Set-Based User Interaction

A Thesis
Presented to
The Academic Faculty

by

Michael Terry

In Partial Fulfillment of the
Requirements for the Degree
Doctor of Philosophy in
Computer Science

Georgia Institute of Technology
August, 2005

Set-Based User Interaction

ACKNOWLEDGEMENT

I owe a debt of gratitude to my younger cousins, Ellen and Cameron McKnight (8 and 11 at the time), who originally inspired this work. When I first came to Georgia Tech, I rented an apartment in their house, and they would often come down to visit. To keep them busy while I worked, I introduced them to the GNU Image Manipulation Program (the GIMP). They quickly settled into a routine: Using one of the built-in scripts, they generated a blue sphere, then proceeded to exhaustively try and undo filters until pleasing results were found. Two things were noteworthy. First, they were able to generate some amazing compositions using filters alone (often bearing no resemblance to the original blue sphere), and second, the application seemed to have so much power hidden away – one had to continually probe the application to discover what it was capable of. This seemed the wrong model for creative work, and led me to investigate ways computer interfaces could more easily support the creative process.

Knowing nothing about HCI research when I first arrived, I had the good fortune of finding Beth Mynatt and joining her Everyday Computing Lab. I could not have asked for a better mentor. Throughout the years Beth has supported my research in all ways possible and been a model researcher and advisor. Her Everyday Computing Lab, too, is an incredible collection of people. Its members – Jeremy Goecks Elaine Huang, Jessica Paradise, Jim Rowan, Quan Tran, Joe Tullio, Amy Voida, and Steve Voida – have offered guidance, support, and friendship in great abundance throughout the years. Lena Mamykina, a former ECL member, deserves special mention for always providing last minute feedback on papers and talks, and for showing me around NYC with her husband Kliment on weekends away from Atlanta.

While Beth Mynatt was my official advisor, Gregory Abowd served as my unofficial co-advisor, always providing enthusiastic encouragement throughout the years. His students, too, have lent a helping hand, especially Khai Truong.

In addition to Beth and Gregory, I had a fantastic committee to guide this work. I was fortunate enough to join Kumiyo Nakakoji in Japan for one summer via NSF's East Asia Summer program, and Scott Hudson at CMU the next summer. Both experiences helped shaped this work directly and indirectly. Blair MacIntyre, too, has influenced this work through our many discussions, and is the person I turn to when I need to talk about graphics, digital imagery, or photography.

A number of people have readily and selflessly offered their time to support this work in various capacities. In particular, I thank Wendy Newstetter for grounding me in the methods of anthropologists; Molly Stevens for always coming in on short notice to test things out days before deadlines; Diane Gromala for opening my eyes to the world of artists, digital art, and new media; Jeff Weese and Grace Ou, two artists who continually inspire me to build more tools (and who also served as guinea pigs for tools-in-progress); Laura Dabbish for lending her statistical expertise to analyze study data; Jim Davies for keeping me laughing; Gabe Brostow for his model work ethic; and Ruomei Gao for her support, delicious baked goods, and the time the work stole away from doing more fun stuff.

Finally, I would like to dedicate this work to my family, who made everything possible: My father Peter, my mother Sarah, my sister Alison, and my brother David; my grandparents Sherwin and Jean Terry, and Frank and DeDe Reynolds; and Joe and Margaret Dombrowski, my home-town grandparents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Compared to physical media, digital media are extremely plastic: Previews enable one to experiment with a command before actually modifying data, while Undo provides the ability to return to a previous state should an entire series of actions prove undesirable. However, despite this plasticity, an observational study we conducted of artists and graphic designers suggests that current user interfaces make it difficult to quickly develop and explore *sets* of alternative solutions in parallel.

Exploring sets of alternatives is a common practice among expert practitioners since it allows them to better understand the problem and its space of potential solutions. Rather than think in the abstract, sets of alternatives enable one to directly compare and contrast the relative merits of each possibility.

While it is not impossible to explore sets of alternatives with current interfaces, it must be manually orchestrated by the user: Time must be spent preparing the user interface for exploration (e.g., by making copies of a solution before deriving a new standalone solution), and few mechanisms exist to ease the tasks of comparing the alternatives and keeping them in sync. Ultimately, these costs can be high enough to discourage users from exploring in both breadth and depth. As a result, current interfaces tend to lead to highly linear problem solving processes whereby a single solution is continually revised until deemed acceptable. We term such interfaces *point-based interfaces* because of their focus on one solution at a time.

To better support exploration, we introduce the concept of a *set-based interface*, or an interface that provides explicit support for simultaneously generating, manipulating, evaluating, and managing *sets* of alternative solutions for the same problem. Such an interface is intended to enable a problem solving process characterized by broad exploration and the simultaneous consideration of multiple solution instances.

We present two tools demonstrating the concepts of a set-based interface, *Side Views* and *Parallel Pies*. Both are designed for use in image manipulation tasks. Side Views is an enhanced previewing mechanism that simultaneously displays sets of previews for one or more commands and their parameters. Parallel Pies, on the other hand, streamlines the process of "forking" so one can more easily pursue promising alternatives in parallel. In contrast to history-tracking tools that simply record all the states a person has visited, Parallel Pies allows an individual to easily spawn new *standalone* alternatives: As promising possibilities are discovered while interacting with a command, individuals may use Parallel Pies to duplicate their current document, apply the command to the copy, and insert the new result into the same workspace – all via a single action. Parallel Pies' visualization tool then allows one to compare and contrast the generated alternatives.

To understand the impact these tools have on the problem solving process, we conducted two controlled, within-subjects laboratory studies and a third think-aloud study. In the first controlled study, subjects color-corrected images to make them match a known (visible) target image. Though it employed a well-defined task, the solution path was ill-defined, making experimentation essential to solving the problem well. In the second controlled study, subjects developed color schemes for watches, offering a view of how these tools perform in much more open-ended tasks. The final study paired individuals to work cooperatively on the same color scheme task as the second study. This third study design helped to externalize individuals' thought processes as they

worked through problems as subjects needed to communicate with one another to solve each problem.

The collective results from these studies indicate that Parallel Pies leads to a discernibly different problem solving process that subjects perceive to be more desirable. In particular, Parallel Pies leads to broader exploration and the creation of more solution alternatives in the same amount of time. The studies further indicate that this problem solving process can result in more optimal solutions when solving tightly constrained tasks, where optimality is measured by the number of operations required to derive a solution state from a given start state.

While Parallel Pies can lead to broader exploration, we found that too much exploration can actually *lower* solution quality under certain circumstances. In particular, when solving open-ended problems under tight time constraints, individuals can initially overuse the capability to broadly explore and not spend enough time maturing a single solution. However, there are indications that this issue would disappear after users have fully acclimated to this capability.

Parallel Pies was designed to enable broad exploration by increasing the ease with which one can spawn a new, separate solution instance. One unexpected side-effect of this capability is that it can increase the likelihood of backtracking. Specifically, as individuals work, they sometimes use Parallel Pies to create back-up copies of different stages of their work, thereby increasing the frequency with which they return to a previous state to pursue alternative paths. Though the tool was not expressly designed to support this practice, this use suggests the need for history mechanisms that more completely track all states a user visits.

In contrast to Parallel Pies, our studies show that the multiple previews of Side Views have a far more subtle effect on the problem solving process. While we find no evidence that multiple sets of previews can affect one's performance, we did find that they can lead to individuals serendipitously discovering viable solution alternatives. That is, while interacting with a command, the multiple previews can suggest ways of solving a problem that the user had not originally considered.

Side Views' multiple sets of previews per parameter can also reduce the need to scan for settings of interest. In our studies, subjects moved commands' sliders about half as much when Side Views were present, suggesting that users can find values of interest with less effort by using this tool.

Finally, our studies indicate that Side Views' multiple previews can help ground communication between individuals as they cooperatively solve problems. Rather than abstractly describe how they wish to proceed, users can simply point to a Side View that is close to the desired direction.

From these three studies, we conclude that user interface mechanisms that support the rapid generation and automatic management of new standalone alternatives are highly valued and useful facilities for solving complex, ill-defined problems. By making it easy to create sets of potential solutions that are simultaneously active, these capabilities streamline the process of exploring the design space and help users to perform concrete, side-by-side comparisons of their options.

We also conclude that current interface designs lead to highly linear problem solving processes due to their lack of mechanisms to support broad exploration. When such mechanisms are available, they are eagerly adopted and lead to much wider explorations of the problem space, *despite them being completely optional components to the problem solving process*. Thus, this work argues for the need to more fully investigate how process support tools (such as history mechanisms and previewing tools) impact the problem solving process.

While this research has documented how Side Views and Parallel Pies can assist in experimentation and exploration, this work also makes significant contributions in the methodology and metrics used to perform the evaluations. For over 20 years, mechanisms such as branching histories and enhanced versions of Undo have been proposed to facilitate experimentation. However, very few evaluations have been performed on these types of tools. The dearth of evaluations can be partially attributed to the lack of analytical methods appropriate to evaluating how these types of mechanisms affect the problem solving process.

Noting this lack of evaluation instruments, this work contributes a suite of metrics that formalize concepts such as backtracking, breadth and depth of exploration, and "dead-ends." These metrics enable statistical comparisons to be performed across these dimensions, allowing a researcher to investigate whether their user interface mechanisms lead to significantly different problem solving strategies. A visualization called a *process diagram* aids in conveying results by rendering an individual's problem solving process through a set of integrated directed graphs.

CHAPTER 1

INTRODUCTION

Toyota's "2nd Paradox": "delaying decisions, communicating 'ambiguously,' and pursing excessive numbers of prototypes, enables Toyota to design better cars faster and cheaper" [p.44, WAR95]

In the quote above, Ward et al. describe Toyota's vehicle design practices: Whereas many car designers attempt to develop precise design specifications early in the design process, Toyota's engineers delay commitment to final solutions and continually explore the design space *throughout* the problem solving process. Solution development thus progresses by constantly generating, evaluating, and pruning *sets* of possibilities (Figure 1). This process provides a more comprehensive view of the solution space and, correspondingly, what will, and will not, work in practice. It also increases the chance that optimal solutions are found, while ensuring a back-up solution always exists. Because of this emphasis on exploring several viable alternatives in parallel, the authors dub the practice *set-based concurrent engineering* [WAR95, SOB99].

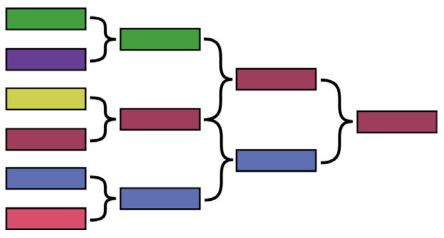


Figure 1. Set-based engineering. Set-based engineering generates many possibilities, and gradually narrows the choices to a final solution.

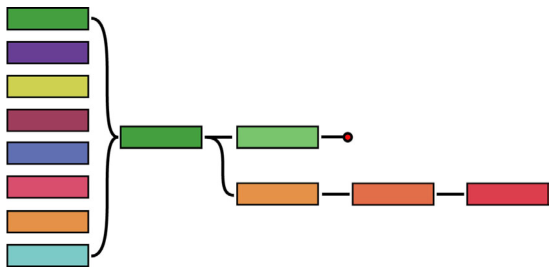


Figure 2. Point-based engineering. Point-based engineering initially considers many alternatives, but quickly commits to one that is revised until acceptable. Dead-ends require backtracking.

The authors contrast Toyota's practices with that of American car manufacturers, who seek to quickly converge on a single, fixed solution point, a process Ward et al. term *point-based concurrent engineering*. Point-based concurrent engineering uses the initial conceptual design phase to explore alternatives, but at its conclusion, a single solution target is chosen. The remainder of the design process consists of continual refinement of that single solution until it reaches an acceptable end state (Figure 2).

Set-based and point-based concurrent engineering represent two philosophically different approaches to decomposing and solving complex, ill-defined problems. At one extreme, set-based practices intentionally generate alternatives so that more than one possibility is always under investigation. At the other extreme, individuals commit to just one solution instance that matures through continual revision.

These methods are not unique to car design, and can be easily found elsewhere. Design disciplines have long advocated and practiced methodologies similar to set-based practices, though they describe the process in other terms, such as ideation or design space exploration [GUN99]. Recognizing that differences in both terminology and process details exist, we nonetheless generally refer to problem solving methods that emphasize investigating sets of alternatives as *set-based problem solving*, while referring to the alternative in which a single solution is continually revised as *point-based problem solving*.

In this dissertation, these alternative conceptualizations of the problem solving process serve to frame the central problem investigated by this work. Based on results from a field study of work practices [TER02a], we argue that current user interfaces cater to point-based problem solving practices to the exclusion of fluidly supporting set-based practices. In particular, we claim that current interfaces assume a highly linear problem solving process in which a single solution instance is continually revised. As a result, few facilities exist to support the rapid, broad exploration of possibilities, forcing users to manually orchestrate the process when they wish to engage in set-based practices. While set-based problem solving is not impossible, we provide evidence that the transaction costs can be high enough to discourage its use.

These findings lead us to argue for the need to reconceptualize how interfaces structure the problem solving process. In particular, we advocate an alternative type of user interface, the *set-based interface*, which provides explicit support for generating, manipulating, evaluating, and managing sets of alternatives. These facilities are intended to enable one to more easily explore sets of alternatives in parallel by reducing the need to commit to just one solution path at a time.

To explore this notion of a set-based interface, we constructed two tools in the domain of image manipulation, *Side Views* and *Parallel Pies*. Side Views is an enhanced previewing mechanism that affords the rapid, broad exploration of near-term possibilities by automatically generating sets of previews for one or more commands and their parameters [TER02b]. Parallel Pies, on the other hand, affords sustained, long-term exploration of a select subset of alternatives via a combination of features that streamline the process of "forking" a solution to explore divergent paths [TER04]. In contrast to previous approaches that support exploration by maintaining a more complete history of all states one has visited, this tool enables the *proactive* generation of alternatives, allowing users to quickly spawn new standalone solution instances as they are discovered when interacting with commands.

Given this backdrop, this research's thesis statement is formulated below.

1.1 Thesis Statement

User interfaces that offer mechanisms to facilitate the generation, manipulation, evaluation, and management of multiple solution alternatives will enable individuals to develop a high quality solution in less time and with lower cognitive load than interfaces that do not offer these same services. These facilities will also lead to users developing more alternatives in the same amount of time.

In exploring this thesis, we limit our scope to the domain of image manipulation.

To investigate the claims of the thesis, we evaluated Side Views and Parallel Pies in two controlled laboratory studies and a think-aloud study. The first study utilized a tightly-constrained task in which subjects color-corrected images given a start state and a known (visible) target state. The second study changed the nature of the task to make it more open-ended: Subjects were asked to develop color schemes for watches to make them evocative of one of the four seasons of the year. Finally, the third study paired individuals to work together on the color scheme task of the second study, asking them to think aloud as they worked.

From these three studies, we conclude that Parallel Pies' ability to quickly generate new standalone alternatives results in individuals adopting problem solving strategies more characteristic of set-based problem solving: They explore more broadly and generate more solution alternatives when the tool is present. We also find evidence that this strategy can lead to more optimal solutions, where optimality is measured by the number of operations required to derive the solution state from a given start state. However, results from one study indicate that this strategy of broadly exploring can backfire if individuals are under a strict time constraint and they spend too much time exploring, and not enough time maturing a single solution. Despite this negative result, subjects in our study insist that the ability to broadly explore is highly desirable, and tools that facilitate this process are welcome.

While Parallel Pies is often used as a catalyst for exploration, its ability to generate new standalone alternatives can also lead to its use as a "bookmarking" facility. That is, rather than rely on the undo stack, individuals accumulate open, accessible copies of past states using Parallel Pies, leading to higher rates of backtracking when the tool is present. This use argues for the need for histories that make past states highly visible and accessible.

In contrast to Parallel Pies, we found Side Views to have a far more subtle effect on the problem solving process. While there was no discernable effect on the quality of solutions developed nor the problem solving strategy employed, four uses of the tool became apparent. First, users can serendipitously discover new ways of solving a problem via Side Views' multiple previews. In our think-aloud study, this use became apparent as subjects expressed surprise at some of the previews that appeared while interacting with the tool. Second, Side Views can help one to quickly eliminate an entire area of the design space. In our think-aloud study, we found individuals dismiss an entire command if it was clear that no combination of parameter settings met their needs. Third, Side Views can reduce the need to manually scan a parameter's range of settings. Similar in spirit to the previous finding, each study consistently found that individuals use sliders approximately 50% less when Side Views are available, suggesting that multiple sets of previews can help people find values of interest with less effort. Finally, our think-aloud study revealed that Side Views can serve as a tool to help coordinate problem solving. As people work cooperatively on a problem, the multiple sets of previews ground communications by giving people concrete reference points to explain how they wish to proceed.

Despite these observed effects, we found no support for the thesis claim that either of these tools will lead to better solutions faster. However, we did find mixed support for the claim that the tools will reduce cognitive load. In the open-ended task of designing a color scheme for a watch, we found Parallel Pies can increase mental demand while lowering frustration. Data collected from surveys and interviews explain these results. When solving problems, subjects develop more solution alternatives when Parallel Pies is present, leading to a decrease in frustration because of the greater number of choices available for solving the problem. However, by the same token, these extra choices increase mental demand because they result in more data on the screen at one time. These findings again suggest the desirability of exploration, but also indicate that the interface must take care not to overwhelm the user with information.

The unexpected negative results of these studies – the negligible impact of Side Views' multiple previews on performance and the discovery that broad exploration can actually lower solution quality under some circumstances – are instructive in that they underscore the need to critically examine how such mechanisms actually affect the problem solving process. Intuitively, capabilities that facilitate experimentation and exploration are desirable and useful. For example, few would argue with the utility of Undo or a history mechanism that fully tracks every state visited. However, to date, these types of user interface mechanisms have rarely been evaluated. These results are some of the first to illustrate the potential benefits and drawbacks of such tools.

One of the barriers to evaluating such process support tools has been a lack of analytical methods suitable for describing how these tools affect problem solving practices. As such, one of the contributions of this work is a suite of metrics that formalize concepts such as breadth and depth of exploration, backtracking, and "dead-ends." These formalizations enable one to perform statistical comparisons across these dimensions to understand whether a given tool significantly affects how individuals solve problems. A visualization called a *process diagram* aids in communicating these results by highlighting aspects of the problem solving process such as branching, backtracking, abandoned states, and the number of states active at any point in time.

In the rest of this dissertation, we develop the notion of a set-based interface in the following phases. Chapter 2 reviews the literature examining ill-defined problems and general problem solving practices, independent of any computer-based support. We also discuss the Toyota studies in more detail, further delineating the differences between point- and set-based problem solving practices.

Chapter 3 describes results from an observational study of expert users of an image manipulation application and presents three representative case studies. These case studies reveal problem solving strategies similar to those described in the general problem solving literature, but also show how computer-based tools integrate into the process. From these case studies, we synthesize a table mapping problem solving techniques to specific interface tools, and consider the relative strengths and weaknesses of each existing interface mechanism. This mapping uncovers a number of deficiencies in current interfaces with respect to solving ill-defined problems. In particular, we argue that interfaces are designed to support point-based problem solving at the cost of fluidly supporting experimentation and exploration.

In Chapter 4 we review existing research in user interface design that attempts to address some of these shortcomings. Three main types of tools are considered, each of which allows individuals to work with sets of alternatives: augmented history tools, what-if tools, and enhanced previewing mechanisms. We identify a number of ways this existing research can be extended, which leads to the thesis and one of the contributions of this work: The notion of a set-based interface. The principle components of a set-based interface are defined and distinguished from other efforts in Chapter 5, and the concepts of a set-based interface are illustrated in Side Views and Parallel Pies in Chapter 6.

With this foundation in place, we turn our attention to the task of evaluating the claims of the thesis. In Chapter 7, we synthesize a set of metrics for describing the problem solving process and present a visualization called a *process diagram* to aid in comparing problem solving sessions. Chapter 8 covers three studies conducted to evaluate the impact of Side Views and Parallel Pies on the problem solving process. Results are presented in detail and interpreted with respect to the research questions of this work. We conclude in Chapter 9 by enumerating the contributions of this work and describing open research questions. Appendix A documents the best results from the study in which subjects designed watch color schemes, while Appendix B speculates on how these principles may apply to the design of a programming language.

CHAPTER 2

BACKGROUND

The ultimate goal of this work is to better support individuals as they solve *ill-defined problems*. Ill-defined problems are highly complex, intricate problems with no single right answer. They can be found in design disciplines (e.g., architecture, software engineering, etc.), as well as in relatively common activities, such as writing a paper. In this chapter, we review literature describing the nature of these problems and how individuals approach their solution.

2.1 Characteristics of Ill-Defined Problems

Ill-defined problems are problems lacking clearly defined goals and solution methods. They arose as a focused topic of research in the 1960's when Walter Reitman made an important distinction between *well-defined problems*, and what he termed *ill-defined problems*, or problems with poorly defined operators and goals [REI65]. Prior to making this distinction, research in problem solving primarily concentrated on well-defined tasks. For example, studies conducted by Newell and Simon (summarized in [NEW72]) investigated how people solved problems with well-defined goals and states, such as cryptarithmetic. In differentiating these different types of problems, Reitman claimed that most real-world problems, such as design tasks, are ill-defined and qualitatively different from well-defined problems in form, complexity, and method of solution.

