CHAPTER 6 GAINING INSIGHT THROUGH FEEDBACK

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

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

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

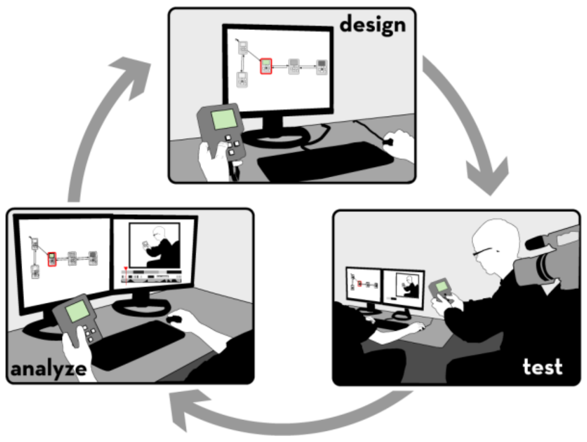


Figure 6.1: d.tools supports design, test & analysis stages through integration with a video editor.

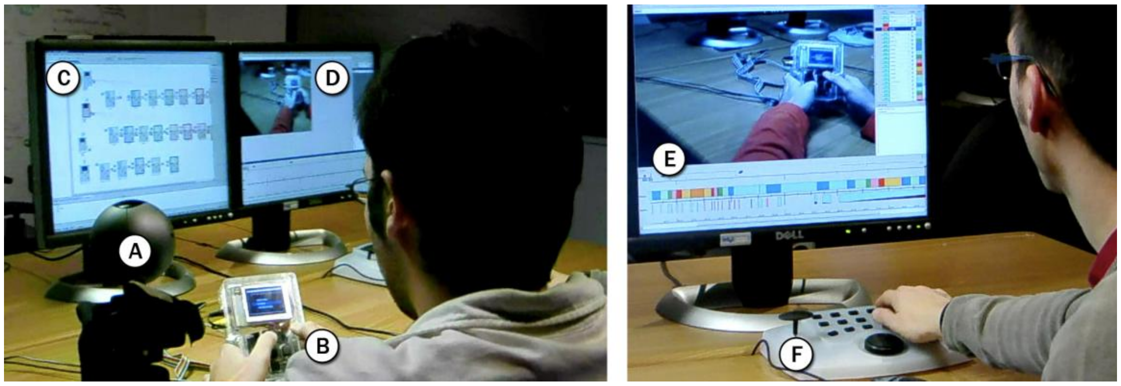


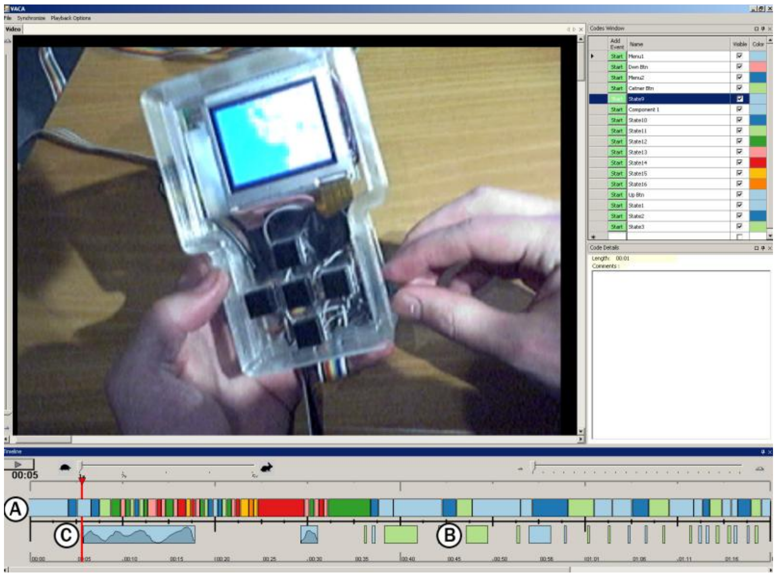
Figure 6.2: Testing a prototype built with d.tools: A camera (A) is aimed at the tester and the physical prototype (B), which is driven by a storyboard (C) in d.tools. Live video of the test is recorded in the video editor (D) and annotated with events and state changes (E). Designers can add additional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

