. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2007), 1333-1342.
98. Guimbretière, F., Stone, M., and Winograd, T. Fluid interaction with high-resolution wall-size displays. *Proceedings of the 14th annual ACM symposium on User interface software and technology*, ACM (2001), 21-30.
99. Hackos, J.T. and Redish, J.C. *User and Task Analysis for Interface Design*. Wiley, 1998.
100. Haenlein, H. Personal Communication. 2007.
101. Hailpern, J., Hinterbichler, E., Leppert, C., Cook, D., and Bailey, B.P. TEAM STORM: demonstrating an interaction model for working with multiple ideas during creative group work. *Proceedings of Creativity and Cognition 2007*, ACM (2007), 193-202.
102. Haller, M., Brandl, P., Leithinger, D., Leitner, J., Seifried, T., and Billinghurst, M. Shared Design Space: Sketching Ideas Using Digital Pens and a Large Augmented Tabletop Setup. In *Advances in Artificial Reality and Tele-Existence*. 2006, 185-196.
103. Hammontree, M.L., Hendrickson, J.J., and Hensley, B.W. Integrated data capture and analysis tools for research and testing on graphical user interfaces. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (1992), 431-432.
104. Hansen, M. and Rubin, B. The Listening Post. http://www.earstudio.com/projects/listeningpost.html
105. Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3 (1987), 231-274.

106. Harrison, C. and Hudson, S.E. Scratch input: creating large, inexpensive, unpowered and mobile finger input surfaces. *Proceedings of the 21st annual ACM symposium on User interface software and technology*, ACM (2008), 205-208.
107. Hartmann, B., Abdulla, L., Mittal, M., and Klemmer, S.R. Authoring sensor-based interactions by demonstration with direct manipulation and pattern recognition. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2007), 145-154.
108. Hartmann, B., Doorley, S., and Klemmer, S.R. Hacking, Mashing, Gluing: Understanding Opportunistic Design. *IEEE Pervasive Computing 7*, 3 (2008), 46-54.
109. Hartmann, B., Klemmer, S.R., Bernstein, M., et al. Reflective physical prototyping through integrated design, test, and analysis. *Proceedings of the 19th annual ACM symposium on User interface software and technology*, ACM (2006), 299-308.
110. Hartmann, B., MacDougall, D., and Klemmer, S.R. What Would Other Programmers Do? Suggesting Solutions to Error Messages. *Adjuct Proceedings of UIST 2009*.
111. Hartmann, B., Morris, M.R., Benko, H., and Wilson, A.D. Pictionaire: Supporting Collaborative Design Work by Integrating Physical and Digital Artifacts. *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW). In Press.*, ACM (2010).
112. Hartmann, B., Morris, M.R., and Cassanego, A. Reducing Clutter on Tabletop Groupware Systems with Tangible Drawers. *Adjunct Proceedings of Ubicomp 2006*, IEEE (2006).
113. Hartmann, B., Wu, L., Collins, K., and Klemmer, S.R. Programming by a sample: rapidly creating web applications with d.mix. *Proceedings of the 20th annual ACM symposium on User interface software and technology*, ACM (2007), 241-250.
114. Hartmann, B., Yu, L., Allison, A., Yang, Y., and Klemmer, S.R. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. *Proceedings of the 21st annual ACM symposium on User interface software and technology*, ACM (2008), 91-100.
115. Heckel, P. A technique for isolating differences between files. *Communications of the ACM 21*, 4 (1978), 264-268.
116. Heer, J., Mackinlay, J.D., Stolte, C., and Agrawala, M. Graphical Histories for Visualization: Supporting Analysis, Communication, and Evaluation. *Proceedings of IEEE Information Visualization 2008*, IEEE (2008).
117. Heidenberg, J., Nåls, A., and Porres, I. Statechart features and pre-release maintenance defects. *Journal of Visual Languages & Computing 19*, 4 (2008), 456-467.
118. Herring, S.R., Chang, C., Krantzler, J., and Bailey, B.P. Getting inspired!: understanding how and why examples are used in creative design practice. *Proceedings of the 27th international conference on Human factors in computing systems*, ACM (2009), 87-96.
119. Hilbert, D.M. and Redmiles, D.F. Extracting usability information from user interface events. *ACM Computing Surveys 32*, 4 (2000), 384-421.
120. Hinckley, K., Baudisch, P., Ramos, G., and Guimbretiere, F. Design and analysis of delimiters for selection-action pen gesture phrases in scriboli. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2005), 451-460.
121. Hinckley, K., Pierce, J., Sinclair, M., and Horvitz, E. Sensing techniques for mobile interaction. *Proceedings of the 13th annual ACM symposium on User interface software and technology*, ACM (2000), 91-100.