Subsequent research has refined Reitman's distinctions and added weight to his claims. For example, Goel and Pirolli [GOE92] identified a number of common characteristics of design problems through studies of designers in three different disciplines, architecture, mechanical engineering, and instructional design. Their findings echo a number of other related research efforts (e.g., [RIT84, SCH83, SIM73]), and thus serve as a useful summary of the characteristics of ill-defined problems. In describing design problems, they claim these problems:

- are large and complex with a number of interconnected parts that affect one another;
- have underdefined start and goal states, with equally underdefined transformation functions; and
- lead to solutions that can only be described as "better or worse," as opposed to "right or wrong."

These characteristics have some important corollaries that further define the nature of these types of problems. For example, since there is no "right" answer, a solution can always be improved upon [REI65, RIT84]. Thus, declaring a problem "solved" is as much a function of the availability of resources (e.g., time, money) as it is a measure of the suitability of the solution to the task [SIM73].

The lack of well-defined goals also means that the goals themselves are flexible and subject to reinterpretation. This further increases the set of possible solutions. As an example, Goel and Pirolli found that individuals often attempt to negotiate the problem's goal to make it better match their skills and experience.

Finally, the underdefined nature of these problems requires a practitioner to develop the *solution method* itself, which is itself an ill-defined problem [REI65] – there is no "textbook" method that can be blindly applied to solve these problems.

In sum, the lack of a single “best” solution leads to a significant space of possibilities for individuals to consider. This large space of possibilities is critical to keep in mind when we later consider user interface support for these problems.

2.2 Strategies for Solving Ill-Defined Problems

In their study, Goel and Pirolli also identified a number of common problem solving strategies. They found:

- Distinct problem-solving phases: Preliminary design, refinement, and detailed design;
- Structuring and decomposition of the problem into smaller subcomponents;
- Incremental development of the solution and its subcomponents;
- The practice of limiting commitment to precise solutions; and
- Personalized stopping rules and evaluation functions.

These findings correlate with many other descriptions of problem solving practices. We highlight two relevant characterizations, one by Donald Schön, the other by Ward et al. [WAR95] and Sobek et al [SOB99].

2.2.1 Schön’s Theory of Reflection-In-Action

Through studies of professional designers, Donald Schön developed the theory of *reflection-in-action*, a characterization of the design process that likens it to a kind of informed improvisational act [SCH83]. As he describes it, reflection-in-action is the process by which skilled practitioners act on the problem, reflect on the consequences of that action, then modify their future actions accordingly. These individual acts can thus be thought of as a series of small experiments to discover how to solve the problem most effectively. Notably, throughout the process, the practitioner is not applying a set of predefined rules to solve the problem, but rather is calling upon a repertoire of past experiences to guide experimentation with the solution.

In his observations, Schön found that individuals use what he calls “virtual worlds,” or alternative representations of the problem and its solution. As he uses the term, a virtual world is a “context for experiment within which practitioners can suspend or control some of the everyday impediments to rigorous reflection-in-action” [SCH83, p. 162]. Often, these virtual worlds are highly ambiguous representations, allowing a designer to continually reinterpret the ambiguity and delay commitment to precise solutions [GRO96]. Casting the solution into another representation can also exploit the characteristics of the alternative media by suppressing some details of the problem while highlighting others. For example, when designing a building, an architect may employ hand-drawn sketches, physical models, or 3D mock-ups. Each has its own unique affordances and enables the problem to be considered from alternative viewpoints.

In relation to Goel and Pirolli’s characterizations, Schön’s characterizations add further support for the notion that solving these types of problems is a highly iterative and unpredictable process: it is not a straightforward path from beginning to end. Rather, the solution path must be slowly discovered and constructed. In effect, the challenge is not only in constructing a path to the goal, but also developing an understanding of what that goal should actually *be*. A comparative study of the design practices of Toyota and other Japanese and American car manufacturers makes this point clear, while uncovering two fundamentally different philosophies to constructing this solution path.

2.2.2 Set-Based Concurrent Engineering vs. Point-Based Concurrent Engineering

In an attempt to understand how Toyota produces cars faster, more efficiently, and with fewer people than its competition, a set of studies conducted by Ward et al. [WAR95] and Sobek et al. [SOB99] uncovered a design culture unique to Toyota. In comparison to its competitors in the US and Japan, Toyota generates a significant number of prototypes throughout the design process. Further, they design to *sets* of possibilities, rather than to precise specifications. The authors term this problem-solving strategy *set-based concurrent engineering*. In contrast, Toyota's competitors initially explore a number of alternatives, but quickly converge on a single target that is developed into the final product. The authors dub this process *point-based concurrent engineering* because of its emphasis on pushing towards one envisioned, uncompromising goal[1].

Point-based concurrent engineering strives to determine, with great precision, the exact specifications of the vehicle as early as possible. For example, CAD diagrams are created and sent to suppliers to create the parts or subcomponents exactly as prescribed. As the authors note, "key decisions are made early on in order to simplify interactions among subsystems. These decisions *maximally constrain design* to achieve the desired effect" (emphasis added).

Toyota, on the other hand, deals with *ranges* of tolerable values for every part of the car. Suppliers are given general targets, and components can be built anywhere within the given design tolerance. As a result, a number of alternatives are developed and explored. For example, Toyota's supplier of exhaust systems prototypes 10-50 different

[1] In this context, "concurrency" refers to multiple groups of people working on different sub-problems simultaneously.

systems while its competition normally prototypes just one. Additionally, while American car manufacturers derive and fix CAD drawings early on in the process, it is only *after* the right fit for a component has been found in practice that Toyota updates its CAD drawings to reflect the desired specifications.

Toyota's design process explicitly acknowledges limitations of human cognition when solving complex, ill-defined problems:

> *The engineers recognize that even though they are familiar with this product, they cannot know everything about it. There are subtle relationships between parts of the system that can be explored only through tests of real prototypes. In set-based terms, experimentation is a way of exploring the design space. Decisions are made only after such experimentation. [WAR95, p. 56]*

To arrive at a final solution, Toyota begins with a set of viable alternatives, bounded by experience and knowledge of what is feasible, then gradually narrows this set by eliminating sub-optimal alternatives. A large tree of possibilities is originally developed, then gradually pruned until the best fit can be found amongst all the sub-components (Figure 3). American manufacturers, on the other hand, start with a range of theoretical alternatives, but quickly prune to a single design that is iterated upon (Figure 4). When aspects of that vision do not work as planned, a new offshoot must be created to develop a workable alternative. In the end, both Toyota and its competitors explore multiple alternatives; the difference is that Toyota explores more, and does so in an informed, purposeful way, rather than as a reaction to unexpected circumstances.

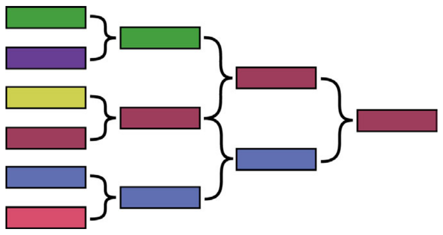


Figure 3. Set-based engineering.

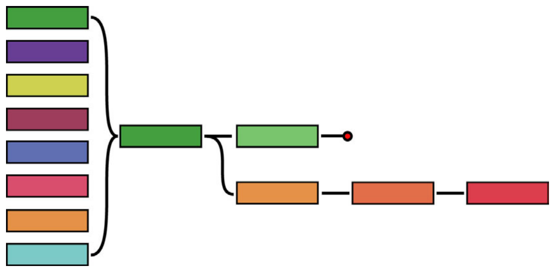


Figure 4. Point-based engineering.

To be clear on our use of the terminology, we consider an alternative solution to be a separate, unique solution to the same problem. This is in contrast to an *iteration*, which simply represents a revision to a single solution. While we make a distinction between these two types of solution instances, we are not claiming they are mutually exclusive. For example, a previous iteration can sometimes be considered an alternative solution if it is sufficiently different from the current solution state. The critical difference is the *role* the solution instance plays for an individual at a specific point in time.

A similar distinction can also be made with regard to how solution alternatives arise. In set-based problem solving, alternatives are *proactively* generated as part of a purposeful process, while alternatives in point-based are *retroactively* generated, after a particular solution path proves inadequate. These two solution types have also been referred to as *generative variations* and *corrective variations* [GUN99], respectively, because one either generates the alternatives in *parallel* (with the intention of comparing them), or as a result of correcting past actions through a *serial* process. Generative variations lead to multiple solution possibilities that are simultaneously active, while corrective variations typically result in a single solution instance being active at a time.

Point- and set-based concurrent engineering represent two alternative philosophies for how to solve an ill-defined problem. As previously mentioned, these methods are not unique to car design, and instances of each can be readily found in descriptions of design practice (e.g., [CRO01, GUN99]). Again, recognizing that differences in both terminology and process details exist, we nonetheless refer to problem solving methods that emphasize investigating sets of alternatives as *set-based problem solving*, while using the phrase *point-based problem solving* to refer to strategies typified by the continual revision of a single solution instance.

In contemplating these different problem solving approaches, an important consideration is which method will generally yield better solutions. Ward et al. and Sobek et al. argue that set-based practices help Toyota develop better solutions faster and cheaper, and many design disciplines advocate practices akin to set-based practices. However, evidence of the existence of this strategy does not, by itself, indicate its optimality.

A theoretical comparison of search algorithms suggests that exploring sets of alternatives in parallel does indeed lead to better solutions faster, at least in a highly idealized environment. In work comparing the expected performance of algorithms intended to find an optimal solution, Aldous and Vazirani [ALD94] prove that consideration of several alternatives in parallel results in the discovery of an optimal solution faster, on average, than depth-first searches. The key, as they put it, is to "go with the winners": When exploring alternatives in parallel, paths that lead to sub-optimal solutions should be abandoned with the resources reallocated to exploring derivations of an already "good" solution. Compared to depth-first search strategies, they prove that this strategy has a significantly higher probability of finding an optimal solution faster. While this example draws upon an idealized world in which state transitions are discrete functions that yield states that can perfectly assessed (conditions which don't exist when solving ill-defined problems), it does provide evidence that, all other things equal, set-based approaches have the potential to outperform point-based approaches.

2.2.3 Summarizing Basic Attributes of Ill-Defined Problems and Problem Solving Strategies

Ill-defined problems lack concrete goals and well-defined evaluation criteria. As a result, any number of solutions can be developed, each with its own unique set of strengths and weaknesses. Solving these types of problems thus requires experimentation and exploration of multiple alternatives, with the continual evaluation of one's progress towards the envisioned goal.

Set-based and point-based problem solving represent two philosophically different approaches to solving ill-defined problems. The former explicitly considers and develops ranges of possibilities throughout the design process, while the latter attempts to define a single goal early in the process. Multiple solution alternatives arise in both cases, though for different reasons. Set-based problem solving results in generative variations, or solution alternatives that are proactively created to explore the design space. Point-based problem solving, on the other hand, results in corrective variations, or solution alternatives that arise due to backtracking. There is evidence, from real-world practices and theoretical perspectives, that set-based practices yield a more optimal problem solving process, making it worthwhile to consider how to better support this process in user interfaces.

CHAPTER 3

NEEDS AND PRACTICES WHEN SOLVING ILL-DEFINED PROBLEMS WITH COMPUTER-BASED TOOLS: THREE CASE STUDIES

To understand how current interfaces support solving ill-defined problems, we conducted an observational study of users of an image manipulation application [TER02a]. Though we focused on the use of one particular application, our goal was to identify domain-independent needs and practices. A modest degree of generality was achieved by studying how the tool is used across a variety of tasks, from color correcting images for a newspaper, to the design of user interfaces for multimedia software.

Eight users, all but one expert users of the application, were interviewed using open-ended, qualitative interviewing techniques [SEI98, WEI94]. In these interviews, our goal was to understand subjects' typical day-to-day workflow, including the types of problems they must solve and how they solve them. Importantly, interviews were *not* structured to elicit information regarding perceived strengths and weaknesses of the application, but rather, were intended to uncover how the tool integrates into their work process.

After obtaining a general sense of the type of work performed, we asked each subject to demonstrate how they solved a recent problem. As they solved the problem, we videotaped their actions, capturing the computer screen, keyboard, mouse, and person in the same frame.

The results of this study uncovered problem solving practices and strategies similar to those enumerated in the previous chapter. Furthermore, we found evidence of both set-based and point-based practices. However, we also discovered that current interfaces tend to be aligned with the point-based problem solving philosophy, creating a gap between practices desired by users and those well supported by current interfaces. We present three representative case studies here to illustrate these points, then summarize and reflect on the findings.

3.1 Newspaper Image Control Desk: Image Toning

In the first case study, we convey the work practices of a former employee of a major metropolitan newspaper. At the newspaper, her primary task consisted of toning (color correcting) images so they print well on the newspaper's printing press.

When toning images for a newspaper, employees must improve the quality of the images without altering the editorial content. This process includes cropping and sizing the image, and adjusting its colors so they print well on newsprint. Because newsprint tends to soak up ink in unique ways, new employees must continually monitor how their images print to cater their toning process to idiosyncrasies of the newspaper's printing press and paper.

To achieve consistent results, employees rely on a set of color-toning heuristics they develop through experience. An overarching goal is to apply as few changes as possible, because each change causes original visual information to be lost. Therefore, individual operations must be chosen with care so that the *total number* of operations is minimized. Furthermore, each operation should contribute as much as possible, without causing side-effects that make subsequent corrections difficult or impossible. These constraints and goals necessitate a highly iterative process that often requires users to explore and evaluate several different approaches before arriving at an acceptable result. Because experimentation and evaluation play such crucial roles in toning images well, we describe how they instantiate these activities next.

Settings for commands are typically explored by “scrubbing” commands’ parameter sliders back and forth. With real-time previews, scrubbing a slider enables the user to scan the range of possibilities for a particular parameter before committing to any one value. We found three distinct types of scrubbing.

In the first case, users do not know which parameter value will produce desirable results, so they make broad sweeps with the slider to quickly sample possibilities. These broad sweeps are gradually narrowed until a final value is chosen.

In the second case, users know the approximate value they are seeking and immediately move the slider to that vicinity. To confirm their choice and evaluate nearby settings, they scrub the control within a narrow range to refine the parameter setting.

In the final case, a value has been found, but the user wishes to verify that another, completely different setting would not provide better results. In these situations, the user quickly moves the slider to the opposite side, then typically returns to a narrow scrub around the original value of interest. We call these “sanity checks” because they help the user confirm she is on the right track.

Each instance of parameter scrubbing enables experimentation and allows the user to delay commitment to a final value until a set of alternatives have been evaluated. However, since only one preview is available, comparisons must be made in *time*, rather than side-by-side in space.

After applying a command, progress is evaluated in a number of ways. Since image toning is essentially an optimization task, evaluation is most critical between *consecutive* states, that is, immediately after applying a command. Thus, we found users engage in *undo-redo cycles* whereby they would repeatedly invoke Undo and Redo in quick succession to "flash" the current and previous versions of the image on the screen. This evaluation technique has the advantage that comparisons can be made on the entire image at any scale. Furthermore, since the two versions occupy the same space, users do not need to perform any spatial reorientation as would be necessary if the two versions were placed side-by-side.

If earlier states have previously been saved, users sometimes reopen them to evaluate how much the image has improved, or, potentially, degraded. This type of evaluation can reveal when details of the original image have been lost as a result of improving other aspects of the image (such as its overall tonal quality). In such situations, users must decide whether the result is an acceptable trade-off or whether they should backtrack and try an alternative approach.

Progress is also assessed by viewing alternative representations of the image. For example, users sometimes examine their work by viewing each color channel separately, as a grayscale image. These views provide information that may be hidden or difficult to discern in the normal representation.

Though users continually evaluate their progress at every step, a state can still be reached that proves unsatisfactory. To support backtracking to a previous state, individuals create several back-ups of the image at critical points, often before doing something risky or drastic. However, though several copies of different image states may be created, attention is still typically focused on one image state at a time. That is, problem solving usually progresses by continually revising one solution instance at a time, with users backing up to a previous state when dead-ends are encountered.

3.1.1 Relating Practices to Problem Solving Styles

Mapping this description of work practice to those previously described, we find that it most closely resembles point-based problem solving and Schön's theory of reflection-in-action: It is a highly iterative process that evaluates only one path to the goal at a time. When results do not meet expectations, users must backtrack and redo their work to find a better solution. Thus, alternative solutions represent *corrective variations* rather than generative variations. Because backtracking occurs so frequently, back-up copies of one's work are routinely generated and stored in separate files.

Evaluation is critical throughout this task. Every action is closely scrutinized, from the selection of commands and their parameters, to the results that appear after applying a command. Comparisons between consecutive states are performed the most frequently, with comparisons occasionally made between the current state and a previous milestone.

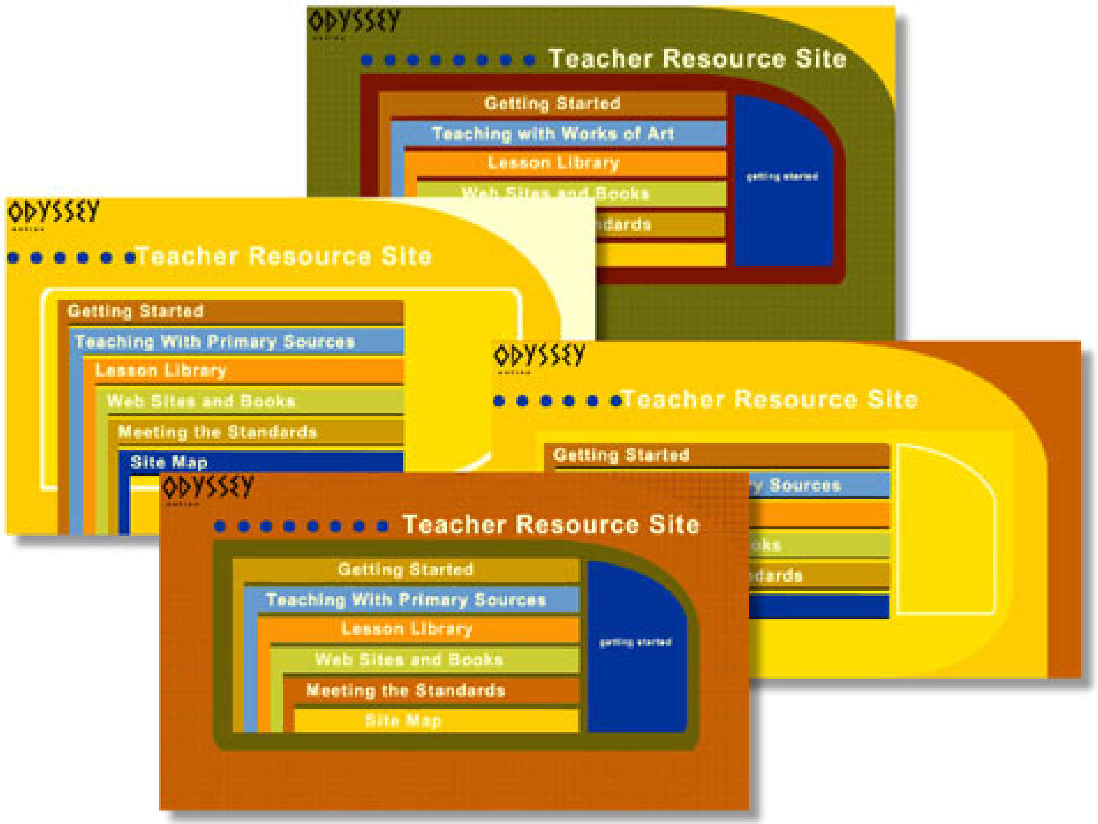


Figure 5. Four (out of eleven) alternative solutions produced for a website.

3.2 Interactive, Multimedia Software: User Interface Design

Our second case study considers how a professional graphic designer uses the image manipulation program as part of producing interactive, multimedia websites and CD-ROMs. While the designer uses the tool for a range of tasks (including image toning), we focus on the portion of her work involving the design of graphical user interfaces.

When designing interfaces, the designer uses the image manipulation application primarily as a drawing and painting program. In this prototyping phase, she develops sets of alternatives for the user interface, exploring possibilities at all levels of granularity.

For example, she initially creates variations of the entire interface that explore alternative layouts for the content. These explorations result in sets of coarse-grained, high-level alternatives. When the general layout has been established, she may then turn her attention to exploring potential designs for individual components of the interface, such as buttons. This leads to much finer grained alternatives for the project. Figure 5 displays a collage of four interfaces prototyped for a website where both large- and small-scale differences are visible.

As with the previous case study, exploration and evaluation are critical processes. However, because of the nature of the task, we found a different set of strategies employed.

When exploring alternatives for small items such as buttons, the designer often does so by creating a large, blank canvas and laying out each possibility side-by-side. These variations enable her to discover and develop the right "look and feel" for the particular object by facilitating direct comparisons.

The art director with whom she works also creates variations of her work, but often embeds each alternative in a separate layer of the image, instead of placing them on the same canvas. By using layers, she can selectively evaluate the alternatives in the context of the entire interface by turning the layer with each variation on or off.