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

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

6.1.1 TESTING PROTOTYPES

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



Codes Window

	Add Event	Name	Visible	Color
▶	Start	Menu1	✓	🟦
	Start	Dwn Btn	✓	🟥
	Start	Menu2	✓	🟦
	Start	Cetner Btn	✓	🟩
	Start	State9	✓	🟦
	Start	Component 1	✓	🟦
	Start	State10	✓	🟦
	Start	State11	✓	🟩
	Start	State12	✓	🟩
	Start	State13	✓	🟥
	Start	State14	✓	🟥
	Start	State15	✓	🟨
	Start	State16	✓	🟧
	Start	Up Btn	✓	🟦
	Start	State1	✓	🟦
	Start	State2	✓	🟦
	Start	State3	✓	🟩

Code Details

Length: 00:01

Comments :

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

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

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

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

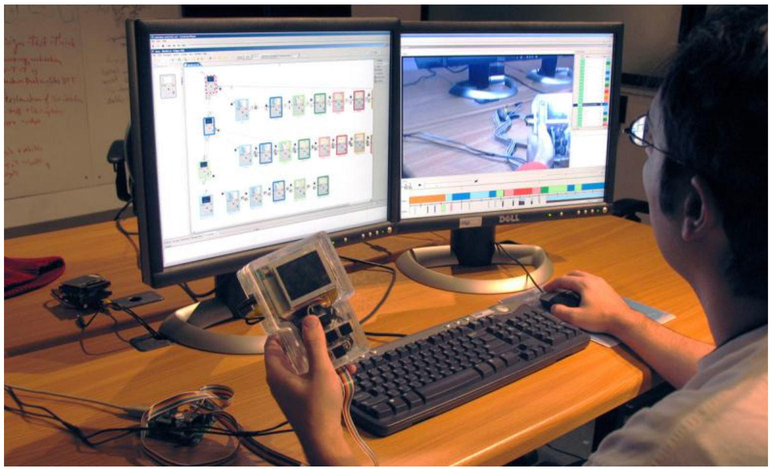


Figure 6.4: In analysis mode, a dual-screen workstation enables simultaneous view of state model and video editor.

6.1.2 ANALYZING TEST SESSIONS

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

6.1.2.1 Single User Analysis

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

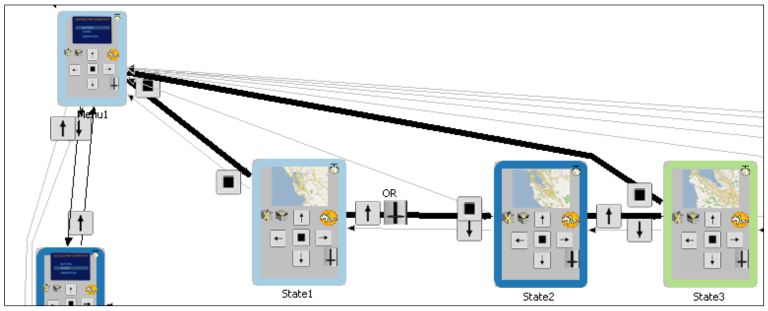


Figure 6.5: Line thickness in analysis mode shows how many times a given transition was taken.

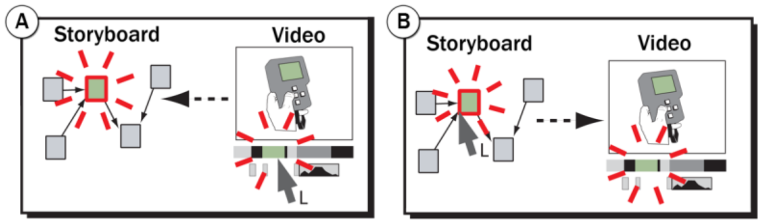


Figure 6.6: Two query techniques link storyboard and video. **A**: Selecting a video segment highlights the state that was active at that time. **B**: Selecting a state in analyze mode highlights the corresponding video segment(s).

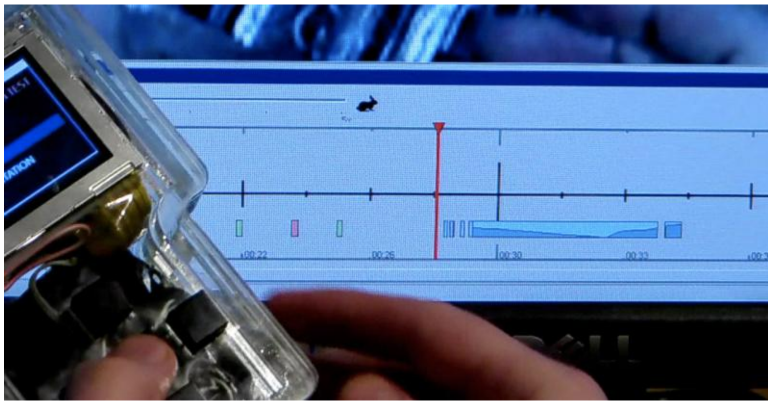


Figure 6.7: Designers can *query by demonstration*: Generating input events in analyze mode filters recorded video so that only those sections where similar events were received are shown.

