Reflective physical prototyping through integrated design, test, and analysis

Björn Hartmann, Scott R. Klemmer,
Michael Bernstein, Leith Abdulla, Brandon Burr, Avi Robinson-Mosher, Jennifer Gee
Stanford University HCI Group
Computer Science Department, Stanford, CA 94305-9035, USA
{bjoern, srk, mbernst, eleith, bburr, avir, jgee}@stanford.edu

ABSTRACT
Prototyping is the pivotal activity that structures innovation, collaboration, and creativity in design. Prototypes embody design hypotheses and enable designers to test them. Framing design as a thinking-by-doing activity foregrounds *iteration* as a central concern. This paper presents d.tools, a toolkit that embodies an iterative-design-centered approach to prototyping information appliances. This work offers contributions in three areas. First, d.tools introduces a *statechart-based visual design tool* that provides a low threshold for early-stage prototyping, extensible through code for higher-fidelity prototypes. Second, our research introduces *three important types of hardware extensibility* — at the hardware-to-PC interface, the intra-hardware communication level, and the circuit level. Third, d.tools *integrates design, test, and analysis* of information appliances. We have evaluated d.tools through three studies: a laboratory study with thirteen participants; rebuilding prototypes of existing and emerging devices; and by observing seven student teams who built prototypes with d.tools.

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

General terms: Design, Human Factors

Keywords: Toolkits, information appliances, design tools, prototyping, integrating physical & digital, design thinking

INTRODUCTION
Ubiquitous computing devices such as information appliances — mobile phones, digital cameras, and music players—are growing quickly in number and diversity. To arrive at usable designs for such physical UIs, product designers commonly build a series of prototypes — approximations of a product along some dimensions of interest. These prototypes are the pivotal media that structure *innovation*, *collaboration*, and *creativity* in design [21, 32]. 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.

Reflective practice, the framing and evaluation of a design challenge by *working it through*, rather than just *thinking it through*, points out that physical action and cognition are interconnected [23, 30]. Successful product 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 [31]. The epistemic production [22] of concrete prototypes affords unexpected realizations that a designer could not have arrived at without producing a concrete artifact. This articulation of design as a thinking-by-doing activity foregrounds *iteration* as a central concern of design process. And indeed, product designer Michael Barry argues that, "the companies that want to see the most models in the least time are the most design-sensitive; the companies that want that one perfect model are the least design sensitive." [33]

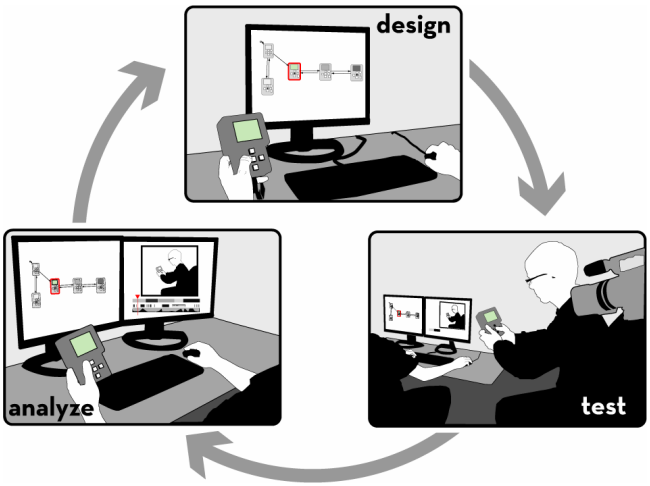


Figure 1. Toolkit support for design thinking: d.tools integrates design, test, and analysis for physical prototyping.

In this paper, we suggest iteration as a core concern for UI tools and present d.tools, a design tool that embodies an iterative-design-centered approach to prototyping physical UIs (see Figure 1). This work offers three contributions.

The first contribution is a set of interaction techniques and architectural features that enable d.tools to provide a *low threshold* for early-stage prototyping. d.tools introduces a visual, statechart-based prototyping model (see Figure 2) that extends existing storyboard-driven design practice [19]. To provide a higher ceiling than is possible with visual programming alone, d.tools augments visual authoring with textual programming.