Finally, like the newspaper employee, alternatives are also stored in separate files. As such, three methods are used to manage variations: They are created and stored side-by-side in large canvases, within layers, and in multiple files. Because of this range of possibilities, a common protocol must be agreed upon when sharing alternatives. Given the art director's preference for layers, layers often serve this purpose. Across these strategies, it is important to note that the application does not provide native support for managing these alternatives, so ad-hoc strategies and protocols must be developed.

3.2.1 Relating Practices to Problem Solving Styles

While the user in the previous case study generates alternatives only after obtaining disappointing results, the designer in this study explicitly creates sets of alternatives to explore the design space, making her work practices most closely resemble set-based problem solving. Thus, her alternatives can be considered *generative variations* that are proactively created.

Like the previous study, evaluation of solutions is critical. However, evaluation occurs by comparing sets of alternatives after they are all developed. That is, evaluation of consecutive states of the same solution is not as important as the evaluation of separate solution instances that are more fully developed.

3.3 Amateur Artist: Coloring of a Pen Drawing

Our final case study considers the practices of an amateur artist using the image manipulation application to color a scanned-in pen drawing of a science-fiction scene. His goal is to decide on a color scheme for a line drawing before using real paints to paint a permanent version on wood. Since he has little experience painting in color, he uses the image manipulation program to help him iterate through variations before committing to using real paints.

To paint the pen-drawing, the artist uses the application to fill regions of the drawing with color. After making a change, the artist sits back and contemplates the changes. If the operations prove satisfactory, he continues. Otherwise, he undoes the operation and tries another variation. We call these small experiments *try-undo* cycles because he repeatedly tries and undoes commands until he finds a satisfactory result.

When results are uninspiring, the artist sometimes challenges himself by painting the drawing using the same palette as another painting, for example, one by Matisse. For these explorations, he creates folders labeled with the type of experiment (e.g., grayscale-only, versions based on a palette by Matisse, etc.) and names each file with a version number.

To evaluate the image, he often sits at a distance and squints his eyes. In so doing, he blurs the image in an attempt to get an overall sense of the balance of the composition. Less frequently, he prints out a grayscale version to reflect on the overall tonal quality of the image. Both techniques thus use alternative representations of the solution to aid in evaluation.

On occasion, we observed the artist consciously refuse to try and fix an unsatisfactory result. For example, after applying one command, he sat back and considered the state, commented that it was not as good as hoped, but remarked that he did not want to take the effort to undo it and try another alternative. In short, he forced himself to be satisfied with a suboptimal solution when it may have been advantageous to go back and try another alternative.

3.3.1 Relating Practices to Problem Solving Styles

Again, exploration and evaluation are common themes in this individual's work. This case study also reveals a combination of point- and set-based problem solving practices: The try-undo cycles are indicative of a point-based problem solving strategy and recall

Schön’s reflection-in-action, while the proactive exploration of different paint palettes reflect a set-based strategy that yields generative variations.

The subject’s use of squinting and black-and-white printouts is similar to Schön’s description of the use of “virtual worlds” (alternative representations) to facilitate the process of evaluating a solution and its overall fitness. In both cases, it is worth noting that only one state is considered at a time, rather than many in parallel.

While experimentation is just as common as in the previous cases, we also note that it is sometimes cut short when the cost of experimentation is perceived to be too high, even when the subject is dissatisfied with the result of the most recent action. This indicates that the transactional costs of experimenting can be too great, even when they may be beneficial.

3.4 Relating Problem Solving Techniques to User Interface Support

Table 1 lists each general problem solving strategy identified above, the method to enact that strategy (including user interface mechanisms employed), and the advantages and disadvantages associated with each approach. From this chart, we can consider how certain decisions in user interface designs can adversely affect the problem solving process.

Table 1. Mapping Practices to Interface Mechanisms

Problem Solving Practice	**Methods and Interface Mechanisms Used**	**Advantages**	**Deficiencies/Limitations**
Short-term experimentation (temporary generation of variations)	Undo: Try and undo different commands.	Strategy is easy to learn and enact.	Must invoke a command before getting a sense of its effect. Cannot compare multiple commands simultaneously.
	Parameter scrubbing with real-time previews.	Practice is part of process of setting parameter's values.	Modal dialog boxes prevent simultaneous previews of multiple commands. Single preview limits ability to survey multiple options for a command's parameters: Can only scan/compare alternatives for one parameter at a time. Scrubbing allows comparison of nearby parameter values, but comparisons can be made only in time and across only one dimension (parameter) at a time.
"Forking" current state to apply same command differently to copies of the *same* state	Duplicate current state, re-invoke command, find parameter settings of interest, apply command, repeat for each alternative.		Alternatives cannot be produced (forked) at moment of operating on data, such as when interacting with a command.

(Continued on next page…)

Table 1 (continued).

Problem Solving Practice	**Methods and Interface Mechanisms Used**	**Advantages**	**Deficiencies/Limitations**
Long-term exploration: Creating new standalone solution instances	Separate files: Each variation saved in a separate file.	Alternatives are separated into distinct files, achieving a 1:1 correspondence between an alternative and its existence. File names can indicate their contents.	Separate document instances can be created, but are treated as unrelated items by interface: Multiple states, whether stored in the file system or the undo stack, are nominally mutually exclusive states of which only one can be active at a time. Content from several files cannot be simultaneously manipulated. Evaluation tools typically not available for comparing alternatives stored in separate files (with the notable exception of textual data and corresponding diff tools).
	Embedded variations: Document is enlarged to hold multiple variations (e.g., through layers, side-by-side in a large canvas, etc.).	Separate, common content can be manipulated simultaneously. Little time required to duplicate data in same document (essentially requires "copy and paste" of data).	Multiple solutions in same document breaks WYSIWYG model: User must remove/hide undesired variations from document when evaluating overall solution.
	Undo stack: Restoring previous state using Undo.	Requires no extra effort on part of user to "store" an alternative. Backing up to a previous state a relatively easy operation.	Easy to lose alternatives (undoing to previous state then applying a new action; going beyond limits of undo stack; application clearing undo stack when document saved, etc.). Alternatives cannot be compared side-by-side. User must mentally track all variations in undo stack since there are no persistent reminders in the user interface.

(Continued on next page…)

Table 1 (continued).

Problem Solving Practice	Methods and Interface Mechanisms Used	Advantages	Deficiencies/Limitations
Simultaneous manipulation of separate alternatives	Select all alternatives, apply an operation to them (only possible when variations embedded in the *same* document).		If alternatives exist in separate documents, they cannot be simultaneously manipulated in the user interface (non-interactive batch processing sometimes possible, but lacks rich feedback loop).
Evaluation: Comparison of two different states	Comparisons in time: Undo and Redo used in quick succession to flash two different states on the screen. Clicking on previews temporarily displays current version, rather than preview.	Method works well when comparing discrete changes to small portions of document (user does not need to reestablish spatial context to perform comparisons).	Cannot quickly survey many options at the same time. Limited to comparing two contiguous states in history. Multiple commands cannot be simultaneously compared.
	Side-by-side comparisons of different documents.	Document windows can be arbitrarily arranged for comparisons.	Highly manual process unless appropriate version control/diff'ing software available for domain.
	Side-by-side comparisons of embedded alternatives.	Alternatives ready-at-hand.	Breaks the WYSIWYG model, requiring users to eventually prune lesser alternatives from document.
Evaluation: Isolated analysis of a single state (e.g., judging result of last action)	Pan and zoom for holistic analysis or detailed investigation. Using alternative representations, such as separate color channels or grayscale print-outs. Squinting.		Some methods are ad-hoc approaches that repurpose existing functionality for evaluative purposes. In these cases, evaluation mechanisms are not standardized for the benefit of everyone.

In general, deficiencies listed above result from interfaces lacking capabilities to treat alternatives as first-class objects. That is, there is an emphasis on displaying and manipulating only one state at a time, with few facilities to generate, manipulate, and evaluate multiple possibilities through streamlined operations. We call interfaces embodying this philosophy *point-based interfaces*.

3.4.1 Point-Based Interfaces

Point-based interfaces, like the problem solving technique from which their name is derived, assume that a user's mode of working is to choose a single goal (point) and continually revise a single solution instance until it matches the envisioned end state. In this scheme, a document can be in one, and only one, state at a time; users progress through tasks by applying an operation, then working on the new state that results.

Though conceptually simple for both user and implementer, point-based interfaces impose a serial, linear progression through a task that leaves little opportunity to engage in the non-linear, experimental practices typified by set-based problem solving. Thus, while a user may need to simultaneously explore multiple alternatives, point-based interfaces constrain one's ability to efficiently produce and navigate several possibilities in parallel. Specifically, the following attributes of point-based interfaces can impede set-based problem solving:

- *Interface tunnel vision*. Point-based interfaces display only one preview at a time, making it difficult to quickly and broadly assess multiple, potential future states. Thus, users cannot compare options *within* commands (i.e., different parameter settings), nor can they simultaneously perform comparisons *between* commands. Similarly, alternatives coincident in history can be evaluated using Undo-Redo in quick succession, but users must devise their own strategies to compare alternatives further back in the history or in separate documents.

- *Impaired ability to generate and pursue alternatives in parallel.* Point-based interfaces require that the interface be *prepared* for exploration. For example, users must explicitly save copies of the current state before exploring new, standalone alternatives. This setup-time detracts from the main task and introduces barriers to exploration.
- *Premature commitment to commands and arguments*. At times, multiple commands or parameter settings may seem equally viable. However, point-based interfaces require commitment to just one course of action at a time, since the underlying assumption is that a single state is continually revised until a solution is found. When users wish to deviate from this model, they must take the time to back up and prepare to explore by making copies of their current state.
- *Highly modal interfaces*. Highly modal interfaces are acceptable for point-based interfaces since it is assumed only one state can be considered at a time. However, modal interfaces are primarily of convenience to user interface implementers and not to users, as they constrain the ability to do things such as compare previews of multiple commands simultaneously.

When users wish to engage in set-based activities, these aspects of point-based interfaces create a tension between desired problem solving practices and those supported by the interface. Experimentation and exploration are costly to enact, and may be avoided simply because the transaction costs are perceived to be too high, as seen in the last case study with the artist.

To address these limitations, we suggest the need for interface tools that support set-based practices – mechanisms that support the simultaneous production, manipulation, evaluation, and management of alternatives. Before fully developing this concept, we first consider how previous efforts have addressed these needs in the HCI community.

3.5 Summary

Results from our observational study indicate that individuals wish to engage in both point- and set-based problem solving practices: We observed users develop corrective and generative variations, and found a great need to compare multiple solution states, whether they were consecutive instances of the same solution, or separate, standalone alternatives. However, we found that current interfaces cater to point-based practices, making it difficult and costly to generate, evaluate, and manage alternatives. Users were thus required to develop ad-hoc workarounds to achieve desired practices. To explore potential future states, they engaged in try-undo cycles and "scrubbed" commands' parameters to find desired settings. Users stored alternatives in files, within the same canvas, or within layers in the document. Finally, to evaluate the alternatives, they repeatedly invoked Undo-Redo to compare consecutive states, physically arranged the alternatives side-by-side, or cast the data into other representations (including grayscale versions and those that result from squinting one's eyes). Each of these activities could be streamlined with appropriate user interface-level support.

CHAPTER 4

INTERFACE-LEVEL SUPPORT FOR EXPERIMENTATION: RELATED WORK

The need to explore alternatives has been recognized by many within the HCI community, and is often seen as one of the ideal ways in which computers can assist the problem solving process. A number of interface design recommendations and tools suggest the forms this assistance may take. We review this existing work, and conclude the chapter by analyzing how these proposed tools have been evaluated.

4.1 Theoretical Guidelines

Drawing upon a wide range of studies of highly creative individuals, Shneiderman has created a framework of items interface designers should consider when seeking to support the creative process [SHN99, SHN00]. Most related to the practice of developing alternative solutions are suggestions that interfaces include support for tracking a user's history, offer what-if tools, and provide data visualizations. History tracking tools allow a user to more easily branch from past states, while what-if tools offer a structure for experimentation. Data visualizations can provide alternative representations of one's data similar in spirit to Schön's virtual worlds, thereby facilitating evaluation and understanding.

Thomas Green has developed a set of heuristics for user interface design, many of which are relevant to designing interfaces that enable exploration [GRE]. In particular, the notion of *viscosity* considers the amount of "resistance" the interface presents when users wish to make changes and revisions to data; *visibility* and *juxtaposibility* indicate the ease with which users can make comparisons between data; and *premature commitment* embodies the idea of an interface forcing users to commit to a state or action before having enough information to make an informed decision. These concepts echo many of the concepts of set-based problem solving, and many of these concepts can be seen in the findings of our observational study. For example, the notion of delaying commitment to a single solution is central to set-based practices. As related to our observational study, we found the interface particularly "viscous" when users were unsatisfied with a result and reluctant to undo it to pursue an alternative. These heuristics thus not only assist in evaluating user interfaces and the degree to which point-based or set-based practices are supported, but also provide a language for describing observed behaviors.

A number of researchers have translated findings from situated observations of domain experts into specific interface-level guidelines. For example, a study of website designers by Newman [NEW00] confirms that designers explore multiple alternatives in the initial conceptual phases of design. The various practices observed in this study led to a number of suggested applications and interface design guidelines. In particular, the authors advocate applications that afford rapid, informal prototyping, and enhanced history tools that more fully capture all states visited.

4.2 Specific Tool Implementations

Given the range of recommendations and guidelines offered for supporting exploration, and the types of tools that have been constructed, we cluster existing tool offerings into three primary categories:

1. Augmented histories
2. What-if tools
3. Enhanced previewing mechanisms.

Many improvements to the traditional stack-based history have been proposed. Automated history tracking tools, such as Timewarp [EDW97] or the Designer's Outpost [KLE02], replace the undo stack with a tree-like data structure that automatically records all states visited. With these tools, users do not need to take any explicit action to store states visited, enabling them to freely explore with the knowledge that they can access any point in the past. Other approaches retain the undo stack, but attempt to overcome some of its limitations by letting users easily record snapshots of important states [ADO, VER02]. These tools lower the barrier to exploration by streamlining the process of saving states prior to embarking on a path that may prove unsatisfactory.

Some approaches to augmenting histories seek to increase the ease with which one can make changes to past actions. Tools such as Editable Graphical Histories [KUR90], Timewarp [EDW97], and Selective Undo [BER94] all enable users to directly edit past actions, without needing to undo to the previous state. Changes made to earlier actions “trickle down” the history, automatically updating later, dependent states. These tools are especially valuable for point-based problem solving processes, because they allow more convenient revision of past states in place (i.e., users do not need to undo to a past state, make changes, then replicate their former steps). These capabilities also make them ideal for optimization tasks that require precise tuning of each step.

What-if tools, in contrast, enable users to more easily explore sets of alternatives in parallel. The common spreadsheet is often considered the best example of a what-if tool [CHI98a, JAN01]. In terms of set-based problem solving, its primary virtues are in providing the structure to embed alternatives (for example, alternative financial scenarios, each in its own column), and the tools to evaluate these differences (e.g., a graph plotting both columns and their scenarios). The basic principles of a spreadsheet, particularly its grid and cell-based semantics, have been imitated and applied to other domains, such as scientific visualization [CHI98a, JAN01]. In these latter cases, multiple views of a large data set are simultaneously presented to a user. These visualizations provide the ability to explore a problem from multiple angles.

Other tools maintain the spirit of a what-if tool, but instantiate the core concepts in a different way. Aran Lunzer's subjunctive interface [LUN98, LUN99] allows user interface objects to exist in multiple states at the same time. For example, a cannon in a physics simulator can be set to multiple angles, then "fired" to simultaneously view the trajectory of a cannonball for each angle specified.

The ART system [NAK00] is a what-if tool geared towards supporting writing practices. In contrast to conventional word processors, ART offers users three concurrent views and modes of working with text: a WYSIWYG document view, a text editing panel, and a two-dimensional space that allows free-form spatial organization of segments of text. In this latter view, the vertical ordering of the text segments determines their ordering in the WYSIWYG view. Apart from this application-imposed meaning of spatial relationships, users are free to arrange the text segments in whatever way best fits their particular needs and work styles. Thus, the free-form space provides the structure to store, consider, and manipulate multiple alternatives. These same concepts have been extended to the design of interfaces for dealing with other types of data, such as video data [YAM01, NAK02].

Magic Lenses and other see-through tools [BIE93, BIE94, HUD97, FOX98] facilitate experimentation without forcing commitment to any particular path. However, their use of physical metaphors (e.g., overlapping lenses), can make it difficult to simultaneously consider multiple alternatives for the same region.

Informal prototyping tools (e.g., [LAN01, KLE01, LI04]) also enable the rapid production of alternative solutions, but, in general, these applications do not provide the mechanisms for managing and comparing sets of alternatives in parallel.

While what-if tools provide explicit structure for holding alternatives, the creation of these alternatives is, by-and-large, manual – users must indicate which alternatives they would like to create and compare. Enhanced previewing mechanisms take a more proactive stance and automatically generate sets of potential future states for a user to consider. Design Galleries [MAR97, QUI02] are a good example of this concept. Design Galleries automatically generate hundreds of alternatives for a particular task, then select the most semantically distinct elements to present to the user. For example, in lighting a 3D scene, a Design Gallery will systematically vary the number, location, and intensity of lights, then present users with the most visually distinct results. Similar approaches can be found in systems that employ genetic algorithms to generate the alternatives [HEP02, HEP03].

The Suggestive Interface [IGA01] also generates multiple previews for users, but does so by attempting to infer the user's intentions from her most recent actions. The most probable future actions are then presented to the user to speed up her workflow. This concept has been instantiated in domains other than the original 3D modeling environment, as well [TSA04].

4.3 Additional Opportunities for Supporting the Exploration of Alternatives

All of these tools provide a valuable starting point for better supporting problem solving practices. However, there are a number of ways we can build upon these concepts to improve the caliber of user interface support for working with sets of alternatives.

4.3.1 Better Support for the Production and Management of Alternatives

In general, the most significant deficiency in existing approaches is the lack of tools to streamline the production of sets of alternatives that are simultaneously active. Augmented histories can more fully capture explored states, but are retrospective, passive tools that do not take an active role in the generation of new, standalone alternatives. As such, they are most likely to be used as safety-nets for point-based problem solving, facilitating the ease with which users can backtrack to a previous state. This use of the tool will lead to corrective variations being produced, rather than generative variations.

What-if tools offer the structure to hold alternatives, but current implementations provide little more than copy-and-paste operations to generate individual alternatives. Lunzer's subjunctive interface is one notable exception since it automatically generates sets of alternatives based on the set of potential values the user has specified.

Enhanced previewing tools generate sets of possibilities, but these sets are transient: users cannot instantiate multiple, standalone alternatives from these tools. Instead, they must commit to just one state at a time. If multiple viable alternatives are discovered and the user wishes to explore them in parallel, she must repeatedly reestablish the originating context to manually spawn each alternative. This is cumbersome, and discourages in-depth exploration. Thus, an opportunity exists for providing the ability to instantiate a new standalone alternative at the *moment of data manipulation*.

4.3.2 Simultaneous Manipulation of Alternative States

Alternative solutions can range from being highly divergent with respect to one another, to being more similar than different. For example, the sample of variations shown in Figure 5 present the four most distinct alternatives, but are, in many ways, more similar than they are different.

When variations are produced that share a number of commonalities, it is likely they will be manipulated in similar ways as they are further developed. Thus, at times, it may be helpful to conceptualize these alternatives as "the same thing" so that they can be *interactively* manipulated as one object. However, few interfaces allow one to easily apply operations to sets of solutions simultaneously. More typically, operations are provided that facilitate merging alternatives (e.g., [EDW97, KLE02]) or for creating a script that can be repeatedly run on different source states (e.g., [ADO]). Thus, while one can typically manipulate several objects in the same document simultaneously, this capability does not hold across sets of solutions.

4.3.3 Extracting and Displaying Embedded Alternatives as Standalone Entities

What-if tools provide the structure to hold alternatives, but in general, existing implementations do not provide mechanisms for the selective extraction of individual alternatives to form a final, standalone version. For example, embedding alternatives in different columns of a spreadsheet provides a convenient storage and comparison mechanism, but is more akin to the practice we observed in which the designer placed alternative user interface elements side-by-side on the same canvas. Providing the structure for holding alternatives is an important first step, but interfaces must go full circle and offer facilities to selectively display each alternative on its own.

4.3.4 Tighter Feedback Loops and Better Navigation Mechanisms

Enhanced previewing mechanisms produce several options for users to consider, but do not always have a tight feedback loop or the ability to arbitrarily navigate the space of possibilities. For example, Design Galleries relies on heuristics to determine which alternatives will be of most interest to users. One of the primary limitations of this approach is that it must be customized to a particular task, making the tool less amenable to other, unforeseen tasks. Generally lacking, then, are navigation tools that grant the user more agency in navigating the possibilities.

Most previewing mechanisms also preview only one step “ahead” – strings of commands cannot be arbitrarily composed to explore several steps ahead. When multiple steps can be previewed, as with Magic Lenses, only one path can be followed at a time; it is not possible to compare two separate paths in depth.

4.3.5 Weak Previews for Direct Manipulation Actions

A significant class of operations – those involving direct manipulation, such as painting on a canvas – have weak preview support. When support does exist, it is generally a static, "canned" preview unrelated to the user's actual data [BAE91]. Thus, there is an opportunity to devise ways in which users can adjust these tools' parameters while viewing dynamic previews of how these tools will affect their data.