VIDEO TO STORYBOARD

During video playback, a dynamic visualization of transition history is displayed in the d.tools storyboard. The state that was active at the current point in time of the video is highlighted. d.tools also animates moving trail along the state transitions, indicating which state a user was coming from, and which state will become active next. This window into the chronology of interactions provides a visual reminder of context. Selecting any state segment in the timeline moves the play head to that segment and, as a result, highlights the corresponding state in the storyboard (Figure 6.6A)

STORYBOARD TO VIDEO

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

6.1.2.2 Group Analysis

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

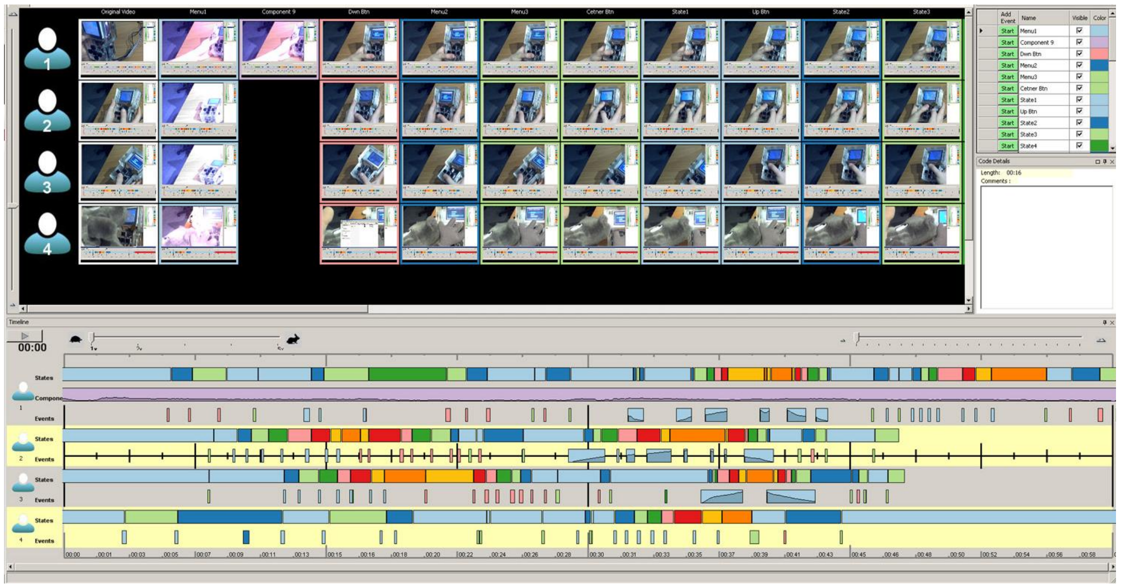


Figure 6.8: Group analysis mode aggregates video and event data of multiple user sessions into one view.

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

6.1.3 IMPLEMENTATION

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

6.1.4 LIMITATIONS & EXTENSIONS

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

6.1.4.1 No Support for Quantitative Analysis

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

Two complementary strategies to support more quantitative analysis suggest themselves: the first is to automatically extract data from the recorded test (e.g., state dwell statistics — how long did users spend in each state, which states were never reached). The second is to provide the reviewer with better tools to conduct such analyses manually. An example of a more fine-grained analysis approach is Experiscope [97], a tool for analyzing user tests of mouse- or stylus-based interaction techniques. Experiscope can both visualize input event data well as produce aggregate reports of event frequency and duration. As an initial step into this direction, d.tools visualizes how many times a transition was taken by changing transition line thickness in the diagram. However, it is not currently possible to extract the precise number of times the transition was taken, or to derive a similar figure for the number of times a state was active during a test.

6.1.4.2 Limited Visibility of Application Behavior During Test

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

6.1.4.3 Cannot Compare Multiple Prototypes in Analysis Mode

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

Tohidi and Buxton [243] note that testing multiple prototypes is preferable to testing a single prototype, since users will feel less pressured to be “nice” to experimenters and can draw comparisons between prototypes. In addition, if prototypes are more refined and the designer has concrete hypotheses in mind, formal comparative testing is required to support or reject these hypotheses. For traditional GUI interactions, tools that support such comparative analysis of alternatives exist. Experiscope [97] enables testers to visually compare event traces of multiple treatment conditions side-by-side. However, existing tools such as Experiscope do not link the recorded trace back to the source of the application being tested. It is an open question how tools can show video, event traces, and software models for multiple alternative designs simultaneously without overwhelming the designer with complexity.