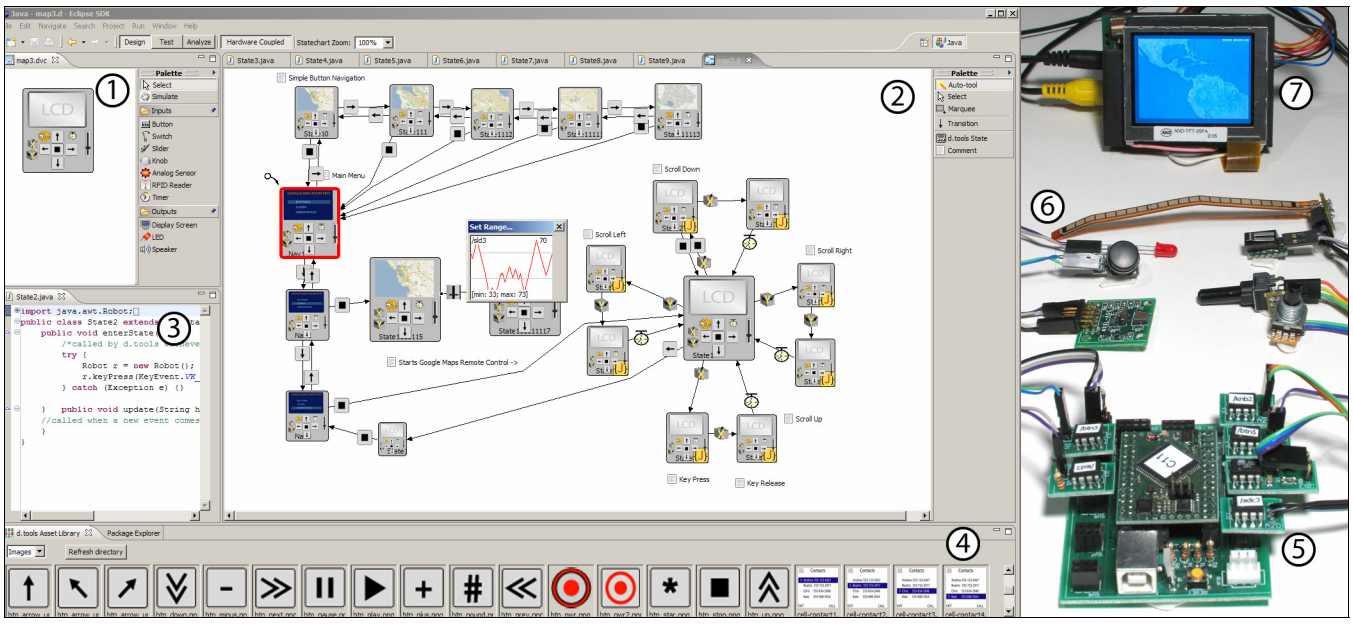


Figure 2. *Left*: The d.tools software authoring environment offers (1) a device designer; (2) a statechart editor; (3) a source code editor; and (4) an image browser. *Right*: The d.tools hardware interface (5) connects compatible hardware inputs (6) to the PC. d.tools includes authoring support for small LCD screens (7).

Second, d.tools offers an extensible architecture for physical interfaces. In this area, d.tools builds on prior work [4, 7, 9, 13, 14, 26] 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 significantly 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.

Third, d.tools integrates *design, test, and analysis* of information appliances. In test mode, d.tools records a video of the user's interaction with the physical device and logs interaction events to structure the video. Analysis mode uses this integration of video and event logs to facilitate post-test review of usability data. While iterative design is central to current practice, few tools—the notable exception being SUEDE [24]—have explored how this cycle can be facilitated through computation.

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

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 the design process. Comprehensive prototypes that integrate form factor (*looks-like* prototypes) and functions (*works-like* prototypes) are mostly created as expensive one-offs that serve as 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 [10] while leaving room to expand into higher-fidelity designs for presentations.

REFLECTIVE PROTOTYPING WITH D.TOOLS

In this section we discuss the most important interaction techniques that d.tools offers to enable the rapid design and evaluation of interactive physical devices. d.tools supports *design thinking* rather than *implementation tinkering*. Using d.tools, designers place physical controllers (*e.g.*, buttons, sliders), sensors (*e.g.*, accelerometers, compasses),

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. In *design mode*, software duals of physical I/O components can be graphically arranged into a visual representation of the physical device (see Figure 2, part 1). On the PC, designers then author behavior using this representation in a visual language inspired by the statecharts formalism [16] (see Figure 2, part 2). d.tools employs a PC as a proxy for an embedded processor to prevent limitations of embedded hardware from impinging on design thinking.

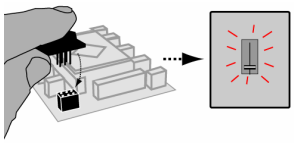
Designers can test their authored interactions with the device at any point in time, since their visual interaction model is always connected to the "live" device. When seeking to gather feedback from others, designers switch to *test mode*. In test mode, d.tools records live video and audio of user interactions with the prototype—important for understanding ergonomics, capturing user quotes, and finding usability problems. d.tools also logs all user interaction events and uses this log to automatically structure the test videos. Video 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 [27]. d.tools interactions with structured video enable *rapid usability analysis* through aggregate data visualization, *fast access to video data* through the visual interaction model and vice versa, and finally *comparative evaluation* of multiple user tests in a video matrix.

DESIGNING A PROTOTYPE

This section presents d.tools support for authoring interaction models with physical I/O components. As an example scenario, consider a designer creating a handheld GPS unit featuring tilt-based map navigation.

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. Physical components announce themselves to d.tools, creating virtual duals in this editor. Alternatively—when the physical components are not at hand or 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, even manipulate the behavior via Wizard of Oz [20] at test time.