122. Hoffmann, R., Fogarty, J., and Weld, D.S. Assieme: finding and leveraging implicit references in a web search interface for programmers. *Proceedings of the 20th annual ACM symposium on User interface software and technology*, ACM (2007), 13-22.
123. Hollan, J., Hutchins, E., and Kirsh, D. Distributed cognition: toward a new foundation for human-computer interaction research. *ACM Trans. Comput.-Hum. Interact. 7*, 2 (2000), 174-196.
124. Hong, J.I. and Landay, J.A. WebQuilt: a framework for capturing and visualizing the web experience. *Proceedings of the 10th international conference on World Wide Web*, ACM (2001), 717-724.
125. Hopcroft, J.E., Motwani, R., and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2000.
126. Houde, S. and Hill, C. What do Prototypes Prototype? In M. Helander, T.K. Landauer and P. Prabhu, eds., *Handbook of Human-Computer Interaction*. Elsevier Science BV, 1997.
127. Hudson, S.E. and Mankoff, J. Rapid construction of functioning physical interfaces from cardboard, thumbtacks, tin foil and masking tape. *Proceedings of the 19th annual ACM symposium on User interface software and technology*, ACM (2006), 289-298.
128. Hunt, J.W. and McIlroy, M.D. An Algorithm for Differential File Comparison. *Computing Science Technichal Report #41, Bell Laboratories*, (1976).
129. Huot, S., Dumas, C., Dragicevic, P., Fekete, J., and Hégron, G. The MaggLite post-WIMP toolkit: draw it, connect it and run it. *Proceedings of the 17th annual ACM symposium on User interface software and technology*, ACM (2004), 257-266.
130. Hutchins, E. *Cognition in the Wild*. MIT Press, 1995.
131. Hutchins, E. Material anchors for conceptual blends. *Journal of Pragmatics 37*, 10 (2005), 1555-1577.
132. Hutchins, E.L., Hollan, J.D., and Norman, D.A. Direct manipulation interfaces. *Human-Computer Interaction 1*, 4 (1985), 311-338.
133. Ivory, M.Y. and Hearst, M.A. The state of the art in automating usability evaluation of user interfaces. *ACM Computing Surveys 33*, 4 (2001), 470-516.
134. Johanson, B., Fox, A., and Winograd, T. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. *IEEE Pervasive Computing 1*, 2 (2002), 67-74.
135. Jones, J.C. *Design Methods*. Wiley, 1992.
136. Jørgensen, A.H. On the psychology of prototyping. In R. Budde, ed., *Approaches to Prototyping*. Springer Verlag, 1984, 278-289.
137. Ju, W., Lee, B.A., and Klemmer, S.R. Range: exploring implicit interaction through electronic whiteboard design. *Proceedings of the ACM 2008 conference on Computer supported cooperative work*, ACM (2008), 17-26.
138. Kelleher, C. Looking Glass: Supporting Learning from Peer Programs. Presentation at the IBM New Paradigms for Using Computers (NPUC) Conference. 2009. http://www.almaden.ibm.com/cs/user/npuc2009/slides/NPUC09-CaitlinKelleher.pdf
139. Kelleher, C., Cosgrove, D., Culyba, D., Forlines, C., Pratt, J., and Pausch, R. Alice2: Programming without Syntax Errors. *Adjunct Proceedings of UIST 2002*, ACM (2002).
140. Kelley, J.F. An iterative design methodology for user-friendly natural language office information applications. *ACM Transactions on Information Systems (TOIS) 2*, 1 (1984), 26-41.