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

6.1.4.4 Limited Query Language

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

6.1.4.5 Interaction Techniques Have Not Been Formally Evaluated

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

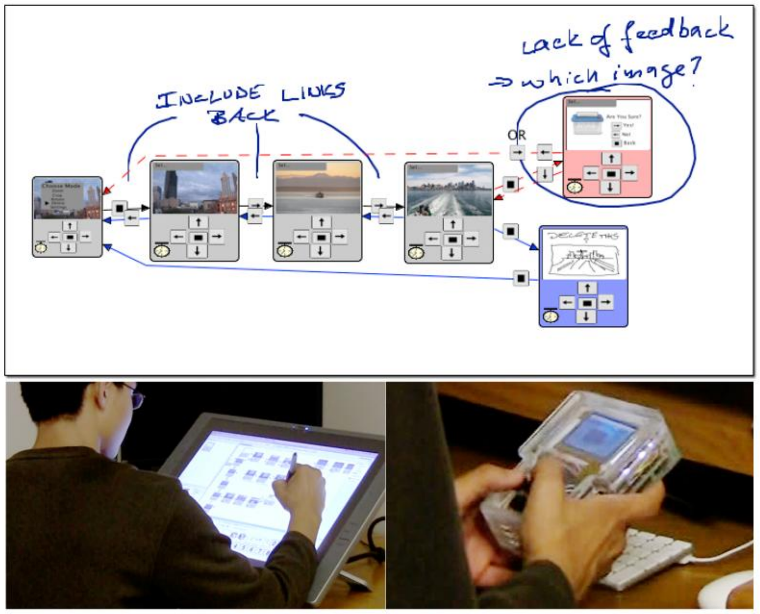


Figure 6.9: d.note enables interaction designers to revise and test functional prototypes of information appliances using a stylus-driven interface to d.tools.

6.2 CAPTURING FEEDBACK FROM OTHER DESIGNERS: D.NOTE

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

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

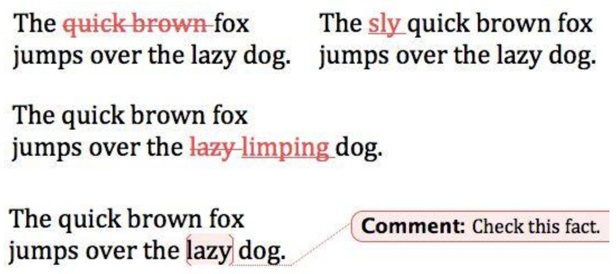
This section demonstrates how three primary text revision techniques can be applied to interaction design: commenting, tracking changes, and visualizing those changes. It also introduces revision tools unique to interaction design: immediate testing of revisions and proposing alternatives. The novel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

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

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

6.2.1 REVISION PRACTICES IN OTHER DOMAINS

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



The ~~quick brown~~ fox
jumps over the lazy dog.

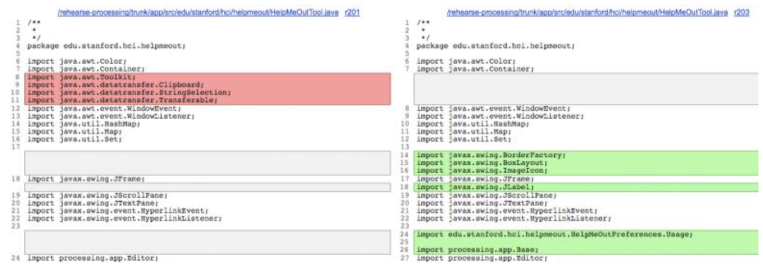
The sly quick brown fox
jumps over the lazy dog.

The quick brown fox
jumps over the ~~lazy~~ limping dog.

The quick brown fox
jumps over the lazy dog.

Comment: Check this fact.

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



/rehearse-processing/trunk/app/src/edu/stanford/hci/helpmeout/HelpMeOutTool.java r201

```
/**
 *
 */
package edu.stanford.hci.helpmeout;

import java.awt.Color;
import java.awt.Container;
import java.awt.Toolkit;
import java.awt.datatransfer.Clipboard;
import java.awt.datatransfer.StringSelection;
import java.awt.datatransfer.Transferable;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

import javax.swing.JFrame;

import javax.swing.JScrollPane;
import javax.swing.JTextPane;
import javax.swing.event.HyperlinkEvent;
import javax.swing.event.HyperlinkListener;

import processing.app.Editor;
```

