Interaction History Support for Web Applications

Peter Simonyi

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

Master of Computer Science

Carleton University
Ottawa, Ontario
September 2015

© Copyright 2015, Peter Simonyi

Table of Contents

List of Figures	v
Abstract	ix
Acknowledgements	x
Chapter 1 Introduction	1
1.1 Goal	4
Chapter 2 Background	5
2.1 History systems	5
2.1.1 Traditional Undo	5
2.1.2 Back buttons	6
2.1.3 File-like saving	8
2.1.4 Source control	10
2.1.5 Manual version control	11
2.2 Managing and visualizing history	11
2.2.1 Linear models	11
2.2.2 Graphs	12
2.3 Sources of history management	13
Chapter 3 Design	15
3.1 Requirements	15
3.2 Visualizing history	16
3.3 Applications	19
3.4 Ra user experience	21
3.4.1 Visual walkthrough	25
3.4.2 Persistence and collaboration	26
Chapter 4 Technology Platform	34
4.1 General approach	34
4.2 Alternatives	35

4.3	The DOM	36
4.4	Technical architecture	37
4.4.1	Storing state	39
4.4.2	Restoring states	39
4.4.3	The sidebar	40
4.4.4	Sol: a server for persistence and collaboration	41
4.4.5	Technology choices	41
4.5	Ra API	42
4.5.1	State requirements	42
4.5.2	Core	43
4.5.3	Suggest restore points	43
4.5.4	Shared server	44
4.6	Practical considerations	44
4.6.1	Supporting V8	44
4.6.2	Supporting other architectures	45
Chapter 5	Usability Evaluation	48
5.1	Method	48
5.2	Participants	48
5.3	Equipment	49
5.3.1	System setup	49
5.3.2	Applications	49
5.4	Series changes	52
5.5	Tasks	53
5.5.1	Maze introduction	54
5.5.2	Maze with choice of targets	54
5.5.3	Travelling Salesman maze	55
5.5.4	Parallel coordinates	56
5.5.5	Parallel coordinates presentation	57
5.6	Observations	59

5.6.1	Maze tasks	59
5.6.2	Parallel coordinates tasks	64
5.6.3	Hardware	66
5.6.4	Other comments	67
5.7	Questionnaire responses	68
5.8	Interpretation	70
Chapter 6	Discussion and Conclusion	74
6.1	Discussion	74
6.1.1	Capturing state	74
6.1.2	Mutability and source control	75
6.1.3	Kinds of history tasks	75
6.1.4	Merges	77
6.2	Conclusion	77
6.2.1	Summary	77
6.2.2	Contributions	78
6.2.3	Limitations and future work	78
Bibliography		79
Appendix A	Usability evaluation documents	82

List of Figures

1.1	ACH Walkthrough, an intelligence analysis tool. This screenshot shows an interactive parallel coordinates graph. (Image from [29].)	2
1.2	A study participant using our prototype software Ra (the dark blue sidebar) with an interactive data analysis tool.	3
2.1	WorldWideWeb, the original prototype Web browser and editor, had navigation controls in a window/menu.	6
2.2	Mosaic, an early Web browser, had back and forward buttons near their modern position.	7
2.3	The Undo History window for the Inkscape vector graphics editor, showing some sample history. Time flows down. The highlighted entry marks the state currently shown in the editor.	12
3.1	gitk displays the DAG of the Git repository	17
3.2	The <code>git log</code> command can also generate a DAG visualization of the Git repository.	18
3.3	A simple merge. James Antill and Seth Vidall began working in parallel, both starting from James Antill’s “Huge memory decrease” commit, when the paths diverge. Then Seth Vidal merged both their work before going on to commit “die tabs, die” to the now-unified history.	18
3.4	Large repositories, such as the one for Git, can have complicated history. gitk has added colours to the visualization to help track each branch.	19
3.5	Adventures of the Green Square: a simple maze, the first application made to work with Ra.	20
3.6	TodoMVC React: another application we modified to use Ra. .	21

3.7	The parallel coordinates application (showing the months Ottawa had at least some snow and the temperature never dropped below zero – a rare occurrence). The balloon popup for a node is showing the label and timestamp. The user has already returned to that state and started a new branch; the “Load this” button would let them do so again.	22
3.8	Ra divides the page to make room for a sidebar.	23
3.9	An automatic snapshot	24
3.10	An app-suggested snapshot	24
3.11	A starred snapshot	25
3.12	A simple maze and a fresh history tree. Since nothing has happened yet, it’s pretty empty.	26
3.13	As we move the token in the maze (drawing a path behind it), Ra is creating automatic snapshots on the side.	27
3.14	Hm...should we go back to the left a bit, or try to keep going right? We’ll save this spot for later by writing a label and pressing Ra’s “Save” button.	27
3.15	Having marked the decision point with a star, we tried the right path. But it’s got a rather big wiggle; this isn’t looking like the best path any more.	28
3.16	We’ve gone back to the starred snapshot to try the left-hand fork this time. Ra still remembers that big wiggle we didn’t like, so we can go back if this doesn’t pan out. (Also, notice our label from figure 3.14?)	28
3.17	The automatic snapshots of our new path also go under the star, but in a new branch. This path looks much better from here.	29
3.18	We can head straight to the finish. Our path is 32 segments long.	29
3.19	But was the path we abandoned really inferior? Let’s go back and bring it to completion.	30

3.20	The new automatic snapshots continue in the old branch as we continue from where we left off. Indeed, this was the worse path, by two segments – 34 instead of 32.	30
3.21	We’ll select the better path as our final solution.	31
3.22	Collaboration enabled by Sol. Two users share a history tree, but each has an independent current state.	32
4.1	The components of a system using Ra. The application’s model state is actually made up of proxies for the real state objects stored as Ra’s current state, so the two components remain synchronized. The current state, in turn, is expanded from the tree of serialized historical states. Sol is a separate server that keeps the history tree synchronized between clients.	38
5.1	One configuration of the maze application.	50
5.2	The parallel coordinates application.	51
5.3	Comparison of changes made between the series: <i>Left</i> : series 1 savepoint; <i>Right</i> : series 2 savepoint.	53
5.4	Our demonstration of Ra used a very simple maze.	54
5.5	One participant’s solution to the first maze.	55
5.6	This series 1 participant found two likely solutions to the second maze and marked them with big dots; the current path is two segments longer than the other solution.	56
5.7	One participant’s solution to the third maze. The current state has been marked with a star.	57
5.8	The weather data (series 2) parallel coordinates graph. This participant has selected the summer months with very high total precipitation.	58
5.9	One state in a participant’s presentation, showing the car data parallel coordinates graph.	58
5.10	The history tree can get large enough to require scrolling. . . .	63

5.11	The touchscreen layers (not to scale) seen from the side show that the detected touch position and the point the user sees are different.	66
5.12	Difficulties using the touchscreen. A participant has accidentally started text-selection mode instead of pressing the “Load this” button, likely because their tap was interpreted as a long swipe.	67
5.13	Box-plot of usability questionnaire responses. Note that the statements marked “(Not)” have been inverted so that all the positive responses are on the left; for example, most participants agreed that there were too many dots, so this has been reversed to disagreeing with the inverse statement.	71

Abstract

All users of complex software make decisions that they may later wish to change. Many computer systems have tools to support this need for revision, such as the undo command. However, the common history tools (like undo) do not support exploratory, epistemic interaction well. And there are common, non-specialized tasks that are difficult in common computer systems, but would be much easier with improved support for managing interaction history. Desktop computing environments have well-established norms for how undo works, but there is room to explore this in newer computing environments, such as the Web and surface computing, as their design culture has not stabilized to the same extent. We argue that history tracking needs to be more accessible to users.

We developed a prototype JavaScript library for Web applications that lets users keep a history of all their interaction states, including those that would be discarded by using a traditional stack-model undo system. The history is presented to users in a tree structure similar to the model used in source control software. We ran a usability study of our system with two applications designed to encourage the kind of exploratory behaviour we wanted to support. We identified usability improvements that could be made, but the study suggests that this kind of system could be generally useful even in non-specialized fields.

Acknowledgements

This thesis would not have happened without the support and encouragement of my supervisor, Prof. Robert Biddle. Thank you for your help. Thanks also to Jeff Wilson and Judith Brown for your help on this project and interesting discussions in the lab.

—Peter

Chapter 1

Introduction

All users of complex software make decisions that they may later wish to change. Software can support this need to revisit past decisions by keeping past versions of the application’s state that the user can go back to. There are several mechanisms for maintaining and presenting this history. Since early in the history of desktop computing in the 1970s and 80s, most user applications have provided users with an “undo” command to revert the most recent change. This capability has generally been improved over time, as more modern desktop applications have more memory and storage available, and undo has become a standard expected feature.

However, new computing environments, such as the World Wide Web, phones, tablets, and large touchscreen surfaces, do not share the same design culture and provide similar undo capabilities only rarely. The most common cases when undo is available in these environments are applications that mimic well-established desktop applications, such as word processors and spreadsheets. We suggest that the general lack of undo – and lack of consistency in its presentation, when an undo command is available – in these new environments is an oversight, perhaps due to the relative immaturity of the platforms, and that users still need these capabilities. This does have one advantage for our research: the lack of established norms allows us the freedom to experiment with different designs for history features without running counter to users’ ingrained expectations.

Following an interesting series of experiments with people playing Tetris, Kirsh and Maglio [13] concluded that users intentionally and rationally perform actions that do not directly contribute to the final state of their work – these actions are later reverted but were not mistakes. Their participants, including the advanced players, were using game actions to perform intermediate computations instead of just moving the Tetris pieces toward their final positions: for example, participants would rotate a piece instead of doing the rotation mentally, or rotate a piece that was partly off-screen

Figure 1.1. ACH Walkthrough, an intelligence analysis tool. This screenshot shows an interactive parallel coordinates graph. (Image from [29].)

to reveal its identity before it had fully appeared.

As a result, Kirsh and Maglio divide (non-errorneous) user interactions into two categories: *pragmatic* actions are those that actually move the user closer to their goal, and *epistemic* actions are those that help the user learn about their situation, exploring to gather information that is either “hidden or hard to compute mentally”. So interaction history systems should be designed to support epistemic interaction as well as error recovery.

Touchscreens make epistemic interaction more compelling. Lee *et al.* [15] argue that touchscreens enable a kind of directness even more direct than the Direct Manipulation described by Shneiderman [26], since Shneiderman was assuming the use of a mouse and keyboard. Large touchscreens also enable new kinds of co-located collaboration possibilities [7]. Sharing a touchscreen is much easier than sharing a keyboard and mouse. Touch interfaces are changing the kind of software we make, and the new types of applications need to support epistemic interaction.

That makes large touchscreens a good interface for collaborative data analysis work, using systems such as Wilson *et al.*’s ACH Walkthrough [29] (see figure 1.1), which implements the Analysis of Competing Hypotheses intelligence analysis technique.

Figure 1.2. A study participant using our prototype software Ra (the dark blue sidebar) with an interactive data analysis tool.

Tools like ACH improve analysis work by reducing the impact of analysts’ cognitive biases. ACH in particular is meant to reduce confirmation bias, where analysts will unknowingly focus on the evidence that supports their pet hypotheses rather than evaluating all evidence fairly.

Another cognitive process that can interfere with effective analysis is *satisficing* [28], in which an analyst will stop when they have reached an answer that seems “good enough”. On its own, this is rational and acceptable as long as the threshold is set right. The problem is that software may impose additional costs to further exploration – at worst, further exploration requires starting all over again – and that lowers the “good enough” threshold.

This is the problem of *premature commitment* from the Cognitive Dimensions of Notations framework [5]. If you have reached a solution but want to try something else, you must decide whether it’s worth the effort to just get back what you had if the “else” isn’t any better. Without a system for storing interaction history, the user is constrained to repeat the steps to achieve the old solution, or else execute the inverse

of all actions taken since then. This may be a significant cost to exploration.

1.1 Goal

We wanted to develop a system to provide users with access to all their historical interaction states, including those that would be discarded by a traditional stack-model undo system. Such a system should encourage more epistemic interaction by allowing users to return to known-good states after exploring and reduce premature commitment and the urge to satisfice by freeing users from the risk of losing good work while investigating other options. We want to make software tools better support data analysis and other kinds of nonlinear tasks that are hard to automate; we want risk-free exploration.

We think more design exploration for undo-like behaviour is needed in the new post-desktop areas of computing. Indeed, such exploration may benefit desktop applications too, as many of the constraints (memory, computing power) that applied when undo became a standard feature are much less constraining today.

In furtherance of these goals, we developed a prototype library called Ra, pictured in figure 1.2, discussed in more detail in the following chapters.

Chapter 2

Background

2.1 History systems

Software has many different methods for handling interaction history and exposing it to users. In this section, we review the methods currently available in software systems.

2.1.1 Traditional Undo

One of the most obvious history systems is traditional undo. Any large desktop application (word processor, data analysis tool, image processor, etc.) would seem incomplete without it. In menu bars, it is nearly universally found under “Edit”; the standard keyboard shortcuts depend on the system in question, such as Ctrl+Z for Windows or Cmd+Z for Mac OS.

Undo has a long history. The original vi text editor (written by Bill Joy around 1976) appears to have included it as the `u` command, and it was likely not the first. Early undo support would only undo the last action, or considered the undo itself to be an editing action that a subsequent undo command would revert, flipping between the most recent two states. As late as 1992, Abowd and Dix [2] write that “[d]espite the general agreement of the importance of undo, few systems supply more than the simplest single-step undo command”.

Typical modern systems can undo older actions as well. (Perhaps this is a result of increases in available memory for storing more than one historical state.) In this model, each editing action is added to a stack, with the most recent on top. The undo command removes the top of the stack, bringing the application to previous states, step by step.

Stack-based undo is regularly paired with a redo command. The undo command then moves the current position in the stack down one step, but does not discard the saved action. The redo command moves the current position one step up again. The

Figure 2.1. WorldWideWeb, the original prototype Web browser and editor, had navigation controls in a window/menu.

undone actions are only discarded when a new editing action needs to be added to the stack. So after an undo, one can redo back again as long as no other command has been issued in the interim.

2.1.2 Back buttons

Back buttons in the browser Web browsers have had back (and forward) commands since the very beginning. The first Web browser, WorldWideWeb, had a navigation menu with “Previous” and “Next” (see figure 2.1). Mosaic, the first widely-available browser [4], had “history-tracking facilities” [3] even in the pre-release prototype, and a released version put them prominently in the toolbar as seen in figure 2.2 and every major browser since.

What exactly the back button *does* is becoming less clear as the Web changes. When Tim Berners Lee put them in the original web browser, the Web was conceived as a document-centric system, and so it became for a while. Naturally, “back” would cause the browser to navigate to the previous URL in the history stack, analagous to the undo described above.

However, a URL does not completely encode the state of a browser. As Web pages support more interactivity, the number of non-URL sources of state increase: scroll position, values in form fields, pages that result from POST form submission, and finally running JavaScript. Browsers try to restore this state as well when going back.

Figure 2.2. Mosaic, an early Web browser, had back and forward buttons near their modern position.

For example, Firefox will freeze the state of a page and keep it in the “bfcache” [19] – within memory limits – so it can be restored by the back button.

As interactive content in Web pages increases in complexity and ubiquity, it is useful to model the Web browser as a virtual machine rather than a document viewer. In this model, the back button’s use is less clear. When should items be added to the history? Regular virtual machines only make snapshots on explicit user commands, so they are under the user’s control. But they are also usually longer-lived than browser tabs.

A new feature, introduced with HTML5, allows Web applications to interact with the browser’s history and let one page have multiple history entries. An application can add entries to the history list without needing a page navigation. (The operation is called `pushState`.) It can provide extra data to represent its state, which the browser will store with the history entry. If the user chooses to return to a history entry created this way, while the current page is the same one that added the history entry, the browser will notify the page instead of performing a page navigation. (The notification event is called `popState`.) It is then up to the application to restore its state, using the data stored previously.