141. Kelley, T. and Littman, J. *The Art of Innovation: Lessons in Creativity from IDEO, America's Leading Design Firm*. Broadway Business, 2001.
142. Kin, K., Agrawala, M., and DeRose, T. Determining the Benefits of Direct-Touch, Bimanual, and Multifinger Input on a Multitouch Workstation. *Proceedings of Graphics Interface 2009*, (2009), 119-124.
143. Kirsh, D. and Maglio, P. On distinguishing epistemic from pragmatic action. *Cognitive Science 18*, 4 (1994), 513-549.
144. Klemmer, S.R. Integrating Physical and Digital Interactions. *IEEE Computer 38*, 10 (2005), 111-113.
145. Klemmer, S.R., Hartmann, B., and Takayama, L. How bodies matter: five themes for interaction design. *Proceedings of the 6th conference on Designing Interactive systems*, ACM (2006), 140-149.
146. Klemmer, S.R., Li, J., Lin, J., and Landay, J.A. Papier-Mâché: toolkit support for tangible input. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2004), 399-406.
147. Klemmer, S.R., Newman, M.W., Farrell, R., Bilezikjian, M., and Landay, J.A. The designers' outpost: a tangible interface for collaborative web site design. *Proceedings of the 14th annual ACM symposium on User interface software and technology*, ACM (2001), 1-10.
148. Klemmer, S.R., Sinha, A.K., Chen, J., Landay, J.A., Aboobaker, N., and Wang, A. Suede: a Wizard of Oz prototyping tool for speech user interfaces. *Proceedings of the 13th annual ACM symposium on User interface software and technology*, ACM (2000), 1-10.
149. Klemmer, S.R., Thomsen, M., Phelps-Goodman, E., Lee, R., and Landay, J.A. Where do web sites come from?: capturing and interacting with design history. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2002), 1-8.
150. Klemmer, S.R., Verplank, B., and Ju, W. Teaching embodied interaction design practice. *Proceedings of the 2005 conference on Designing for User eXperience*, AIGA: American Institute of Graphic Arts (2005), 26.
151. Klokmose, C.N. and Beaudouin-Lafon, M. VIGO: instrumental interaction in multi-surface environments. *Proceedings of the 27th international conference on Human factors in computing systems*, ACM (2009), 869-878.
152. Kumar, R., Kim, J., and Klemmer, S.R. Automatic retargeting of web page content. *Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, ACM (2009), 4237-4242.
153. Kurlander, D. and Feiner, S. Editable Graphical Histories. *IEEE Workshop on Visual Languages*, IEEE (1988), 127-134.
154. Kurlander, D. and Feiner, S. A Visual Language for Browsing, Undoing, and Redoing Graphical Interface Commands. In S.K. Chang, ed., *Visual Languages and Visual Programming*. Plenum Press, NY, 1990, 257-275.
155. Landay, J.A. and Myers, B.A. Interactive sketching for the early stages of user interface design. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (1995), 43-50.
156. Landay, J.A. and Myers, B.A. Sketching Interfaces: Toward More Human Interface Design. *IEEE Computer 34*, 3 (2001), 56-64.
157. Lawson, B. *How designers think, 4th ed.* Elsevier / Architectural Press, 2006.

158. Lee, B.A., Klemmer, S.R., Srivastava, S., and Brafman, R. *Adaptive Interfaces for Supporting Design by Example. Technical Report CSTR 2007-16*. Stanford University Computer Science Department, 2007.

159. Lee, J.C., Avrahami, D., Hudson, S.E., Forlizzi, J., Dietz, P.H., and Leigh, D. The calder toolkit: wired and wireless components for rapidly prototyping interactive devices. *Proceedings of the 5th conference on Designing interactive systems: processes, practices, methods, and techniques*, ACM (2004), 167-175.

160. Levis, P., Madden, S., Polastre, J., et al. TinyOS: An Operating System for Wireless Sensor Networks. In W. Weber, J. Babaey and E. Aarts, eds., *Ambient Intelligence*. Springer, 2005.