/rehearse-processing/trunk/app/src/edu/stanford/hci/helpmeout/HelpMeOutTool.java r203

```
/**
 *
 */
package edu.stanford.hci.helpmeout;

import java.awt.Color;
import java.awt.Container;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

import javax.swing.BorderFactory;
import javax.swing.BoxLayout;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JScrollPane;
import javax.swing.JTextPane;
import javax.swing.event.HyperlinkEvent;
import javax.swing.event.HyperlinkListener;

import edu.stanford.hci.helpmeout.HelpMeOutPreferences.Usage;

import processing.app.Base;
import processing.app.Editor;
```

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

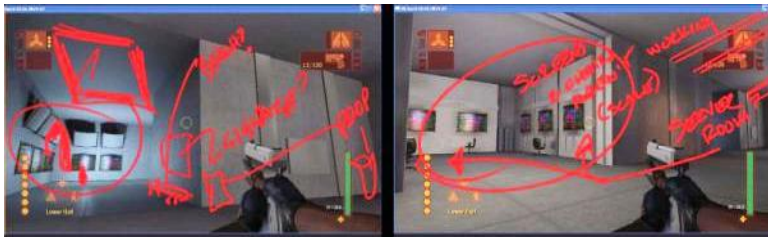


Figure 6.12: Video game designers draw annotations directly on rendered still images (from [55:p. 179]).

TEXT DOCUMENTS

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

SOURCE CODE DOCUMENTS

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

VISUAL MEDIA

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

DESIGN PRINCIPLES

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

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

6.2.2 A VISUAL LANGUAGE FOR REVISING INTERACTIONS

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

6.2.2.1 Revising Behavior

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

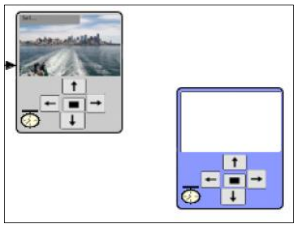


Figure 6.13: States added during revision are rendered in blue.

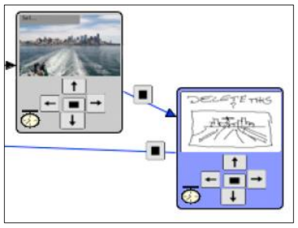


Figure 6.14: New screen graphics can be sketched in states.

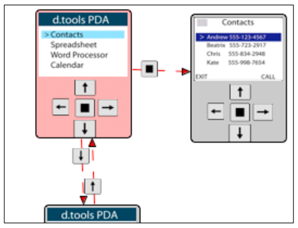


Figure 6.15: State deletions are rendered in red. Connections are marked as inactive.

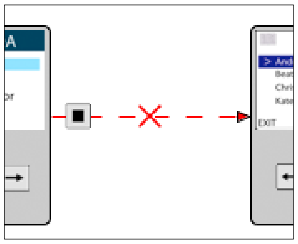


Figure 6.16: Transition deletions are marked with a red cross and dashed red lines.

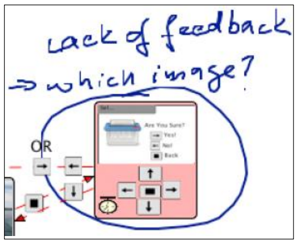


Figure 6.17: Comments can be attached to any state.

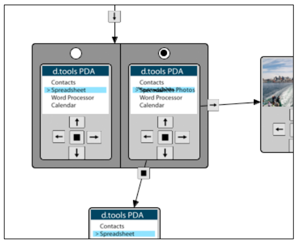


Figure 6.18: Alternative containers express different options for a state.

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

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

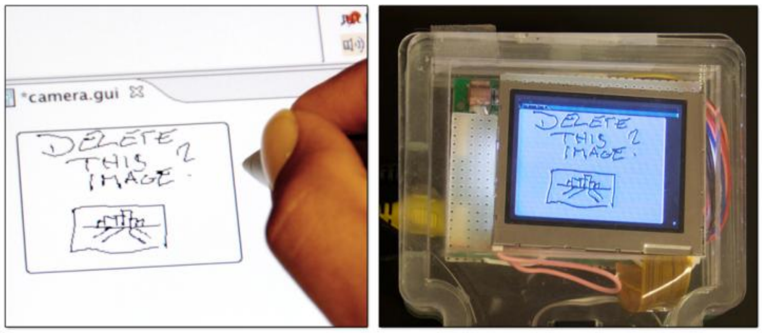


Figure 6.19: Sketched updates to screen content are immediately visible on attached hardware.