So now the stack of history associated with the back button includes both real pages (the usual navigation happens if you go back to it) and entries created by an ancestor real page that represent some altered state of that page (going back doesn't cause page navigation, but runs some page script). The back button used to deal with a stack of URLs, but now the stack can also have serialized state info along with a URL. This feature can be useful when a Web application does something analogous to a page navigation (such as switching folders in an email app) but may cause confusion if it does not match the user's expectations (e.g. we have seen users surprised that panning in Google Maps adds history items).

Back buttons elsewhere The browser's back button has been adopted in other systems at least since Windows 98's attempt at Web integration, when the file manager got a back button. Since then the back button has proliferated in a wide variety of contexts as well, including other document viewers like PDF viewers, multi-step guided processes like wizard dialog boxes, and sometimes as a replacement for modal or hierarchical views like control panels.

The Android operating system for phones has a system-wide back button. It will first go back to previous views within an application, and then even switch to previously-viewed applications.

The back button is essentially a variant of traditional undo where the actions it manages are viewing actions (rather than editing actions as with traditional undo). It uses the same stack model, though some of these non-browser back buttons do not come paired with a forward.

The proliferation of this undo-view-change model is further evidence in support of Nielsen and Mack's [22] suggestion that reversible actions might not be enough. People seem to benefit from a common interface for reversing recent actions, even though those actions can be reversed using individual commands.

2.1.3 File-like saving

VM and filesystem snapshots Virtual machine managers can capture the running state of an entire virtual machine – its disk, memory, and currently executing instructions – to be saved for later. These are usually called “snapshots”.

Similarly, some advanced filesystems such as ZFS and btrfs can capture snapshots of the entire filesystem. A user (usually with special privileges) can then restore the filesystem to an earlier state.

These kinds of snapshots can act as a history system for their contents (the virtual machine or filesystem) in a way that would be impossible working at the lower level. However, they do not provide the management facilities that undo does. In this respect, they share many similarities with manual version control (discussed below); it is their ability to capture special data that sets them apart.

Computer game savepoints In many genres of computer games, the gameplay can extend for long periods of time. The player runs the risk of failure for the duration of the game, so as the player gets closer to the finish, they have more to lose. This may prove frustrating, but the ability to save and roll back history arbitrarily can be seen as a form of cheating.

It has become common for games to offer savepoints that provide limited access to history. For example, the player may be permitted to save only when their character is in a certain location, or at the beginning (or end) of a level. They can be automatic, as in many side-scroll adventure games, which mark such points by stopping the side-scrolling, or they may require manual action, as in the game *Ico* by Kenji Kaido and Fumito Ueda, in which the save is triggered by the character sitting on a bench. The availability of savepoints from which one can resume following failure is so common that its absence – often called “permadeath” – is a defining feature of the genre of Roguelike games such as *NetHack*, though apparently it was not a remarkable feature of *Rogue* at the time [10].

However, games rarely have any structure or management interface beyond deletion for saved games. They are usually offered as a plain list, or perhaps a representation of all possible savepoints (e.g. levels) of which only those that have been reached are available. Saved games are essentially the same as saved files (indeed, this is the typical implementation), as in the manual version control discussion below; it is their deliberate limitation and the possibility for automation that set them apart.

2.1.4 Source control

Software support for managing historical versions and parallel development of things is most advanced in the field of software development. The source code for a software project is kept in a source control system, such as Git, Mercurial, CVS, and many others. Working with old versions is particularly common in software development as updates are made to released versions of the software while development of unreleased versions is ongoing.

Source code is stored as plain text, and usually lines of text are a natural unit of code. Developers rarely have to work with non-text file formats directly. It is no surprise, then, that most source control systems are primarily designed to store text files and operate on lines of text; storing non-text files such as images and word processing documents may be handled less efficiently and useful features (such as version comparison) may not support them at all.

The first widely used source control system was SCCS, the Source Code Control System developed by Rochkind at IBM and Bell Labs starting in 1972 [23]. It introduced several key features of this class of software, including storing only the differences between versions, and support for different branches so that work on versions 1 and 2 can continue in parallel while being managed by the same system. It was superseded by RCS, and then CVS.

The most popular source control system today is surely Git, originally made by Linus Torvalds in 2005 following a period of dissatisfaction with the options available for managing the source code for Linux. Many experimental source control systems were developed around the same time, spurred on by similar motives; several are still in wide use today, including Mercurial and Darcs. Important features that set this generation apart are the distributed model (CVS and Subversion used a client-server model) and alternative views of how to track changes. CVS and Subversion tracked each file separately, so multi-file changes would not be kept together in history; Git and Mercurial solve this by treating each changeset as a snapshot of the whole repository; Darcs takes a similar approach but treats the repository as a set of (possibly multi-file) patches, which may not be strictly ordered.

2.1.5 Manual version control

In the absence of a system for managing history, especially in domains where source control systems are unknown or unsupported, people often work out their own manual systems for keeping history. The typical form is periodically making a copy of the work, then storing one copy and continuing work on the other. The ad-hoc nature of this system often results in folders full of files with names like “thing-v21-final-edited.doc”.

Manual version control is interesting not so much for the systems that people develop, since these are highly constrained by the technology they can use conveniently, but as an indicator of the need for maintaining history in the first place.

2.2 Managing and visualizing history

In the simplest cases of traditional undo, only the commands undo (and possibly redo) are available. There is no attempt to present the history to the user, and so no way to move through the history more than one step at a time. Similarly, the history mechanisms that are essentially like saving files, such as virtual machines and savepoints in computer games, offer only an unmanaged list of files or snapshots.

There are two general structures with which the other history systems discussed above represent history: linear (stacks and lists) and nonlinear graphs including trees.

2.2.1 Linear models

History that can be represented as a stack of states, such as traditional undo and back buttons, is often presented as a list. For example, the Inkscape vector graphics editor has an undo history window (figure 2.3) with a list of editing actions that have been performed.

The Inkscape example is notable because it collapses related sequences of actions (marked with a + sign in the list). The “Create text” entry represents the creation of a text object and the typing of text in the new object, so it can be expanded to show 24 “Type text” entries, one for each keypress. But it’s confusing because the entries really represent states but are labelled with actions. (The entries represent the state after the action named in the label happen.)

Figure 2.3. The Undo History window for the Inkscape vector graphics editor, showing some sample history. Time flows down. The highlighted entry marks the state currently shown in the editor.

Browsers also present history in a list, though the history entries are labelled with the page title rather than the action that produced them. Some display separate lists for back and forward, for example as menus produced by clicking an arrow beside the respective buttons. Some present a unified list with the currently displayed page highlighted, back items below and forward items above.

2.2.2 Graphs

History can also be represented as a graph of states. The most well-known example here is source control systems. Systems like Git describe the history as a directed acyclic graph (DAG). Any individual commit (changeset) has a nonnegative number of parents: the initial commit has no parents; most regular commits have as their one parent the commit the author started work on; and commits with multiple parents merge the code from their parent commits. This structure is discussed further in section 3.2.

In Git, users are directly exposed to the graph structures it uses for storing history. One consequence of making users work directly with the DAG is the additional management burden it may impose. Software projects develop policies about how to contribute patches so that the changes to the DAG will meet other developers' expectations. For example, one question is whether to “squash commits” or not –

meaning that all the intermediate code changes for one logical change are combined in a single commit, or are left as they happened even if the intermediate states do not compile. Another is whether to update the parent of a commit before distributing it to others, leading to a more linear DAG, or let the parent indicate the true point from which development started.

There are several lesser-known graph representations of history. These represent it with trees, which are DAGs but with the constraint of having at most one parent per node and exactly one node with zero parents.

The text editor Vim can track editing history as a tree [17], as can Emacs (with the undo-tree elisp module). These features do not seem to be well-known even within their respective communities, perhaps because the typical user of these editors is also familiar with source control.

Some people have tried to introduce the tree representation of history into other domains as well. There have been at least two extensions for Firefox that extend the history to use a tree representation, but they were not widely used and have been abandoned. This may be partly because of their awkward interfaces: one required a separate window to draw a traditional tree structure, and the other represented the tree as a set of deeply-cascading menus (a well-known usability problem [24]).

2.3 Sources of history management

Some history mechanisms are *external*, imposed by a system operating at a higher level of abstraction (or privilege level). Source control, virtual machine and filesystem snapshots, and (from the perspective of Web applications) the browser's back button are all external history mechanisms.

These have less information about the states in history available to them, so they may rely more on semantics provided by the user, or simply offer less contextual features. The main advantage of operating at a higher level is that they can capture state that might be unavailable at lower levels. For example, virtual machine snapshots can include the registers and instructions being executed on the virtual processor, plus the contents of memory regardless of memory protection techniques. External mechanisms can also be applied to new systems that were not designed to support

them.

In contrast, some history mechanisms are *internal* to the system being managed. They may act at a slightly higher level of abstraction within the system, but they do so only with the consent of the system being managed. Traditional undo is the type specimen for this category, but it also includes game savepoints. Our prototype software Ra is somewhat in between these two categories, being an internal mechanism trying to offer some of the advantages of external mechanisms.

Internal history systems have more information about internal states, so programmers can identify likely restore points, and offer other contextual features. However, each system must take care to capture all state correctly. As Nielsen [21] advises, undo should not be limited to a special category of actions; our own experience is that the rare situations when undo skips some actions are surprising and frustrating. Systems designed to encapsulate separate components make it difficult to fully capture the program state without special privileged access (such as external mechanisms have).

Chapter 3

Design

3.1 Requirements

Our goal was to make a prototype library that would keep track of a user’s interaction history, and let them use it. This establishes the only functional requirement, broad as it is: allow the user to go back to any state they’ve been in before.

For the non-functional requirements, we use the categories from Rogers *et al.* [24]. Since we were making a research prototype, some of these requirements are less strict than they would be for production software.

Data requirements Ra must store all past versions of an application’s state. This naturally leads to the question *what is state?* – a question we mostly dodge by claiming that the answer depends on the application; anything the user would expect to be restored after saving something should count as state. The application may have to maintain other kinds of data that fall outside this definition; see section 4.5.1 for the technical view of this question.

The current state may change frequently, adding a new record to be stored. This might seem to point to a rather large amount of data that Ra must store, but there will be similarities between versions, so compression can reduce the amount of data. Furthermore, our experience with version control in software suggests that this will not be a problem: for example, in the mozilla-central repository (which contains the code for Firefox), the uncompressed current version is 4.4 GiB, but the entire Mercurial repository (about 8 ½ years of history¹) is only about a third that size: 1.6 GiB.

However, while the current state may change frequently, it is the only data that needs to change. Once recorded, past states do not change.

The data must be consistent, since it represents the running state of an application,

¹The first commit in mozilla-central is an import from the CVS repository that was used before the switch to Mercurial, dated 22 March 2007. The repository used for the file measurements was current as of 7 August 2015, changeset 461fc0a6a130.

and without knowledge of the application being restored (which parts are dependent on which), that means it must be accurate.

The data should be persistent for at least as long as the browser tab is open; this is the minimum required for the usability study we conducted (section 5). We also wanted the option of longer-term persistence, but left the details to be explored.

Environment requirements The physical environment will be a standard computer with a mouse or a large touchscreen.

We did not establish the social environment as requirements. Instead, we wanted to explore the possibilities Ra presented so future work can focus on a more specific environment. For example, we were generally assuming a single-user model at first, but as we began work, the possibilities for collaboration soon became evident, as described in section 3.4.2.

We assumed that at least some organizational support would be available to users. It is, of course, preferable to require as little training as possible, but we expect Ra to be used with technical tools such as data analysis applications, which already require some learning investment. We thus concluded that Ra did not have to be a “pick-up-and-use” system.

The technical environment will be a Web browser. We initially chose Firefox for its relatively advanced **Proxy** support, but later added support for Chrome (see section 4.6.1). Additionally, the library should be general enough to support a range of Web applications. It should not require significant, invasive changes to use Ra with an existing application.

Usability goals The presence of Ra should encourage the user to perform more epistemic interaction, to explore more possibilities, and to continue looking for other possible solutions after an acceptable solution is found. Users should not feel committed to an early decision.

3.2 Visualizing history

Some modern version control software such as Git and Mercurial store the history of a project as a directed acyclic graph (DAG). Tools for working with these systems will

Figure 3.1. gitk displays the DAG of the Git repository

often display the history as a DAG as well. See, for example, figures 3.1 and 3.2. Note that in these visualizations, time flows up; the most recent changes are at the top of the diagram.

In software projects, the structure is a DAG because most work ends up in the final product. When developers work in parallel, they are usually not working on alternatives; they will merge both lines of development together. A typical small example is shown in figure 3.3.

In Ra, we represent the history as a tree. (This is also the data structure used internally; see section 4.4.1.) We expected that the exploratory behaviour we want to encourage would result in mostly dead ends, or multiple different results for presentation or comparison, rather than some unification of most of the work. Merges seem like a desirable feature in some cases, but their usefulness may not be worth the extra complexity. It is unlikely that Ra could perform merges automatically, and there is no obvious way for a user to direct the merge of two snapshots of application state. This is in contrast to merging source code, which can often be done automatically, and manual merges of source code requires understanding the code, whereas merging application state would require understanding the (usually not human-readable) representation of that state.

The tree visualization in the Ra sidebar is inspired by the visualizations for version control systems, such as figures 3.1, 3.2, 3.3, and 3.4, and the traditional visualization

```

psimonyi@phyllis:~/src/git
* f474298 Junio C Hamano <gitster@pobox.com>
  Merge branch 'jk/connect-clear-env' into pu
* aab4043 Jeff King <peff@peff.net>
  git_connect: clear GIT_* environment for ssh
* 8e16a19 Junio C Hamano <gitster@pobox.com>
  Merge branch 'gb/filter-branch-progress' into pu
* 9969f14 Gabor Bernat <gabor.bernat@gravityrd.com>
  filter-branch: add passed/remaining seconds on p..
* 5865475 Junio C Hamano <gitster@pobox.com>
  Merge branch 'kn/for-each-tag' into pu
* 4a531d2 Junio C Hamano <gitster@pobox.com>
  SQUASH??? a tag longer than 15 bytes break align..
* bd041ae Karthik Nayak <karthik.188@gmail.com>
  tag.c: implement '--merged' and '--no-merged' op..
* b99e86a Karthik Nayak <karthik.188@gmail.com>
  tag.c: implement '--format' option
:|

```

Figure 3.2. The `git log` command can also generate a DAG visualization of the Git repository.

```

psimonyi@phyllis:~/src/dnf
* ca927b9 Seth Vidal <skvidal@fedoraproject.org>
  die tabs, die
* 08b247e Seth Vidal <skvidal@fedoraproject.org>
  Merge branch 'master' of ssh://login.linux.duke.edu/home/groups/yum/git/yum
* 72569b7 James Antill <james@and.org>
  Special workaround for depsolving, load prco data without keeping hdr
* 4ac0b5c James Antill <james@and.org>
  Add buildtime for hacky pkgid
* e1bb34b Seth Vidal <skvidal@fedoraproject.org>
  make rpmsack_pkglist return direct from the rpm hdr objects and not pass..
* 6daa211 James Antill <james@and.org>
  Huge memory decrease, by using sqlitesack algo. on installed rpm packages.
* 197c473 James Bowes <jbowes@redhat.com>
  Remove some unused imports
* e77b7ca James Bowes <jbowes@redhat.com>
  :|

```

Figure 3.3. A simple merge. James Antill and Seth Vidall began working in parallel, both starting from James Antill’s “Huge memory decrease” commit, when the paths diverge. Then Seth Vidal merged both their work before going on to commit “die tabs, die” to the now-unified history.

Figure 3.4. Large repositories, such as the one for Git, can have complicated history. gitk has added colours to the visualization to help track each branch.

of trees in computer science. Figure 3.4 in particular shows that the dots-and-lines visualization is capable of representing complex structures. It is the latter's influence that determined the direction of tree growth. New nodes are added below, and if necessary to the right of, the parent node (which represents the state that happened immediately beforehand). This also, happily, matches normal English reading order.