161. Levoy, M. Spreadsheets for images. *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, ACM (1994), 139-146.

162. Lewis, C. and Rieman, J. Task-Centered User Interface Design. 1993. http://hcibib.org/tcuid/

163. Li, Y., Hong, J.I., and Landay, J.A. Topiary: a tool for prototyping location-enhanced applications. *Proceedings of the 17th annual ACM symposium on User interface software and technology*, ACM (2004), 217-226.

164. Li, Y. and Landay, J.A. Informal prototyping of continuous graphical interactions by demonstration. *Proceedings of the 18th annual ACM symposium on User interface software and technology*, ACM (2005), 221-230.

165. Li, Y. and Landay, J.A. Activity-based prototyping of ubicomp applications for long-lived, everyday human activities. *Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, ACM (2008), 1303-1312.

166. Lichter, H., Schneider-Hufschmidt, M., and Züllighoven, H. Prototyping in Industrial Software Projects-Bridging the Gap Between Theory and Practice. *IEEE Trans. Softw. Eng. 20*, 11 (1994), 825-832.

167. Licklider, J.C.R. Man-Computer Symbiosis. *Human Factors in Electronics, IRE Transactions on HFE-1*, 1 (1960), 4-11.

168. Lieberman, H., ed. *Your Wish is My Command*. Morgan Kaufmann, 2001.

169. Lieberman, H., Paternò, F., and Wulf, V., eds. *End User Development*. Springer, 2006.

170. Lim, Y., Stolterman, E., and Tenenberg, J. The anatomy of prototypes: Prototypes as filters, prototypes as manifestations of design ideas. *ACM Trans. Comput.-Hum. Interact. 15*, 2 (2008), 1-27.

171. Lin, J., Newman, M.W., Hong, J.I., and Landay, J.A. DENIM: finding a tighter fit between tools and practice for Web site design. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2000), 510-517.

172. Lins, E. *Crumb128 - Atmel ATmega128 module*. chip45 GmbH & Co. KG. http://www.chip45.com/index.pl?page=Crumb128&lang=en

173. Little, G., Lau, T.A., Cypher, A., Lin, J., Haber, E.M., and Kandogan, E. Koala: capture, share, automate, personalize business processes on the web. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2007), 943-946.

174. Liu, A.L. and Li, Y. BrickRoad: a light-weight tool for spontaneous design of location-enhanced applications. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2007), 295-298.

175. Loewy, R. *Never Leave Well Enough Alone: The Personal Record of an Industrial Designer*. Simon and Schuster, 1951.
176. Lunzer, A. Choice and Comparison Where the User Wants Them: Subjunctive Interfaces for Computer-Supported Exploration. *Proceedings of INTERACT '99: IFIP Conference on Human-Computer Interaction*, (1999), 474-482.
177. Lunzer, A. and Hornbæk, K. Subjunctive interfaces: Extending applications to support parallel setup, viewing and control of alternative scenarios. *ACM Transactions on Computer-Human Interaction 14*, 4 (2008), 1-44.
178. MacIntyre, B., Gandy, M., Dow, S., and Bolter, J.D. DART: a toolkit for rapid design exploration of augmented reality experiences. *Proceedings of the 17th annual ACM symposium on User interface software and technology*, ACM (2004), 197-206.
179. Mackay, W.E. EVA: an experimental video annotator for symbolic analysis of video data. *SIGCHI Bulletin 21*, 2 (1989), 68-71.
180. Mackay, W.E., Appert, C., Beaudouin-Lafon, M., et al. Touchstone: exploratory design of experiments. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2007), 1425-1434.
181. Marks, J., Andalman, B., Beardsley, P.A., et al. Design galleries: a general approach to setting parameters for computer graphics and animation. *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM (1997), 389-400.
182. Maynes-Aminzade, D., Winograd, T., and Igarashi, T. Eyepatch: prototyping camera-based interaction through examples. *Proceedings of the 20th annual ACM symposium on User interface software and technology*, ACM (2007), 33-42.
183. McConnell, S. *Code Complete: A Practical Handbook of Software Construction, 2nd ed.* Microsoft Press, 2004.
184. Mehra, A., Grundy, J., and Hosking, J. A generic approach to supporting diagram differencing and merging for collaborative design. *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ACM (2005), 204-213.
185. Mellis, D.A., Banzi, M., Cuartielles, D., and Igoe, T. Arduino: An Open Electronics Prototyping Platform. *Extended Abstracts of the SIGCHI Conference on Human Factors in Computing Systems*, AMC (2007).
186. Merrill, D. *FlexiGesture: A sensor-rich real-time adaptive gesture and affordance learning platform for electronic music control. S.M. Thesis.* Massachusetts Institute of Technology, 2004.
187. Merrill, D., Kalanithi, J., and Maes, P. Siftables: towards sensor network user interfaces. *Proceedings of the 1st international conference on Tangible and embedded interaction*, ACM (2007), 75-78.
188. Miller, P., Pane, J., Meter, G., and Vorthmann, S. Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University. *Interactive Learning Environments 4*, 2 (1994), 140 – 158.
189. Moggridge, B. *Designing Interactions*. The MIT Press, 2007.
190. Myers, B., McDaniel, R., Miller, R., et al. The Amulet environment: new models for effective user interface software development. *IEEE Transactions on Software Engineering 23*, 6 (1997), 347-365.
191. Myers, B., Hudson, S.E., and Pausch, R. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction 7*, 1 (2000), 3-28.