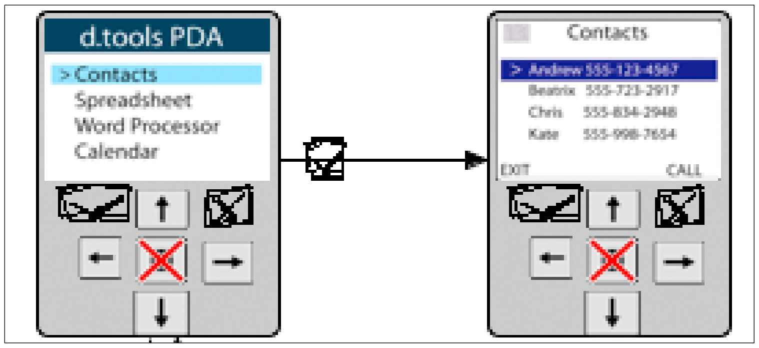


Figure 6.20: Changes to the device configuration are propagated to all states. Here, one button was deleted while two others were sketched in.

6.2.2.2 Revising Appearance

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

6.2.2.3 Revising Device Definition

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

6.2.2.4 Commenting

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

6.2.2.5 Proposing Alternatives

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

6.2.3 SCENARIO

The following scenario summarizes the benefits d.note provides to interaction design teams. Adam is designing a user interface for a new digital camera with on-camera image editing functions. To get feedback, he drops his latest prototype off in Betty's office. Betty picks up the camera prototype, and tries to crop, pan and color balance one of the pictures that Adam preloaded on the prototype. She notices that exiting to the top level menu is handled inconsistently in different screens. She opens up the d.tools diagram for the prototype and, with d.note enabled, changes the transitions from those screens to the menu state. She next notices that the image delete functionality is lacking a confirmation screen – images are deleted right away. To highlight this omission, Betty creates a new state and sketches a rudimentary confirmation dialog, which she connects to the rest of the diagram with new transitions so she can immediately test the new control flow. Betty is not convinced that the mapping of available buttons to crop an image region is optimal. She selects the crop state and creates an alternative for it. In the alternative, she redirects button input and adds a comment for Adam to compare the two implementations. She also thinks that the current interface for balancing colors via RGB sliders is cumbersome. Since she does not have time to change the implementation, she circles the corresponding states and leaves a note to consider using an alternative color space instead.

6.2.4 THE D.NOTE JAVA IMPLEMENTATION

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

6.2.4.1 Specifying Actions Through Stylus Input

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

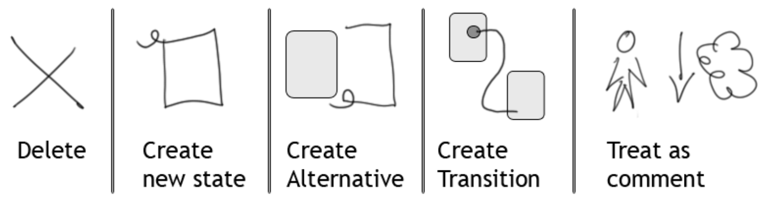


Figure 6.21: The d.note gesture set for stylus operation. Any stroke not interpreted as one of the first four actions is treated as a comment.

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

6.2.5 EVALUATION: COMPARING INTERACTIVE & STATIC REVISIONS

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

6.2.5.1 Study 1: Authoring Revisions

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

METHOD

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

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

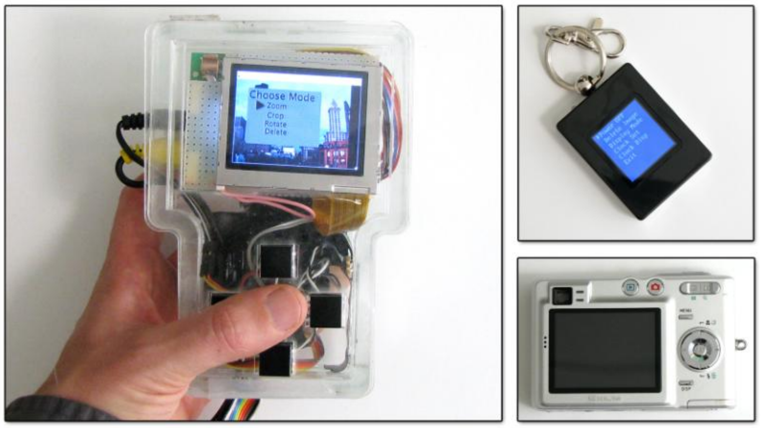


Figure 6.22: Participants were given a prototype device with a color display and button input. They were asked to revise designs for a keychain display and a digital camera, both running on the provided device.