4.4 Evaluation Needs

For over 20 years, various history tools, permutations of Undo, what-if tools, and previewing mechanisms have been proposed. However, despite the range of mechanisms developed and suggested over this period of time, few have been evaluated in terms of their impact on the problem solving process.

Table 2 presents a survey of these tools, whether they have been evaluated, and a brief synopsis of the evaluation method. We restrict ourselves to considering only those tools designed for single-user use in solving a problem. We thus exclude tools primarily developed to support collaboration, or tools developed to support tasks in which something is not being created (such as web browsing).

Table 2. Survey of process support tools

Tool Type	Reference	Evaluated?	Evaluation method
History: Command interpreter history manipulation	How users repeat their actions on computers: Principles for design of history mechanisms [GRE88]	Yes	Log analysis of shell command history
	Investigations into history tools for user support [LEE92]	Yes	Log analysis of shell command history
History: Branching history	Papyrus: A history-based VLSI design process management system [CHI94]	No	
	Zodiac: A history-based interactive video authoring system [CHI98b]	No	
	Data exploration across temporal contexts [DER00]	No	
	Where do web sites come from? Capturing and interacting with design history [KLE02]	Yes	Qualitative evaluation with 6 designers
	A history mechanism for visual data mining [KRE04]	No	
History: History summarization	Translucent history [GEN95]	No	
History: Manipulable history	An editor for revision control [FRA87]	No	
	A visual language for browsing, undoing, and redoing graphical interface commands [KUR90]	No	
	Segmented interaction history in a collaborative interface agent [RIC97]	No	
	Manipulating history in generative hypermedia [KHA04]	No	

(Continued on next page...)

Table 2 (continued).

Tool Type	Reference	Evaluated?	Evaluation method
History: Linear histories	TimeScape: A time machine for the desktop environment [REK99]	No	
	Snapshots and bookmarks as a graphical design history [VER02]	No	
Undo	User recovery and reversal in interactive systems [ARC84]	No	
	US&R: A new framework for redoing [VIT84]	No	
	Concepts and implications of undo for interactive recovery [GOR85]	No	
	A formal approach to undo operations in programming languages [LEE86]	No	
	A selective undo mechanism for graphical user interfaces based on command objects [BER94]	No	
	Object-based nonlinear undo model [ZHO97]	No	
	A temporal model for multi-level undo and redo [EDW00]	No	
	Dynamic hierarchical undo facility in a fine-grained component environment [WAS02]	No	
	A usability study of an object-based undo facility [VAR03]	Yes	Compared preference, task performance, for using Undo in tasks requiring user to return to a previous state.
	Regional undo for spreadsheets [KAW04]	No	

(Continued on next page...)

Table 2 (continued).

Tool Type	Reference	Evaluated?	Evaluation method
Previewing mechanisms	Toolglass and magic lenses: The see-through interface [BIE93]	No	
	A Taxonomy of see-through tools [BIE94]	No	
	Design galleries: A general approach to setting parameters for computer graphics and animation [MAR97]	No	
	Composing magic lenses [FOX98]	No	
	A suggestive interface for 3D drawing [IGA01]	Yes	Informal user study to assess tool's usability.
	Semi-automatic antenna design via sampling and visualization [QUI02]	No	
	Interactive evolution for systematic exploration of a parameter space [HEP03]	No	
	A suggestive interface for image guided 3D sketching [TSA04]	Yes	Informal user study with two users.

(Continued on next page...)

Table 2 (continued).

Tool Type	Reference	Evaluated?	Evaluation method
What-if Tools	Reconnaissance support for juggling multiple processing options [LUN94]	No	
	Principles for information visualization spreadsheets [CHI98a]	Yes	Results not reported.
	Towards the subjunctive interface: General support for parameter exploration by overlaying alternative application states [LUN98]	No	
	Choice and comparison where the user wants them: Subjunctive interfaces for computer-supported exploration [LUN99]	No	
	A spreadsheet interface for visualization exploration [JAN00]	No	
	Two-dimensional positioning as a means for reflection in design [NAK00]	Yes	Qualitative study of tool use with eye-tracking.
	Subjunctive interface support for combining context-dependent semi-structured resources [LUN01]	No	
	Usability studies on a visualization for parallel display and control of alternative scenarios [LUN04]	Yes	Two controlled laboratory studies measuring task performance, accuracy, and cognitive load.

Of the 40 published papers we found meeting our criteria, 8 (20%) report user studies. However, this number is optimistic since only 4 report lessons learned from the evaluation [NAK00, KLE02, VAR03, LUN04]. Of these, only 2 [VAR03, LUN04] are comparative in nature, both being controlled laboratory studies. The study conducted by Lunzer and Hornbaek [LUN04] stands out by virtue of the host of measures collected to understand the tools' effect on task performance, accuracy, and cognitive load.

From this survey, we conclude that a considerable number of mechanisms have been proposed to facilitate looking into the future, reaching into the past, and comparing alternative possibilities in the present. Yet, we are left with a very incomplete picture of how these tools actually influence the problem solving process. Lunzer's study indicates that these types of tools can reduce the time and number of operations required to find a solution, though he found no effect on solution correctness. Furthermore, both Vargas [VAR03] and Lunzer's [LUN04] studies indicate that users like the functionality their respective tools offer. However, none of these studies provide any indication of how the problem solving process *changes* as a result of using these types of tools. For example, do subjects explore more broadly, more frequently, or longer when a previewing or enhanced history tool is available? Since many of these tools are intended to support this more exploratory behavior, these are important questions to consider during evaluation.

Also left unanswered by these studies is how these tools impact the process of solving *ill-defined* problems. Nakakoji et al. [NAK00] and Klemmer et al. [KLE02] both evaluate their tools in the context of solving an ill-defined problem, but neither study employs controls to enable comparisons to be made. Lunzer's study does employ controls, but the tasks are well-defined tasks with verifiably correct answers.

In sum, there is not only a need to deepen our knowledge regarding the influence of these tools on the problem solving process, but also a need for more standardized methods with which to perform evaluations. We will address both issues in subsequent chapters.

4.5 Summary

Computers have long been viewed as excellent vehicles for experimentation. Three classes of general-purpose user interface mechanisms have been proposed to facilitate this process: augmented history tools, what-if tools, and enhanced previewing mechanisms. While each can assist in problem solving, there are a number of research opportunities for extending interface-level support for experimentation. In particular, there is little support for producing and managing sets of alternatives that are simultaneously active; one cannot interactively manipulation sets of alternative solutions; and few interfaces offer previews for direct manipulation operations. Finally, despite 40 different instantiations of these types of tools over two decades, few have investigated *how* these tools affect the problem solving process. There is thus a real need to more fully evaluate the impact of these tools on the problem solving process.

CHAPTER 5

SUPPORTING THE GENERATION, MANIPULATION, EVALUATION, AND MANAGEMENT OF ALTERNATIVES

The goals of this research are to instantiate the concepts of set-based problem solving in the context of a *set-based interface* and to evaluate the impact such an interface has on the problem-solving process. In this chapter, we develop the concepts of a set-based interface in full. We conclude by comparing these concepts to the subjunctive interface, a research effort similar in spirit to the set-based interface.

5.1 The Set-Based Interface: Reasoning About Sets of Alternatives

The intention of a set-based interface is to support the parallel development of sets of alternative solutions. Interfaces can be loosely categorized according to this relatively simple concept: The degree to which a user interface supports the parallel production, development, and evaluation of alternative solutions corresponds to the degree to which the interface is set-based. Conversely, the degree to which users must adopt a serial problem solving process corresponds to the degree to which the interface is point-based. Importantly, these distinctions are not meant to be absolute, but rather, intended to sensitize designers to how their interface designs may influence the problem solving process.

5.1.1 Principle Components of a Set-Based Interface

We consider an idealized set-based interface to provide the following capabilities:

1. *Set Management*. Users can designate, add, remove, access, organize, and selectively view viable alternatives within the same workspace.
2. *Set Generation*. The generation of sets of alternatives is streamlined, and can result in sets that are:
    a. Largely *transient* in nature, or
    b. Relatively *persistent* over time.
3. *Set Manipulation*. Multiple states can be operated upon simultaneously as if they were the same object.
4. *Set Evaluation*. Facilities exist to assist the process of evaluating several possibilities at once.

While we list each of these capabilities separately, the divisions are not absolute. In fact, much of the functionality must, out of necessity, interoperate to ensure a uniform interaction experience for users. However, it is useful to conceptualize this functionality separately when considering what kinds of facilities are necessary to support set-based problem solving practices. We consider each of these capabilities in more detail next.

5.1.1.1 Set Management

When users wish to explore separate, standalone alternatives with current interfaces, they must prepare the interface for exploration by making copies of their data prior to exploration. This activity requires the user to choose how and where to store these alternatives since no infrastructure exists to manage sets of possibilities.

Set management seeks to address this issue by providing the basic services and infrastructure necessary for designating, organizing, storing, selecting, removing, and activating solution alternatives. For example, the interface could provide an area containing snapshots of solution states, offering the user one location to turn to when storing or retrieving possibilities. In essence, set management seeks to elevate solution alternatives to first-class objects that can be as easily manipulated as the domain-specific data itself.

5.1.1.2 Set Generation

Set generation mechanisms facilitate the *production* of alternatives, making it easier to create both transient and permanent variations. Transient sets are realized through previews, which allow users to rapidly survey the space of possibilities through "scouting missions" [LUN94]. Permanent variations, on the other hand, allow one to create a handful of persistent possibilities worthy of longer-term exploration. In the former case, the interface can facilitate set generation by automatically generating navigable sets of previews, while in the latter case, the interface can assist with production by streamlining the process of "forking" and creating new standalone alternatives.

Conceptually, a user could identify the need to generate alternatives at three different times with respect to data manipulation: before, during, or after data manipulation. In the first case, a user may realize she needs to generate a number of alternatives to compare them, and thus prepares for this process by creating copies of the source data (generative variations). Later, while actively manipulating the data, she may discover additional, unexpected sets of possibilities that she wishes to explore in greater depth (serendipitous variations). Finally, after applying an operation, she may realize that other alternatives should be explored because results are unsatisfactory (corrective variations). A set-based interface seeks to provide support for generating alternatives at any of these times.

5.1.1.3 Set Manipulation

Sets of possibilities can by highly diverse, or may share a fair number of similarities. Where appropriate, users should be able to interactively manipulate these sets as a whole. For example, when developing a handful of alternatives, it may be necessary to adjust a common aspect shared by all. Rather than sequentially apply the same operation to each in turn, the set should be addressable as a single entity. In essence, this concept extends the existing convention of being able to select and operate on multiple objects in a single document so that users can operate on multiple solution alternatives simultaneously.

5.1.1.4 Set Evaluation

The final component in a set-based interface is the ability to evaluate any of the alternatives produced, whether they are transient or permanent in nature. Explicit tools should be provided so that any and all alternatives can be compared with one another.

Evaluation tools may take many forms, from simple mechanisms that afford side-by-side comparisons, to those that allow comparisons in time. They may also be proactive and explicitly highlight the differences, as "diff" utilities do for text. Finally, evaluation mechanisms may alter the representational form of the data to facilitate comparisons, as suggested by Schön's virtual worlds.

5.2 Comparison to the subjunctive interface

Aran Lunzer's formulation of the subjunctive interface [LUN98, LUN99, LUN01, LUN04] shares a number of similarities with the concept of a set-based interface. Both research agendas recognize the need to develop and explore multiple alternatives in parallel. The primary difference between these two research tracts lies in how multiplicity is realized in the user interface. In his work, Lunzer has focused primarily on allowing user interface *mechanisms* to exist in multiple states simultaneously. For example, a user may be able to simultaneously select multiple values in a drop-down menu, resulting in the generation of a set of alternatives [LUN04]. As we will see most clearly in the next chapter, a set-based interface focuses more on providing facilities for managing sets of *data*; it maintains the convention that a function, when invoked, accepts only one set of parameters at a time. The subjunctive interface, on the other hand, relaxes this requirement, and allows multiple parameter values to be passed to a function, resulting in several output values being generated.

5.3 Summary

A set-based interface provides explicit support for the generation, manipulation, evaluation, and management of separate, standalone solution alternatives. It is not a specific implementation, but rather, represents a set of concepts to guide interface design.

CHAPTER 6

SET-BASED INTERFACE TOOLS: SIDE VIEWS AND PARALLEL PIES

In this chapter, we describe the design and implementation of Side Views [TER02b] and Parallel Pies [TER04], two tools that instantiate the concepts of a set-based interface. Side Views is a set generation mechanism for rapidly exploring a large terrain of transient possibilities. Parallel Pies, on the other hand, affords more prolonged exploration of a select set of alternatives. Either of these tools can easily exist on its own, but they work particularly well in concert with one another: Side Views can be used to find the most attractive possibilities, while Parallel Pies can turn them into permanent, standalone solution instances.

6.1 Side Views

Side Views provide on-demand, persistent previews of one or more commands and their parameters. They initially appear as a transient pop-up window, similar to the common tool-tip mechanism (Figure 6 and Figure 7). However, they can be made to persist by clicking on the preview window, or invoking the command. Multiple Side View windows can be instantiated simultaneously, affording side-by-side comparisons of commands (Figure 8).

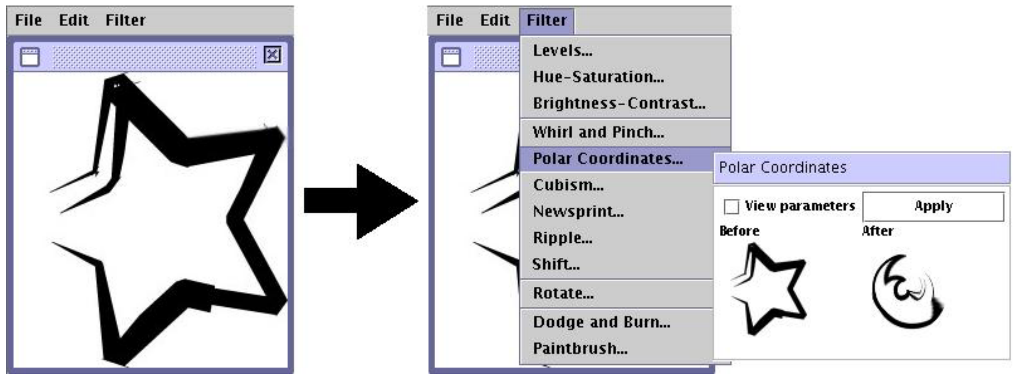
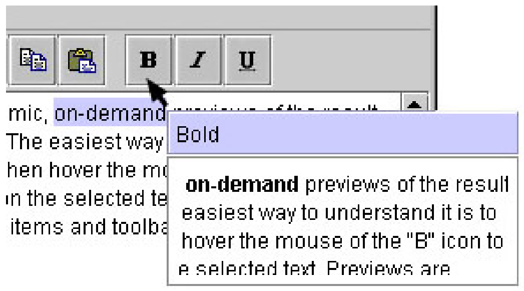


Figure 6. Sequence illustrating an on-demand Side View popping up.



mic, on-demand
The easiest way
hen hover the m
n the selected te
items and toolba

on-demand previews of the result
easiest way to understand it is to
hover the mouse of the "B" icon to
e selected text. Previews are

Figure 7. A Side View for a toolbar item in a text editor.

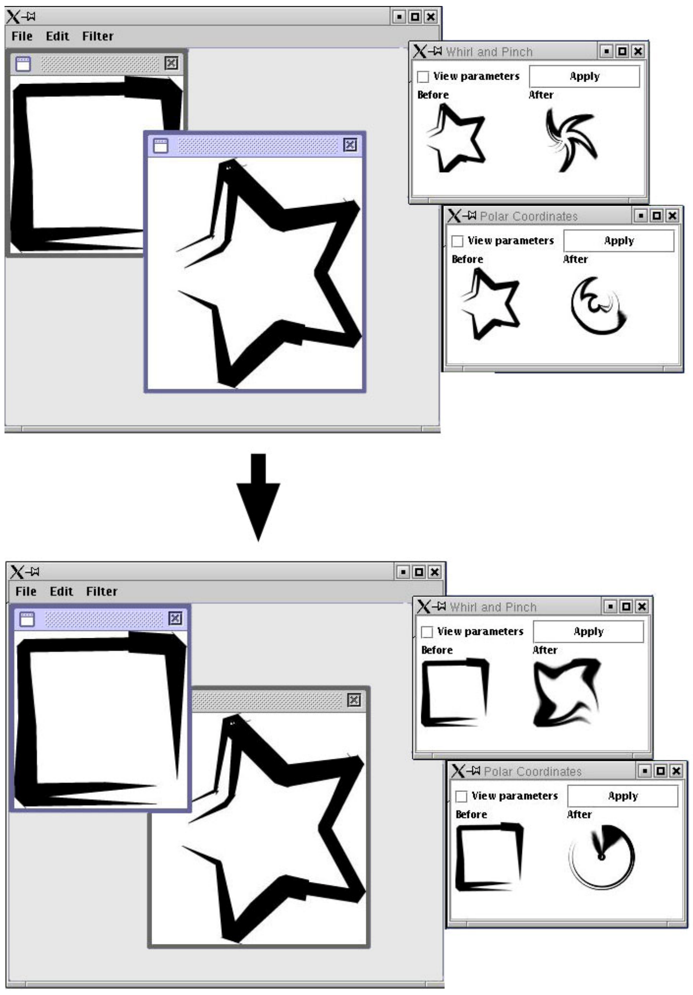


Figure 8. Side Views automatically update as content changes.

Side Views continually update their previews to reflect the current state of the active document. For example, as modifications are made to a document, or the active document is switched to another document, Side Views automatically update their previews to reflect the new active content (Figure 8).

Side Views for commands with parameters initially show a single preview using the default (or the user's last) settings for a command. When a user desires to see and/or interact with a broader range of possibilities for a single command, Side Views can be expanded into *parameter spectrums* (Figure 9 and Figure 10).

Parameter spectrums show a series of previews across the range of values for each parameter. Initially, the range displayed is a sampling of all possible values for a parameter. However, the user can vary the range previewed to focus on a smaller set of values.

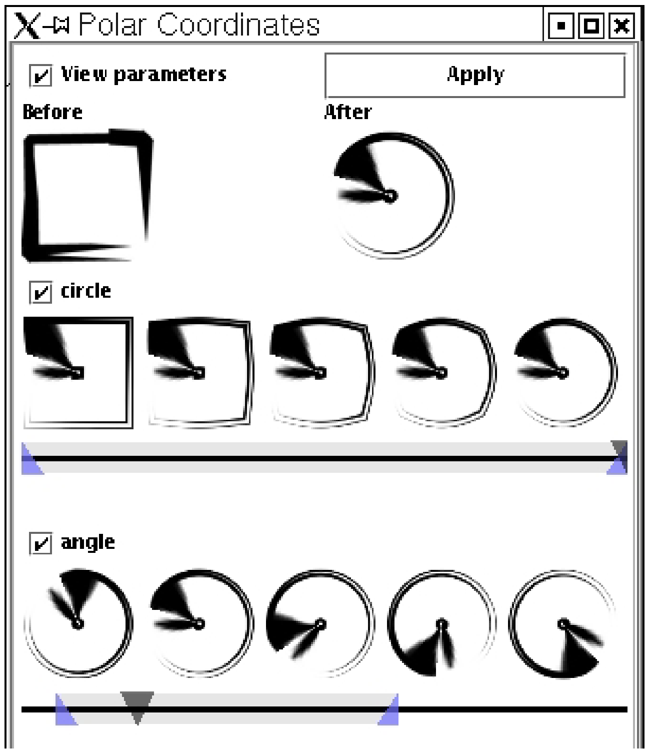


Figure 9. A set of parameter spectrums for the Polar Coordinates command.

