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:

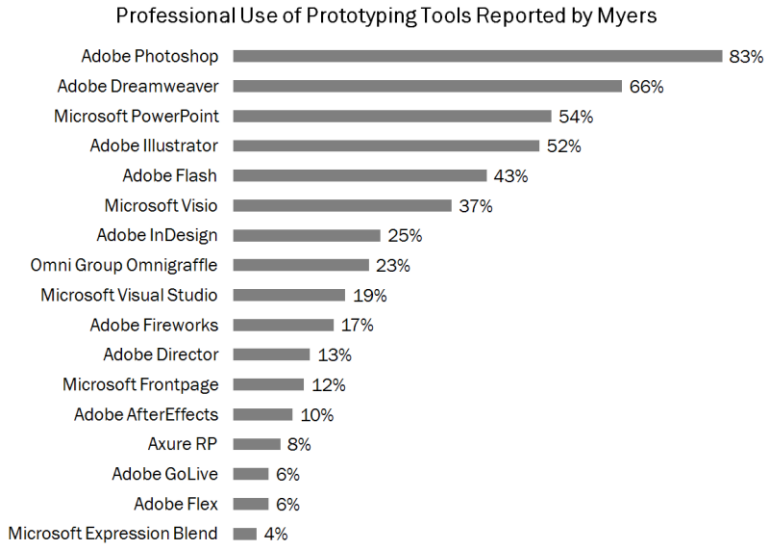

Figure 3.1: Common tools used for UI prototyping as reported in Myers' survey of interaction designers [195]. Figure redrawn by the author.

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.

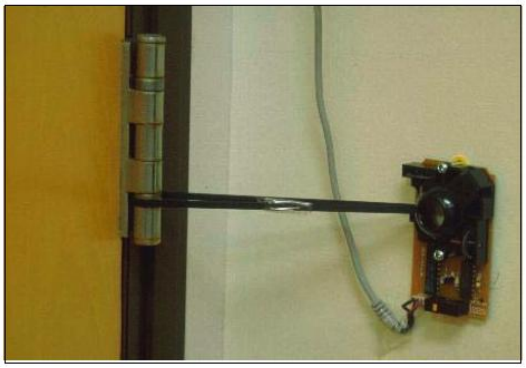

Figure 3.4: Buxton's Doormouse [56] is an example of a "hardware hack" that repurposes a standard mouse.

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.

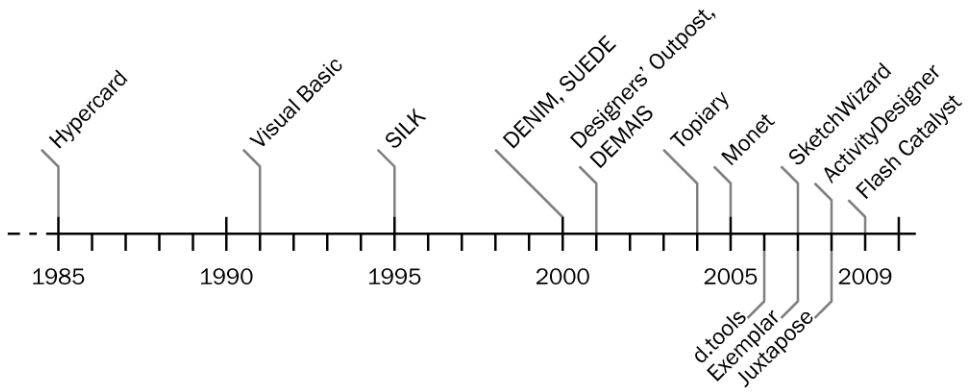

Figure 3.5: Timeline of prototyping tools for graphical user interfaces.

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).

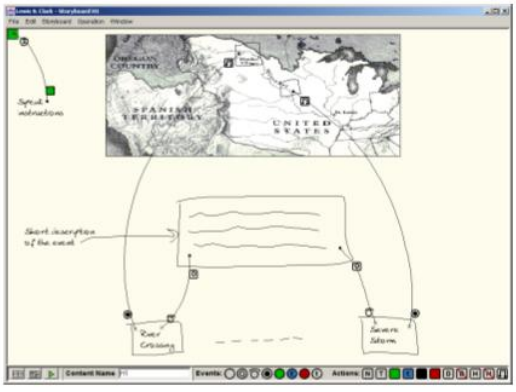

Figure 3.6: Bailey's DEMAIS system introduced a visual language for sketching multimedia applications [40].

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.