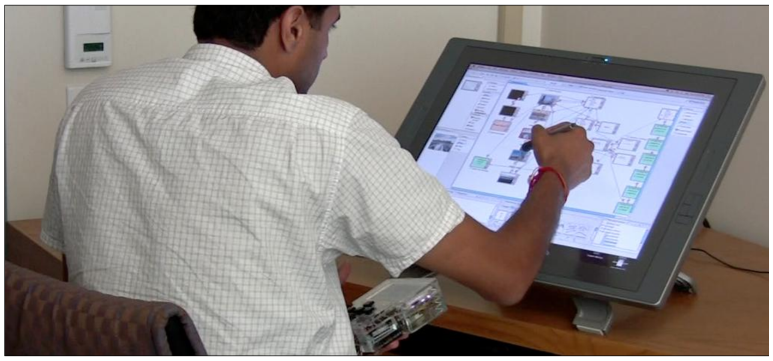


Figure 6.23: Participants in study 1 revised d.tools designs on a large tablet display.

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

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

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

Particpant	Task	Written Comments	Sketches on Canvas	Drawn Transition Arrows	Other Arrows	Drawing/Modifying Screen	Circled States	Circled Groups of States	Circled Transitions	Circled components	Circled State Graphics	Crossed out items	Created States	Created Alternatives	Created Transitions	Deleted Transitions	Deleted States
		Without D.Note															
4	C	||||||||	||	|||||||	||||	|			|	||	||						
5	C	||||||||||||||		|||	||||||	|	|||	||	|								
7	C	||||||||||	|		|||	||											
9	C	|||||||||		||||||||	||		||				||						
10	C	||||||||||||||		||||	|||		|	||	|								
12	C	|||||||||||||		|||||||	|||||||||	||		|	||||		|||||						
1	K	|||||	||||	||||||	|	|								N/A			
2	K	|||||		|||	|||||||		||||	|		||	|||	|||					
3	K	||||||||||	|	|	||	||	||		||								
6	K	||||||||||		||||	||||	|			||			|					
8	K	|||||||||||||			||||||||||	|											
11	K	||||||		|	|||				|								
		With D.note															
1	C					||							||	|	|||||||||		
2	C	|||			|		|	|						|	||||||||||||	|||	
3*	C	||||															
6	C					||||||							||||	||	|||||||||		
8	C	|		||							|				||||||||	|||||||	
11	C	|				||||		|					|||||	|	|||||||||||||	|	
4	K					||							|		||||	|||||	
5	K	|||||			|||			||									
7	K	||||			|			|				|||||||				||||	||
9	K	|			|			|							||||||||||	||	
10	K	||||||||		|	||||		|||	||							|||||||	||	||
12	K	|			|										||||||	||||	||
		Inked Annotations											D.Note Revisions				