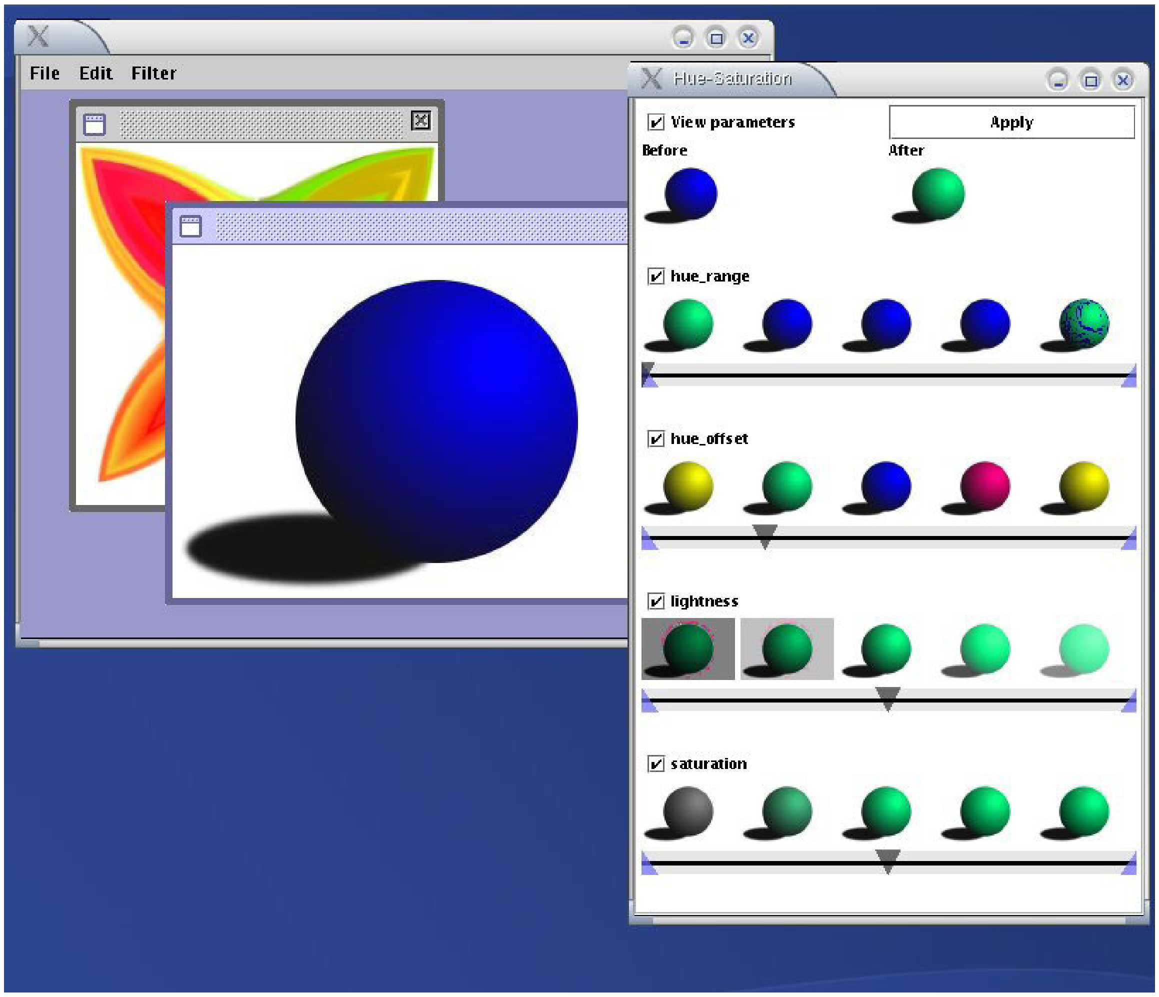


Figure 10. Parameter spectrums for the Hue/Lightness/Saturation command.

Each spectrum varies its previews only on the parameter it represents. For example, if there are two parameters for an oval – height and width – the parameter spectrum for height uses the current setting of the width parameter, and varies its previews in the height dimension. When one parameter's current value is changed, all other parameter spectrums update their previews to reflect this new value. This behavior enables a user to interactively vary one parameter's settings to understand how it affects the other parameters.

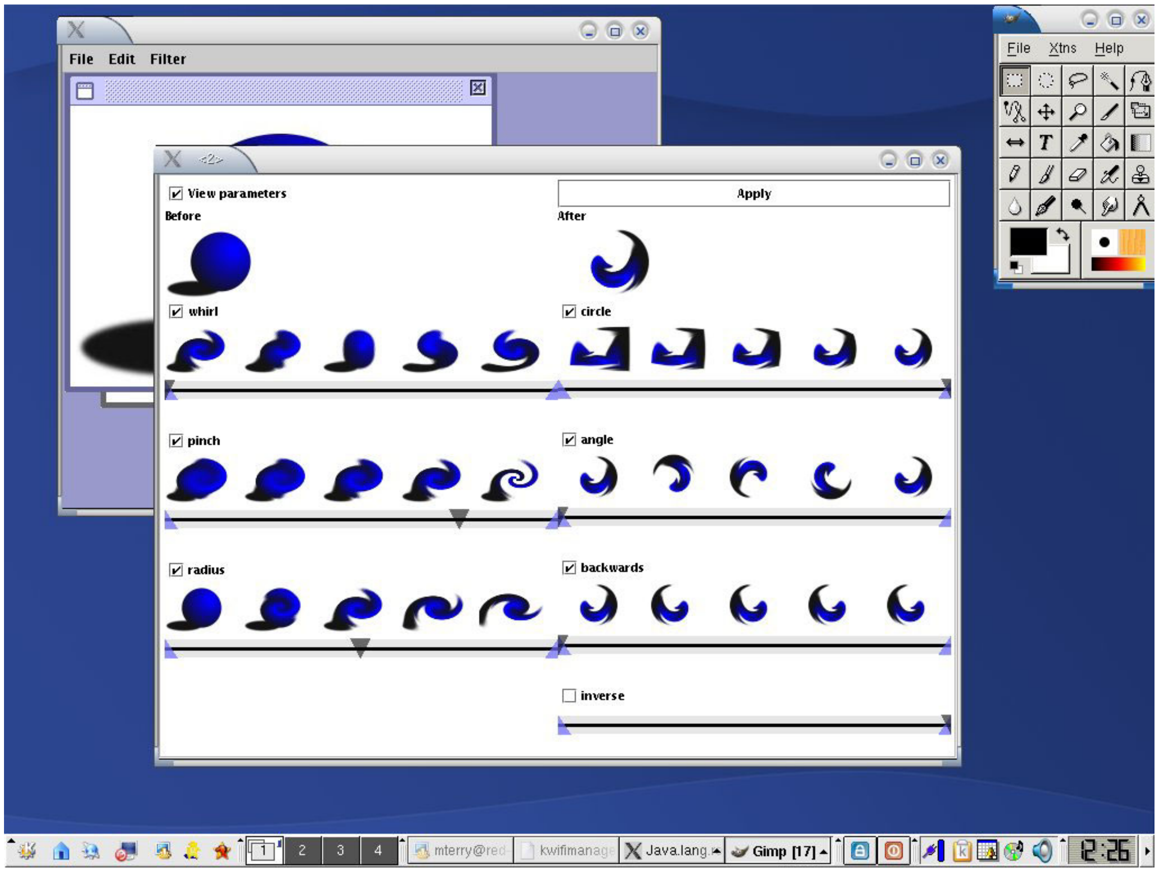


Figure 11. Two commands (Whirl & Pinch, and Polar Coordinates) chained together in a composite Side View.

To view previews of two or more Side Views combined (function composition), users can chain Side Views together. In our current implementation, chaining commands is supported by dragging and dropping one preview on the other. Performing this action causes a new Side View to appear that previews the effect of both commands at once (Figure 11).

As with Side Views for a single command, users can view and vary each command's parameters individually. Changing the parameters in a Side View early in the chain causes subsequent previews to update, since later Side Views are dependent on the output of former Side Views. The effect is one in which users can view changes "ripple" down the chain of Side Views.

Many operations require direct input, such as text entry or mouse input, and some settings affect how that input is interpreted. For example, the size, color, and opacity of a paint brush influence how a user's mouse strokes will be "painted" on a canvas.

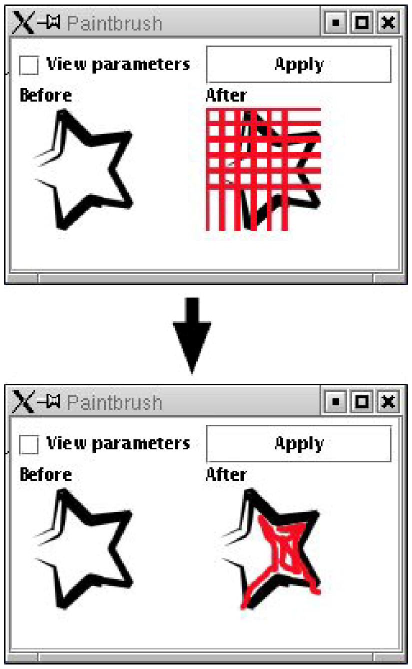


Figure 12. Side Views for the paint brush initially show a grid of paint strokes. These are replaced by a custom stroke when the user moves the cursor over the original image.

Side Views for commands that require direct input present a unique challenge in that the data to be previewed has not yet been entered. Yet it is desirable to be able to preview the settings for the tool before actually using it. In these circumstances, Side Views mimic user input in their initial previews, then allow user interaction to create data to preview. For example, the Side View for a paint brush initially shows an overlapping grid of strokes that enables the user to see the effects of both single and overlapping paint strokes on her image (Figure 12). If the grid does not provide the information needed, the user can move the cursor over the original image. This action clears the default grid in the Side View and replaces it with the user's mouse strokes (Figure 13). These strokes persist after the cursor has left the image, allowing the user to interactively vary parameters and view the effect in the sample paint stroke drawn. This capability effectively enables users to retroactively vary the parameters for previews of commands with direct input.

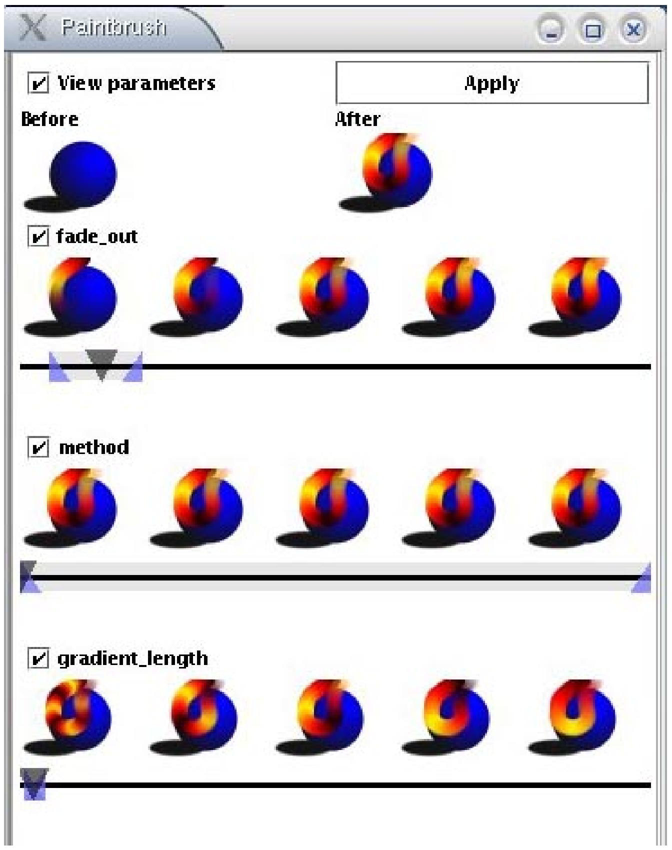


Figure 13. Side Views for the paint brush. Characteristics of the paint brush can be interactively previewed and manipulated as with any other command.

6.1.1 Affordances of Side Views

Side Views' feature set makes them amenable to a range of tasks. At a high level, Side Views enable users to more easily perform breadth- and depth-first searches of possibilities without committing to any one course of action. They can thus be considered a set-generation mechanism that enables the rapid creation of lots of (relatively) transient previews for quick exploration. Within the context of this use, users can continually evaluate results via direct comparisons.

Side Views also have some properties that facilitate sequential manipulation of sets of alternatives. When separate, standalone variations must be developed in similar ways, users may need to slightly modify the command to the particularities of each variation. That is, they may not be able to blindly apply the exact same command to each variation. In these cases, persistent Side Views can ease this process through their parameter spectrums. After applying the command to one variation, a Side View will retain its settings as the user switches to another variation. Users can then refer to the parameter spectrums to adjust the command for the nuisances of the new variation.

6.1.2 Architecture

To date, we have implemented Side Views in two applications, a rich text editor and an image manipulation program. Both were written in Java, with the GNU Image Manipulation Program (the GIMP) [GIMP] used as the image manipulation engine for the latter application.

Side Views' architecture is shared between the two applications, and its design is intended to be flexible and extensible. It makes use of a modular, pluggable design built on a set of design patterns [GAM94] to factor out behavior and functionality that may differ across applications and user needs. While our first implementation is within a graphical user interface, the design itself is not explicitly tied to a GUI, and could be used within other interfaces, such as one whose primary mode of input is speech. We explain its design next.

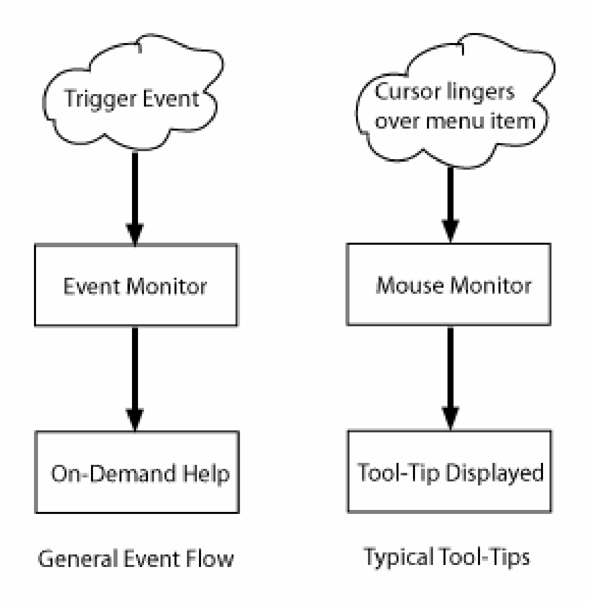


Figure 14. A generalization of on-demand help. A specific example of tool tips is given on the right.

The design of the architecture is driven by the observation that on-demand help is triggered by a specific event, but that this event and the form of help may vary (see Figure 14). For example, a traditional tool-tip is shown after the mouse hovers over an interface object for a short period of time. However, it is reasonable to assume that the trigger event could arise from keystrokes, and that the help given could take another form, such as synthetic speech, rather than visual cues alone. Thus, the design of Side Views breaks the architecture into components that monitor an interface for events, and factory classes to generate and activate application-specific Side Views when needed. Figure 15 shows an overview of this design.

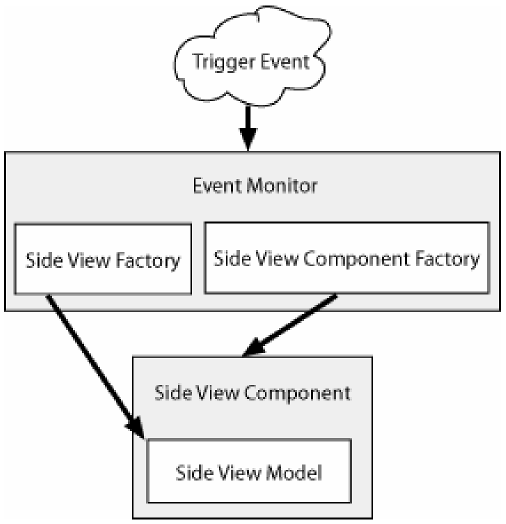


Figure 15. Basic architectural design of Side Views. A trigger event causes factory objects to compose the Side View for the user.

Given this overview of the basic architectural design, we turn now to the specific interaction semantics of Side Views, followed by implementation issues.

6.1.3 Interaction Semantics

6.1.3.1 Invocation

The initial invocation of Side Views within our two sample applications is identical to that of normal tool-tips: Hovering over a user interface object (such as a menu command) causes a Side View to appear after a delay. Existing tool-tips typically appear after a 750ms delay. However, in our informal tests, users requested a shorter delay because they wanted faster access to Side Views when sampling the interface. Thus, our current implementation uses a much shorter 400ms delay. Our hypothesis is that the additional utility of Side Views for experts motivates the request for a shorter delay.

6.1.3.2 Instantaneous Display of New Side Views after Initial Display

As with normal tool-tips, if a transient Side View is visible, other Side Views will instantly appear if the cursor moves to a new object. This instantaneous appearance of new Side Views is important as it facilitates rapid sampling of options within the interface: After one Side View has been made visible, users can simply sweep the interface with their cursor, pausing over items of interest to discern their effect, rather than having to wait the default delay for each interface object.

6.1.3.3 Making a Side View Persistent

To make a Side View persistent, users click in the title bar of a Side View. This behavior changes the Side View into a "regular" window that does not automatically disappear. Persistent Side Views dynamically update their previews to reflect the current state of the active document.

6.1.3.4 Extended and Suspended Dismiss Delays

Normal tool-tips automatically hide themselves after a certain amount of time (~7 seconds), called a dismiss delay. However, Side Views offer a much more information-rich preview, and thus can require more time to view. Thus, we extend Side Views' dismiss delay to 10 seconds, but also add the ability for the delay to be temporarily halted. When a user's cursor enters a transient Side View, the dismiss timer is stopped until the cursor leaves the Side View.

6.1.3.5 Supporting Interaction

Supporting interaction *within* a transient Side View requires a modification to typical tool-tip implementations, because normal tool-tips hide themselves whenever the user moves the cursor out of the component that generated the tool-tip. For transient Side Views to support interaction, we introduce a "grace period" whereby the Side View remains visible for a short period of time after the cursor leaves the triggering object. Currently, we use a 750ms delay for this value: After the cursor leaves the object that initiated the Side View, the Side View will remain visible for an additional 750ms to give the user time to enter a transient Side View. If the cursor enters another interface object (e.g., another menu item), then the original Side View is immediately hidden and a Side View for the second object is shown.

After the cursor enters the transient Side View, an identical 750ms delay is used before dismissing a Side View when the user's cursor leaves the Side View. This delay takes care of the case in which the user accidentally enters and exits a transient Side View when she actually intends to interact with it.

6.1.4 Implementation Issues

6.1.4.1 Preview Content

Determining what information to preview can be challenging for some commands. For commands that produce a visual change in the interface, one challenge is choosing how to render previews for changes that are too large for a Side View window. For example, if a user selects all the text in a multi-page document and wishes to preview a style change (such as a different font size), there is a question of how the Side View should render this rather large preview.

For commands for which there are no visual changes in the interface (such as the print icon in a toolbar, which automatically prints the current document using the last settings), the choice of what to preview is also unclear. There are a number of approaches one could take to address these concerns. For example, for changes to large selections in a text document, the Side View could show the most recently edited text, the beginning or end of the current text selection, a scaled-down version, and so on. However, through the design, development, and use of Side Views, it appears that no single heuristic can reliably predict the content of most interest to the user at any point in time – there are simply too many special cases to consider. Therefore, in our implementation, we chose to apply a consistent, predictable algorithm to the initial display of each Side View, but allow the user to modify the preview.

For Side Views in the text editor, we align the beginning of the text selection in the top-left corner of the Side View, unless the content is at the right margin, in which case we right-align the preview (as in Figure 7). In the image manipulation application, transient Side Views always appear at a fixed thumbnail-size that displays a before and after preview. These previews can later be resized to show larger previews, and users can choose to view the parameter spectrums as well.

For commands that do not produce a visual change, our rule of thumb is to present a summary of the effect of the command. For example, the Side View for the print icon could show a print preview, but this is largely unnecessary since most applications are WYSIWYG. Instead, the Side View could display the printer settings that will be applied: the printer that will be used, the number of copies, the quality, etc.

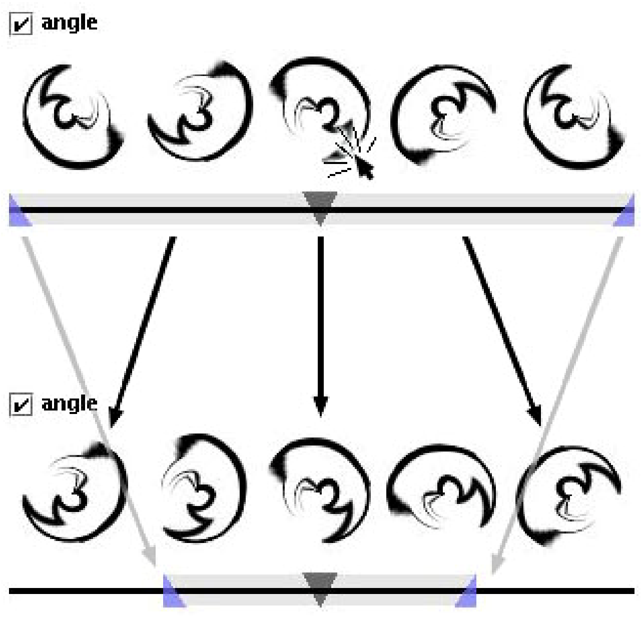


Figure 16. Narrowing the range previewed in Side Views. Clicking on a value narrows the range of previews shown. Above, the user clicks on the middle value, which forces its nearest neighbors to become the new boundary values.

6.1.4.2 Parameter Spectrums

As described above, parameter spectrums originally display a sampling of previews across the full range of possible values for each parameter. Users can vary the range shown in two ways. First, they can directly vary the boundaries of the range via a slider that has three handles. The center handle acts as a normal handle in a slider, selecting the current value. The outer handles – the range handles – serve to vary the boundaries of the range of values shown (see Figure 16).

Second, users can click on a preview of interest. When a user clicks a preview, the values of the previews to the left and right of the one chosen become the new boundaries for the range of values shown (again, refer to Figure 16.)

6.1.4.3 Computationally Expensive Commands

Not all commands can be instantly previewed. Generating a preview might be computationally costly, by virtue of the operations performed, the amount of data that needs to be copied, etc.

In our implementations, we have addressed this problem in several ways. First, we create threads that perform background rendering. These threads enable the interface to remain responsive, while providing updated previews as they become available. Second, in the image manipulation application, we first scale an image down to its thumbnail size, then apply the required filter. This procedure quickly produces accurate previews for most operations (such as those that modify an image's colors, its rotation, etc.), but not all – some filters are not commutative with respect to scaling transformations. (That is, a filter will produce different results if the image is first scaled then transformed, rather than vice versa.) Thus, another background rendering thread is required to apply these filters to a full-scale version of the image before scaling. Because we have been working primarily with operations that are commutative with respect to scaling transformations, we have only implemented the first tier of rendering threads at this time.