192. Myers, B.A. Separating application code from toolkits: eliminating the spaghetti of call-backs. *Proceedings of the 4th annual ACM symposium on User interface software and technology*, ACM (1991), 211-220.
193. Myers, B.A. Graphical techniques in a spreadsheet for specifying user interfaces. *Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology*, ACM (1991), 243-249.
194. Myers, B.A. More Natural User Experiences for Design and Software Development. Presentation at the IBM New Paradigms for Using Computers (NPUC) Conference. 2009. http://www.almaden.ibm.com/cs/user/npuc2009/slides/EUP-Myers-NPUC-overview.pdf
195. Myers, B.A., Park, S.Y., Nakano, Y., Mueller, G., and Ko, A.J. How Designers Design and Program Interactive Behaviors. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE Computer Society (2008).
196. Nardi, B.A. *A small matter of programming*. MIT Press, 1993.
197. Nassi, I. and Shneiderman, B. Flowchart techniques for structured programming. *SIGPLAN Notices 8*, 8 (1973), 12-26.
198. Neuwirth, C.M., Kaufer, D.S., Chandhok, R., and Morris, J.H. Issues in the design of computer support for co-authoring and commenting. *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, ACM (1990), 183-195.
199. Nielsen, J. and Molich, R. Heuristic evaluation of user interfaces. *Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people*, ACM (1990), 249-256.
200. Niemeyer, P. JSR 274: The BeanShell Scripting Language. http://jcp.org/en/jsr/detail?id=274
201. Norman, D.A. *Things that make us smart*. Basic Books, 1994.
202. Partridge, K., Chatterjee, S., Sazawal, V., Borriello, G., and Want, R. TiltType: accelerometer-supported text entry for very small devices. *Proceedings of the 15th annual ACM symposium on User interface software and technology*, ACM (2002), 201-204.
203. Pausch, R., Conway, M., and DeLine, R. Lessons learned from SUIT, the simple user interface toolkit. *ACM Transactions on Information Systems (TOIS) 10*, 4 (1992), 320-344.
204. Pering, C. Interaction design prototyping of communicator devices: towards meeting the hardware-software challenge. *ACM interactions 9*, 6 (2002), 36-46.
205. Perry, M. and Sanderson, D. Coordinating joint design work: the role of communication and artefacts. *Design Studies 19*, 3 (1998), 273-288.
206. Peters, K., Hirmes, D., Jokol, K., et al. *Flash Math Creativity*. friends of ED, 2004.
207. Polanyi, M. *The Tacit Dimension*. Doubleday, 1966.
208. Pomberger, G., Bischofberger, W.R., Kolb, D., Pree, W., and Schlemm, H. Prototyping-Oriented Software Development - Concepts and Tools. *Structured Programming 12*, 1 (1991), 43-60.
209. Puckette, M.S. Pure Data: Another Integrated Computer Music Environment. *Proceedings of the International Computer Music Conference*, ICMA (1996), 37--41.
210. Reas, C. and Fry, B. Processing: programming for the media arts. *AI & Society 20*, 4 (2006), 526-538.
211. Rettig, M. Prototyping for tiny fingers. *Communications of the ACM 37*, 4 (1994), 21-27.