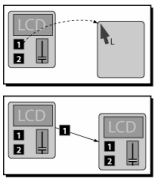
In the device editor, designers create, arrange and resize input and output components, specifying their appearance by selecting images from an integrated image browser. This iconic representation affords rapid matching of software widgets with physical I/O components.

The component library available to designers comprises a diverse selection of buttons, switches, sliders, knobs, and RFID readers. Outputs include LCD screens, LEDs, and speakers. LCD and sound output are connected to the PC A/V subsystem, not our hardware interface. In addition, general purpose input and outputs are available for designers who wish to add custom components. Physical and virtual components are linked through a hardware address that serves as a unique identifier of an input or output.

Authoring interaction models

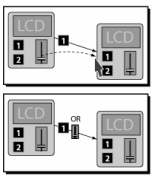
Designers define their prototype's behavior by creating interaction graphs in the *statechart editor* (see Figure 2). States are graphical instances of the device design. They describe the content assigned to the outputs of the prototype at a particular point in the UI: screen images, sounds, LED behaviors. States are created by dragging from the statechart editors palette onto the graph 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 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.

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.

Conditions for state transitions can be composed using the Boolean AND and OR. A single such connective is applied to all conditionals on a transition arrow, as complex Boolean expressions are error-prone. More complex conditionals can be authored by introducing additional states. This allows authoring conditionals such as "transition if the accelerometer is tilted to the right, but only if the tilt-enable button is held down simultaneously."



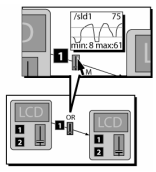
Within the visual editor, *timers* can be added as input components to a device to create automatic transitions or (connected with AND to a sensor input) to require a certain amount of time to pass before acting on input data. Automatic transitions are useful for sequencing output behaviors, and timeouts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

While the statechart's visual representation aids a designer's understanding of the control flow, complex designs still benefit from explanation. d.tools supports

commenting with text notes that can be freely placed on the statechart canvas.

Demonstrating transitions

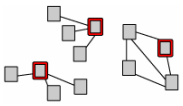
Through our own prototyping practice and through student projects built with d.tools, we discovered that fine-tuning parameters of continuous sensors is a time-consuming, trial-and-error process. Mapping sensor values to discrete categories is further complicated by noise and non-linear responses. The time taken "tuning the dials" could be better spent exploring the design space.



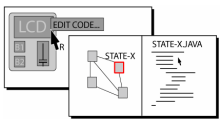
d.tools facilitates parameter setting in two ways. First, the *Sensor Data View* presents a real-time visualization of all attached continuous sensors. Second, ranges of sensor data that trigger transitions can be *authored by demonstration*. The designer selects the input icon on the transition that represents the desired continuous input, bringing up a real-time display of the sensor's current value and history. The designer then performs the desired interaction with the physical prototype (*e.g.*, tilting an accelerometer to the right or moving a slider) and presses keys to define upper and lower thresholds for the transition. This technique replaces needing to set numerical sensor values through trial-and error parameter modification with a physical demonstration technique. This approach lends itself to future work on machine-learning by demonstration for capturing more complex input patterns (cf. [12]).

Raising the ceiling

The statechart-based visual programming model embodied in d.tools enables rapid design of initial comprehensive prototypes, but the complexity of the control flow and interactive behavior that can be authored is limited. To support later phases of design, when labor and expertise permit higher-fidelity prototyping, d.tools provides two mechanisms that enable more complex interactions: parallel statecharts and extending statecharts with code.



Expressing parallelism in single point-of-control automata 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 subgraphs (*e.g.*, *the power button can always be used to turn the device off, regardless of the other state of the model*).



Designers can attach Java code to visual states to specify behaviors that are beyond the capability of the visual environment (*e.g.*, dynamically generate graphics such as map annotations). The right-click context menu for states offers actions to edit and hook or unhook Java code for each state. The first time a designer chooses to add code, d.tools generates a skeleton source code file and opens a Java editor. We leverage the Eclipse programming environment to provide auto-completion, syntax highlighting, and integrated help. Eclipse automatically compiles, loads, and updates code. d.tools offers a compact API that calls designers' functions on transition and input events, allows designers to query input state of any attached hardware, gives write access to attached outputs (*e.g.*, to programmatically change the image shown on the LCD screen), and allows remote control of third party applications (see Table 1). Using this API, two of the authors prototyped accelerometer-based zoom and pan control for the Google Earth application in less than 30 minutes.

Executing interaction models at design time

Designers can execute interaction models in three ways. First, they can *manipulate the attached hardware*; the prototype is always live. Second, they can *imitate hardware events* within the software workbench by using a simulation tool where the cursor can be used to click and drag virtual inputs that will then generate appropriate event transitions. Finally, designers can employ the *Wizard Of Oz* [20, 24] technique by operating the prototype's visual representation. In all cases, the prototype is fully interactive.

TESTING & ANALYZING PROTOTYPES

d.tools provides integrated support for designers to *test* prototypes with users, and *analyze* the results to inform subsequent iteration. Manual video annotation and analysis for usability tests is enormously time consuming. Even though video recording of user sessions is common in design studios, resource limits often preclude later analysis. We introduce d.tools support for video analysis through

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 received 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 [1]) to remote control external applications.