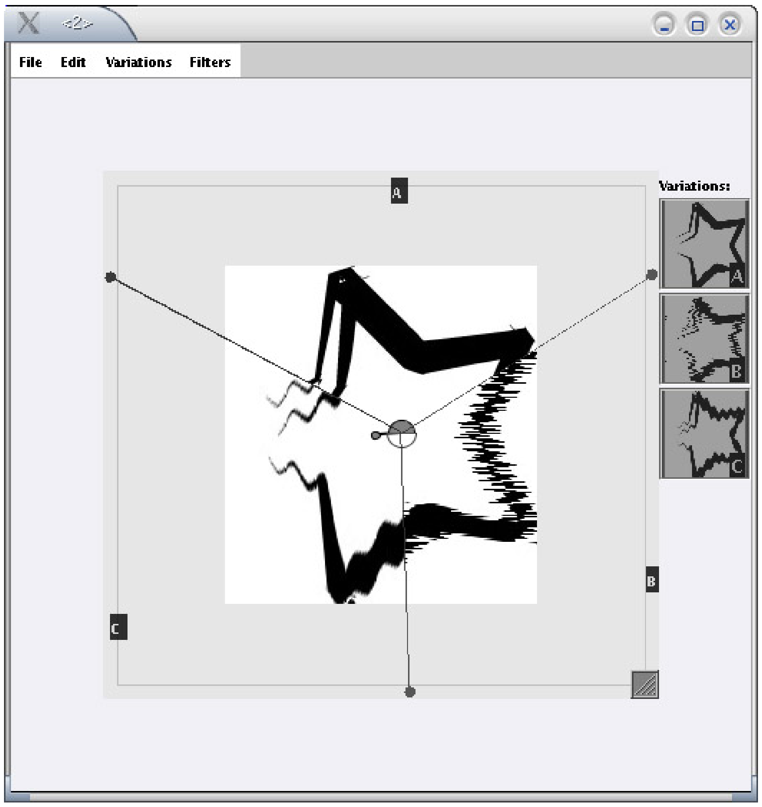


Figure 17. Parallel Pies. Parallel Pies allow multiple alternatives to be embedded in the same workspace.

6.2 Parallel Pies

While Side Views provides an explicit mechanism to generate and survey one's options, Parallel Pies provides the structure to generate, manage, and simultaneously manipulate a smaller, more permanent set of solution states. Parallel Pies consists of a visualization tool that facilitates evaluations of alternatives (Figure 17) and a number of smaller changes made throughout the interface to enable variations to be added or removed from the workspace. At a high level, Parallel Pies allows users to:

- Create new alternatives before, during, and after invoking a command (set management and generation),
- Embed alternative solutions within the solution workspace (set management and evaluation), and
- Manipulate each alternative independently or collectively (set manipulation).

To realize this tool, a number of changes were necessary across the interface. In our implementation, the interface is augmented in the following ways:

- Command dialog boxes introduce a new option, *Add Variation*[1], which allows users to add the currently previewed result as a new alternative to the given solution (Figure 18)
- Any variation can be *duplicated* to create a new alternative
- Each variation maintains a *complete history* of all its prior states, initially adopting the history of its source. Thus, users can duplicate a variation, then return to a previous state to pursue an alternative path
- Alternatives are embedded directly within the same solution workspace and viewable through the Parallel Pies visualization (Figure 17). This visualization evenly divides the workspace to show portions of each alternative side-by-side
- Commands are augmented to allow users to modify one or more alternatives simultaneously

We describe each of these features, and their motivation, in turn.

---

[1] In the implementation used for our controlled studies, this feature was renamed to "Create New Version." Originally, we chose "Add Variation" to distinguish it from revision control systems, but later settled on "Create New Version" to better convey the functionality to the end user.

6.2.1 Set Generation

As discussed in Chapter 5, there are three conceptually different situations in which a user may discover it is necessary to explore alternatives:

1. *Before a command is invoked*. In this situation, the user realizes that the current state of the problem will necessitate exploring a number of alternatives
2. *While interacting with the command*. At this point, the user may discover a number of interesting alternatives, or be unable to find one that perfectly fits the problem
3. *After a command has been applied*. In this case, the results obtained are not as hoped, though not without value. This realization may come immediately after invoking a command, or several steps later, when it becomes clearer that earlier actions must be refined

To support the generation of variations under these three conditions, users must be able to duplicate a document state, create a variation while interacting with a command, and revisit past states of a document without losing the current state. We describe how Parallel Pies supports each activity next.

6.2.1.1 Document Duplication

In our application, users can duplicate the current state by invoking the Duplicate Variation command from a pull-down menu. The entire document state, including its history, is copied and embedded within the same solution workspace. Visualization of the multiple alternatives is handled by the Parallel Pies visualization tool, described below.

6.2.1.2 Adding Variations within Commands

To support the creation of variations while modifying data through a command dialog box, users can insert the currently previewed result as a new variation to the workspace. Invoking the Add Variation command (Figure 18) duplicates the document, applies the command with its current settings, and inserts the result in the solution workspace.

Exploration of variations while manipulating data is further enhanced via Side Views. Side Views presents a broad overview of the set of options available from a particular command, while Parallel Pies enables users to quickly add any and all relevant results to the solution workspace via the Add Variation command.

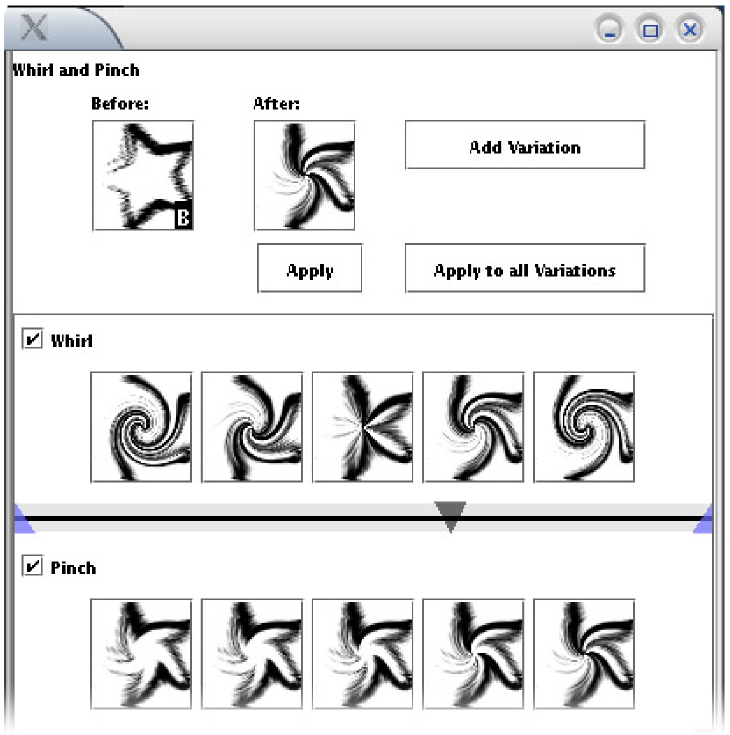


Figure 18. Parallel Pies' additions to dialog boxes. Command dialog boxes include the ability to add a variation or apply the command to all variations.

6.2.1.3 Duplicating Lineages and Skating Through Time

To support the creation of variations after a command has been applied, each variation maintains a history of all prior states and commands leading to its current state. When users duplicate a particular variation, its lineage is also duplicated. By copying the history of a state, users can more easily backtrack to a previous state while retaining the most recent state. However, rather than use the Undo mechanism to return a previous state, we introduce a function called *skating*.

Skating allows users to traverse the timeline of a solution instance independent of actions performed in the interface. For example, consider a scenario where a user has applied three operations to a document, but would like to explore an alternative path from a previous state of the document. In this instance, she may duplicate the document, resulting in two, identical variations, each of which has an identical history of past states. With either variation, she can then skate through its lineage to access previous states. Note that using Undo in this situation would not achieve the desired effect, as Undo would be interpreted as undoing the last action, in this case the Duplicate operation. (This issue is similar to those encountered in the design of Flatland, a whiteboard application that hosts self-contained workspaces that can also interact with one another [EDW00].)

6.2.2 Set Evaluation and Manipulation

One advantage of manually embedding variations within the same solution workspace (for example, two versions of a paragraph, one after the other) is that the alternatives are ready-at-hand: There is no intermediate layer required to load or save them. Instead, they are highly accessible, making basic comparisons straightforward. They also allow other parts of the solution to be manipulated independent of the variations: Changes do not need to be merged or duplicated as they would be if separate document instances were created for each variation in a history tracking system.

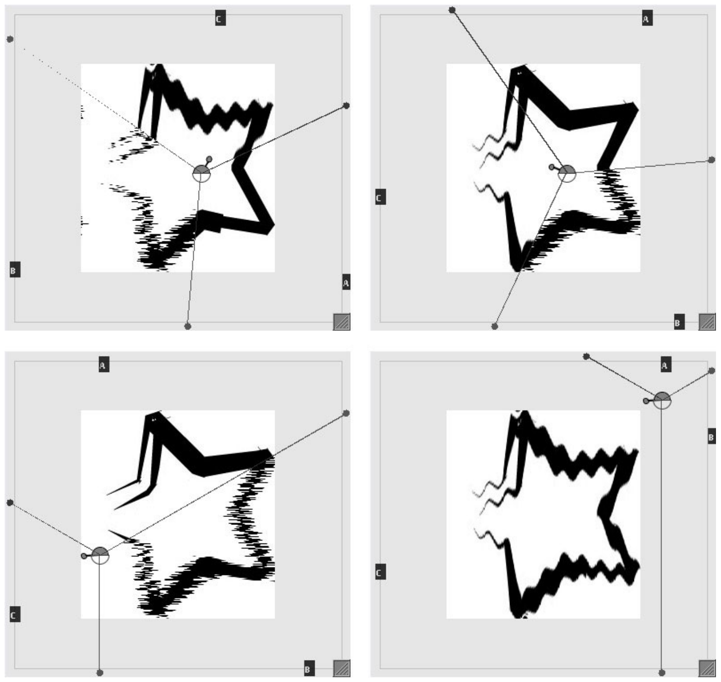


Figure 19. Parallel Pies visualization tool. The central hub of Parallel Pies can be rotated and moved selectively to display and compare separate alternatives.

Building on these concepts, Parallel Pies embeds alternatives directly within the same solution workspace and slices the space to show a different variation in each slice. Wedges radiate outwards from a central hub that can be repositioned and rotated to reveal different areas of the individual variations (Figure 19). A gutter surrounding the image provides space for the hub to be dragged off to the side. When pulled into the gutter, the wedge affords a larger view of a variation, allowing users to focus on only one at a time. Notably, using other schemes to divide the space (such as a grid), would not provide this same affordance.

The Parallel Pies visualization also acts as the mechanism by which users select the default variation when applying commands to only one state at a time. Users select a variation by allocating it the most screen space (for example, by pulling the hub to the side). The choice of this mechanism was driven through user testing that revealed that users expected their actions would be applicable to only the most visible variation.

The visualization provided by Parallel Pies works particularly well when the differences between images are relatively minor. When alternatives are more divergent, the ability to reposition the hub in the gutter helps reduce visual confusion by showing only one variation at a time.

6.2.2.1 Selective Manipulation of One or More Variations

Users can choose to modify a single variation or all variations at once. An Apply button in a command's dialog box affects the most visible variation without dismissing the window, while all variations can be modified at once by pressing the Apply to All Variations button. This capability helps users keep the sets of alternatives in sync.

6.2.3 Distinguishing Between Variations

Moving to a set-based interface with Parallel Pies necessitates a few other changes to the interface. Most obviously, it requires additional feedback to the user so they know what variations they have created and which they will effect when applying a command. Accordingly, we provide a number of cues throughout the interface.

A thumbnail-based summary of all variations is placed on the right side of the document window (Figure 17). As an additional cue, we decorate variations with tags throughout the interface to help users differentiate between them when they are visually similar. Small black boxes with a unique letter are positioned over the variations' thumbnails on the side of the window, in the before view of a preview, and on the edge of variations' slices in the workspace.

6.2.4 Discussion: Design Rationale for Selecting the Default State

In our initial design, users clicked on pie slices to select the default variation in the workspace. While this approach had the advantage that users could directly click on the item to manipulate it, it also had the consequence of adding an extra layer of selections: Users could now select variations as well as individual objects in the document. This was an obvious point of confusion, so we explored alternatives.

The next mechanism we implemented allowed users to select the variation from within the command's dialog box. Buttons in the shape of arrows below the before preview let users cycle between variations. However, users did not readily understand the buttons' functionality, nor could they easily discover this method for switching between variations.

Our current design builds on a behavior that emerged through testing, namely moving the pie's hub to the side to concentrate on one variation at a time. When the interface showed only one variation at a time, users expected that commands would only affect that variation, since others were not visible. Therefore, we adopted this convention and modified commands to update their previews accordingly.

6.2.5 Architecture

Normally, a document is assumed to be the high-level organizational structure for a solution. Parallel Pies extends this concept to create a hierarchical ordering of alternatives within a single entity called a *solution* (Figure 20). In this scheme, what is usually considered a document becomes a *document state* (shown on the right side of Figure 20).

References to these separate document states are contained with *document sets*. Document sets provide the set management infrastructure in a set-based interface by clustering all viable alternatives together. A solution maintains references to the *active document set* (the set of document states currently active) and to the *default document state* (the state that will be modified when applying a command to a single state).

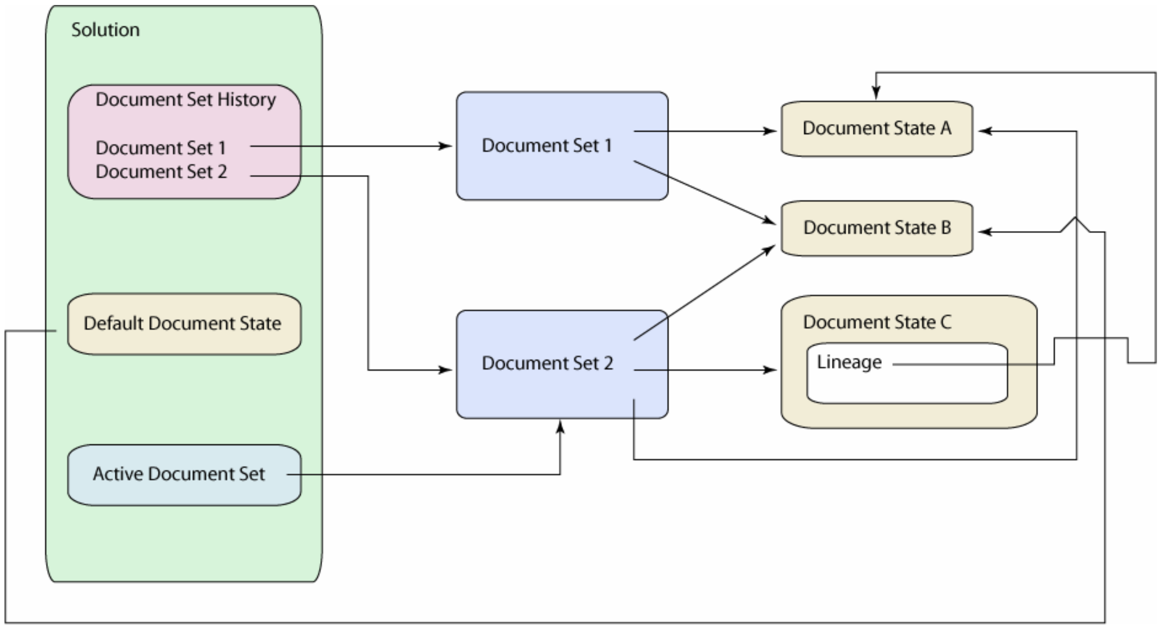


Figure 20. Organizational architecture for Parallel Pies.

When an operation is applied to a solution, it is applied to the default document state. Two new objects result. First, the result of the operation yields a new document state. Second, a new document set is created to contain both the new document state, and any unmodified document states contained in the previously active document set. Thus, a new document set is created each time an operation is performed. These sets form the history of operations to support Undo and Redo, and are maintained in a *document set history*.

When an operation is applied to a document state, the resultant document state also includes a lineage of all states prior to reaching this particular state. This lineage enables a user to duplicate a document state, then traverse its history via skating.

Figure 20 makes these abstract concepts more concrete. In this example, Document Set 1 is composed of references to Document States A and B. However, the user has applied an operation to Document State A, yielding Document State C. While the operation is not shown in the diagram, we can infer that the operation was an Add Variation command because the resultant document set, Document Set 2, includes references to states A, B, and C. That is, state C was added to the solution and did not replace the previous version, A.

Undoing this last operation would reload Document Set 1 and its references. Since a document set is composed of references to document states, the creation of Document Set 2 minimally increases memory in this example: Only Document State C is created; the other two document states (A and B) already exist, and are merely referenced.

Cast in terms of the model-view-controller (MVC) user interface design pattern [KRA88], the solution represents the model, and its corresponding window the view and controller. In our implementation of Parallel Pies, there are two external, visual representations of a solution: the Parallel Pies visualization and the solution window's thumbnails. The latter represent each document state and toggle their visibility in the visualization, making them both views and controllers. Users can also move the visualization tool to selectively dedicate screen real estate to one particular solution at a time. The version with the most screen real estate becomes the default version in the solution.

6.2.6 Discussion: Considerations for Toolkits and Interaction Semantics

One of the greatest challenges in developing Parallel Pies was deciding the granularity of support to offer for creating new alternatives. For example, should the interface support alternatives at the level of the document, the layer, or even individual objects within a layer?

In our current scheme, we duplicate an entire document state when spawning a new standalone alternative. This is perhaps the simplest way to deal with alternatives architecturally, but it is also somewhat wasteful of memory in cases where two or more alternatives share most of their data. For example, in editing a text document, one may devise multiple alternatives for a paragraph, but keep all other content the same. If a word processor enabled one to generate new document states for each paragraph and used our scheme as a model, these alternatives would yield separate, standalone document states with a significant amount of duplicated content.

To be more efficient, this scheme could be enhanced with a copy-on-write scheme in which two or more document states with identical data share that data until it is changed in one of the alternatives. This would allow a developer to continue to work with and reason about alternatives at the level of entire document states, rather than fragments of document states, even though these document states may be internally represented via references to shared data. Therefore, at the toolkit level, we believe the preferable solution is to allow the developer to present the toolkit with a state, the operation to apply to that state, and an indication of whether the operation should replace the existing state or spawn a new standalone alternatively. Internally, the toolkit should then decide, given the operation requested and the state provided, how to derive the new state most efficiently. We elaborate on a potential architecture for realizing this scheme in the final chapter.

A second challenge in the design of these features was deciding how much to transform interaction semantics to account for multiple, co-existent possibilities. For example, should an operation apply to one alternative, all alternatives, or only those that are visible? Should we allow these alternatives to exist in different interaction states? For example, should we allow the user to select a particular region in one image, another region in another image, then allow the same operation to be applied to both (incongruent) selections? Similarly, what should the interface do if the user tries to apply an operation to multiple document states, but the operation is not compatible with all document states?

While our current prototype does not allow document states to assume incompatible interaction states (e.g., one cannot make selections on a per-document-state basis), our experience suggests that the way to handle these ambiguities is very similar to the way interfaces handle this issue when applying commands to multiply-selected objects. For example, visual user interface builders often allow the user to modify the properties of multiple, selected objects (such as buttons, scrollbars, and labels) simultaneously. When there are properties that are not shared (e.g., buttons and labels have a "text" property, but scrollbars do not), the interface does not allow unshared properties to be set. In other words, only those operations which can be applied to all selected objects are available for invocation. This model seems the most reasonable, though there are ways it could be enhanced for the benefit of the user. For example, the interface may preview which document states are compatible with one another with respect to a particular operation before the user attempts to apply the command.

6.3 Summary

This chapter has introduced two tools illustrating the concepts of a set-based interface: Side Views and Parallel Pies. Their specific implementations have been described, as have general architectural considerations.

CHAPTER 7

DESCRIBING AND DISTINGUISHING BETWEEN PROBLEM SOLVING PRACTICES

Up to this point, we have described point- and set-based problem solving practices, interfaces, and interaction in fairly abstract terms. However, if we are to understand whether tools truly influence the problem solving process as desired, we need ways of measuring *how* they influence the process. In this chapter, we synthesize a set of metrics and a visualization designed to help one distinguish between point- and set-based problem solving when using computer-based tools. These techniques are not meant to specify hard boundaries between what constitutes point- and set-based problem solving, but rather are intended to serve as metrics by which comparisons can be made.

We construct these measures and the visualization by first reviewing Ward et al. [WAR95] and Sobek et al.'s [SOB99] original work describing point- and set-based problem solving, then consider prior techniques used to depict branching histories in web browsing and other applications.

7.1 Measures

In the original formulation of set-based concurrent engineering, the following activities most clearly distinguish set-based approaches from point-based:

- Sets of alternatives are simultaneously developed and considered, and
- The alternatives are continually evaluated and pruned in order to narrow the space of possibilities under consideration.

Point-based problem solving, in contrast, is marked by the continual refinement of a single solution instance, leading to a linear solution process with occasional episodes of backtracking.