212. Riddle, W.E. Advancing the state of the art in software system prototyping. In R. Budde, ed., *Approaches to Prototyping*. Springer Verlag, 1984, 19-26.

213. Rittel, H. and Webber, M. Dilemmas in a General Theory of Planning. *Policy Sciences 4*, (1973), 155-169.

214. Rogers, Y. and Muller, H. A framework for designing sensor-based interactions to promote exploration and reflection in play. *International Journal of Human-Computer Studies 64*, 1 (2006), 1-14.

215. Rozin, D. Wooden Mirror. http://www.smoothware.com/danny/woodenmirror.html

216. Rubin, J. and Chisnell, D. *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests, 2nd ed.* Wiley, 2008.

217. de Sá, M., Carriço, L., Duarte, L., and Reis, T. A mixed-fidelity prototyping tool for mobile devices. *Proceedings of the working conference on Advanced visual interfaces*, ACM (2008), 225-232.

218. Sakoe, H. and Chiba, S. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, (1978).

219. Salber, D., Dey, A.K., and Abowd, G.D. The context toolkit: aiding the development of context-enabled applications. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (1999), 434-441.

220. Schneider, K. Prototypes as assets, not toys: why and how to extract knowledge from prototypes. *Proceedings of the 18th international conference on Software engineering*, IEEE Computer Society (1996), 522-531.

221. Schon, D.A. *The reflective practitioner*. Basic Books, 1983.

222. Schrage, M. *Serious play*. Harvard Business Press, 2000.

223. di Sciascio, E. and Mongiello, M. Query by Sketch and Relevance Feedback for Content-Based Image Retrieval over the Web. *Journal of Visual Languages & Computing 10*, 6 (1999), 565-584.

224. Segerståhl, K. Utilization of pervasive IT compromised? *Proceedings of the 7th International Conference on Mobile and Ubiquitous Multimedia - MUM '08*, (2008), 168.

225. Selker, T. A bike helmet built for road hazards. 2006. http://news.cnet.com/2300-1008_3-6111157-1.html?tag=mncol

226. Sharp, H., Rogers, Y., and Preece, J. *Interaction Design: Beyond Human-Computer Interaction, 2nd ed*. Wiley, 2007.

227. Shneiderman, B. Overview + Detail. In S.K. Card, J.D. Mackinlay and B. Shneiderman, eds., *Readings in Information Visualization*. Morgan Kaufmann, 1996.

228. Shneiderman, B. *Leonardo's Laptop: Human Needs and the New Computing Technologies*. The MIT Press, 2003.

229. Shneiderman, B., Fischer, G., Czerwinski, M., et al. Creativity Support Tools: Report From a U.S. National Science Foundation Sponsored Workshop. *International Journal of Human-Computer Interaction 20*, 2 (2006), 61.

230. Simon, H.A. *The sciences of the artificial*. MIT Press, 1996.

231. Smith, D.C., Cypher, A., and Spohrer, J. KidSim: programming agents without a programming language. *Communications of the ACM 37*, 7 (1994), 54-67.