3.3 Applications

We modified several Web applications to use Ra. We wanted relatively simple applications to experiment with so that we could try several and so that we could focus on Ra. Using Ra with these applications guided both the implementation and the interface design. The following applications are pictured in this chapter:

Green square This is an extremely simple maze game. All it does is let you move the green square around the screen with the arrow keys. (Figure 3.5)

Figure 3.5. Adventures of the Green Square: a simple maze, the first application made to work with Ra.

Although it was useful in development, its state consists entirely of the position of the green square. Ra is not a compelling addition because there is so little state and it is easy to change.

TodoMVC React TodoMVC² is a project to implement the same todo-list application in many Model-View-Controller frameworks to help programmers compare and choose one of the frameworks. We modified the one using the React framework.

This showed us that Ra can be added to an existing application without too much work, but a todo list does not exercise the exploratory possibilities either. (See figure 3.6.)

Maze See section 5.3.2. This simple maze application draws a path through a configurable maze where there is more than one solution, so the exploratory possibilities are more apparent.

²<http://todomvc.com/>

Figure 3.6. TodoMVC React: another application we modified to use Ra.

Parallel coordinates See section 5.3.2. This is an interactive data analysis application. It is somewhat more complex than the previous simple applications, which let us experiment with the app-suggested snapshot feature described in section 3.4.

Interactive data visualizations involve the exploration Ra is designed to support. As shown in figure 3.7, Ra keeps track of the brushes (selections), so the user can return to interesting queries, or walk back through the queries that came before a given state.

3.4 Ra user experience

When Ra is part of a Web application, it adds a sidebar, shrinking the available application space, as shown in figure 3.8. Ra does not try to intercept or manage user interaction with the application part, so aside from being narrower, the application works exactly as it would without Ra.

As the user uses the application, Ra records the state of the application when it changes. We call the recorded state a “snapshot”. These are shown as nodes in the

Figure 3.7. The parallel coordinates application (showing the months Ottawa had at least some snow and the temperature never dropped below zero – a rare occurrence). The balloon popup for a node is showing the label and timestamp. The user has already returned to that state and started a new branch; the “Load this” button would let them do so again.

Figure 3.8. Ra divides the page to make room for a sidebar.

tree visualization in the sidebar, where each snapshot follows from its parent in the tree (“time flows down”).

The user can return the application to a previous state from the sidebar. Tapping or hovering with the mouse brings up a balloon popup for each node, as shown in figure 3.7, from which the saved state can be loaded. The node in the tree that represents the current application state is marked in yellow.

When the user returns to a previous state, they can still interact with the application – making different decisions this time. Instead of replacing the history from that point forward, as a traditional undo system would, Ra starts a new branch in the tree, as shown in figure 3.7, so both timelines are available.

Ra can record states for different reasons, and these get different glyphs for the nodes in the tree visualization. There are three³ types of nodes:

- Automatic snapshots are shown as small dots (figure 3.9). They are all given the

³There is a fourth type: the snapshot of the initial state is special, but it only happens once. Its appearance matches app-suggested snapshots.

Figure 3.9. An automatic snapshot

Figure 3.10. An app-suggested snapshot

same default label (“autosave”) because no semantic information about the state is available. Ra creates them automatically when it detects that the state has changed, though this is rate-limited to at most one per second, and the other snapshot types supersede automatic snapshots.

- App-suggested snapshots are shown as bigger, brighter dots (figure 3.10). The application can tell Ra to create one of these snapshot when it is in a state the user is likely to want to return to, which is why these nodes are more prominent. The application provides the label for these snapshots. This kind of snapshot depends on the support of the application, and some applications (such as our simple maze) may not create any. Our parallel coordinates application uses these to mark the creation of new brushes (selections).
- Starred snapshots are shown as stars (figure 3.11). They are created explicitly by the user. They may have a label set, also provided by the user. To make a starred snapshot, the user enters the label in the textbox in the sidebar, then presses the “Save” button. Note that in previous versions of Ra, starred snapshots had the same appearance as app-suggested snapshots; see sections 5.4 and 5.6 for details.

The application may have two influences on how Ra works. As mentioned above, it can trigger the creation of a more prominent snapshot, and provide the label for it. Our simple maze does not do this (it’s hard to predict where a user might want

Figure 3.11. A starred snapshot

to branch, and it is similarly hard to generate useful text labels), but our adaptation of the parallel coordinates application does. When the user creates a new brush (selection) on any axis, the application creates a snapshot labelled “New brush on $\langle axis\ name \rangle$ ”. Moving and resizing existing brushes just creates automatic snapshots.

The other influence the application has is its choice of state. In our sample applications, we tried to include everything that would change the user-visible display. However, the application may choose to exclude some kinds of state such as that only relating to the view (e.g. which of several panels is selected). Further study is needed to see whether this is a useful feature.

3.4.1 Visual walkthrough

In the following series of screenshots, we present a step-by-step visual walkthrough demonstrating the use of Ra with our simple maze application:

- Figure 3.12 shows the application when it is first loaded. The Ra sidebar shows only the initial state snapshot.
- Figure 3.13 shows automatic snapshots being created as we work in the maze.
- Figure 3.14 shows label text being entered to make a starred snapshot.
- Figure 3.15 shows that the history continues linearly after creating the starred snapshot.
- Figure 3.16 shows that, having continued to work after the snapshot, we can return to the starred state.
- Figure 3.17 shows that new work goes into a new branch, creating a tree of alternate timelines.

Figure 3.12. A simple maze and a fresh history tree. Since nothing has happened yet, it's pretty empty.

- Figure 3.18 shows the final path.
- Figure 3.19 shows that even after reaching a solution, we can continue an old abandoned branch.
- Figure 3.20 shows the final alternative path.
- Figure 3.21 shows that we can pick the best of several results.

3.4.2 Persistence and collaboration

On its own, Ra keeps its history records in memory. That means that the history is erased when the browser tab is closed, reloaded, or navigated elsewhere. In order to keep history more persistently, we made a server called Sol (see section 4.4.4) to act as a storage system for Ra. This is an optional component, since it has to be run as a separate server.

If Ra is using Sol, then the user can close, reload, or reuse the browser tab and history will still be preserved. When they return to the application, Ra will load

Figure 3.13. As we move the token in the maze (drawing a path behind it), Ra is creating automatic snapshots on the side.

Figure 3.14. Hm... should we go back to the left a bit, or try to keep going right? We'll save this spot for later by writing a label and pressing Ra's "Save" button.

Figure 3.15. Having marked the decision point with a star, we tried the right path. But it's got a rather big wiggle; this isn't looking like the best path any more.

Figure 3.16. We've gone back to the starred snapshot to try the left-hand fork this time. Ra still remembers that big wiggle we didn't like, so we can go back if this doesn't pan out. (Also, notice our label from figure 3.14?)

Figure 3.17. The automatic snapshots of our new path also go under the star, but in a new branch. This path looks much better from here.

Figure 3.18. We can head straight to the finish. Our path is 32 segments long.

Figure 3.19. But was the path we abandoned really inferior? Let's go back and bring it to completion.

Figure 3.20. The new automatic snapshots continue in the old branch as we continue from where we left off. Indeed, this was the worse path, by two segments – 34 instead of 32.

Figure 3.21. We'll select the better path as our final solution.

the saved history tree from Sol rather than starting fresh. The user can even switch computers and continue using the same history.

We noticed that this has the potential to act as a different sort of collaboration tool. If multiple people load the same application, they will see the same history tree. Ra gets real-time updates from Sol when the tree changes, so all users see the same history, but each is running an independent instance of the application.

When one user makes a change in the application, a new snapshot is added to the history tree, which gets copied to all the other users. But each user has their own current state, which is not updated when the new snapshot is added. A user can load their colleagues' snapshots just as they would load their own; it just doesn't happen automatically. Thus two users could work in parallel, and compare their work later. They share the same history tree, and can continue work from their collaborators' states as well as their own. Further work on this feature should provide a way to identify which snapshots were created by which user, since they are currently indistinguishable.

Figure 3.22. Collaboration enabled by Sol. Two users share a history tree, but each has an independent current state.

We have seen Ra from a user's perspective in this chapter, and it meets the requirements from section 3.1. In the next chapter, we will describe the technology behind Ra.

Chapter 4

Technology Platform

Recall that the general technical goal is to capture snapshots of the running state of a Web application, and then be able to load snapshots without too much delay. There are several ways this could be accomplished, each with its own drawbacks.

4.1 General approach

For ease of prototyping, we chose to implement Ra as a JavaScript library, to be included in the Web application with some (but preferably minimal) supporting application changes. We wanted Ra to be non-invasive enough that it can be added to an existing application without restructuring the whole thing.

The central part of keeping required changes to the host application localized is the use of **Proxy** objects. The newly-finalized ECMAScript 2015 Language Specification [1] (ECMAScript 6) introduces them, though prominent JavaScript engines such as SpiderMonkey in Firefox [20] implemented versions specified in drafts of the specification well before the final publication.

A **Proxy** imitates an existing object, but it can intercept almost all interaction with that object. In the specification [1], a **Proxy** object is defined as an “Exotic Object”, meaning that it is not required to display normal JS object semantics. For example, immediately after setting a property on a regular object, retrieving the same property must return the previously stored value (unless an exception was raised); **Proxy** objects are not required to act this way. The **Proxy** object has a special handler function that can override the normal object semantics. Following the same example, retrieving a property value as `obj.prop` would call a function provided when the proxy was created, and the expression would evaluate to the return value of that function. The function can usually return any value it chooses, although there are some more complicated edge cases requiring the semantics of certain features such as non-writable properties to be respected [1, 18].

An application using Ra substitutes **Proxy** objects created by Ra for the objects that hold its state. When all objects that hold state in the application are actually **Proxy** objects managed by Ra, the application code continues to interact with Ra implicitly when it uses those objects, yet all other code can continue to use the objects as if they were the real state objects. This allows us to update state objects on demand, wherever they may be inside the application at the time, and whoever may have references to them. From the perspective of the application code, when the user loads a saved snapshot, the state objects immediately become the saved values, without requiring the application to actually make any changes. This is accomplished by setting all the traps in the **Proxy** to return the value from the current saved state object instead of the original.

If the application was already written in an object-oriented style following the Model-View-Controller (MVC) pattern [14], with state stored as properties of long-lived objects, then the state objects do not have to be tracked through their entire lifecycle; to support Ra, changes are needed only where state objects are *created*. Additionally, since the “objects storing application state” that Ra needs correspond to objects in the Model component of MVC, all the state objects are already identified and ready to be replaced by proxies.

This approach does have several disadvantages. It does not work well if the application was written in another paradigm or a style that does not generally store application state in long-lived objects; we discuss workarounds for this problem in section 4.6.2. It also requires some sort of redrawing mechanism. When the user loads a snapshot, the application’s model changes, but other parts of the application are unaffected, and the page rendered by the browser is still unchanged. The application needs to update everything to match the new model. Nevertheless, we think that these disadvantages are outweighed by the advantages, particularly the limited invasiveness in MVC applications.

4.2 Alternatives

We could have taken several different approaches to this project. We address a few of the possibilities we rejected here.

We could have required that the application serialize and deserialize its model on its own, and Ra would have told it when to do so, and kept track of the serialized records. This approach would have required that the application be able to track down all the state objects in use at any time, which seems like a rather onerous burden. And if the application were already keeping its state separate in a model component, it might include state that doesn't make sense to save from a user's perspective, such as text selection or cached data; this would have to be excluded every time the model was serialized.

We could instead have required that the application explicitly using Ra-provided ordinary objects for state instead of using `Proxy` objects. However, that would make it difficult to be sure that deeply-nested object structures are always using the current version from Ra, it would be possible for the application to make a state object property non-writable (thus making it impossible to restore an older version), and – most significant – Ra would not know when state objects changed, so the application would have to notify it.

We could have implemented Ra as part of the browser. This would have been the most robust option if done correctly, since the browser is essentially a virtual machine for web applications, so it should, in principle, be able to capture a snapshot just as regular virtual machine managers can take snapshots today. However, this is clearly a far more complicated task for prototype software, as JavaScript engines are not designed for capturing running state, and it would have to include capturing the browser DOM and perhaps other components as well. There are also the usual disadvantages of virtual machine snapshots: they would be larger and harder to compress, they would not have the option of being selective about what constitutes state, and the system would not have semantic information available such as for the application to suggest and name snapshots or to distinguish when an automatic snapshot should be taken.

4.3 The DOM

No discussion of ways to store state would be complete without mentioning the browser DOM. In a web browser, a page is represented to the page's scripts as the Document

Object Model. (“The DOM” can mean either the API in general or one model representation in particular.) It initially reflects the tree structure of the page’s HTML, but can then be modified by scripts; these changes are then handled by the browser (such as for displaying new content, loading new resources, or changing display styles).

The DOM is a natural place for a web application to store state because it is already storing the state of the application’s display for input to the browser’s rendering engine. It can also store arbitrary JavaScript objects as properties of the DOM objects. Many simple web applications keep their state in the DOM out of convenience: you don’t have to be able to regenerate part of the page from scratch, and you don’t end up with two copies of your state data (one canonical, the other a copy in the DOM) to manage.

However, there are several reasons the DOM is not a good place to store state, both in general and specifically for use with Ra. For one, retrieving data from regular JavaScript objects is faster than retrieving it from the DOM, so storing state elsewhere can improve performance. Several current efforts in JavaScript application development focus on minimizing interaction with the DOM to take advantage of this improved performance. For another, the DOM is a complicated structure with hooks into several other parts of the browser. The DOM APIs allow access to data from other sources, such as stylesheets and information about the HTTP transfers involved in loading the page; these are not the kinds of data that Ra should store, so some filtering mechanism would be needed to select only the important state anyway. And most of the DOM that does reflect state that should be stored can be generated from much less data stored in JavaScript objects, so we did not make Ra work with state stored in the DOM.

4.4 Technical architecture

The components of a system using Ra are illustrated in figure 4.1. There are three points of interaction between Ra and the host application: the Ra API (not illustrated – see section 4.5) allows the application to initialize and direct Ra; the state objects used by the application are proxies managed by Ra, allowing Ra to track the current state; and the browser DOM is shared between the application and Ra, as Ra adds its

Figure 4.1. The components of a system using Ra. The application’s model state is actually made up of proxies for the real state objects stored as Ra’s current state, so the two components remain synchronized. The current state, in turn, is expanded from the tree of serialized historical states. Sol is a separate server that keeps the history tree synchronized between clients.

user interface to the page in the form of a sidebar (see section 4.4.3).

To add Ra to an existing application, the programmer must substitute calls to the Ra API in place of the creation of state objects. Ra’s object-creation functions return the proxies that allow it to watch the application state for changes, save the current state for later, and restore old states. This sets up the main link between the application and Ra, so that while the application can treat the proxies as normal state objects, all access actually goes through Ra.

This substitution of proxies for real state objects relies on the separation of state objects from other parts of the application, including the browser DOM and possibly other kinds of state that Ra should not track (see section 4.5.1). If the application is structured following the MVC pattern, as in the diagram, this is usually already the case, and the application will not have to be changed very much to support Ra. Otherwise, the application will need to be changed to keep the state objects separate. (Some workarounds can reduce the amount of restructuring required; see section 4.6.2.)

4.4.1 Storing state

Current state objects are kept as objects (not serialized). If an object has another object as a property value, that object must also be a state object; instead of storing the second object itself as the value, a numeric reference is used. This allows the serializer to handle cycles without much extra work, and makes it easier to enforce the requirement that state objects contain only simple types and other state objects.

To save the current state, all objects in the store are serialized to JSON. The serialized state is added to a tree structure of all the serialized states as a child of the previously-current node.