Given these descriptions, one way to distinguish between approaches in user interaction is to construct a tree (i.e., a directed graph) representing all states visited. In such a tree, each node refers to a particular document state, and arcs between a parent and child node represent the function used to derive the child state from the parent state.

For set-based approaches, trees will appear wide at top and gradually narrow to the final solution. In contrast, point-based approaches will yield, in the extreme, highly linear trees with no branches at all (Figure 21).

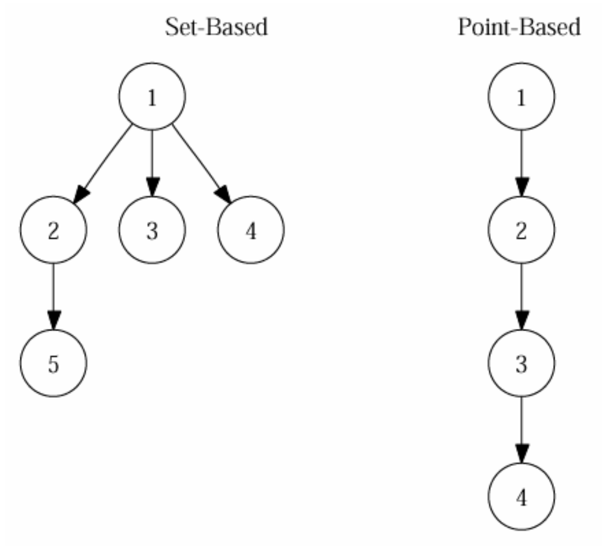


Figure 21. Set- and point-based interaction trees.

These expected differences suggest that the following attributes of the trees are important in distinguishing between practices:

- *Number of leaf nodes*, as a measure of overall breadth of exploration
- *Average number of children per node*, as a measure of breadth of exploration as related to the overall tree size
- *Height of the tree*, (the longest path from the root node to a leaf node) as a measure of overall depth of exploration
- *Total number of nodes*, as a measure of the total number of states considered

The total number of nodes can be divided by the total problem solving time to derive:

- The *problem solving rate*: The number of states visited per a period of time

Within these trees, we can decorate the individual nodes with additional information to help us better distinguish between practices. In particular, the following measures are relevant:

- *Node birth time*: The absolute time the node was first visited/created (e.g., 12 seconds from the start, 30 seconds from the start, etc.)
- *Node birth order*: The relative order in which the node was first visited/created (1st, 2nd, 3rd, etc., derived from an ordering of node birth time)
- *Solution fitness*: The node's judged proximity to an ideal solution
- *Abandonment*: Whether the node was ultimately undone

Arcs within the tree can also be augmented by noting:

- *Command destructiveness*: Whether the resultant state replaced the parent node by way of its creation (i.e., the function was applied to the parent and the child resulted), or whether the child represents a new, standalone state *in addition* to the parent state

A timeline can also be kept to maintain:

- An *active state list*, or the number of states active at any point in time

At first glance, it may seem that some of these measures are somewhat redundant. For example, we use the number of leaf nodes as well as the average number of children per node to indicate breadth of exploration. Figure 22 demonstrates why these multiple perspectives are necessary: Both trees have the same number of leaf nodes, but the tree on the left has a higher number of children per node than the tree on the right.

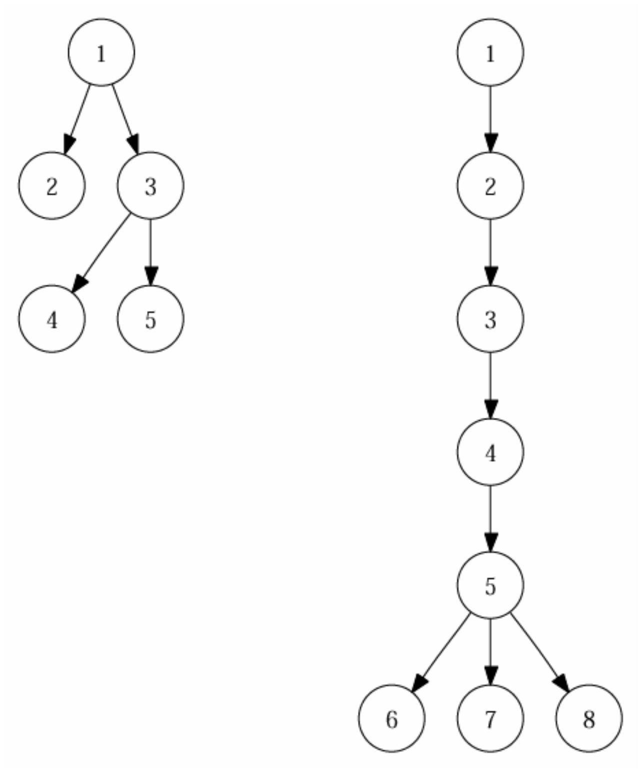


Figure 22. Two trees with different characteristics.

Given these attributes, we can now describe expected differences between trees representing the two different problem solving processes. Across these dimensions, set-based approaches will yield trees with more leaves and a higher average number of children per node. If a constant pace is assumed across problem solving processes, the same number of states (nodes) will be generated across trees. Thus, if the problem solving rate is constant between two trees, but one tree is broader, this implies that it will also be shorter. Set-based approaches will also result in more states being active at any point in time, and fewer instances of backtracking.

Measures of solution quality (fitness) help determine when and where (in a tree) acceptable solutions are developed. In general, set-based approaches should yield good solutions closer to the root node since users more thoroughly explore nearby state spaces. (Whether these solutions should appear earlier in time is unclear.) Set-based approaches should also result in fewer nodes undone, since the expectation is that people are proactively exploring the design space, and not retroactively exploring it after a particular path proves inadequate.

For set-based approaches, sibling nodes should be generated at approximately the same time, one right after the other, as users explore sets of alternatives at a particular point (Figure 21). For point-based approaches, on the other hand, sibling nodes are more likely to be separated in time, since the expectation is that siblings only arise after backtracking (Figure 23), a concept we develop next.

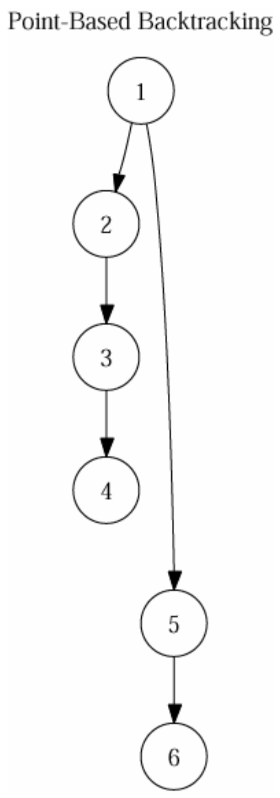


Figure 23. An example of backtracking in point-based problem solving.

7.1.1 Backtracking

Backtracking refers to revisiting a past state to derive a new state. Since a new child is derived from a parent after a previous child's path proves less than adequate, we measure backtracking events based on birth order differences of nodes.

To calculate birth order differences, we first order children of a node according to when they were born, then calculate the differences between adjacent pairs. Successive siblings born one right after the other have a birth order difference of 1, as in Figure 22. Siblings with a birth order greater than 1 indicate that the user first derived the first sibling, turned their attention to deriving other states, then returned to derive a new sibling. Figure 23 provides an example of this where birth order differences between the root nodes' children is 3.

We treat all birth order differences greater than one to be instances of backtracking. This requirement filters out try-undo cycles and alternatives generated via mechanisms such as Parallel Pies' Add Variation command.

We consider the *magnitude* of a backtracking effort to be the difference in birth order between two nodes.

Given these definitions of backtracking, we use the following measures to fully characterize it:

- Total number of backtracking occurrences (number of birth order differences > 1)
- Average magnitude of backtracking (sum(all birth order differences) / number of birth order differences)
- Degree of backtracking as a function of the number of sibling pairs (number of backtracking events / number of sibling pairs in a tree)
- Degree of backtracking as a function of the tree size (number of backtracking events / number of nodes in the tree)

All measures should be self-explanatory, except perhaps the degree of backtracking as a function of sibling pairs. This measure provides an indication of *why* people are constructing alternatives: If this ratio is high, it indicates that when branching occurred in the problem solving process, it was a result of what we define to be backtracking.

7.1.2 Dead-ends

Our definition of backtracking captures any moment when a user returns to a previous state after exploring other states in depth. We can also consider a more extreme version of backtracking in which all previous states are abandoned prior to pursuing a new branch from a common parent. We call these abandoned states *dead-ends*.

Dead-ends can be identified by examining the birth and death times of nodes, where a node's death is the time at which it no longer exists in the interface. If the death times of child's lineage are all less than the birth time of a sibling's lineage, then the former is considered to be a dead-end. As an example, consider Figure 23: If the death times of nodes 2, 3, and 4 all occur before the birth of state 5, then this is an abandoned branch of the tree, and thus a dead-end.

Given these measures of the problem solving process, we turn now to a visualization that highlights these concepts.

7.2 Process Diagram

To visualize one's problem solving process, we build on previous efforts for visualizing website navigation [COC96, AYE99], histories of collaborative editing [EDW97], histories of design points visited [KLE02], and common conventions for drawing tree-based data structures. The overall result is a diagram of a user's actions that we call a *process diagram* (Figure 24).

A process diagram is composed of three interrelated components:

A. A state tree depicting nodes visited
B. An active state timeline, where each entry represents a set of actives states at that point in time
C. A command timeline, indicating every command issued by the user

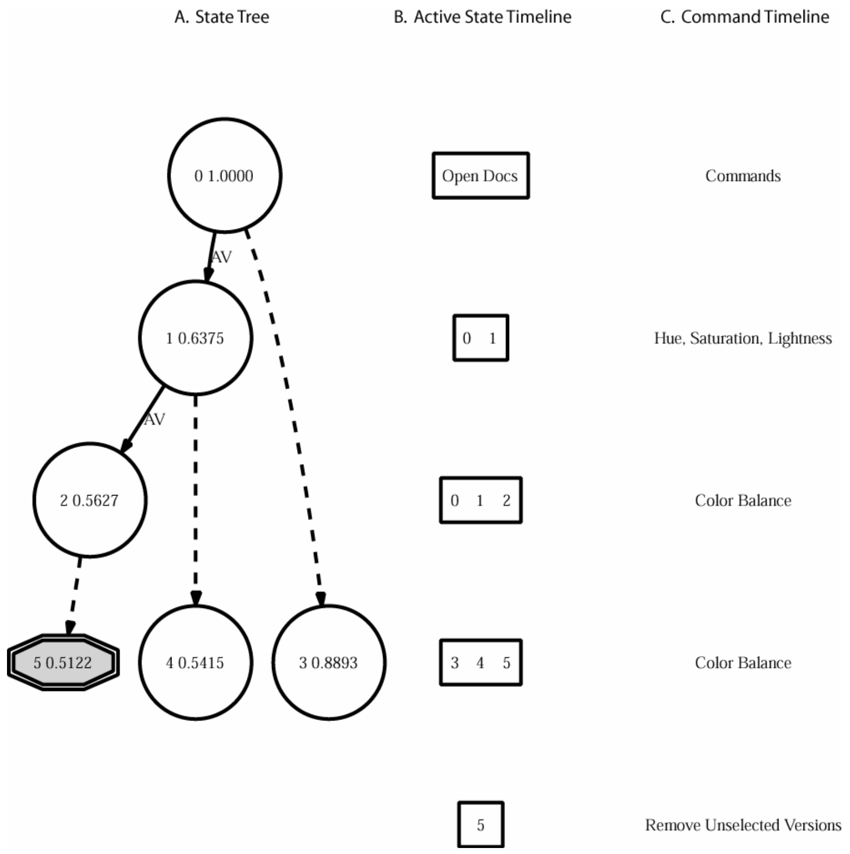


Figure 24. Process diagram.

We describe the various components in detail next, then describe the overall affordances of process diagrams.

Like previous history visualization systems, each unique state is represented by a node. Thus, as a document is modified, each modification results in a new state corresponding to a new node.

Each node contains two numbers, its birth order and a measure of its overall solution quality. For tasks in which the start and ideal solution states are known and quantifiable (that is, solution quality can be quantified in terms of its relation to the start and ideal target state), a normalized value can be used for solution quality. In our work, we use the convention of assigning the start state a normalized value of 1, and the target state a normalized value of 0. Thus, as a solution approaches the target state, its solution quality approaches 0. (This can be conceptualized as the distance of a given state from the ideal solution state. Under this scheme, if a state is worse than the start state, it takes on values greater than 1, and values are non-negative.) The use of normalized values aids in comparisons within and between studies, such as comparing rates of change.

The root node represents the starting point with a birth order of 0 and a solution quality of 1. It is placed at the top of the process diagram.

Active states appear directly to the right of every node in the active state timeline. Since node numbers correspond to nodes' birth order, one can analyze an active state set to see when each state was derived in the overall problem solving process and how long it lived. If more than one instance of a state is currently active (e.g., it has been duplicated), the node is followed by a number in parentheses representing the number of instances of that particular state currently active.

Arcs represent state transitions from parent to child states. The function responsible for this transition is listed in the command timeline on the right.

Every element in all three components is vertically aligned to the same relative timeline, with a top-down, oldest-newest temporal ordering of events. Command timeline entries are aligned to the node that they produce.

If states are derived by invoking a function that replaces the current state with the resultant state (the normal mode of interaction in user interfaces), the parent-child edge is represented using a dashed line. This visual indicator serves to identify when an operation is partially destructive (that is, when it causes the current state to be replaced by the resultant state). If the original state is preserved (for example, the state was first duplicated in some manner before applying the operation), a solid edge is used and labeled according to how the new state was spawned from the parent state. For example, if the parent state was originally duplicated, the edge is labeled with a “D.” If Add Variation was used, it is labeled “AV.” The width of an active state box grows every time a new state is added providing additional confirmation of when new states are generated.

When a state is undone, a dashed edge returns to the parent state. If the state is permanently undone and never returned to (i.e., one does not redo back to it), then the state’s node is depicted using a dashed line (Figure 25). This convention indicates which states and paths have been permanently abandoned by way of Undo.

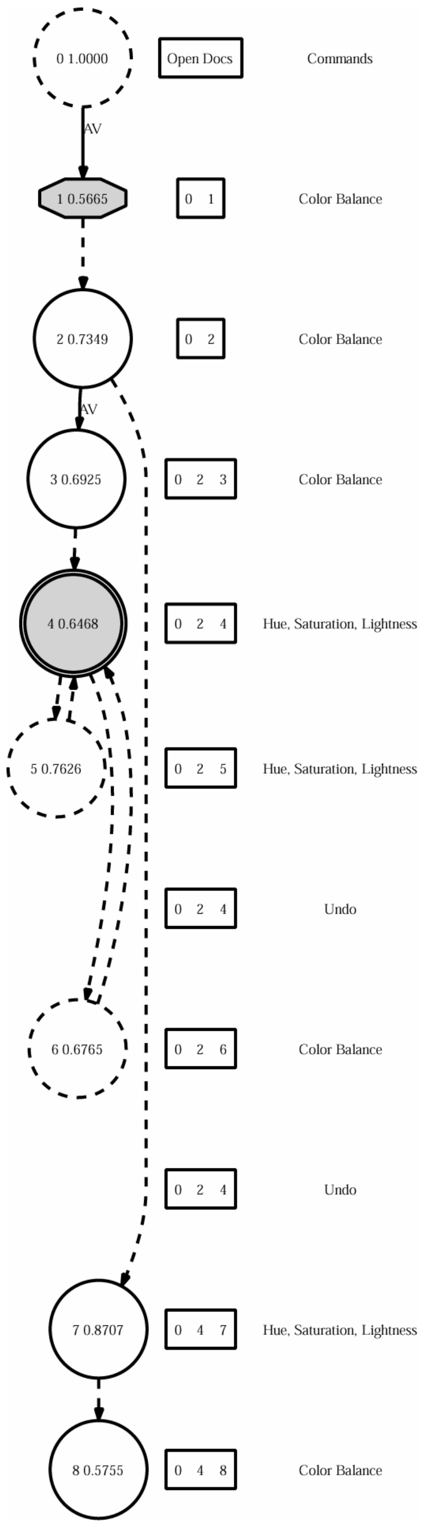


Figure 25. A process diagram illustrating undone nodes and chosen solution states. The octogonal node is the best state visited, while the double-ringed gray state is the state chosen by the user.

At the conclusion of a task, a user may identify the state which she think best solves the problem. For some tasks, the best solution can also be judged computationally by comparing all states with an ideal solution state. We use two methods to help identify and distinguish between these two node types. User-selected solutions are highlighted and include a double border, while computationally-chosen nodes are also highlighted, but assume an octagonal shape rather than a circle (Figure 25). If the two chosen states coincide (that is, both the human and the computer choose the same state), it becomes a highlighted, double-ringed octagon (Figure 24).

7.2.1 Affordances

Process diagrams are designed to explicitly highlight aspects of the problem solving process that distinguish point- and set-based problem solving practices.

The active state timeline draws attention to the number of possibilities under consideration at any point in time, information not available in traditional tree-based representations of task progression. Thus, the addition or removal of states is clearly visible as the width of an active state set grows or shrinks, respectively. This allows one to obtain, at a glance, a sense of whether people are engaging in point- or set-based problem solving by examining how the active state timeline's elements change over time. In point-based problem solving, their sizes will mostly remain constant, whereas in set-based approaches, the sizes will expand and contract over time.

The use of dashed lines in arcs highlights operations that result in states becoming partially or fully removed from active consideration. This information gives the viewer a sense of how many explorations result in dead-ends, and the degree to which they keep sets of alternatives under active consideration.

Aligning tree nodes' vertical ordering to a timeline has the advantage of exposing when each path is explored. In the context of point- and set-based problem solving, this helps to draw out whether sets of alternatives are explored close to one another in time, or as the result of backtracking. In the former case, when nodes are explored close to one another in time, they cascade evenly, whereas is in the latter case, children nodes are greatly separated in vertical space.

Given these sets of measures and the process diagram visualization, we will apply these analytical tools to the analysis of data from two studies, described next.

7.3 Summary

This chapter has developed a set of measures for aiding in distinguishing between point- and set-based problem solving. The following concepts have been formalized to enable statistical comparisons:

- Breadth of exploration
- Depth of exploration
- Degree and magnitude of backtracking
- Dead-ends
- When high quality solutions are developed
- How many solution alternatives are developed

Furthermore, a visualization called a process diagram has also been introduced to help one visually distinguish between point- and set-based problem solving.

CHAPTER 8

ASSESSING THE IMPACT OF SET-BASED TOOLS ON THE PROBLEM SOLVING PROCESS

In this chapter, we turn to the evaluation of Side Views and Parallel Pies. This analysis considers these tools from a number of perspectives: How they influence the problem solving process, how they affect performance, and how they may be redesigned to better accommodate user needs. Two controlled laboratory studies and one think-aloud study serve as the foundations for these analyses.

We begin with the two controlled laboratory studies. These studies evaluate the impact of the tools under two typical tasks, a tightly constrained color-correction task and a more open-ended problem in which subjects must devise a color scheme for a product. The results of these studies provide strong evidence that users wish to engage in set-based practices and readily do so when tools are available to facilitate this process. Most notably, we will show that the ability to quickly spawn new standalone alternatives via Parallel Pies is the most heavily used and highly rated feature of the tools, having greater impact than the multiple previews of Side Views.

We then turn to a think-aloud study that paired subjects to work collaboratively on the open-ended task of producing a color scheme for a product. This think-aloud design was modeled on constructive interaction [MIY86], and served to reveal people's thought processes as they worked with the tools. These data were largely absent from the controlled studies, providing us with additional information to aid in the interpretation of the quantitative data from the first two studies. The data also point to a number of ways the tools could be improved.

Following the presentation of results from individual studies, we highlight the ways in which the tools differently affect each task. These comparisons suggest the roles the tools play in the problem solving process, and the type of user interface support that is most useful for well-defined optimization tasks versus those that are useful for open-ended, ill-defined tasks.

We conclude this chapter by comparing our results with those from Lunzer and Holbaek's [LUN04] study of a subjunctive interface, then summarize how these tools may be better designed in the future.

8.1 Experimental Design

The central thesis of this work is that interface mechanisms designed to support set-based practices will:

- Result in high quality solutions in less time, with lower cognitive load
- Lead to users developing more alternatives in the same amount of time

We hypothesize that these effects will result from users adopting more set-based practices when these mechanisms are present than when absent.

To investigate these hypotheses in the domain of image manipulation, we needed two tasks that were representative of real-world problems and amenable to measurement (e.g., we would like to be able to precisely measure the quality of a solution developed by a subject). However, these two requirements are somewhat at odds with one another: Representative tasks, in this context, are ill-defined problems, but ill-defined problems lack concrete evaluation criteria, preventing us from precisely measuring solution quality.

Conversely, a task in which the solution can be unambiguously judged as "right or wrong" does not, by definition, represent an ill-defined problem since it has a well-defined goal. Thus, in the strictest definition of an ill-defined problem, we can either have an ill-defined problem with a subjective evaluation metric, or a well-defined problem with objective evaluation criteria.