Table 1. The d.tools Java API allows designers to extend visual states with source code. The listed functions serve as the interface between designer's code and d.tools runtime system. Standard Java classes are also accessible.

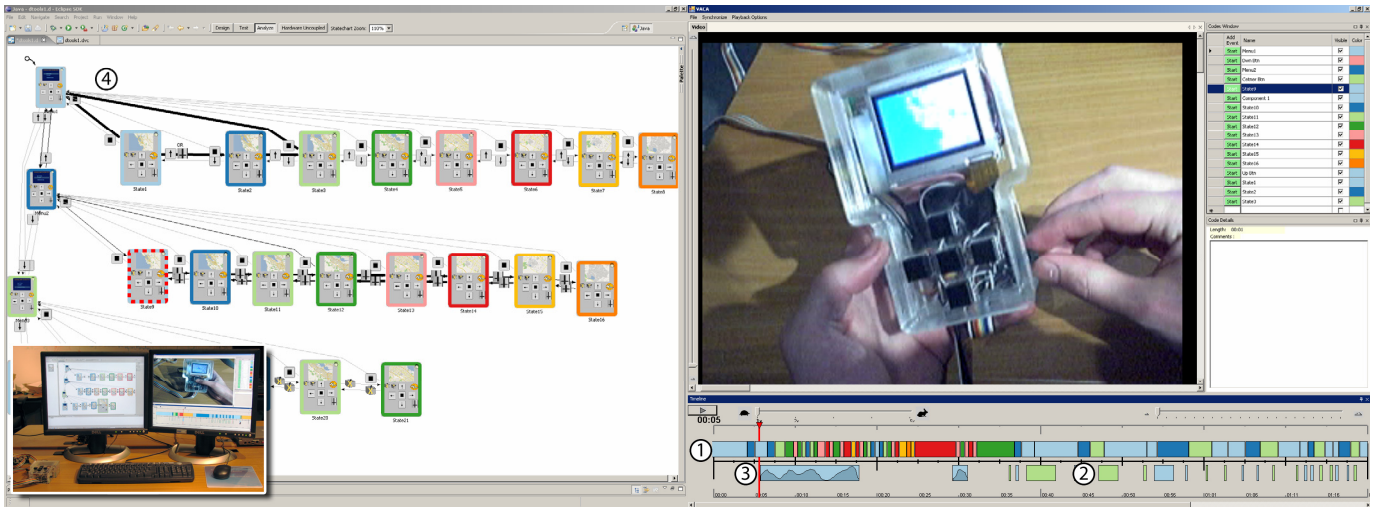
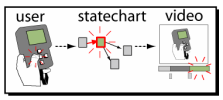


Figure 3. In Analysis mode, statechart and recorded video are synchronized and each can be used to access the other. *Inset:* simultaneous interaction with statechart and video editing is possible on a dual-screen workstation.

timestamp correlation between video and statechart (see Figure 3); this video functionality is implemented as an extension to the VACA video analysis tool [11]. d.tools automatically creates timeline annotations that capture the complete set of state transitions and device events at test time. After completing a test, at analysis time, the *video view* enables designers to access video segments from the statechart authoring environment and vice versa. This interaction allows for fast video query and enables accessing interaction code (the statechart) from a record of its execution (the video). The video view also enables comparison of multiple test sessions (see Figure 4).

Test

In test mode, d.tools executes user interactions just as in the design phase. Interactions with the physical prototype are reflected in the statechart, and outputs are reflected back in the device. Additionally, however, d.tools logs all device events and state transitions for video synchronization.



Switching to test mode initiates video capture. Then, as events and transitions occur they are displayed on the video view timeline in real-time. To clarify correspondence between statechart and video views, a consistent color-coding is used for states and hardware components in both. One row of the timeline corresponds to the state events for each independent subgraph of the statechart (see Figure 3, part 1), and an additional row displays hardware events. Three types of hardware events are displayed. *Instantaneous events*, such as a switch changing from on to off, appear as single slices on the timeline. *Events with duration*, such as the press and release of a button, show up as block segments (see Figure 3, part2). And *continuous events*, such as slider movements, are drawn as small line graphs of that event's value over time (see Figure 3, part 3).

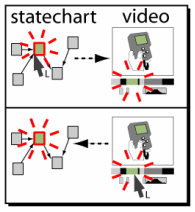
During the test session, the designer can make live annotations. d.tools offers dedicated buttons on an attached video control console to quickly mark positive (*e.g.,* interesting quotes) or negative (*e.g.,* usability problems) sections for later review. The experimenter's annotations are displayed in the video view as a separate row on the timeline.

Analyze

Analyze mode allows the designer to review the data from user test sessions. The video view and statechart editor function in tandem as a *multiple view interface* [6] into the test data to aid understanding of the relationship between the user experience and the interaction model underlying it. d.tools supports both s*ingle user analysis* and *group analysis*, which enables designers to compare data across multiple users.