There are three ways a state can be added to the history tree, and each produces a different kind of tree node:

The user can explicitly request that the current state be saved by clicking the “Save” button in the Ra sidebar. The state at that moment is saved with the given label. This produces a starred snapshot, the most prominent kind of item in the tree.

The application can suggest saving the current state with `ra.stateChanged`. The state at the moment of this function call is saved, and the application can provide an optional label as well. App-suggested snapshots are less prominent than user-requested snapshots.

Whenever a state object is changed, Ra knows, because the change is made through its proxies. This starts an autosave timer. When the timer expires, the state at that time is saved, or the timer is cancelled if the state is saved for some other reason first. These automatically-saved states are given the least prominence in the user interface, and all have the label “autosave”. The timer serves two purposes: it causes rate-limiting, so at most one change per timer period (currently one second) is recorded, and it takes advantage of the single-threaded nature of browser JavaScript execution to ensure that Ra does not save an unstable intermediate state, because the timer function will not run while application code is still running.

4.4.2 Restoring states

Ra can restore a given state when the user requests it. It deserializes the representation of that state from the JSON, replacing the current state objects. The application still

has proxies representing state objects, but those `Proxy` objects always look up the current object in the store so that they appear to be the newly-restored versions of the state objects. This means that the application does not need to do any special work to handle restoring an old state – except that the user interface won’t be changed automatically.

Ra will call registered listener functions (see the API in section 4.5) after a saved state is loaded. The application should provide a redraw function to update the browser DOM to make the interface match the modified state objects.

Ra is able to restore objects whose prototypes are not the default `Object`, but in order to do so reliably, it is important that the required prototypes can exist at any time; otherwise, there could be times when some states would not load properly. To ensure that the prototype is always available, Ra requires that the application provide all the prototypes it might use by registering them with Ra when the application first starts. This requirement is similar to object-serialization requirements in other contexts, such as “pickling” in Python, which requires that class definitions be at the module level (not nested inside functions). Unfortunately, the typical structure of a Python program already meets this requirement, whereas a similar requirement in JavaScript would contradict the common advice to “avoid polluting the global object”. Registration of prototypes is a compromise.

4.4.3 The sidebar

The Ra sidebar shows a tree representation of states. Each glyph (node) in the tree represents a state; following the line up goes to the state that came before. (See section 3.4.)

The sidebar is injected into the page by `ra.startup()`. It is inserted into the DOM as a child element of the document element (typically `<html>`) in the hope that this will be less disruptive to the application. Although Ra and the application necessarily share the DOM, Ra should not interfere with the application’s user interface beyond making it narrower. Typically, all visible HTML elements are children of the `<body>` element, so this is likely to help keep Ra’s DOM objects separate from the application’s.

4.4.4 Sol: a server for persistence and collaboration

On its own, Ra stores historical states in a tree in memory. But since it is part of the Web page, this is erased when the page is closed. There is a server component that can be connected to keep the tree of states more permanently. The server is called Sol.

Sol is a Python program that communicates with Ra over WebSockets. The server is quite simple. It does very little more than simply report the states it has stored and accept more states to store. If Ra is connected to Sol, it will load the history that Sol reports, and upload new states as they are created. Then the history tree will be persistent across page loads and on different computers.

Sol also has experimental support for collaborative use. If more than one Ra instance is connected to the same Sol server, they will share the history tree, but not the current state. To support this, Sol assigns unique identifiers to states and announces new states to all connected clients. When client 1 makes a change, the new state is uploaded to Sol, stored in the database, and announced to client 2, which adds it to the local history tree.

4.4.5 Technology choices

Ra itself is written in JavaScript, and it uses ECMAScript 6 language features such as **Proxy** support, the arrow function syntax which is briefer than function expressions, and modules to separate the code into multiple files for easier development. The multiple files are merged together in a compile step. We are currently using Esperanto for this, though there are alternatives that should work just as well. In addition to keeping the code neater, this allows Ra to be used as a “traditional” JavaScript library that adds a **ra** property to the global **window** object or as an ES6 module, using `import ra from 'ra'`.

Sol is written in Python using the **asyncio** module, part of the standard library since Python 3.4. For asynchronous operation, **asyncio** uses generators to let the programmer write coroutines for handling communication. This allows the code to be written in an imperative style with explicit control flow, and avoids the concurrency problems of threading. The part of Ra that communicates with Sol has similar needs for asynchronous operation. It uses **Promise** objects (another ES6 feature) to avoid

the mess of callback functions usually associated with asynchronous JavaScript code.

Sol stores the history records in an SQLite database. Although it could perform optimizations such as recording only the differences between versions, we did not encounter any performance problems that suggested this was necessary at this prototype stage.

4.5 Ra API

This section presents the API, and is written as if to the application programmer.

The Ra API lets you use Ra in any web application that meets the state requirements.

4.5.1 State requirements

Decide what “state” means in the context of your application. The state should be what the user will want to go back and forward in time with; this will usually correspond to the model in an MVC architecture. For example, runtime state associated with the ephemeral parts of the view is unlikely to be useful as state for Ra’s purposes.

Ra deals with “state objects”, which must meet the following criteria:

- A state object contains only other state objects and simple values (“primitive values” in [1]).
- A state object is always modifiable — no properties are non-writable or non-configurable.
- A state object cannot be a function.
- A state object cannot be an exotic object (as defined in [1]). This includes DOM nodes. Arrays, which are exotic objects, are a special exception.

Since the DOM cannot be part of the state, Ra will not update the DOM when old states are restored. The application must have some way to update the DOM to match any valid state, even if that state has never been reached before (since it might be loaded from the server).

4.5.2 Core

Identify the objects that store state. If necessary, you may have to change the application so that state objects meet the requirements above. For particularly difficult cases, see also section 4.6.2.

For each state-storing object, find where it is created/instantiated and transform the code as follows:

- For object literals, rewrite `o = {}` as `ra.stateObject({})`.
- For object literals, rewrite `o = {a: 1, b: 2}` as `ra.stateObject({a: 1, b: 2})`, and make sure that any object values are similarly wrapped.
- For objects created with `new`, rewrite `o = new Thing(a, b, c)` as `o = ra.newSO(Thing, a, b, c)`.

If you used `newSO`, you will also have to register the constructor in your setup code with `ra.registerConstructor(Thing)`.

If you have a function that updates the display to match the current state, register it with `ra.addListener(redraw)`. Otherwise, you'll need to create a function to do this. The function will be called (with no arguments) whenever Ra changes the state.

Finally, call `ra.startup()` *after* any calls to `registerConstructor` or `addListener`.

The features described in the following sections are optional.

4.5.3 Suggest restore points

Normally, Ra will create little autosave nodes whenever the state changes, rate-limited to one per second. But you have more information about which changes will be most important to the user. After you make a change that the user is relatively likely to want to revisit, call `ra.stateChanged("label goes here")` with an appropriate label for the change. The current state will be saved immediately, and the node will have the given label.

Don't use this for every change, since Ra does that automatically. Instead, try to choose which changes will be the most significant points in history. These nodes will be visually more prominent than the autosave nodes.

4.5.4 Shared server

Ra comes with Sol, a server for storing and sharing the history tree. Everyone on the same server will share a history tree. For a single person, this means the history is not lost when they close the browser window. For a group, each person can be in a different state, but all changes are contributed to the same tree, so one person can start working from someone else's state.

If you're running Sol, pass the URL as an argument to `ra.startup`.

4.6 Practical considerations

In developing and running applications with Ra, we encountered some application- and browser-specific problems.

4.6.1 Supporting V8

We did most of the development work in Firefox, which has support for `Proxy` objects that is mostly compatible with the ECMAScript 6 specification. However, for practical reasons we decided to use Google Chrome for the user studies (section 5).

The V8 JavaScript engine used in the Google Chrome and Chromium browsers has experimental support for `Proxy` objects, but it follows an early draft of the specification. We had to use the Babel¹ transpiler to convert ECMAScript 6 code into the ECMAScript 5 code compatible with V8, and the Reflect² polyfill to update the `Proxy` API to match the current ECMAScript 6 specification that Ra relies on. In addition, V8's support for `Proxy` objects is turned off by default (since it no longer matches the specification). The Reflect polyfill can update the old version to match the new specification, but the old `Proxy` support is needed as a starting point. The browser must be started with the `--js-flags=--harmony_proxies` command-line argument to get this support.

¹<https://babeljs.io/>

²<https://github.com/tvcutsem/harmony-reflect>

4.6.2 Supporting other architectures

State objects must meet the requirements listed in section 4.5.1, including the requirement that they contain only simple values and other state objects. This corresponds very nicely to traditional MVC or three-tiered application architectures, since the Model component keeps the state objects isolated from the other code. But MVC is not required; for example, Ra does not need all state to be kept *together*.

However, the state object requirements are impractical in some programming paradigms and architectural styles (or lack thereof) used in JavaScript. Storing state in the DOM, as discussed in section 4.3, is a common technique that is problematic for Ra. Trying to recover the important state from the DOM from Ra’s position³ would be complicated and error-prone at best.

Some applications keep state in closures, in variables local to a function but available to any other functions that are lexically inside the function. Programs written in the functional paradigm generally rely on this rather than mutating objects. It is common to use closures to avoid adding properties to the global object in top-level code, and some store state in variables in that scope. There is also a well-known pattern for “private members” in code trying to emulate Java-style object-oriented programming by using closures to restrict access to variables, since variables cannot be updated from outside their scope.

Even some popular JavaScript frameworks that present themselves as supporting the MVC pattern store non-model data in objects within the model component. A common technique is to add a list of listeners (callback functions) to each object in the model. When a model object is changed, it notifies all its listeners. This is incompatible with the current implementation of Ra, which serializes all state objects but can’t serialize functions. Functions cannot be serialized reliably because there is no standard way to get the code, and the function may have had access to closures that can’t be recreated when it is restored. It might be possible to overcome this limitation in some well-controlled cases, but it would require support specific to the framework and would add some difficult requirements as well. Any functions that

³The browser itself is in a better position to track this, but it is still likely a difficult task, considering how long it took before browsers could even keep form fields filled out after restarting.

could be restored by a snapshot would have to exist at application startup, otherwise states stored in a previous session or shared by another user could not be loaded. That requirement is also unlikely to be met, as most of these cases happen inside closures. The model-with-listeners architecture could be implemented in a way compatible with Ra by managing access to the model objects and storing the callbacks apart from the state objects, but we are not aware of any systems that work this way.

We found that the parallel coordinates application we used had several of these problems. It uses the D3 framework [6], which maintains listeners on the state objects in the model component, and the code was written in a partly functional style with significant state variables in closures. Restructuring the application to meet Ra’s requirements would have been a large undertaking comparable to rewriting the application.

We developed a mediation mechanism to allow an application to use Ra without significant restructuring when it can’t meet the state object requirements. It puts more responsibility for managing state on the application, so this may be of limited practical value in an application with complex state. However, it was sufficient for the parallel coordinates application.

The Ra API is extended to include *priests*, which are special objects provided by the application that act as interpreters between Ra and the application state. Objects that store state but do not meet Ra’s requirements are still marked with a call to `ra.stateObject` on creation, but a priest name can be supplied as well, as in the call `ra.stateObject(b, "brush")`. Instead of returning a `Proxy` object wrapping `b`, this will return `b` itself. Ra will then delegate responsibility for monitoring, saving, and restoring that object to a priest registered with the given name.

A priest must be registered with `ra.registerPriest(name, priest)` before it can be used. Once the priest is registered, Ra grants it access to internal functions by endowing it with a `rai` property (short for “Ra internals”, but pronounced “ray”). This lets the priest notify Ra when the objects it represents have changed.

A priest object must implement the following member functions that Ra can call:

- `save(object)`, which returns a string representing the state of the `object`.

- `restore(serialization)`, which restores an object from the serialization produced by `save`.
- `destroy(object)`, which destroys the `object` because it no longer should exist in the state Ra is loading.

A method for Ra to notify the priest when it is made responsible for a new object would be needed for more general use; it is omitted only because we did not need it for the parallel coordinates application.

Priests are still compatible with regular use of Ra. When a priest is specified, responsibility for the object is delegated; when the priest argument is omitted, Ra substitutes a proxy and manages the object as usual.

This mechanism was sufficient to adapt our parallel coordinates application to work with Ra. However, it places nearly all of the burden of managing state on the application (through priest objects). The priests must reimplement features that would have been provided by the `Proxy`-based mechanism: Ra will not detect changes to these objects, so the priest has to trigger autosaves; cyclic object graphs must be handled by the priest, and the code for saving and restoring objects both have to be kept synchronized with each other and any changes to the application. Essentially, this mechanism allows the application to use the Ra user interface while forgoing its object management features.

Chapter 5

Usability Evaluation

We conducted a usability evaluation of Ra. We wanted to identify usability problems that would help us improve the interface and to better understand what realistic use of Ra would look like. It is important that using a history system appear to be more efficient than recreating historical states, so we wanted to see how much users would take advantage of Ra when not actually required to use it. The evaluation also allowed us to make refinements to the software and interface design.

We hoped to learn something about the usefulness of a system like Ra to inform future work. Our observations in this study could also be used in designing a more formal long-term experiment to observe how users' behaviour changes when they have access to all previous interaction states, while engaged in real work activity.

5.1 Method

We conducted a usability test of the type described by Rogers *et al.* [24]. We followed the thinking-aloud protocol described by Shneiderman [27], Nielsen [21], and others. At the end, we asked participants to fill in a brief questionnaire.

For collecting data, we recorded on paper our observations and some of the comments participants made. We also configured the system to record the screen and user interaction for later review (and many of the screenshots in this chapter). In a few cases, with the participants' explicit consent, we took photographs of participants interacting with the system.

The study's ethics protocol was cleared by the Carleton University Research Ethics Board on 10 July 2015 as Project #103316. For details, see appendix A.

5.2 Participants

The study comprised two series, held with different participants on different days. Between the series, we made a few easy refinements to the software, described below

in section 5.4.

We recruited a total of twelve participants using the poster shown in appendix A. No participants withdrew after beginning the study. Seven of the participants were in the first series and five in the second. This is within the range of participant counts recommended by Rogers *et al.* [24] for usability testing, both overall and for the series individually.

Of the twelve participants, four (33%) were female (two in each series). All were students or university employees. Most were students in technical and scientific fields. Their ages ranged from 19 to 35, with a median of 25.

All participants reported using a computer daily, and had been using computers for the majority of their lives. All seemed familiar with the standard computer technology we used.

Two participants reported being ambidextrous, and two reported being left-handed.

5.3 Equipment

5.3.1 System setup

Participants interacted with our software using a large wall-mounted touchscreen, composed of a nominally 55-inch television screen with a 69 cm $\times$ 122 cm PQ Labs G3 Plus (32-touch) frame mounted in front of a sheet of glass in front of the screen (see figure 5.11). No other input devices were provided.

The prototype applications were run in Chrome 44.0 (with Proxy support specially enabled as described in section 4.6.1) running on Windows 7. We used a Dell Vostro 460 with an Intel Core i7-2600 processor (circa 2009).

The questionnaire was printed on paper; it is reproduced in appendix A. We reminded participants that all the questions were optional and invited them to ignore the printed directions and fill it in however they liked, in spite of the strict printed directions: “It’s just paper. Write anywhere.”

5.3.2 Applications

We used two of the sample web applications (see section 3.3 in this study, each of which had been modified to include Ra. We held sessions with participants individually, so

Figure 5.1. One configuration of the maze application.

we did not use Sol.

The first application, shown in figure 5.1, was a simple maze game which we developed specifically for this study. The mazes were randomly generated anew for each participant. Unlike traditional mazes, these had multiple solutions (paths through the maze). As the participant moved the position token through the maze, it drew a path of the route taken; all moves (even backtracking) extended the path. There was a counter at the bottom indicating the path length. The path fades from dark at the current position to light at the start position; this helps show any backtracking or crossing more clearly.