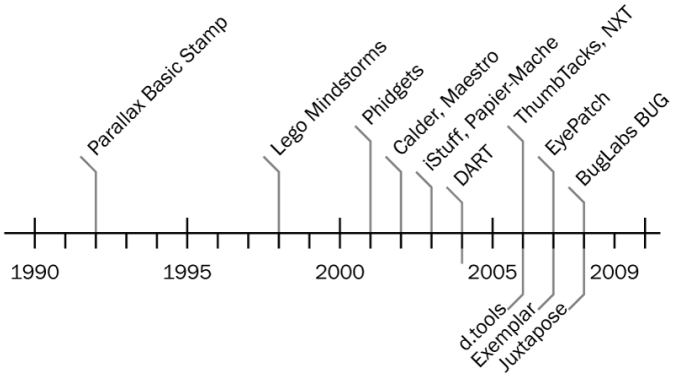

Figure 3.8: Timeline of selected physical computing toolkits.

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].

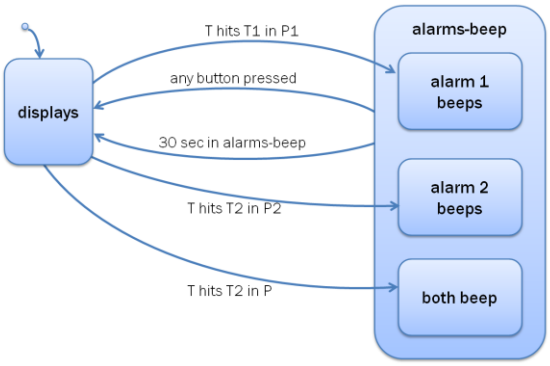

Figure 3.9: A partial, hierarchical statechart for a wrist watch with alarm function; redrawn from an example in Harel [105].

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

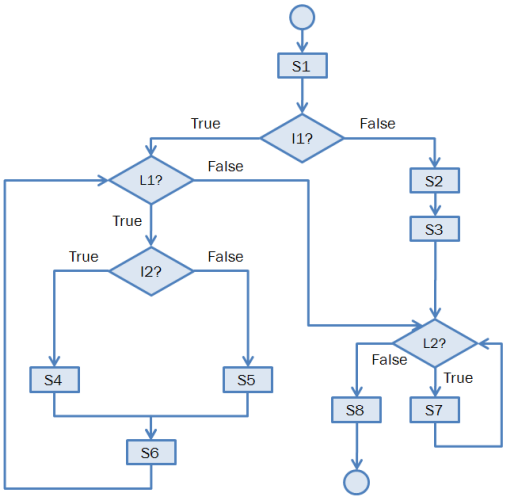

Figure 3.10: Example of a flowchart, adapted from Glinert [87].

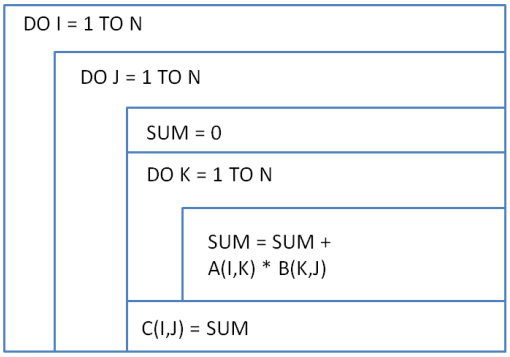

Figure 3.11: Example of a Nassi-Shneiderman structogram, adapted from Glinert [87].

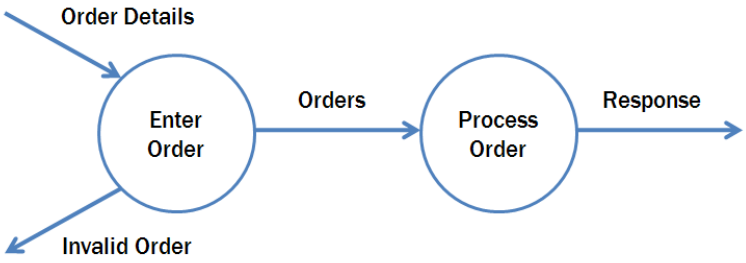

Figure 3.12: Example of a data flow diagram, redrawn by the author from Yourdon [259: p. 159]