Single User Analysis

Single user mode provides playback control of a test session video using a glanceable timeline visualization of the flow of UI state and data throughout that session. d.tools speeds up video analysis by enabling designers to work both from their interaction models to corresponding video segments *and* from video exploration to the statechart, facilitating analysis within the original design context. In addition to this dynamic search and exploration, the statechart also shows an aggregation of all user interactions during the test: the line thicknesses of state transitions are modified to indicate how often they were traversed (see Figure 3, part 4). This macro-level visualization shows which transitions were most heavily traversed and which were never reached.



Statechart to video: To access video from the interaction model, the designer can select a state in the statechart – the video annotations are automatically filtered such that only corresponding video clips are shown on the timeline and played. Similarly, the designer can *query by demonstration*: 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. Designers can also select multiple categories

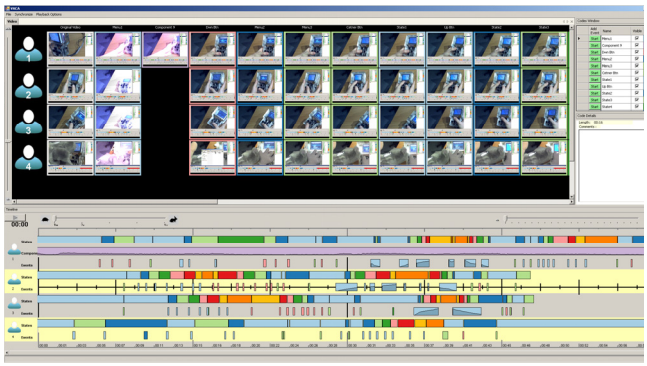


Figure 4. Group Analysis mode aggregates timeline and video data of multiple user sessions into one view.

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 re-enacting the interaction on the prototype itself.

Video to statechart: During video playback a dynamic visualization of transition history is displayed on top of the d.tools statechart. Active states get highlighted and d.tools also animates a real-time moving trail along the state transitions, indicating which state was previously active and which will be active next. This window into the chronology of interactions provides a visual reminder of context.

Group Analysis

Group mode collects all of the user capture sessions corresponding to a given statechart and displays them together. The timeline now aggregates flows for each user. The video window displays an **n** × **m** table of videos, with the rows corresponding to the **n** *users*, and the columns corresponding to the **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.

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.

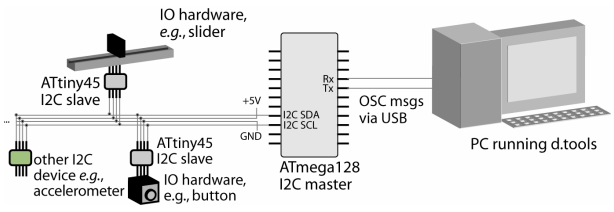


Figure 6. The d.tools architecture uses standardized, open protocols for hardware and PC communication.

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 PC (see Figure 5). Components plug into the interface board to talk on a common I2C serial bus (see Figure 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 implement I2C slave protocols. A USB connection to the host computer provides both 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 on a Crumb128 development board from chip45. I/O components use Atmel ATtiny45 microcontrollers. 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 subsystem provides 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 (*e.g.*, Purdy AND-TFT-25PAKIT). This screen is controlled by a secondary video card connected to a video signal converter.

Hardware Extensibility

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

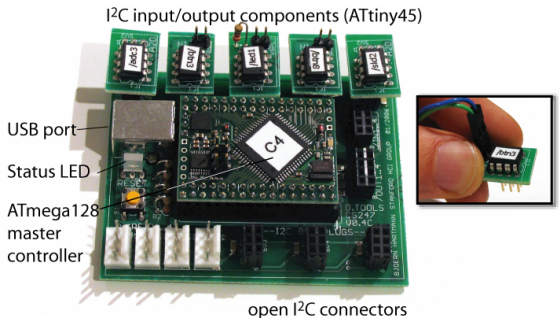


Figure 5. The d.tools board offers plug-and-play interfacing for I/O components.

box" is an important concern for physical computing tools. In the d.tools software, extensibility is provided by its Java hooks. In the d.tools hardware architecture (see Figure 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. OSC was chosen for its open source API, existing hardware and software support, and human readable addressing format (components have short path-like addresses – *e.g.*, buttons 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 PD and Max/MSP can receive sensor input from d.tools hardware. Connecting other physical UI toolkits to d.tools involves developing an OSC wrapper for them. As a proof of concept, we have written such a wrapper to connect Phidgets InterfaceKits to the d.tools software.

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 a source code file for the master microcontroller; in future work this configuration will 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 generic discrete outputs with on/off control 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 analog-to-digital converters.

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. d.tools uses the Eclipse Graphical Editing Framework (GEF) for graphics handling. d.tools file I/O is done via serialization to XML using XStream, which enables source control of device and statechart files in ASCII format using CVS or similar tools.