Normally, a maze can be viewed as a spanning tree on a graph whose vertices represent maze cells and edges represent connected cells (i.e. the lack of a wall between adjacent cells). This guarantees that there is exactly one path (not counting backtracking) between any two positions in the maze (equivalently, vertices in the graph). Such a maze can be generated using a randomized version of a spanning-tree algorithm. We used a randomized variant of Prim's algorithm, in which the tree is built starting with one arbitrary vertex and adding edges to connect vertices not yet in the tree. Prim's algorithm selects the edge of minimum weight; in a maze, these are the edges (always of length 1) connecting the border region of the maze-so-far to an adjacent cell, and the selected edge is randomly chosen from those options.

Figure 5.2. The parallel coordinates application.

To get a maze with multiple paths, we started several instances of the above algorithm in parallel from random starting positions. Each sub-maze was allowed to expand into the unclaimed parts of the space but not the areas already covered by another sub-maze. Thus the graph was divided into several disconnected trees. Then for each border between sub-mazes, at least one new edge was added, and for longer borders, more than one edge.

The second application was a parallel coordinates graph [12], shown in figure 5.2, an interactive information visualization. This application was based on a sample application [9] for demonstrating the brush component introduced in version 2.5 of D3 [6]; we modified it to use Ra and to be slightly easier to use on a touchscreen. In particular, since the original application was designed to be used with a mouse, the interaction targets were small – only a few pixels across. We increased the width of the brushes, but not the size of the brush edges, which would likely have required more significant internal changes, so resizing brushes unfortunately remained difficult on the touchscreen.

In the first series, the parallel coordinates graphs showed the sample data about cars from the original D3 demo application. We had three identical tabs loaded for the three tasks described in section 5.5. We worried that switching between identical tabs might seem strange to participants, so for the second series, we changed the second

tab to show data about the historical weather in Ottawa instead.

We loaded all the pages, one for each task, in separate tabs before the participant arrived. This allowed us to regenerate the random mazes to try to make sure they were not trivial before the participant arrived. It also protected us in case of a network fault during the study.

5.4 Series changes

We made some small refinements to the applications between the two series of participants. These changes were informed by our experience with the first-series participants, but with limited time available, we were mostly limited to fixing the worst problems.

In Ra, we changed the glyph that represents user-named snapshots from a big dot to a star, making it distinct from the app-promoted snapshots. We did this because we noticed that it was hard to use the feature effectively in the parallel coordinates tasks. (The maze application did not make any app-promoted snapshots, so it was not a problem there.) We also changed the timestamp shown in the balloon popup for each node from an absolute timestamp with date and time to a relative time description (“Just now” or “3 minutes ago”). Since participants did not seem to be reading the labels at all, we wanted to reduce the apparent clutter of a complete timestamp. We also fixed a bug that sometimes left the balloon popup showing even after the user began interacting with the app; the popup should be dismissed as soon as the user taps anywhere outside it.

In the parallel coordinates app, we added a new dataset showing the weather in Ottawa, as recorded by Environment Canada, from 1939 to 2011. We were using a separate tab for each task with the parallel coordinates application because it would keep the history for each task separate and it matched what we were doing with the maze (where the configurations were different for each task). We were concerned that switching to another identical tab might seem strange to participants, so we wanted an obvious justification for switching tabs. We also fixed a bug that caused loading the initial state to fail.

Figure 5.3. Comparison of changes made between the series: *Left*: series 1 savepoint; *Right*: series 2 savepoint.

5.5 Tasks

Participants were verbally led through the following tasks:

1. In a maze, draw a path from the top-left to bottom-right corner.
2. In a maze, draw a path from the origin to any one of several targets on the opposite side.
3. In a maze, draw a path that visits each of several targets scattered randomly throughout the maze without backtracking or crossing the path.
4. With a parallel coordinates graph, answer a simple question (to get accustomed to interacting with the graph).
5. With a parallel coordinates graph, answer a question about the relationship between several of the variables presented in the graph.
6. With a parallel coordinates graph, find an answer to a question and then present the answer using the graph as a visual aid.

We chose these tasks in this order to introduce new concepts incrementally, so that the final task with each application is complex enough to take full advantage of Ra. We could then help participants get started using the software as the study progressed, so that participants were able to operate more or less independently for the last task in each series.

The tasks are described in more detail below.

Figure 5.4. Our demonstration of Ra used a very simple maze.

5.5.1 Maze introduction

We demonstrated how to use the maze and the Ra sidebar to return to previous versions in a very simple maze, shown in figure 5.4. The start position is in the top-left corner, and the target is the six-pointed star in the bottom-right corner.

The participant was then asked to solve a larger maze. They were encouraged to “draw” the “best” path, imagining that their path could be shown as a sample solution for others, rather than simply solve the maze.

When the participant reached the end, we announced the path length (displayed all the time as part of the application) and encouraged them to find a better path, if possible. (If the random maze happened to have a trivial solution, we reloaded the maze so the participant would have a reason to interact with Ra.) One participant’s solution is shown in figure 5.5.

5.5.2 Maze with choice of targets

We then demonstrated the named-save feature (see section 4.5.3) which in the first series made a big dot with a named label and in the second made a star, also with a label. Figure 5.6 shows two big dots as they worked in the first series; stars are shown in figure 5.7.

The participant was then shown a maze with many randomly-placed targets on

Figure 5.5. One participant's solution to the first maze.

the far side. We told participants they could choose any target as the goal, but should choose the one that would result in the best (shortest) path.

As before, the participant was encouraged to improve their solution if possible.

5.5.3 Travelling Salesman maze

The final maze had several randomly-placed targets over the whole map. The participant was asked to find a path that visited each target once without backtracking or crossing that path, such as the one shown in figure 5.7. Although we tried to select random mazes that permitted this, sometimes it was not possible; we then asked the participant to minimize backtracking.

This task was inspired by the famous Travelling Salesman Problem. The task is equivalent to finding a Hamiltonian path on a multigraph where the vertices are the targets (and the start position) and the edges are all paths through the maze between the targets (and the start position), with the additional restriction¹ of some edges being mutually incompatible.

Although the Travelling Salesman Problem is usually formulated as requiring a Hamiltonian *circuit* of minimum weight, we chose the path form because the start

¹It could also be seen as a relaxed variant with more vertices, of which only a subset must be visited. We present it as the usual Hamiltonian path problem with additional restraints instead to make it clear that the maze is at least as hard as the smaller graph problem.

Figure 5.6. This series 1 participant found two likely solutions to the second maze and marked them with big dots; the current path is two segments longer than the other solution.

position was the top-left corner, which makes it hard to return without crossing the path left on the way out. The problems are equivalent in complexity [8].

Finding a Hamiltonian path, even without regard to its length, is an NP-complete problem [25]. And as Dasgupta *et al.* [8] note, “[e]ven in [a] tiny example, it is tricky for a human to find the solution” to a Travelling Salesman Problem.

We tried to adjust tasks as necessary to keep the session length near the 45 minutes we advertised to participants in the poster and consent form, though we did not mention timing to participants so they would not feel rushed. If there was time available, we encouraged the participant to find a shorter path, but we did not insist on finding the shortest path. Most participants did make at least some effort to keep the path short.

5.5.4 Parallel coordinates

We showed participants the parallel coordinates graph and demonstrated how to brush (select) to highlight part of the data. Participants were then given a chance to familiarize themselves with the application. Some participants were eager to try it out and started exploring right away; for other participants we gave a simple question to answer.

Then on a new graph, we asked participants to answer a question about the data

Figure 5.7. One participant's solution to the third maze. The current state has been marked with a star.

that required comparing more than two variables. In the first series, the graph showed sample car data and we asked whether older cars were less performant (fuel economy and acceleration time). In the second series, the graph showed historical weather data for Ottawa (see figure 5.8) and we asked about the relationship between precipitation (snow and rain) and temperature (highs, lows, averages). We did not try to encourage using Ra if the participant wasn't inclined to do so.

5.5.5 Parallel coordinates presentation

We asked participants to answer another question about data on a new graph. In this case, we told participants to figure out the answer, and then be prepared to present their findings to someone who had not been watching the discovery process (imaging that their boss asked for a report), using the graph as a visual aid to support their conclusions. We gave the participants time to figure something out and then asked for their conclusions.

In both series, we asked how the engine (cylinders, size, power) affects performance (acceleration, fuel economy). Although we intended that participants would find several relevant queries for the graph and mark those states for the presentation, as in figure 5.9, we did not explicitly ask them to do so.

Figure 5.8. The weather data (series 2) parallel coordinates graph. This participant has selected the summer months with very high total precipitation.

Figure 5.9. One state in a participant's presentation, showing the car data parallel coordinates graph.

5.6 Observations

Our observations are grouped first by which of the two applications they relate to. Within each subsection, then, observations that were specific to a task are collected together and topics that span multiple tasks are grouped after.

5.6.1 Maze tasks

Recall that there were three mazes: a plain top-left-to-bottom-right one, one that offered a choice of targets, and one based on the Travelling Salesman Problem.

Plain maze Participants always solved – drew a path to the target – the plain maze without really reconsidering the path set down until after reaching the target, but often did put effort into planning a good route before moving in the maze. Most of these solutions could later be improved, however.

One participant had been using Ra for small undo steps, but then when we suggested changing part of the path, they appeared to suddenly realize the possibilities; it seems they hadn’t noticed that Ra would let them do that without starting over entirely, despite having used it to do just that on a smaller scale.

One participant forgot to press the “Load this” button after tapping a dot in the sidebar, and started to interact with the maze again before realizing that the maze was still in the old state and they hadn’t actually changed anything.

Many participants wanted to touch the path to undo, expecting the head to rewind to that point. Some participants tried this several times. We suspect this problem is specific to our simple maze application, since most practical applications will not be as simple, and we did not observe similar behaviour in the parallel coordinates tasks.

Some participants did not take advantage of Ra on their own: they spent significant effort planning the best route before taking action, and then indicated that they had completed the task. If the participant didn’t use Ra on their own, we prompted them to find a better route, which required using Ra to go back (even if it was to start over entirely.) One participant used Ra only the once in this first maze, at our prompting.

Some participants did take quite quickly to the possibilities Ra provided: “oops, I have to make an undo” – “I don’t know whether it’s going to be better, but it’s worth

a try.” – “I’m not used to having undo, I’m used to planning ahead, so it’s nice not to have to worry about that.”

Maze with choice of targets Many participants were quite willing to reconsider early choices and try alternative routes, though as in the plain maze, they still displayed a preference for getting one route down on the screen before questioning too much.

“I’m very happy that there is an undo or I’d be making more of a fool of myself.”

Some participants were still able to solve this puzzle before taking action. For example, one participant preferred to work backwards from the targets to the start position.

Travelling Salesman maze Most participants did use Ra on their own, without prompting. However, one participant never tried to undo anything, and one other participant made a mistake (“oops”) but just continued solving the maze; we had to suggest undoing the error.

One participant was able to plan ahead quite effectively, and used Ra mostly for short undo steps; they realized their path was the wrong approach before it had progressed very far from the better path they eventually chose.

We found it helpful to discuss paths relative to the path currently on the screen. Participants also generally talked about alternative paths relative to the current path.

“It’s nice to have the undo so you don’t have to start over if you make a little mistake.”

Finding the right state: the blind-dot problem “Back to here – I think it’s this one.”

Participants often had at least some difficulty choosing the node that represented the state they wanted to return to. As one participant commented, it’s “hard to tell where you’re going back to”. Some participants expressed this by suggesting things such as thumbnails to preview the state a node represents, or (in series 1) different shaped markers. Even series 2 participants, who could make stars after the first maze, had this problem; they often did not know which state they would later want to return to until much later.

Participants displayed varying levels of dedication to finding the state they wanted. One participant tried about 5 nodes before getting the right one, whereas another would accept any state before the desired one if the path forward was reasonably clear. Most participants appeared to have some good-enough threshold beyond which they would prefer to redo some work rather than search for the right state to restore.

Participants had different ways of coping with this problem. One participant, in going back to revise part of the path, held their left hand over the point on the path they wanted to change while selecting different states with their right until the maze position was close to the left-hand point. Another technique was to deliberately create branches in the history tree at significant nodes (see *Starring*, below). One participant, for example, reused just one branch point, which made it easy to find, rather than search for a more precise point nearby. And some participants estimated how many changes had been made and counted the dots to go back (see *Granularity*, below).

Diff, Patch, Merge In the latter two mazes, especially the Salesman maze, participants often wanted to change the *middle* part of their path, but the desired new path had the same ending as the old one. In Ra, there is no way to do that without actually repeating the identical part. One participant copied part of a different history branch by flipping between the two branches and repeating the path from the old branch in small steps: “I’ll take advantage of that undo history to remind myself of what wobble we were trying to do”.

As another participant commented after starting a new branch and doing the middle bit differently, “then I’m going to do what I did before”. In at least one case, the need to repeat exactly the same end part seemed to be a deterrent to making the change at all.

One participant switched between branches to compare experimental starts with their first solution to help evaluate their viability.

Granularity Some users may have wished for finer-grained automatic snapshots in some places. One participant spent some time trying to find the right position near the beginning, but ended up starting over entirely. Another appeared to want to return to a point between two adjacent snapshots, expressing resignation at choosing

the earlier one.

A third participant had assumed (in the plain maze) that each dot in the tree corresponded to a movement command. We told them that the snapshots were limited to one per second, and later they mentioned that they were slowing down in order to get more intermediate snapshots now that they knew the dots were time-based.

However, participants also had some difficulty finding the desired state to restore, and may have wished for coarser granularity in some places. One went back two steps at a time, and did not go forward the single step when they had gone farther than necessary: “trying to figure out which point is which, I guess... oh, well” (close enough).

Several participants treated the snapshot dots as if they were related to the position or distance in the maze, or one per tap. One participant was counting “so we’ll go back 1, 2, 3, 4, 5,” expecting one snapshot for each tap; they said they were trying to tap only once per straight line so that there would be one history dot per turn. That participant later, on going back, made comments about having gone back farther or not quite as far as expected.

Starring Recall that we introduced user snapshots after the plain maze, so participants could use this feature in the maze with a choice of targets and the Salesman maze. In series 1, user snapshots were represented as big dots in the history tree; in series 2, they were stars. In both cases, they were created by optionally entering a name in the text field, and then pressing the “Save” button beside it.

Most participants at least tried it out after we introduced it, and most of those participants gave a name the first time they created a user snapshot. Most names were short, but one participant named their first point “[name]-1st-43” for their first solution, which had a path length of 43; they used shorter names later on.

One participant didn’t seem to read the labels at all. In the questionnaire, they wrote that they did not understand some of the dots.

Several participants would have benefitted from better feedback when they pressed the save button: one participant regularly pressed the button two or three times in a row (creating that many new snapshots), and another clearly went to check the newly-created dot by tapping it. “Is it saved?”

Figure 5.10. The history tree can get large enough to require scrolling.

In making savepoints, one participant still wanted to select the dot to be marked (made bigger). They would tap the (current state) dot, tap “Load this”, then tap the dot again before using the save button.

In the maze with a choice of targets, one second-series participant did make effective use of starring: they marked a likely branch point, and then returned to it to try the other option. And in the Salesman maze, a different participant marked a significant branch point with a star and later returned to it. Most participants did not proactively mark likely branch points like this.

Stars did seem useful for marking states that would later be compared.

However, another participant made a savepoint every three targets, but never used them. When we suggested finding a shorter route, they started again at the beginning.

One participant used stars effectively in the other tasks but not at all in the Salesman maze.

Tree size and management One participant did very little planning ahead and ended up with an extremely large and wide tree. This was the most branching we observed in this study. In addition to one-step undo, this participant also made moves in the maze with little advance planning, preferring to go back when it was clear that the current route was not good. Figure 5.10 shows part of their history tree near the

end of the Salesman maze task. (Some of the tree requires scrolling to view.)