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

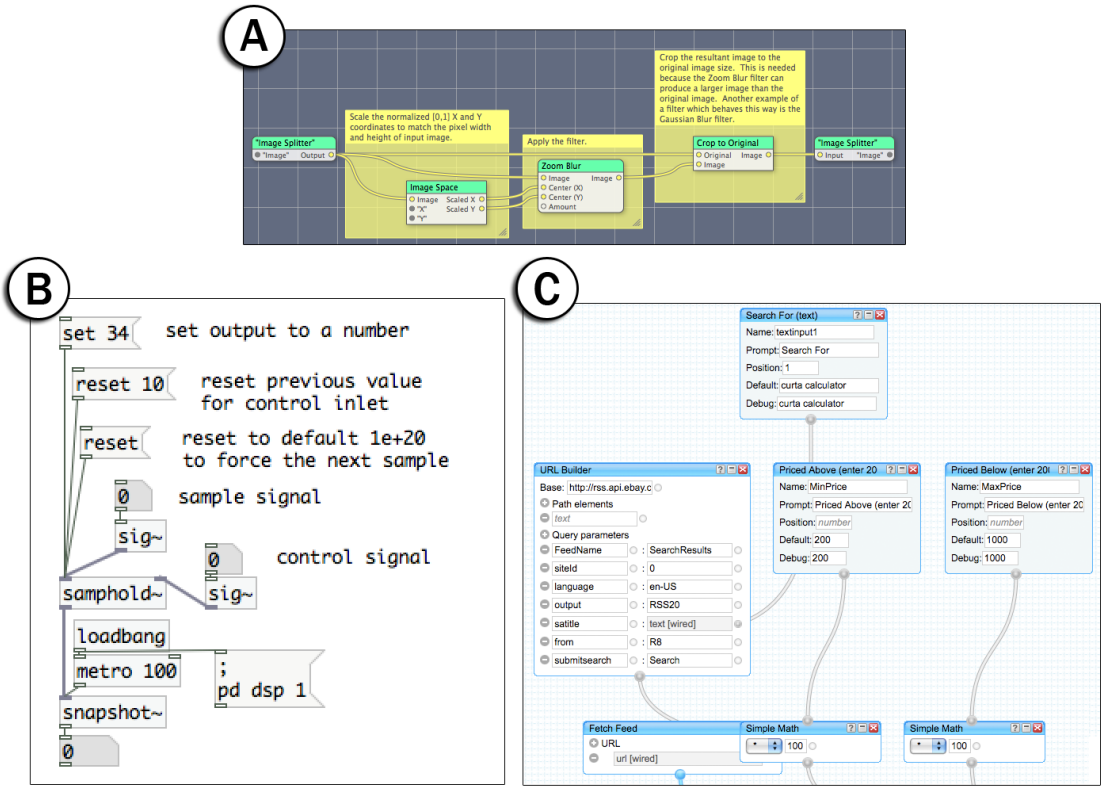

Figure 3.13: Examples of commercial or open source data flow languages. **A**: Quartz Composer; **B**: Pure Data; **C**: Yahoo Pipes

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

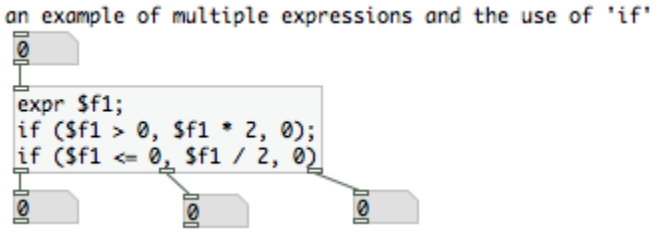

Figure 3.14: Example of hybrid authoring in Pure Data: a visual node contains an algebraic expression.

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.

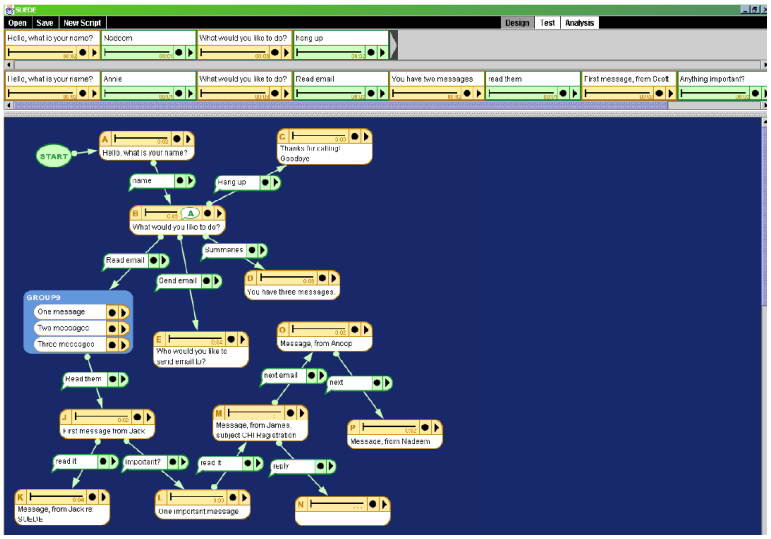

Figure 3.15: SUEDE introduced techniques to unite design, test, and analysis of speech user interfaces.

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.