The video viewer is implemented in C# and uses Microsoft DirectShow technology for video recording and playback. Synchronization between the statechart and video views is accomplished by passing XML fragments over UDP sockets between the two applications. DirectShow was chosen because it allows synchronized playback of multiple streams of video.

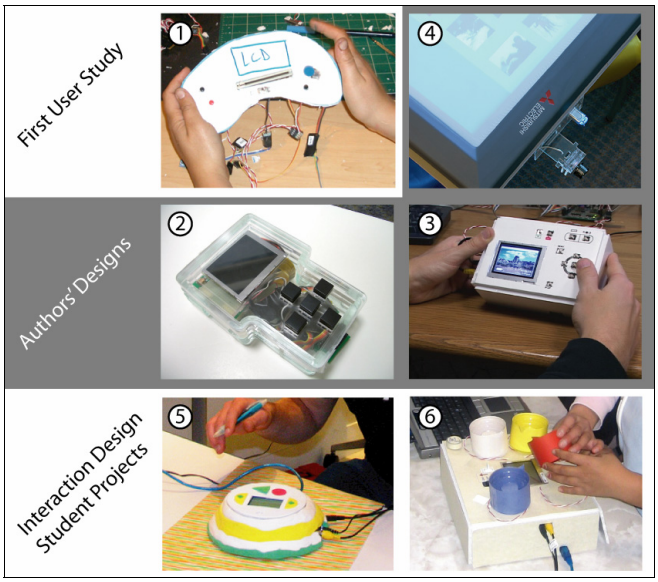


Figure 7. A selection of projects built with d.tools. (1) music player for children; (2) media player; (3) digital camera back; (4) tangible drawer for a tabletop display; (5) voice message trading pebble; (6) tangible color mixer.

EVALUATION AND ITERATION

In this section, we outline the methodological triangulation we employed to evaluate and iteratively refine our tool. Evaluations were carried out at different points during a seven-month period. First, an early version of the tool was tested by thirteen design students and professional designers in a first-use lab study to ascertain the use threshold. Second, the authors rebuilt prototypes of three existing devices and used it in a research project. Third, we made d.tools hardware kits available to students in a project-centric interaction design course at our university. Figure 7 shows some of the projects and devices built with d.tools as part of these evaluations. These evaluations addressed designing with d.tools and motivated the design-test-analyze integration; we reserve evaluation of test and analysis modes for future work.

Establishing threshold with a First Use Study

We conducted a controlled study of d.tools in our laboratory to assess the ease of use of our tool; the study group comprised 13 participants (6 male, 7 female) who had general design experience. 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.

Successes

Automatic recognition of hardware connections and visual statechart authoring were intuitive and well-received. Refining default behaviors through text properties and expressing functional independence in a statechart was less intuitive; 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.

In a post-test survey, 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).

Shortcomings discovered

One significant shortcoming discovered through the study was the lack of *software simulation* of an interaction model: the evaluated version did not provide 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.

Building existing and novel devices

To evaluate the expressiveness of d.tools' visual language, we recreated prototypes for three existing devices — an Apple iPod Shuffle music player, the back panel of a Casio EX-Z40 digital camera, and *Hinckley* et al.'s Sensing PDA [18]. We distilled the central functionality of each device and prototyped these key interaction paths.

Additionally, a novel project built with d.tools explored physical drawers as a file access metaphor for a shared tabletop display [29]. The first author built four drawer mechanisms mounted underneath the sides of a DiamondTouch interactive table. Opening and closing these drawers controlled display of personal data collections, and knobs on the drawers allowed users to scroll through their data.

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 statecharts taxing, a limitation shared by other visual authoring environments. An area for future work is the design of techniques that selectively display transitions based on the current context.

In building these systems, the percentage of *implementation-related* work (as opposed to graphic design or physical construction) was less than 30% of total prototyping time, enabling the prototyping to be driven by design concerns. In the drawers project, the presence of multiple independent drawers prompted the need for multiple concurrently active states as well as sensor data access from Java.

HCI Design Studio

We deployed the d.tools hardware and software to student project teams in a masters level HCI design course at our institution [25]. 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 this real-world deployment, we provided technical assistance, and tracked usability problems, bug reports and feature requests.

Successes

Students successfully built a range of innovative interfaces. Examples include a wearable "sound pebble" watch that allows children to record and trade secret messages with their friends, 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.

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 four mechanical tilt switches into their project.

Shortcomings discovered

Remote control of third party applications (especially Macromedia Flash) was a major concern – in fact, because such support was not integrated into the graphical tool, two student groups chose to develop their project with Phidgets [14], as it offers a Flash API. To address this need, we released a Java API for the d.tools hardware with similar connectivity and added Java execution ability to d.tools statecharts. We observed that student groups that used solely textual APIs ended up writing long-winded statechart representations using switch or nested conditional statements; the structure of their code could have been more concisely captured in our visual language.

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.

RELATED WORK

The d.tools system draws on previous work in two areas: prototyping and evaluation tools, and physical computing tools. This section summarizes how d.tools relates to each body of work.