Two participants whose history trees grew taller than the screen didn't immediately notice what had happened. One was trying to interact with the nodes visible at the bottom, expecting them to be more recent than they actually were. That participant also clearly preferred using on-screen nodes over scrolling.

5.6.2 Parallel coordinates tasks

Recall that we asked participants to use the parallel coordinates graph in two ways: to answer a question interactively, and to present data-based findings to others.

Interactive discovery Most participants made very little use of Ra. Of the participants who did go back at all, most used it almost exclusively for going back to the initial state. One participant complained, it “takes so much time to find the point that you want”. Some did use it for trying other approaches, but this was uncommon and they usually did not appear to get a significant benefit from doing so.

Participants generally did not use Ra to undo small mistakes. Since the selections are quite easy to move and independent on each axis, there is often little to gain from small undo steps (compared to just moving the brushes). Their history tree structures were highly linear compared to the maze tasks.

Presentation Some participants in the first series did not use Ra for presenting their findings at all, and either selected just one graph, or operated the graph while giving the presentation. Several of the participants who did use Ra for the presentation had trouble finding the states they were interested in, since (in series 1) they appeared the same as app-suggested snapshots (see sections 3.4 and 5.4 and figure 5.3). One participant had used Ra in the first parallel coordinates tasks but had given up and mostly just used the graph directly for the presentation task.

Some participants also used interactive brushing as well as jumping between saved views.

Diff One participant who only created branches from the initial state did also look at some other recent versions without creating new branches.

Another participant used saved states to compare different queries by flipping between saved states repeatedly. They wanted a way to do so without having to look at the sidebar between looking at the two versions in question, and mentioned doing this in other applications with the keyboard shortcuts for undo.

Tree management Some participants wanted to manage the states shown in the history tree, such as when one participant accidentally changed the graph after making a savepoint, they wanted to delete the extra state nodes, asking “is there a way to cancel these?”. (This happened in the first series where savepoints and automatic snapshots were both big dots; the change to stars may reduce the need for this kind of management.)

Another participant spent some time browsing the data just out of interest. When they were ready to begin the study task, they wanted to clear out the old history from just browsing around the data.

Starring In the first series, the only way to distinguish between application-savepoints and user-savepoints was by the label, which participants mostly ignored. The distinction was not visible without tapping the node, which made them difficult to identify in the tree. One participant deliberately created branches at the savepoints so they would be visually distinct in the tree.

Many series 1 participants did use explicitly saved states to present their findings. However, they had difficulty identifying those states in the tree. Long chains made this more difficult; the participants who also branched from savepoints returned to those branch points for the presentation. “Hmm. . . trying to navigate here to figure out what we were doing.”

One series 2 participant never made any starred nodes, in any of the parallel coordinates tasks. In the presentation, they just changed the brushes to show the data. The other series 2 participants used stars effectively to mark states they wanted to show in the presentation, though they generally did not assign text labels to the stars.

Participants in series 2 expressed a need for some sort of presentation mode where the history tree would show only the starred states, perhaps linearly and perhaps in a customizable order. One participant who had not labelled their stars said they would

Figure 5.11. The touchscreen layers (not to scale) seen from the side show that the detected touch position and the point the user sees are different.

like to label them afterwards to mark the order.

5.6.3 Hardware

Many participants had some difficulty with the touchscreen. The touch-detection frame we used in this study uses infrared light above the surface to detect fingers (or anything) touching the screen. It registers an object as touching (blocking the infrared light) before the object actually comes in contact with the glass underneath, which itself is above the surface of the screen. This is not a common touch-detection system in consumer devices (such as phones and tablets), whose capacitive touchscreens will also detect a finger slightly before it comes in contact with the glass, but the tolerance is significantly smaller than the PQ Labs frame.

The distance between the touch-detection layer above the glass and the display surface below it causes a parallax effect, where the apparent alignment of touches with the display is distorted at the edges and especially the corners of the screen, as seen from a typical position standing in front of the middle of the screen. That meant that interacting with the Ra sidebar, especially at the bottom, could be difficult, as touches registered beside the point the participant wanted to tap. This effect is illustrated in figure 5.11.

The distance between the plane of touch detection and the surface of the glass also

Figure 5.12. Difficulties using the touchscreen. A participant has accidentally started text-selection mode instead of pressing the “Load this” button, likely because their tap was interpreted as a long swipe.

meant that some touches that were clearly meant to be taps appeared to last longer to the operating system than the participant intended. One consequence of this is shown in figure 5.12 where text-selection mode has been activated by a long press-drag even though the participant’s finger only briefly touched the glass.

5.6.4 Other comments

One participant commented that they “found the undo far more helpful in the maze than the graph.”

One participant mentioned particularly liking Ra’s branching, as most software doesn’t provide that: “All of a sudden I lose my forward history, which is no fun.”

Granularity and the blind dot problem Some participants wrote comments in the questionnaire that also identified what we call the “blind dot problem” of knowing what state a tree node represents:

“I think extended undo would be great in many applications. One challenge I

experienced was that very many undo states were created — perhaps grouping would make them more manageable[...] Also would have been easier if my saved states had been easier to distinguish.”

“I think that it was kind of trick[y] to find the dot that would bring me back to the place that I wanted to go.”

Tree structure Three participants had computer-science backgrounds, and so were already familiar with tree structures. The other participants generally did not have experience with trees, but nobody was confused by the tree. Several participants noted other common uses of tree structures as something similar they’d seen, such as level selection in some games, family trees, flowcharts, and organizational charts.

Left-handedness Two participants reported being left-handed. They seemed generally more reluctant to use the Ra than the other participants. Note that the sidebar is on the right-hand side of a large screen, so it is impractical to operate with the left hand. Left-handed operation might also exacerbate the parallax problems described in section 5.6.3. One of the left-handed participants switched between using their left and right hands to operate the sidebar when they had trouble operating it. Given the small sample size, however, we do not want to conclude that this is a result of left-handedness without further research.

5.7 Questionnaire responses

Following the interactive tasks, we asked participants to fill out a questionnaire on paper. The questionnaire paper is reproduced in appendix A. The responses to the demographics questions were summarized in section 5.2; participants’ free-form comments are included in section 5.6. The questionnaire also included eleven statements which participants were asked to respond to on a five-point Likert scale.

We are not sure how reliable these responses are, because some of them seem to contradict participants’ actual performance. In particular, most participants agreed that they found naming useful, but for most of them, this did not seem to be the case.

The responses are also shown in a box-plot in figure 5.13.

Understood need “I understood the need for the extended undo.” — It is reassuring that all participants but one outlier fully agreed.

Difficult to go back “I found it difficult to go back to earlier states.” — Although no-one fully agreed that it was difficult, the median response was neutral. As our observations suggest, there is room for improvement. Nevertheless, contrast this with *More time than saved*, below.

Understood sidebar “I found it easy to understand how the sidebar worked.” — Most participants agreed; none disagreed. We explained the sidebar and helped participants learn to use it, so this suggests the effort was successful.

Understood dots “I understood the meaning of the different kinds of dots.” — Although most participants at least somewhat agreed, the consensus is weak, with two participants fully disagreeing. Since there were only two (series 1) or three (series 2) kinds of glyphs in the tree, this suggests that introducing any new glyphs should be approached with caution.

Saw how change reflected “When I made a change to the application, I saw how it was reflected in the sidebar.” — This question is very similar to *Understood sidebar*, above, and indeed the distribution of responses matches. Most participants fully agreed. Strangely, however, the individual participants who gave responses other than “agree” are (mostly) not the same as the ones who gave responses other than “agree” to *Understood sidebar*.

More time than saved “I spent more time using the sidebar than it saved me.” — Most participants at least somewhat disagreed, so the general conclusion is that using Ra was worthwhile. However, the contrast with *Difficult to go back*, above, seems strange, since that question was much less conclusive yet it asks much the same thing. Perhaps a future study should compare Ra with a traditional undo system.

Wanted to retry “I reached a point where I wanted to go back to something I’d tried before.” — There is clear agreement, with only two outliers at somewhat

agree and neutral. Since the tasks were designed to put participants in this position, this is not surprising.

Would like elsewhere “I would like to have this extended undo feature in other software or web sites I use.” — Most participants agreed, with some staying neutral, but none disagreed. Again, participants seem generally positive about Ra.

Found touchscreen hard “I found the touchscreen hard to use.” — The responses to this seem pretty evenly spread, and it matches our observations: some participants had no trouble, some got used to the hardware, and others struggled with it. Further discussion of the hardware problems is in section 5.6.3.

Found naming useful “I found it useful to give names to some states (dots).” — More than half the participants fully agreed, yet some of those were the participants who never even tried to assign a name to a snapshot. Furthermore, while many participants tried naming a snapshot at least once, those who continued to make starred snapshots usually did not continue to assign names to them.

Too many dots “I ended up with too many dots to manage.” — The consensus is in the range between neutral and agree; participants thought there were too many dots in the tree. The two outliers who disagreed with this did not make much use of Ra, and their history trees had few branches. For the others, the responses are in alignment with our observations of the granularity problem, and we suggest that this is an important problem if Ra is ever to be used for real work rather than our small, simple study tasks.

5.8 Interpretation

There seem to be three broad categories of reasons for going back to a previous state with Ra. There are accidents to fix, in which only a short time passes between making a mistake and restoring the previous state: this is the traditional “undo”. There are past deliberate decisions to revisit, in which the time difference is greater, more work has been done on the state being discarded, and it may be less obvious that the work

Figure 5.13. Box-plot of usability questionnaire responses. Note that the statements marked “(Not)” have been inverted so that all the positive responses are on the left; for example, most participants agreed that there were too many dots, so this has been reversed to disagreeing with the inverse statement.

was erroneous: we call this “retry”. There are past memories to recall, in which the user views old work not to resume working from there but to remind themselves of some part of that state, either to do the same thing or to avoid making the same mistake: we call this “review”. These tasks are discussed further in section 6.1.3.

These different kinds of undo seem to need different levels of state granularity. Traditional undo requires fine-grained control over recent states; one saved state per interaction is probably ideal, and recalling a previous state is likely to be sequence-based rather than time-based. Retry and review are more likely to cover a greater time period, so coarser granularity and time-based state identification are likely to be more useful.

Naming snapshots does not seem to be a compelling feature. Participants didn’t seem to read the labels, and although some were willing to assign good names to their user snapshots, most soon stopped assigning any name at all. This may be because the study was quite small, spending only around five minutes with each task, and just loading the state to see what was there was easier than naming states in advance. This suggests that labelling states is likely to happen after – perhaps long after – the state is created, when the user realizes that they have too many states to manage otherwise. Or, as in the case of the presentation task, users might benefit more from a label about the use of a snapshot rather than a description, which could not be determined at creation time.

Three of the four series 2 participants made good use of user snapshots. The switch to a distinct star glyph seems to have greatly improved their usability. It certainly made the feature easier to describe. It also decreased the need to name the starred snapshots.

It seems that “save” is the wrong way to present the user-snapshot feature. The term “save” already brings to mind traditional file semantics that do not apply to Ra. In particular, one participant’s initial reaction to assign a name like a filename and another’s repeated pressing of the save button are typical behaviour with traditional files that does not translate well to Ra.

There is a need for some tree management features, but care is needed to ensure the features’ benefit outweighs the management burden (discussed in section 2.2.2).

This study revealed two management tasks worth further study: In the first case, a user may be working on two different tasks, either sequentially or in parallel, and want to keep their history from getting mixed. (We essentially enforced this in the study by switching to different applications in different tabs.) In the second, a user may want to view only a subset of the nodes in their history tree. The user may have created states to which they are sure they will never return (either an accident or a discarded branch), and want to hide those states to focus on the rest of the tree. The user may want to present a specific portion of their history to others. And for “retry” operations, “undo”-level nodes need not be seen. Perhaps hiding the history tree until it is needed would reduce the urge to manage the tree.

The blind dot problem affected all the participants who made good use of Ra. Some mechanism is needed to help users choose the right state from the tree when they have a fragile mental image of that state. This problem is related to the problem of comparing two different states, and a feature such as a preview mode might solve both.

In the maze application, being able to merge states or copy part of one state to another would have helped participants. We did not think participants needed to merge in the parallel coordinates application. It is likely that the idea of merging states will be application-specific, but having all states presented to the user will raise the issue.

Presenting snapshots with Ra is more versatile than taking screenshots of the states because the application can still be used interactively, and without interfering with the rest of the presentation.

The results of the study suggest that while there are some important improvements to be made, as discussed above, Ra is generally usable by non-experts for small tasks as used in the study. It suggests that a system like Ra is useful and worth exploring further.

Chapter 6

Discussion and Conclusion

6.1 Discussion

We implemented a library for recording and exploring interaction history in Web applications. It presents past states in a tree beside the application, so users can return to, and resume work from, any past state. We performed a usability evaluation in which participants used a simple puzzle and an interactive data analysis tool with Ra. Our experience implementing Ra and our observations from the study reveal several important themes.

6.1.1 Capturing state

Ra operates at the same level as the application it is attached to – as a library, it must be included by the application author. This limits Ra’s access to internal application state, which is naturally protected by standard encapsulation techniques that have been promoted by the software engineering community at least since the rise of object-oriented programming. To work around the constraints of encapsulation, we used **Proxy** objects to intercept interaction with state objects. This relies on the application to select which objects Ra can see, but it also puts the application in a position to provide useful contextual information that can benefit the user, such as leaving some types of state (such as the state of a view in contrast to data) out of Ra’s control, and providing Ra with useful labels for important states the user might want to review.

Although some common languages have built-in support for serialization of objects, this is not a useful technique for capturing application state. These systems do not track which objects will need to be serialized, and they do not support replacing existing state objects; unpacking a serialized object creates the objects in a new environment, but other parts of the application may still have references to the old objects. These problems are solved by the **Proxy** system Ra uses.

6.1.2 Mutability and source control

The facilities Ra provides are similar to those provided by source control systems. However, our usability evaluation suggests that there should be some notable differences.

Source control systems store metadata with each changeset (or commit), such as the author, the date, a title and comments, and others, depending on the system. Ra, too, stores the date and a title with each snapshot. This metadata is immutable: once the changeset or snapshot is created, the associated metadata can't be changed either. (In Git, the changeset can be deleted and a new one with new metadata created, but it will have a different identifier. The Mercurial project takes a stricter view of history: by default, history cannot be changed; although the re-create method used in Git is possible, it requires the user to explicitly enable history-rewriting functionality.)

Our observations from the usability evaluation suggest that at least some of this metadata *should* be mutable in Ra. The significance of a state may not be apparent until some time after it has been passed, and asking users to try to label states as they are created may cause more problems with premature commitment.

Similarly, whether a state is specially marked as a star or not should not be fixed forever when the state is first stored. This is parallel to the concept of tags, which (in some source control systems, including Git) can be changed after being created.

A further question of mutability concerns deleting unwanted states and other tree-management tasks. Two dangers are that tree management may take more time and effort than it's worth (see section 2.2.2), and that a user might inadvertently delete something they later want to recover. (The idea of having an undo for an interaction history support tool seems silly at first, but it is not unprecedented: Git's reflog is essentially that.) Losing history unexpectedly could be worse than not having it in the first place if the user has come to rely on having that history available.

6.1.3 Kinds of history tasks

In the study, we observed three kinds of history tasks. This categorization is not directly about the user's intent, for which there would surely be more than three categories, but the relationship between the state the user was in (old) and the state

to which the user went (new).

Correcting mistakes In the “oops-undo” case, the user has made a mistake recently, or tried to perform an action but the computer did something unexpected. The old state was clearly wrong; the user does not expect to need it again, and perhaps it should be hidden from view. This is the case that traditional undo was designed for, and it is reasonably well-suited to it.

Trying alternatives In the “undo-retry” case, the user wants to try some alternative, usually starting from further back in history than the oops-undo case, or from a parallel timeline. New work will be based on the new state, but the user may not be certain that the old state should be discarded; the old state may still be useful.