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

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

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

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

RESULTS

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

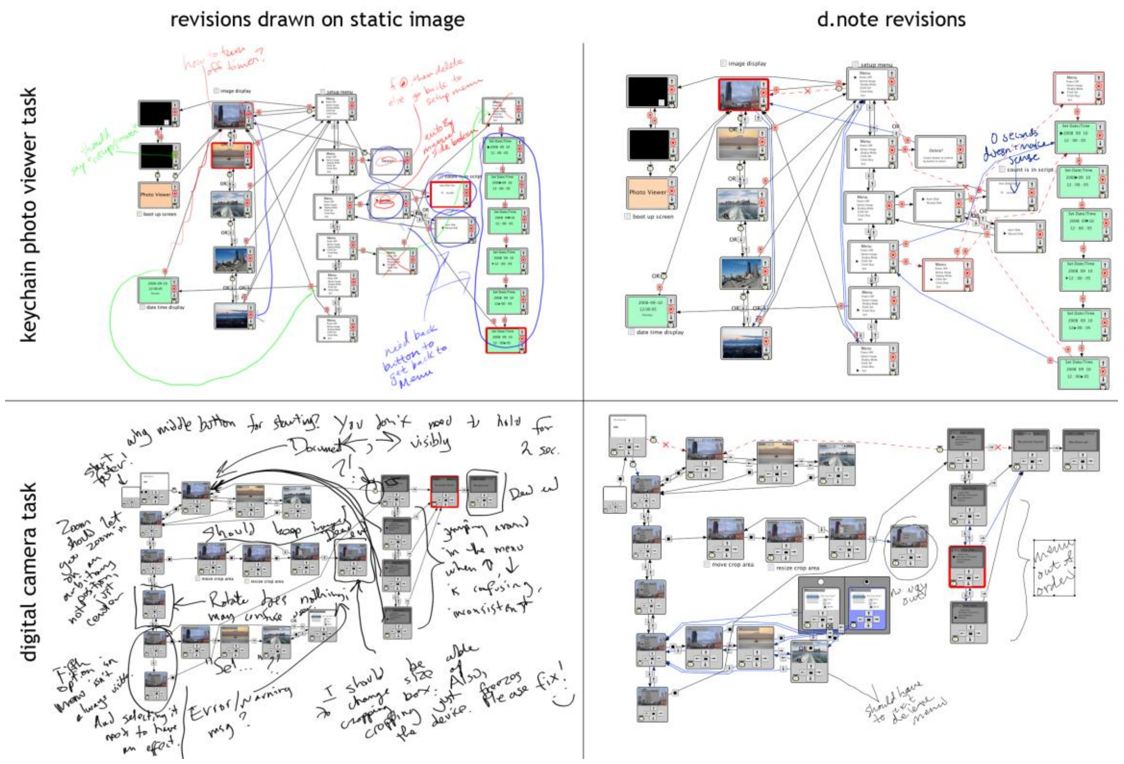


Figure 6.24: Two pairs of revision diagrams produced by our study participants. Diagrams produced with Sketchbook Pro in the control condition are shown on the left; diagrams produced with d.note are shown on the right.

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

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

DISCUSSION

Why did participants write less with d.note? One possibility is that that users wrote more with Sketchbook because it was easier to do so (Sketchbook is a polished product, d.note a research prototype). To the extent this is true, it provides impetus to refine the d.note implementation, but tells us little about the relative efficacy of a static and dynamic approach to design revision.

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

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

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

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

6.2.5.2 Study 2: Interpreting Revisions

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

METHOD

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

RESULTS

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

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

DISCUSSION

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

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

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

Perceived advantages of d.note for interpreting revisions		Perceived disadvantages of d.note for interpreting revisions	
Changes are concrete and specific	IIII	Visual clutter in regions of dense changes	IIIII
Contains proposed solutions	III	Hard to glean motivation for changes	IIII
Can automatically apply changes	I	Hard to keep track which changes were already examined	II

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

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

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

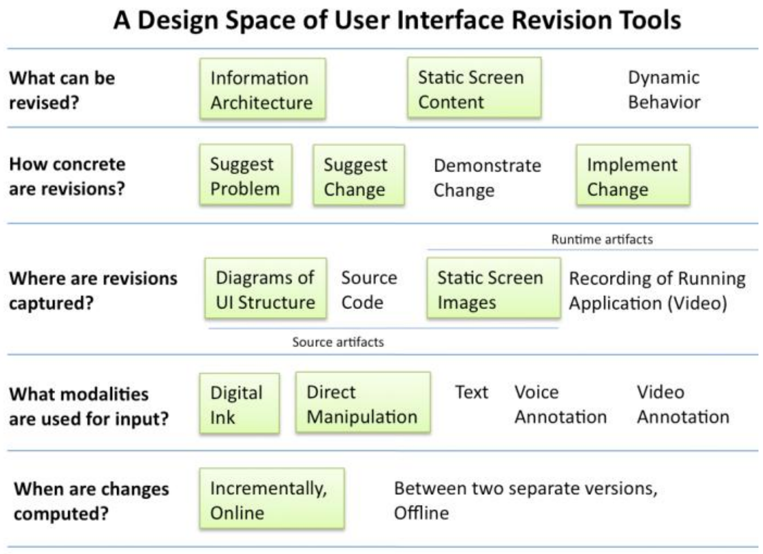
Visual complexity of annotated diagrams: Visual programs become harder to read as the node & link density increases. Showing added and deleted states and transitions simultaneously in the diagram sometimes yielded "visual spaghetti": a high density of transition lines made distinguishing and following individual lines hard. The connection density problem becomes worse when state alternatives are introduced because each alternative for a state has an independent set of outbound transitions.

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

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

6.2.6 LIMITATIONS & EXTENSIONS

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



A Design Space of User Interface Revision Tools

What can be revised?	Information Architecture	Static Screen Content	Dynamic Behavior		
How concrete are revisions?	Suggest Problem	Suggest Change	Demonstrate Change	Implement Change	
			Runtime artifacts		
Where are revisions captured?	Diagrams of UI Structure	Source Code	Static Screen Images	Recording of Running Application (Video)	
	Source artifacts				
What modalities are used for input?	Digital Ink	Direct Manipulation	Text	Voice Annotation	Video Annotation
When are changes computed?	Incrementally, Online	Between two separate versions, Offline			

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

6.2.6.1 *Cannot Comment on Dynamic Behavior*

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

6.2.6.2 *Cannot Revise Dynamic Behavior*

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

6.2.6.3 How To Support Identified Revision Principles for Source Code?

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