Tool support for prototyping and rapid video evaluation

Most closely related to the design methodology embodied in d.tools is SUEDE [24], a design tool for rapidly prototyping speech-user interfaces. 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. SUEDE has been used and extended by several speech UI firms. SUEDE's open

architecture enabled these firms to extend the visual environment to support complex interactions. d.tools extends SUEDE's framework into a new application domain – physical user interfaces. It contributes a model for applying design-test-analyze to applications that transcend software development and adds integration of video analysis into the cycle. Like SUEDE, the d.tools system supports early-stage design activities.

This research also draws on prior work on structuring and accessing usability video of GUI tests through user interface event records; Hilbert and Redmiles present a comparative survey of such systems in [17]. Mackay described challenges that have inhibited the utility of video in usability studies, and introduced EVA, which offers researcher-initiated annotation at record time [27]. Hammontree *et al.* recorded test-generated event data to index video tapes and for comparing UI prototypes [15]. I-Observe by Badre et al. [5] enabled an evaluator to access synchronized UI event and video data of a user test by filtering event types through a regular expression language. While Weiler [34] suggests that proprietary 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 extends prior research and commercial work in three ways. First, it moves *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, it offers a *bi-directional link* between model and video where video can also be used to access and replay flow of control in the model. Third, it introduces *comparative evaluation* techniques for evaluating multiple user sessions.

Tool support for physical computing

The Phidgets [14] system introduced physical widgets: programmable ActiveX controls that encapsulate communication with USB-attached physical devices, such as a switch, pressure sensor, or servo motor. 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 (*e.g.*, for 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. d.tools' three extension points enable users with knowledge of mechatronics to add to the library of supported devices. Second, on the software level, d.tools targets *prototyping by designers*, not development by programmers. Textual APIs have too high a threshold and too slow an iteration cycle for rapid UI prototyping; they have not generally been adopted by product designers. The d.tools visual authoring environment contributes a lower threshold tool and provides stronger support for rapidly developing the "insides of applications" [28]. 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 [4, 26] integrates RFID buttons and other wired and wireless devices with C and the Macromedia Lingo language. Fluid integration with physical mock-ups is aided by the small form factor of the devices. Calder shares with d.tools its focus on design; it also describes desirable mechanical attachment mechanisms and electrical properties (battery-powered RF transceivers) of prototyping components. Like Phidgets, Calder's user interface is a textual API and only supports the design stage.

iStuff [7] extended the idea of programmatic control of physical devices to support wireless devices, a loose coupling between input and application logic, and the ability to develop physical interactions that function across an entire ubiquitous computing environment. iStuff, in conjunction with the Patch Panel [8], enables standard UIs to be controlled by novel inputs. iStuff targets room-scale applications. The size of hardware components make it infeasible to design integrated devices like information appliances.

The Lego Mindstorms Robotic Invention System [2] offers a visual environment based on control flow puzzle pieces to control sensors and actuators. While a benchmark for low-threshold authoring, Lego Mindstorms targets robotics projects; the programming abstractions are inappropriate for designing physical user interfaces. Mindstorms supports developing autonomous stored programs which runs counter to storyboard-driven development and eliminates designer access to model behavior at runtime.

Maestro [3] 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 the informal prototyping activities that d.tools targets. The availability of such a commercial tool demonstrates the importance of physical UI design tools to industry.

CONCLUSIONS AND FUTURE WORK

This paper introduced d.tools, a prototyping environment that lowers the threshold for creating functional physical prototypes and integrates support for prototype testing and analysis into the workflow. We have released d.tools to the design community as open source (see http://hci.stanford.edu/dtools/). Further work is underway to cut the tether to the PC by executing interaction models directly on embedded platforms to enable development of truly mobile prototypes. Finally, beyond individual tools, we are looking at creating entire design spaces that enable and support iterative design for ubiquitous computing.

ACKNOWLEDGEMENTS

We thank Mike Krieger and Scott Doorley for their help with video production; Merrie Morris for collaboration on the virtual drawers project; Arna Ionescu for discussions on professional design practice; Intel for donating the PCs used in this research; and the many product designers, students and study participants for working with d.tools and sharing their insights.

REFERENCES

1 *Java 2 Platform SDK: java.awt.Robot*, 2006. Sun Microsystems. http://java.sun.com/j2se/1.5/docs/api/java/awt/Robot.html

2 *LEGO Mindstorms Robotic Invention System*. http://www.mindstorms.lego.com/

3 *Maestro*, 2005. Cybelius. http://www.cybelius.com/products

4 Avrahami, D. and S. E. Hudson, Forming interactivity: a tool for rapid prototyping of physical interactive products, in *DIS: ACM Conference on Designing interactive systems*. ACM Press. pp. 141-46, 2002.

5 Badre, A. N., M. Guzdial, S. E. Hudson, and P. J. Santos. A user interface evaluation environment using synchronized video, visualizations and event trace data. *Software Quality Journal* **4**(2): Springer Netherlands. pp. 101-13, 1995.