Ra was intended to support this task in particular. Traditional undo mechanisms force the user to give up one branch to work on another, which requires the user to commit to a decision before they see the result; they may have to resort to manual version control (saving the file separately for each experiment) or make a decision with incomplete information. Ra allows the user to keep any number of parallel alternatives without the extra costs of saving and managing alternatives in files.

Comparing versions In the “undo-review-redo” case, the user just wants to *look at* a previous version of their work. It might be to compare two alternatives, or to copy a particular piece of a previous solution, or even to remind themselves of what *not* to do. The old state is still the working copy where new editing work will happen; the new state is not something the user wants to keep.

Using traditional undo for this task is particularly dangerous because any accidental edit will discard the redo stack, leaving the user in a state they intended to abandon. Ra happens to support it better, since all versions are accessible, but there are features that could improve the experience, such as some way to keep track of the current *working* branch separately from the version being viewed. We did not think of this task when initially designing Ra, so it is an interesting outcome of the study that participants did this anyway.

Reviewing sequences of past states without editing them may also be able to help other people understand the final state. For example, Farah and Lethbridge [11] developed a linear timeline for reviewing the development of software engineering models. In the field of intelligence analysis, the system could be used for a kind of traceability, allowing analysts to review the decisions that led them to a conclusion. Ra could emphasize this capability by making it easy to explore the path from a state to the root of the tree – that is, the work that went into a selected state, ignoring parallel timelines.

6.1.4 Merges

There have been several indications that some kind of facility to merge selected parts of two or more states would be a good idea. This sort of function is not generally available in everyday software, but current research such as Meng *et al.* [16] suggests that the best way is grouping the product of the application into components for working with history. There are similar approaches in some source control systems. In CVS and Subversion, each file has its own independent history rather than having one timeline for all files in a project. Darcs manages history as a set of patches, with dependencies between patches dictating how some patches may be removed or re-ordered even if they are not the most recent changes.

6.2 Conclusion

6.2.1 Summary

We developed Ra, a prototype JavaScript library for Web applications that lets users keep a history of all their interaction states. In contrast to other models of history, Ra tracks state automatically and does not discard state at all. We ran a usability study of our system with two applications designed to encourage the kind of exploratory behaviour we wanted to support. We identified usability improvements that could be made, but the study suggests that this kind of system could be generally useful even in non-specialized fields.

6.2.2 Contributions

Our main contributions in this thesis are as follows:

- We presented Ra, a system for storing and working with interaction history in Web applications. In the interface design, we suggests that a tree is a good model for presenting history to the user. Ra also demonstrates that `Proxy` objects in JavaScript are a practical, though not ideal, way to overcome encapsulation as needed to capture state.
- We described general problems based on users' interaction in a usability study that should affect the design of any interaction history support system.

6.2.3 Limitations and future work

Ra does not support several features that our study observations suggest are wanted. For example, Ra's user interface maintains a clear separation from the application: Ra is the sidebar on the right; the application takes the space on the left. If the application could be divided into components, the user might be able to gain the benefits of merges without Ra having to understand the semantics of the state it stores.

Our evaluation of Ra was limited to a small usability test. Future work should include a longer-term study of a history system like Ra. In particular, the study should engage users in real work activity, rather than simple five-minute mazes. Such a study would be important for drawing stronger conclusions about how human behaviour changes when epistemic interaction is better supported.

Bibliography

- [1] ECMAScript 2015 language specification. Standard ECMA-262, Sixth Edition, Ecma International, June 2015. <http://www.ecma-international.org/publications/standards/Ecma-262.htm>.
- [2] Gregory D Abowd and Alan J Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.
- [3] Marc Andreessen. NCSA Mosaic technical summary. Technical report, National Center for Supercomputing Applications, February 1993.
- [4] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The world-wide web. *Communications of the ACM*, 37(8):76–82, August 1994.
- [5] Alan Blackwell and Thomas Green. Notational systems—the cognitive dimensions of notations framework. *HCI Models, Theories, and Frameworks: Toward an Interdisciplinary Science*, 2003.
- [6] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D³ data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2301–2309, 2011. See <http://d3js.org/>.
- [7] Judith Brown, Jeff Wilson, Stevenson Gossage, Chris Hack, and Robert Biddle. *Surface Computing and Collaborative Analysis Work*. Number 19 in Synthesis Lectures on Human-Centered Informatics. Morgan & Claypool Publishers, 2013.
- [8] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2008.
- [9] Jason Davies. Parallel coordinates [example]. <http://bl.ocks.org/jasondavies/1341281>, November 2011. Git commit 85c1cedeba1ba1b9bd-d2f5e9aac16ef7dc5bf2e1.
- [10] A.K. Dewdney. Computer recreations. *Scientific American*, 252(2):18–21, February 1985.
- [11] Hanna Farah and Timothy C. Lethbridge. Temporal exploration of software models: A tool feature to enhance software understanding. In *Proceedings of the 14th Working Conference on Reverse Engineering, WCRE '07*, pages 41–49. IEEE Computer Society, 2007.

- [12] Alfred Inselberg. Parallel coordinates: Interactive visualisation for high dimensions. In Robert Liere, Tony Adriaansen, and Elena Zudilova-Seinstra, editors, *Trends in Interactive Visualization*, Advanced Information and Knowledge Processing, pages 49–78. Springer London, 2009.
- [13] David Kirsh and Paul Maglio. On distinguishing epistemic from pragmatic action. *Cognitive Science*, 18(4):513–549, 1994.
- [14] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.
- [15] Bongshin Lee, P. Isenberg, N.H. Riche, and S. Carpendale. Beyond mouse and keyboard: Expanding design considerations for information visualization interactions. *Visualization and Computer Graphics, IEEE Transactions on*, 18(12):2689–2698, December 2012.
- [16] Chii Meng, Motohiro Yasue, Atsumi Imamiya, and Xiaoyang Mao. Visualizing histories for selective undo and redo. In *Computer Human Interaction, 1998. Proceedings. 3rd Asia Pacific*, pages 459–464. IEEE, 1998.
- [17] Bram Moolenaar. The undo tree. In *Vim User Manual*. Vim version 7.4 edition, July 2010. `usr_32.txt`.
- [18] Mozilla Develop Network. Proxy. https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Proxy. Accessed: 2015-03-17.
- [19] Mozilla Develop Network. Working with bfcache. https://developer.mozilla.org/en-US/docs/Working_with_BFCache, October 2014. Accessed: 2015-09-09.
- [20] Mozilla Developer Network. ECMAScript 6 support in mozilla. https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_6_support_in_Mozilla. Accessed: 2015-03-17.
- [21] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1993.
- [22] Jakob Nielsen and Robert L. Mack, editors. *Usability Inspection Methods*. John Wiley & Sons, Inc., 1994.
- [23] Marc J Rochkind. The source code control system. *Software Engineering, IEEE Transactions on*, SE-1(4):364–370, December 1975.
- [24] Yvonne Rogers, Helen Sharp, and Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction*. Wiley, third edition, 2011.
- [25] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. Sixth edition.

- [26] Ben Shneiderman. Direct manipulation: A step beyond programming languages. *SIGSOC Bull.*, 13(2-3):143–, May 1981.
- [27] Ben Shneiderman and Catherine Plaisant. *Designing the User Interface: Strategies for effective human-computer interaction*. Addison-Wesley, fourth edition, 2005.
- [28] Herbert A. Simon. Rational choice and the structure of the environment. *Psychological Review*, 63(2):129, 1956.
- [29] Jeff Wilson, Judith M. Brown, and Robert Biddle. Ach walkthrough: A distributed multi-device tool for collaborative security analysis. In *Proceedings of the 2014 ACM Workshop on Security Information Workers*, SIW '14, pages 9–16, New York, NY, USA, 2014. ACM.

Appendix A

Usability evaluation documents

On the following pages, the following documents related to our usability study are reproduced: the completed ethics review application form, the ethics review clearance notice, the recruiting poster we used, the consent form signed by study participants, and the questionnaire we asked participants to complete at the end of the study.

CUREBs - Research Ethics Submission Form

1. Project Snapshot		
1A. Lead Researcher's Name (Detailed instructions)	Academic Staff	Last name/First name
	Library or Other Staff	Response: Simonyi/Peter
	Post Doc Fellow	
	Ph.D. Student	
	☒ Master's Student	
	Undergraduate	
	Student Association/Clubs	
	Other	
1B. Project Title (Detailed instructions)		
		Title of Research Project
		Response: Usability of Ra, an interaction history manager
1C. Academic Supervisor (Detailed instructions)		
☐ Same as lead researcher	List the academic supervisor(s) here by providing the following information: 1. Last name/First name 2. email address 3. affiliation	
	Response: Biddle/Robert robert.biddle@carleton.ca Professor of Computer Science, Carleton University	
1D. Project Team Members (Detailed instructions)		
☐ Not applicable/No other team members	List the project team members here. For each team member, provide the following: 1. Last name/First name 2. email address 3. role in project 4. affiliation	
	Response: Wilson/Jeff jeffwilson3@cmail.carleton.ca Assisting in conducting study Master's student, Carleton University	
1E. Start Date of Project (Detailed instructions)		
☐ Not applicable	When will you start recruiting participants? (DD/MM/YYYY)	
	Response: 8 July 2015	
1F. End Date of Project (Detailed instructions)		
☐ Not applicable	When will you stop interacting with human participants? (DD/MM/YYYY)	
	Response: 30 April 2016	

CUREBs - Research Ethics Submission Form (updated 2015-02-12)

1/10

1G. Study Goal (Detailed instructions)		What question will your research answer (1-2 sentences)?
		Response: Our prototype software allows users to explore and return to earlier interaction states: a more general kind of undo. Our study will explore how people will use this facility.
1H. Study Purpose and Benefits (Detailed instructions)		Why should the research question be answered, what are the benefits, and to whom?
		Response: All users of complex software make decisions that they may later wish to change. Our prototype software is designed to support this; we therefore wish to explore this specific design to see how it supports their interaction with web applications.
1I. Minimal Risk Review Request (Detailed instructions)		Should this protocol be considered for minimal risk review? If so, please briefly justify.
☒ Yes		Response: The participants are not part of a vulnerable population. Participants will just be using our web sites and our prototype software in order that we can determine its usability.
☐ No		
1J. Project Funding (Detailed instructions)		Who is funding this project? (f applicable, Include the funding source, program, and award name.)
☒ Tri-Council Funded		Response: NSERC Strategic Research Network, Surface Computing Applications RT 35645 (NETGP 37091-08)
☐ Other Award/Grant		
☐ Contract Funded		
☐ Personal Consulting or Personal Work		
☐ Scholarship		
☐ Unfunded		
1K. Researcher Funding (from contracts only) (Detailed instructions)		How much of the funding for this project is going directly to the researcher(s) as income? Include the dollar amount and the percentage of the total funding amount. Will this create a real or perceived conflict of interest and how will it be managed? Provide the title and date of any contracts. (The REB may review the contract.)
☐ The researcher will receive a portion of the funds as personal income		Response:
☐ No funds are paid directly to the researcher as personal income		
☐ A copy of the contract/agreement has been submitted to the Research Office		
☒ Not contract funded research		

CUREBs - Research Ethics Submission Form (updated 2015-02-12)

2/10

2. Methods: Participants		
2A. Participant Interactions (Detailed instructions)	☒ Directly interacting with participants	Describe how the research team will interact with the participants. If the research involves secondary analysis of data that has already been collected, describe participants, the data, and how the data will be used. Response: Participants will be given a brief demonstration on our prototype (~5 min). They will then be asked to perform a sequence of simple tasks requiring use of web sites and our prototype. They will be asked to use the "think aloud" protocol [see Sharp et al. <i>Interaction Design</i> (2 nd ed.) sect. 7.6].
	☒ Observing participants	
	Secondary Analysis of Anonymous Data	
	Secondary Analysis of Anonymized Data	
	Secondary Analysis of Coded or De-identified Data	
	Secondary Analysis of Indirectly Identifying Data	
☒ Secondary Analysis of Directly Identifying Data		
2B. Description of Participants (Detailed instructions)		Describe the participants and any inclusion criteria. Include any control groups (if applicable). Response: Anyone who is an experienced computer user, aged 18 years or over.
2C. Exclusion Criteria (Detailed instructions)	☒ Not Applicable	If applicable, describe any exclusion criteria. Response:
2D. Number of Participants (Detailed instructions)		What is the number of participants requested? (Provide statistical rationale if available.) If the number is large or there are significant risks, provide a justification. Response: 30 (The usability technique is qualitative, so statistical rationale is not appropriate.)
2E. Vulnerable Population (Detailed instructions)	☒ Not Vulnerable Population ☐ Vulnerable Population	Describe any pre-existing vulnerabilities associated with the proposed participant group(s) that may cause additional risks. Describe the associated risks and mitigation strategy. Response:

2F. Participant Relationship to Researcher (Detailed instructions)	☐ Students	Describe any relationship that exists between the researcher and the participants. Indicate how relationships will be managed so there is no undue pressure put on participants. Response:
	☐ Clients	
	☐ Employees	
	☐ Friends/Family	
	☐ Other	
	☒ No previous relationship	
2G. Conflict of Interest (Detailed instructions)	☐ Financial ☐ Commercial Entity Benefits ☐ Other ☒ No conflicts	Describe any real or perceived conflicts of interest that could affect participant welfare. Response:
3. Methods: Recruitment		
3A. Location of Recruitment (Detailed instructions)	☒ Carleton	List all recruitment locations. If some locations require permission prior to recruitment, indicate if permission has been secured. Response: Posters around university.
	☐ Other Canadian School/University	
	☐ Canada	
	☐ Online	
	☐ Other	
	☐ Not applicable	
3B. Third Parties in Recruitment (Detailed instructions)	☐ Third Parties ☒ Not applicable	If using third parties to recruit, indicate who is doing the recruitment and how it will be accomplished. (Third parties refers to people or organizations other than the research team who will be assisting with recruitment.) Response:
3C. Recruitment Materials and Methods (Detailed instructions)	☒ Posters	Describe each step of how participants will be recruited. This includes how contact information is obtained, how participants will be made aware of the study, and how participants can express their interest. Response: Please see attached sample poster.
	☐ Social Media	
	☐ Student Participant Pool (e.g. SONA)	
	☐ Emails	
	☐ Letters	
	☐ Telephone	
	☐ Snowballing	
	☐ Other	
	☐ Not applicable	

3D. Recruitment risks to Participants (Detailed instructions)	☒ No risks or not applicable	Describe any risks to participants during the recruitment phase.
	☐ Mild risks	Response:
	☐ Moderate risks	
	☐ Extreme risks	
3E. Recruitment risks to Researcher (Detailed instructions)	☒ No risks or not applicable	Describe any risks to the research team during the recruitment phase.
	☐ Mild risks	Response:
	☐ Moderate risks	
	☐ Extreme risks	
3F. Researcher Training (Detailed instructions)	☒ No training provided/Not applicable	Describe what training the researcher has (or will receive) to work with the participants.
	☐ Research will be trained	Response:
	☐ Researcher is trained	
3G. TCPS2 (Detailed instructions)	☐ Completed the online TCPS tutorial	The TCPS tutorial can be found here: http://www.pre.ethics.gc.ca/eng/education/tutorial-didacticiel
	☒ Have not completed the online TCPS tutorial	Response:
3H. Benefits (Detailed instructions)	☒ No benefits/Not applicable	Describe any direct benefits to the research participants (as compared to benefits to society or knowledge)?
		Response: No direct benefits. (Indirect benefits include making software easier and more flexible.)
3I. Compensation (Detailed instructions)	☐ No Compensation/Not applicable	Describe all compensation/remuneration and indicate when participants will receive the compensation. What is the monetary value of the compensation/remuneration? What happens to the compensation if a participant withdraws?
	☒ Monetary Gift	Response: \$10 cash
	☐ Gift Card	If a participant begins the study, they may keep the money even if they withdraw part way through.
	☐ Reimbursement of Travel Expenses	
	☐ Refreshments	
	☐ Course Credit	
	☐ Other	