232. Song, H., Guimbretière, F., Hu, C., and Lipson, H. ModelCraft: capturing freehand annotations and edits on physical 3D models. *Proceedings of the 19th annual ACM symposium on User interface software and technology*, ACM (2006), 13-22.
233. Star, S. and Griesemer, J. Institutional Ecology, `Translations' and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology, 1907-39. *Social Studies of Science 19*, 3 (1989), 420, 387.
234. Streitz, N.A., Geißler, J., Holmer, T., et al. i-LAND: an interactive landscape for creativity and innovation. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (1999), 120-127.
235. Stylos, J. and Myers, B.A. Mica: A Web-Search Tool for Finding API Components and Examples. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE (2006), 195-202.
236. Su, S. Visualizing, Editing, and Inferring Structure in 2D Graphics. *Adjunct Proceedings of UIST 2007*, ACM (2007).
237. Teague, W.D. *Design This Day; the Technique of Order in the Machine Age*. Harcourt, Brace and Company, New York, 1940.
238. Teitelbaum, T. and Reps, T. The Cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM 24*, 9 (1981), 563-573.
239. Terry, M. *Set-based user interaction. Unpublished Ph.D. Dissertation.* Georgia Institute of Technology, 2005.
240. Terry, M. and Mynatt, E.D. Recognizing creative needs in user interface design. *Proceedings of the 4th conference on Creativity & cognition*, ACM (2002), 38-44.
241. Terry, M. and Mynatt, E.D. Side views: persistent, on-demand previews for open-ended tasks. *Proceedings of the 15th annual ACM symposium on User interface software and technology*, ACM (2002), 71-80.
242. Terry, M., Mynatt, E.D., Nakakoji, K., and Yamamoto, Y. Variation in element and action: supporting simultaneous development of alternative solutions. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2004), 711-718.
243. Tohidi, M., Buxton, W., Baecker, R., and Sellen, A. Getting the right design and the design right. *Proceedings of the SIGCHI conference on Human Factors in computing systems*, ACM (2006), 1243-1252.
244. Toomim, M., Begel, A., and Graham, S.L. Managing Duplicated Code with Linked Editing. *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, IEEE Computer Society (2004), 173-180.
245. Tufte, E.R. *Envisioning information*. Graphics Press, 1990.
246. Wang, G. and Cook, P.R. On-the-fly programming: using code as an expressive musical instrument. *Proceedings of the 2004 conference on New interfaces for musical expression*, National University of Singapore (2004), 138-143.
247. Weiler, P. Software for the usability lab: a sampling of current tools. *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, ACM (1993), 57-60.
248. Weiser, M. The Computer for the Twenty-First Century. *Scientific American 265*, 3 (1991), 94-104.

249. Wellner, P.D. Statemaster: A UIMS based on statechart for prototyping and target implementation. *SIGCHI Bulletin 20*, SI, 177-182.

250. Wieringa, R. A survey of structured and object-oriented software specification methods and techniques. *ACM Comput. Surv. 30*, 4 (1998), 459-527.

251. Winograd, T., ed. *Bringing design to software*. ACM, 1996.

252. Winter, D.A. *Biomechanics and Motor Control of Human Movement*. Wiley-Interscience, 1990.

253. Wobbrock, J.O., Wilson, A.D., and Li, Y. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. *Proceedings of the 20th annual ACM symposium on User interface software and technology*, ACM (2007), 159-168.

254. Wojahn, P.G., Neuwirth, C.M., and Bullock, B. Effects of interfaces for annotation on communication in a collaborative task. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (1998), 456-463.

255. Wong, Y.Y. Rough and ready prototypes: lessons from graphic design. *Posters and short talks of the 1992 SIGCHI conference on Human factors in computing systems*, ACM (1992), 83-84.

256. Wright, M. Open Sound Control: an enabling technology for musical networking. *Organised Sound 10*, 3 (2005), 193-200.

257. Yee, K., Swearingen, K., Li, K., and Hearst, M. Faceted metadata for image search and browsing. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2003), 401-408.

258. Yeh, R.B., Paepcke, A., and Klemmer, S.R. Iterative design and evaluation of an event architecture for pen-and-paper interfaces. *Proceedings of the 21st annual ACM symposium on User interface software and technology*, ACM (2008), 111-120.

259. Yourdon, E. *Just Enough Structured Analysis*. Yourdon Press, 2006. http://www.yourdon.com/jesa/pdf/JESA_mpmb.pdf

260. Zhang, H. *Control Freaks. Unpublished Master's Thesis.* Interaction Design Institute Ivrea, 2006. http://failedrobot.com/thesis/

261. Zhang, H. and Hartmann, B. Building upon everyday play. *CHI '07 extended abstracts on Human factors in computing systems*, ACM (2007), 2019-2024.