6 Baldonado, M. Q. W., A. Woodruff, and A. Kuchinsky, Guidelines for using multiple views in information visualization, in *Proceedings of the working conference on Advanced visual interfaces*. 2000, ACM Press: Palermo, Italy.

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

8 Ballagas, R., A. Szybalski, and A. Fox. Patch Panel: Enabling Control-Flow Interoperability in Ubicomp Environments. In Proceedings of *PerCom 2004 Second IEEE International Conference on Pervasive Computing and Communications*: IEEE Press. p. 241–52, 2004.

9 Barragan, H., *Wiring: Prototyping Physical Interaction Design*, Interaction Design Institute, Ivrea, Italy, 2004.

10 Buchenau, M. and J. Fulton Suri. Experience prototyping. In Proceedings of *DIS: ACM Conference on Designing interactive systems*: ACM Press. pp. 424-33, 2000.

11 Burr, B. and S. R. Klemmer, VACA: A Tool for Qualitative Video Analysis, in *Extended Abstracts of CHI: ACM Conference on Human Factors in Computing Systems*. 2006, ACM Press.

12 Fails, J. and D. Olsen. A design tool for camera-based interaction. In Proceedings of *CHI: ACM Conference on Human Factors in Computing Systems*: ACM Press. pp. 449-56, 2003.

13 Greenberg, S. and M. Boyle. Customizable physical interfaces for interacting with conventional applications. In Proceedings of *UIST: ACM Symposium on User Interface Software and Technology*: ACM Press. p. 31–40, 2002.

14 Greenberg, S. and C. Fitchett. Phidgets: easy development of physical interfaces through physical widgets. In Proceedings of *UIST: ACM Symposium on User Interface Software and Technology*. p. 209–18, 2001.

15 Hammontree, M. L., J. Hendrickson, J., and B. W. Hensley. Integrated data capture and analysis tools for research and testing on graphical user interfaces. In Proceedings of *CHI: ACM Conference on Human Factors in Computing Systems*: ACM Press. pp. 431-32, 1992.

16 Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* **8**(3). pp. 231-74, 1987.

17 Hilbert, D. M. and D. F. Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys* **32**(4). pp. 384-421, 2000.

18 Hinckley, K., J. Pierce, M. Sinclair, and E. Horvitz. Sensing techniques for mobile interaction. In Proceedings of *UIST: ACM Symposium on User Interface Software and Technology*: ACM Press. pp. 91-100, 2000.

19 Houde, S. and C. Hill, What do Prototypes Prototype? in *Handbook of Human-Computer Interaction*, M. Helander, T.Ê. Landauer, and P. Prabhu, Editors. Elsevier Science B. V: Amsterdam, 1997.

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

21 Kelley, T., *The Art of Innovation*: Currency. 320 pp. 2001.

22 Kirsh, D. and P. Maglio. On distinguishing epistemic from pragmatic action. *Cognitive Science* **18**. pp. 513-49, 1994.

23 Klemmer, S. R., B. Hartmann, and L. Takayama. How Bodies Matter: Five Themes for Interaction Design. In Proceedings of *Design of Interactive Systems*, 2006.

24 Klemmer, S. R., A. K. Sinha, J. Chen, *et al.* SUEDE: A Wizard of Oz Prototyping Tool for Speech User Interfaces. In Proceedings of *UIST: ACM Symposium on User Interface Software and Technology*. p. 1–10, 2000.

25 Klemmer, S. R., B. Verplank, and W. Ju. Teaching Embodied Interaction Design Practice. In Proceedings of *DUX: Conference on Designing for User eXperience*: AIGA, 2005.

26 Lee, J., D. Avrahami, S. Hudson, *et al.* The Calder Toolkit: Wired and Wireless Components for Rapidly Prototyping Interactive Devices. In Proceedings of *DIS: ACM Conference on Designing Interactive Systems*: ACM Press. p. 167–75, August, 2004.

27 Mackay, W. E. EVA: an experimental video annotator for symbolic analysis of video data. *SIGCHI Bulletin* **21**(2): ACM Press. pp. 68-71, 1989.

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

29 Ringel Morris, M., *Supporting Effective Interaction with Tabletop Groupware*, Unpublished PhD, Stanford University, Computer Science, Stanford, CA, 2006.

30 Schön, D. A., *The Reflective Practitioner: How Professionals Think in Action*. New York: Basic Books. 374 pp. 1983.

31 Schön, D. A. and J. Bennett, Reflective Conversation with Materials, in *Bringing Design to Software*, T. Winograd, Editor. ACM Press: New York. pp. 171-84, 1996.

32 Schrage, M., Cultures of Prototyping, in *Bringing Design to Software*, T. Winograd, Editor. ACM Press: New York. pp. 191-205, 1996.

33 Schrage, M., *Serious play - How the world's best companies simulate to innovate*. Cambridge, MA: Harvard Business School Press. 245 pp. 2000.

34 Weiler, P. Software for the usability lab: a sampling of current tools. In Proceedings of *CHI: ACM Conference on Human factors in computing systems*: ACM Press, 1993.