4. Methods: Informed Consent

4A. Obtaining informed consent (Detailed instructions)	☒ Signed consent	Describe the method for obtaining informed consent from the participants. In addition: Oral consent - Indicate why oral consent is being used instead of signed consent. Parent/Guardian - How will consent be obtained from the parent (or legal guardian) when a participant cannot fully consent? How will the participant then assent? Implied consent - Indicate why implied consent is being used.
	☐ Online consent	
	☐ Oral consent	
	☐ Implied consent	
	☐ Parent/Guardian consent	Response:
	☐ Assent	Consent form attached.
	☐ Other	
5. Methods: Data Collection		
5A. Overview (Detailed instructions)	☐ Not applicable	Describe what will happen to, or will be required of, the participants during the course of the research. (Break down by phases if required.)
		Response: 0. Participants will be presented with the consent form and asked to read, agree, and sign it before proceeding. 1. Participants will be given a brief demonstration on our prototype (~5 min). 2. They will then be asked to perform a sequence of simple tasks requiring use of our web sites (sample applications such as a simple maze and graph) and our prototype. They will be asked to use the "think aloud" protocol [see Sharp et al. <i>Interaction Design</i> (2 nd ed.) sect. 7.6]. 3. They will be given a brief demographic and usability questionnaire.
5B. Location of Data Collection (Detailed instructions)	☒ Carleton	Specify where the research will take place. (Include room/lab number if applicable.) If the location has special requirements, such as police checks for researchers, indicate it here.
	☐ Canada	
	☐ Online	Response: HCI building room 2110/2116
	☐ Other	
5C. Data Collection Methods (Detailed instructions)	☒ Questionnaires / Surveys	Describe the method of data collection being used and attach a copy. If data collection is being done online, visit the FAQ section (insert link) for full details on what information the REB requires.
	☐ Interviews	Response:
	☐ Focus Groups	We will use the "think aloud" protocol [see Sharp et al. <i>Interaction Design</i> (2 nd ed.) sect. 7.6] and a brief survey (attached).
	☐ Oral and/or Visual Stimuli	
5D. Photography or Recordings (Detailed instructions)	☐ Not applicable	If the participant will be photographed, video-recorded or audio-recorded, indicate how the data will be acquired and protected.
	☒ Photographs	Response:
	☐ Audio Recording	With their oral permission, we will occasionally photograph participants' interaction with the computer screen (including their hands but not including their face or body in the photo).
	☐ Video Recording	All data will be non-identifiable and will be maintained on password-protected computers.

5E. Translation or Transcription (Detailed instructions)	☐ Translation	If you require the services of a translator or transcriber, describe what services you will use and how you will interact with the translator and/or transcriber. Response:
	☐ Transcription	
	☐ Researcher will Translate or Transcribe	
	☒ Not applicable	
5F. Bio-interactions (Detailed instructions)	☐ Biological specimens/fluids	Describe the apparatus and methods to acquire biological specimens or fluids. (e.g., blood, saliva, tissue samples.) How will specimens be safely destroyed? If any will be kept, explain why, how and for how long. Response:
	☒ Not applicable	
5G. Bio-instruments (Detailed instructions)	☐ Bio-instruments	Bio-instruments touch or send energy into the body. (e.g., electrodes, MRI/X-ray.) Describe the apparatus and its use. If applicable, explain any significant risks and compare the dose (e.g., electrical, radiation) to established safety standards. Response:
	☒ Not applicable	
5H. Bio-interventions (Detailed instructions)	☐ Bio-interventions	Describe the apparatus and methods associated with the bio-intervention. (e.g., drug, stress, medical devices.) Explain any risks to the participants and compare it to established safety standards. Response:
	☒ Not applicable	
5I. Magnitude and Probability of Psychological Harm (Detailed instructions)	☒ No risks or not applicable	Describe the nature of the risks and provide a rationale for your selection (i.e., why you selected "no risks", "mild", "moderate", or "extreme"). Response: No risks beyond normal computer use.
	☐ Mild risks	
	☐ Moderate risks	
	☐ Extreme risks	
5J. Magnitude and Probability of Physical Harm (Detailed instructions)	☒ No risks or not applicable	Describe the nature of the risks and provide a rationale for your selection (i.e., why you selected "no risks", "mild", "moderate", or "extreme"). Response: No risks beyond normal computer use.
	☐ Mild risks	
	☐ Moderate risks	
	☐ Extreme risks	
5K. Magnitude and Probability of Social and/or Economic Harm (Detailed instructions)	☒ No risks or not applicable	Describe the nature of the risks and provide a rationale for your selection (i.e., why you selected "no risks", "mild", "moderate", or "extreme"). Response: No risks beyond normal computer use.
	☐ Mild risks	
	☐ Moderate risks	
	☐ Extreme risks	

CUREBs – Research Ethics Submission Form (updated 2015-02-12)

7/10

5L. Magnitude of Deception (Detailed instructions)	☒ No deception or not applicable	If using deception, please describe the nature of the deception, why it must be used, and the procedures that will be used to protect the participants. Response:
	☐ Mild deception	
	☐ Extreme deception	
	☐ Not applicable	
5M. Debriefing (Detailed instructions)	☐ Participants will be debriefed	Describe the debriefing process which will be used and attach a copy of any documents that will be provided to participants. Response:
	☒ Not applicable/not required	
5N. Debriefing risks to Participants (Detailed instructions)	☒ No risks or not applicable	Describe any risks to participants during the debriefing phase. Response:
	☐ Mild risks	
	☐ Moderate risks	
	☐ Extreme risks	
5O. Incidental Findings (Detailed instructions)	☒ Not applicable/No incidental findings anticipated	Describe possible incidental findings and how they will be managed. (Your approach should be described in the informed consent.) Response:
	☐ Low probability of incidental findings	
	☐ High probability of incidental findings	
	☐ Not applicable	
5P. Withdrawal (Detailed instructions)	☐ Not applicable	Describe the procedures for a participant to withdraw, including any limitations and the withdrawal deadline. Response: Participants may withdraw at any time before leaving.
	☐ Participants can withdraw after the study is complete	
	☒ Participants can only withdraw during the study	
	☐ At some points participants withdraw safely	
	☐ Participants cannot withdraw	

6. Methods: Data Storage and Analysis

CUREBs – Research Ethics Submission Form (updated 2015-02-12)

8/10

6A. Identifiability of data (Detailed instructions)	☒ Anonymous	Describe the identifiability of research data, including how pseudonyms will be assigned, if applicable. If there are different levels of anonymity for different groups, describe each level here. <i>Response:</i> We will not keep any identifying data about participants (except, of course, their signature on the consent form and for receipt of compensation).
	☐ Pseudonyms/Coded	
	☐ Real participant names with data attributable	
	☐ Real participant names with data non-attributable	
	☐ Different levels of anonymity for different groups of participants	
6B. Data storage (Detailed instructions)	☐ Encrypted	How is the data being stored and kept safe? Provide details for electronic data and hard copies. <i>Response:</i>
	☒ Anonymous data	
	☐ Other	
6C. Data Disposition (Detailed instructions)	☐ Retained	After project completion, describe how the data will be stored for future use, returned to participants, or destroyed. <i>Response:</i>
	☐ Returned	
	☒ Destroyed	
6D. Data Breach Risks (Detailed instructions)	☒ Mild risk to participants	Describe how likely a data breach is to occur and how will it affect the participants. <i>Response:</i> Unlikely, because data will be kept on password-protected computers. Mild risk, because the stored data is anonymous.
	☐ Moderate risk to participants	
	☐ Extreme risk to participants	
7. Comments on the New Form (Optional)		
7A. Ease of Completion	☐ Strongly agree	Do you agree that this form is easier to complete than the previous ethics form? <i>Response:</i>
	☐ Agree	
	☐ Neither agree or disagree	
	☐ Disagree	
	☐ Strongly disagree	
7B. Speed of Completion	☐ Strongly agree	Do you agree that this form is faster to complete than the previous form? <i>Response:</i>
	☐ Agree	
	☐ Neither agree or disagree	
	☐ Disagree	
	☐ Strongly disagree	

7C. Form Preference	☐ Strongly agree	Do you agree that you prefer this form to the previous form? <i>Response:</i>
	☐ Agree	
	☐ Neither agree or disagree	
	☐ Disagree	
	☐ Strongly disagree	
7D. Comments	☐ No comments	Do you have any comments or suggestions on the new ethics application form? Please describe them here. <i>Response:</i> I wish it weren't a docx.

Ethics Clearance Form – New Clearance

This is to certify that the Carleton University Research Ethics Board has examined the application for ethical clearance. The REB found the research project to meet appropriate ethical standards as outlined in the *Tri-Council Policy Statement: Ethical Conduct for Research Involving Human, 2nd edition*, and the *Carleton University Policies and Procedures for the Ethical Conduct of Research*.

Date of Clearance: July 10, 2015

Researchers: Peter Simonyi (Student Research: Master's Student)

Jeff Wilson (Student Research: Master's Student)

Department: Faculty of Science\Computer Science (School of)

University: Carleton University

Research Supervisor (if applicable): Prof. Robert Biddle

Project Number: 103316

Alternate File Number (if applicable):

Project Title: Usability of Ra, an interaction history manager

Funder (if applicable): NSERC Strategic Research Network, Surface Computing Applications

Clearance Expires: May 31, 2016

All researchers are governed by the following conditions:

Annual Status Report: You are required to submit an Annual Status Report to either renew clearance or close the file. Failure to submit the Annual Status Report will result in the immediate suspension of the project. Funded projects will have accounts suspended until the report is submitted and approved.

Changes to the project: Any changes to the project must be submitted to the Carleton University Research Ethics Board for approval. All changes must be approved prior to the continuance of the research.

Adverse events: Should a participant suffer adversely from their participation in the project you are required to report the matter to the Carleton University Research Ethics Board. You must submit a written record of the event and indicate what steps you have taken to resolve the situation.

Suspension or termination of clearance: Failure to conduct the research in accordance with the principles of the Tri-Council Policy Statement: Ethical Conduct for Research Involving Humans, 2nd edition and the Carleton University Policies and Procedures for the Ethical Conduct of Research may result in the suspension or termination of the research project.

Louise Heslop

Chair, Carleton University Research Ethics Board

Andy Adler

Vice-Chair, Carleton University Research Ethics Board

Help us understand undo!

All users of complex software make decisions that they may later wish to change. Our prototype software is designed to support this by allowing users to explore and return to earlier interaction states: a more general kind of undo. This study explores how people will use this facility to see how it supports their interaction with web applications.

Participants will be asked to complete a few tasks using our software and answer a few questions about it. We expect this will take about 45 minutes.

To participate in this study, you must be 18 or older.

All participants will be given \$10 in cash.

For more information about this study, or if you're interested in participating, contact the researcher, Peter Simonyi:

undo@hotsoft.carleton.ca

*The ethics protocol for this study has been reviewed and cleared by the Carleton University Research Ethics Board. You can contact them at 613-520-2517 or ethics@carleton.ca
Ethics protocol clearance date: 10 July 2015*

Undo software study
undo@hotsoft.carleton.ca

Undo software study
undo@hotsoft.carleton.ca

Undo software study
undo@hotsoft.carleton.ca

Undo software study
undo@hotsoft.carleton.ca

Undo software study
undo@hotsoft.carleton.ca

Undo software study
undo@hotsoft.carleton.ca

Undo software study
undo@hotsoft.carleton.ca

Undo software study
undo@hotsoft.carleton.ca

Undo software study
undo@hotsoft.carleton.ca

Undo software study
undo@hotsoft.carleton.ca

Undo software study
undo@hotsoft.carleton.ca

Undo software study
undo@hotsoft.carleton.ca

Usability of Ra, an interaction history manager – consent form

About the study: All users of complex software make decisions that they may later wish to change. Our prototype software is designed to support this by allowing users to explore and return to earlier interaction states: a more general kind of undo. This study explores how people will use this facility to see how it supports their interaction with web applications.

This study is funded by the Natural Sciences and Engineering Research Council of Canada.

Your tasks: You will be given a brief demonstration on our prototype. You will then be asked to perform a sequence of simple tasks requiring use of web sites and our prototype. You will be asked to “think aloud” while using the software. Finally, you will be given a brief demographic and usability questionnaire.

We expect this will take about 45 minutes. We do not expect that this will involve any extra risks beyond those you would normally experience from using a computer.

Compensation: As compensation for participating, you will be given \$10 in cash; this is yours to keep even if you withdraw before completing the study.

Photography: While you are using the software, we may ask to photograph your interaction from the back (showing your hands/arms and the screen; your face will not be in any photos). We will keep the photos for analysis and may include them in our published results. We won't do this unless you agree at the time.

Confidentiality: We will not record identifying information except on this form, which will be separate from all other records. Only the researchers listed below will have access to the data. We will erase the raw data by 30 April 2016.

Right to withdraw: You may withdraw from this study at any time before you leave, without penalty. Just tell us you'd like to withdraw. If you do, we will not keep any records other than this form. You do not have to answer any questions you do not want to.

Ethics: The ethics protocol for this study has been reviewed and cleared by the Carleton University Research Ethics Board. Clearance for data collection began on 10 July 2015 and expires on 31 May 2016.

If you have any ethical concerns about this study, you may contact: Louise Heslop, Chair of the Research Ethics Board, Carleton University Research Office, 511 Tory Building, 1125 Colonel By Drive, Ottawa, K1S 5B6; 613-520-2517 or ethics@carleton.ca

Contact: The researchers for this study are Peter Simonyi <peter.simonyi@carleton.ca>, Robert Biddle <robert.biddle@carleton.ca>, Jeff Wilson <jeffwilson3@cmail.carleton.ca>, of the School of Computer Science, Carleton University.

If you would like to hear about any publications using this study data, send an email to Peter Simonyi <peter.simonyi@carleton.ca>.

I agree to participate in this study. I have received the \$10 compensation.

Signature:

Date:

Researcher's signature:

Date:

We will give you a copy of this form to keep for your records.

1 Your age

Please write your answer here:

2 Your gender

Please choose **only one** of the following:

- ☐ Female
- ☐ Male

3 Which is your dominant hand?

Please choose **only one** of the following:

- ☐ I'm left-handed
- ☐ I'm right-handed
- ☐ I'm ambidextrous

4 What is your occupation? If you are a student, please include your program/major.

Please write your answer here:

5 How long have you been using computers?

Please choose **only one** of the following:

- ☐ As long as I can remember
- ☐ Since before 1994
- ☐ 11-20 years
- ☐ 4-10 years
- ☐ A few years or less

6 How often do you use a computer?

Please choose **only one** of the following:

92

- ☐ Daily
- ☐ Most days
- ☐ A few times a week
- ☐ A few times a month
- ☐ More infrequently

7 Please indicate how much you agree or disagree with the following statements.

Please choose the appropriate response for each item:

	Agree	Somewhat	Neutral	Somewhat	Disagree
I understood the need for the extended undo.	☐	☐	☐	☐	☐
I found it difficult to go back to earlier states.	☐	☐	☐	☐	☐
I found it easy to understand how the sidebar worked.	☐	☐	☐	☐	☐
I understood the meaning of the different kinds of dots.	☐	☐	☐	☐	☐
When I made a change to the application, I saw how it was reflected in the sidebar.	☐	☐	☐	☐	☐
I spent more time using the sidebar than it saved me.	☐	☐	☐	☐	☐
I reached a point where I wanted to go back to something I'd tried before.	☐	☐	☐	☐	☐
I would like to have this extended undo feature in other software or web sites I use.	☐	☐	☐	☐	☐
I found the touchscreen hard to use.	☐	☐	☐	☐	☐
I found it useful to give names to some states (dots).	☐	☐	☐	☐	☐
I ended up with too many dots to manage.	☐	☐	☐	☐	☐

8 Do you have any other comments?

Please write your answer here:

94