As a compromise, we can choose tasks with well-defined evaluation criteria, but extremely ill-defined solution paths. Simon, in outlining the differences between well-defined and ill-defined problems, points to chess as an example of this type of problem: While one can readily assess whether a current board configuration results in an end-of-game (e.g., checkmate), evaluation of all other states is rather ill-defined, leading to an ill-defined solution path [SIM73].

In our chosen domain of image manipulation, color correction tasks with known goal states represent a problem that possess these desirable traits. From a given start image, we can successively apply operations to derive a goal state, making the subject's task one of transforming the start state to match the goal state. Because the goal state is known (to both subject and evaluator), we can compute the difference between a subject's solution and the goal state to arrive at a measure of how close a given state is to the goal state. (We later describe how the end state can be derived in such a way that the choice and order of operations are non-obvious, resulting in an ill-defined solution path.)

This color correction task with a known, given target serves as the task for our first study. For our second study, we relax the requirement of matching a known goal state, and instead ask subjects to transform an image until it is indicative of an abstract theme. For example, given a start state, users are asked to create a color scheme indicative of winter. A set of independent judges rate solution quality to provide an overall assessment of how well each subject performs in each task. This second task thus trades the well-defined goal state (and correspondingly well-defined evaluation function) for a task that is more open-ended and subjective in nature. These two tasks are designed to simulate two types of ill-defined problems: those that are relatively constrained (where the overall task can be considered an optimization task), and those that are much more open-ended and subject to interpretation.

From this overview of the tasks, we turn now to the details of the studies' designs. We begin by describing the commonalities between the controlled laboratory studies, then describe features unique to each. We then present the specific research questions the studies are intended to answer, along with our method of answering each question.

8.1.1 Basic Study Design

Both studies employ a within-subjects design. There are four conditions that correspond to the presence or absence of our two tools:

1. Neither tool is available
2. Side Views is available
3. Parallel Pies is available
4. Both Side Views and Parallel Pies are available

The order of conditions follows a Latin square design to systematically vary the order of tool availability. Each subject performs a trial with each condition, for a total of four trials.

Separate from the tool conditions, there are four different tasks, where each task represents a different goal. For example, in the color correction study, there are four different images to color correct. The order of these tasks is varied using a second Latin square. To ensure that pairings of tools and task conditions vary over all subjects, Latin squares were randomly generated with respect to one another.

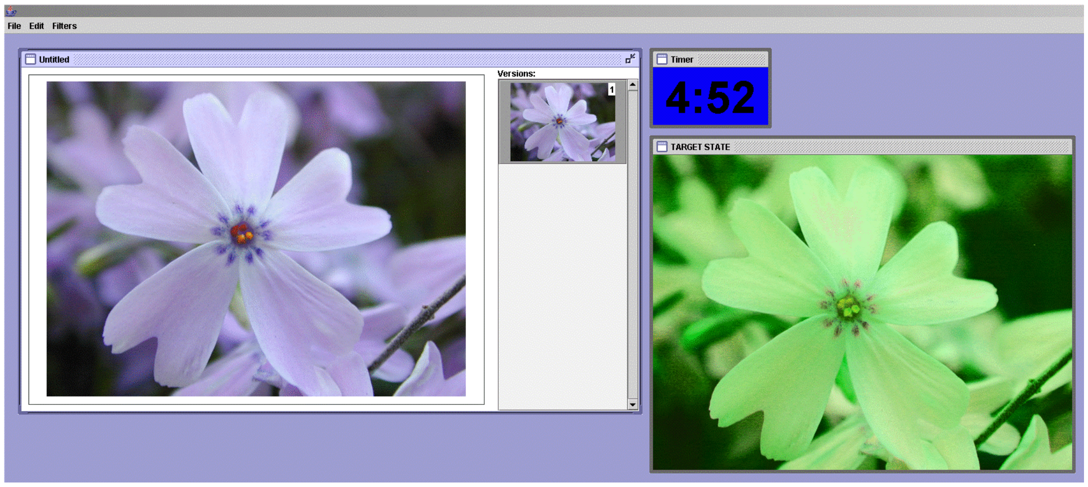


Figure 26. Study task environment (cropped vertically).

Subjects have five minutes for each task in both studies. A countdown timer window (Figure 26) displays the time remaining, and, taking a cue from Time Aura [MAM01], gradually changes color from blue to red as time runs out. The countdown timer window is configured to always appear in the top window layer, so it is always visible, even if other windows occupy the same space. Pilot testing indicated that this timer window was not sufficient to keep people informed about remaining time (since their visual attention was focused on the images), so two sonic alarms were added. A single warning bell sounds one minute before time is up, and a double-bell sounds when 10 seconds remain.

Subjects can end the task early by invoking the End Task command from the File menu. They are prompted to confirm they wish to end the task early. Otherwise, if time runs out, the application window automatically closes, and a dialog box appears to inform them that they will next need to choose their best solution.

8.1.2 Application Design

The application employs the multiple document interface (MDI) [MDI] paradigm common to products such as Adobe Photoshop [ADO]. In this scheme, one application window houses all documents as sub-windows. Using this scheme, we can ensure that all windows remain within our control, rather than under the operating system's control.

A basic set of application services are available to enable people to engage in their common problem solving practices. Documents can be saved to disk via a Save command, a "Save As…" command allows documents to be saved under a new name, and documents can be closed and re-opened. Unlimited Undo and Redo are available, and subjects can duplicate the entire document with a Duplicate command. Selections cannot be made, and data cannot be cut, copied, or pasted.

The application offers three filters that operate holistically on the entire image: Hue, Saturation, and Lightness; Brightness and Contrast; and Color Balance. These three commands provide the only means for modifying image data, and one can only apply them to the entire image. This restricted means of modifying an image was intentionally chosen for methodological reasons: In most image manipulation applications, users have the ability to selectively apply commands by making selections or by using tools that enable their precise application (for example, paint brush-like tools). While these are clearly useful tools for the tasks in both studies, we did not include them for two primary reasons. First and foremost, excluding these tools makes the task inherently more difficult, forcing people to spend more time experimenting with how to achieve desirable results.

Secondly, these tools would provide additional variables to account for when determining the effect our tools have on the problem solving process.

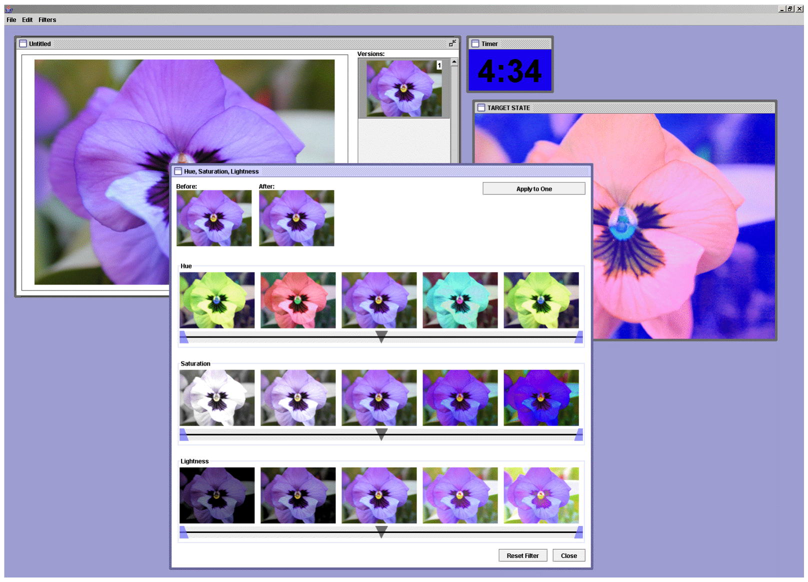


Figure 27. A task environment with Side Views, but no Parallel Pies.

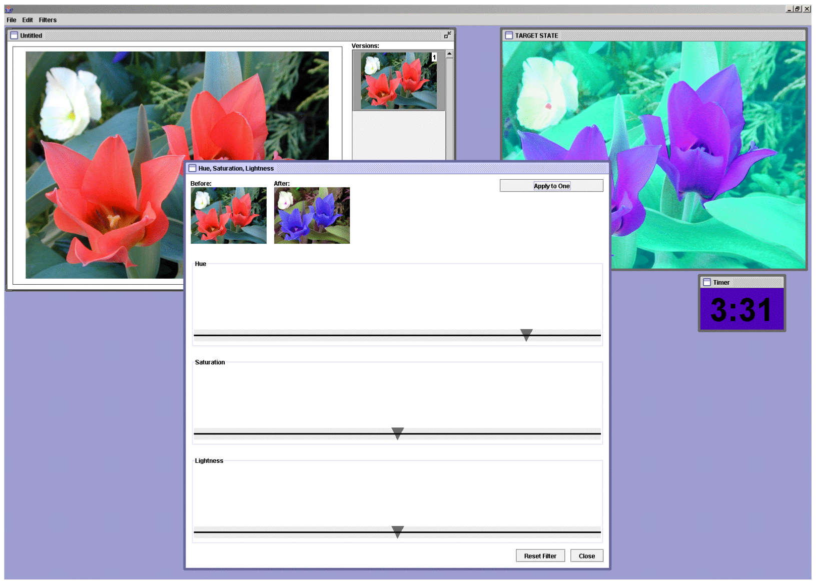


Figure 28. A task environment in which neither Side Views nor Parallel Pies are available.

The three filters are implemented using modeless dialog boxes, meaning more than one dialog box can be open at a time. In all conditions, a before and after preview are available for each command. In the Side Views condition, sets of previews (parameter spectrums) are available for each command parameter. Sliders are also augmented with range handles to allow one to control the range of values previewed by the parameter spectrums. In conditions with no Side Views, the space occupied by the parameter spectrums is blank, and takes up the same amount of space (see Figure 27 and Figure 28 for a comparison). Thus, there is no difference in layout of controls or dialog box size between conditions with and without Side Views. Filter dialog boxes cannot be resized and can only be closed by using the Close button (i.e., no window embellishments are available to close the window, and one cannot use keyboard shortcuts such as the Esc key to close the filter dialog box).

Response time of filters is controlled for in all conditions. Sets of previews take more time to produce than a single preview, so to control for this time difference, previews for *all* parameter spectrums are *always* rendered, regardless of whether they are displayed or not. Additionally, white rectangles are rendered to the window in conditions without Side Views to ensure constant rendering time.

When the active document changes, a filter's previews update to show new before and after previews based on the new document. Because multiple dialog boxes can be open simultaneously (since they are modeless), we again control for preview production time by rendering previews for *all* commands, regardless of whether they are visible or not. Thus, whenever a document changes, before, after, and Side View previews are rendered for all filters, no matter the experimental condition.

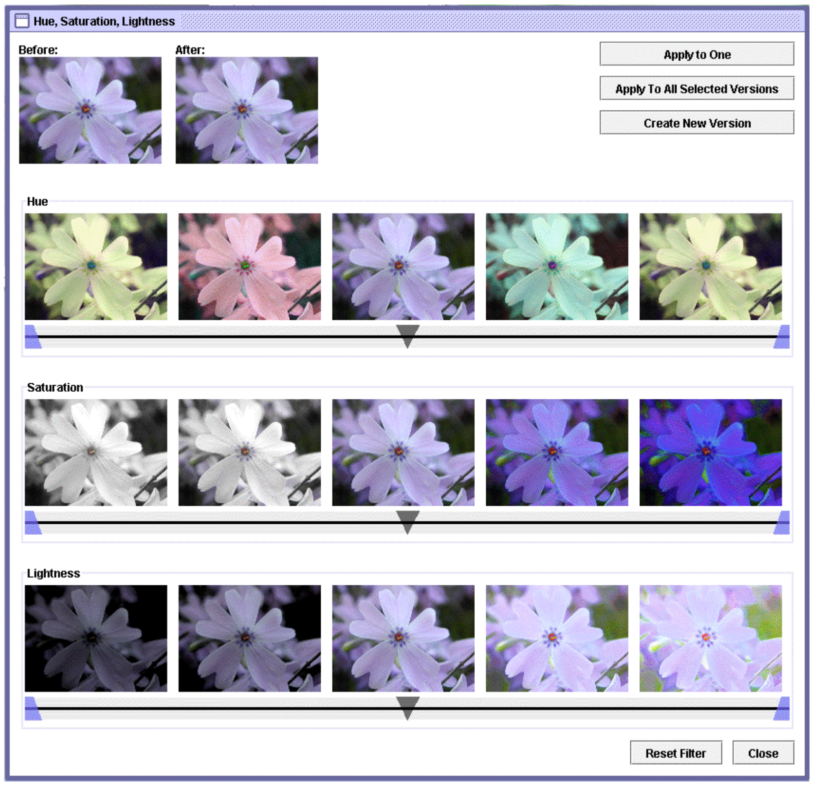


Figure 29. Filter with Apply to All Selected Versions and Create New Version buttons.

In conditions in which Parallel Pies' features are present, two additional buttons are available within the filter dialog box: Apply to All Selected Versions and Create New Version (Figure 29). The Create New Version button[1] performs a number of operations normally performed manually: It duplicates the currently active document state, applies the filter to that copy, and inserts the result into the same document window. The Parallel

[1] As previously mentioned, this button serves the function of the "Add Variation" button described in Chapter 6. In our research, we refer to alternative solutions as "variations" to differentiate from versions, the latter of which are typically associated with previous instances of the same solution in revision control systems. However, for the study, we felt that the phrase "Create New Version" would better convey the function of the command to the subjects.

Pies visualization automatically splits to show both the original and new version, and a new thumbnail is placed in the side of the document window.

In all conditions, thumbnails of every embedded document state are present in the side of the document window (that is, even when Parallel Pies is not available in the interface, a thumbnail of the current state is still present in the document window). The thumbnails are toggle buttons that allow one to turn the visibility of a particular state on or off. If only one version is visible, the toggle button cannot be deactivated, ensuring that at least one document version is always visible.

Like filter dialog boxes, document windows cannot be closed via a keyboard shortcut or through any window embellishments. Instead, one must choose Close from the File menu. However, unlike filter dialog boxes, document windows *can* be resized, though they cannot be minimized or maximized.

No keyboard shortcuts are available for any commands, requiring a mouse to initiate any action (the one exception being that users can type file names in file dialog boxes). This restriction was intended to control for some people's tendency to use keyboard shortcuts to streamline workflow. An optical mouse was provided for subjects, which they were instructed to use instead of the trackpad on the study's computer.

For each session, a unique folder is created on the file system to contain all subject-generated files. Within this folder, four additional folders are created, corresponding to each trial. Within each of these four trial-specific folders, a Documents folder is created. This Documents folder serves as the default directory that appears when opening or saving a document. Providing a fresh, new Documents folder as the default folder for every task presents an uncluttered, consistent file system to subjects, preventing them from the need to take time to organize the existing file system for saving or loading files.

The initial state is automatically saved in the Documents folder prior to the task starting. However, this file is not associated with the initial active document window (even though the initial window represents this start state). This design ensures a back-up of the original start state is always present for the subject's use should she mistakenly close all document windows without first saving any of her work. Subjects are instructed that this file is available should they wish to return to the start state at some point during the trial.

Documents with multiple states are saved as one file. Thus, the entire state of the document window is saved and available when re-opened.

As mentioned, subjects are allotted five minutes to perform each task. At the conclusion of the task, subjects are presented with a list of all active, open states, as well as any states that have been saved in files (Figure 30). They are given an unlimited amount of time to choose their best solution.

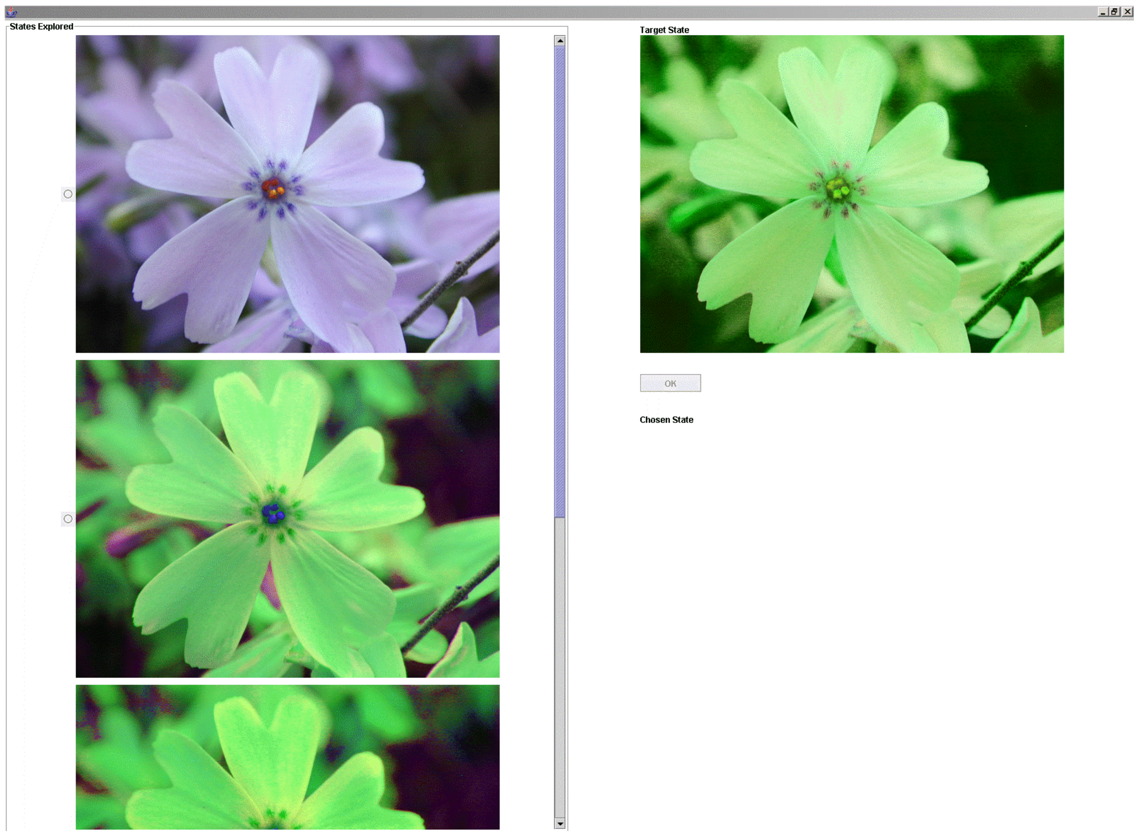


Figure 30. Image selection screen.

All command and control usage are logged and timestamped via custom, internal logging routines within the application. These data enable us to recreate every state visited, and know when each command or user interface mechanism was used.

The software is implemented in Java, with Intel's IPP library [IPP] providing the back-end for image manipulation operations. JIPP [JIP] provides the Java-to-C interface. Custom C-based functions were implemented for Brightness/Contrast and Hue (but not Saturation/Lightness) since these functions are not directly supported by IPP.

All experiments were conducted on the same computer with no other applications running. The computer was a Pentium IV 1.6GHz laptop with 512MB of RAM. Java

JDK 1.4.2_06 was the VM used. With this configuration, preview generation time was highly responsive (no subjects complained of slow response times).

8.1.3 Experimental Procedure

Subjects were recruited through email advertisements to architecture, industrial design, and digital media programs at a university, and by word-of-mouth through contacts at a local newspaper and local design firms. The advertisement requested individuals who were proficient in using image manipulation software.

The first study offered $5 compensation to participants, with the top performer being awarded an additional $75. We found this base compensation too low, so for the second study, we raised the base compensation to $15, with the top performer receiving an additional $50. 24 subjects were recruited for the color correction study, and 10 for the color scheme study.

After providing consent, subjects filled out a questionnaire to assess their experience in the visual arts and design disciplines. Separate questions gauged their experience in these domains with both traditional and computer-based media. A series of questions also inquired about experience with specific software applications such as Adobe Photoshop.

After finishing the questionnaire, subjects were shown a training video to acquaint them with the software and the task. In the first study, no written instructions were provided. However, after the first study, it became clear that some information needed to be reinforced outside the video, such as the fact that saved states were also available to choose from at the end of the task. Accordingly, a small set of written instructions were created to accompany the video developed for the second study.

Following the video, subjects began the first of four tasks. Subjects had five minutes for each task and an unlimited amount of time to choose their best solution. After choosing their best solution, they were asked to complete the NASA TLX workload self-assessment [HAR88]. They could start the next task whenever they were ready.

At the conclusion of the four tasks, subjects were asked to fill out an exit questionnaire about the software. This questionnaire is composed of four sections. Sections one and two present 5-point Likert scale questions about using one or many previews, respectively. Section three asks subjects to rate the utility of software features such as Undo or Create New Version on a 5-point scale. Section four gauges subjects' preferences for using tools in solving the tasks.

8.1.4 Task Descriptions

In this section, we describe the tasks specific to each study.

8.1.4.1 Color Correction Study

In the color correction study, four different images of flowers were chosen and cropped to the same dimensions. Four different scripts were developed to transform each flower to a goal state. Scripts were paired with images so that the same script was always applied to the same image. All scripts share an identical set and sequence of operations, though parameter settings vary between scripts. Each filter is applied once, in this order: Color Balance; Hue, Saturation, and Lightness; and Brightness and Contrast. Parameter settings were chosen so that the total RMS (root-mean-square) difference of all parameter settings is roughly equivalent for each task.

The order of operations was determined through the pilot study. During one pilot, the same script was mistakenly applied to each image. However, despite the exact same transformation being applied in all cases, the subject found it difficult: Not only was he not able to detect a pattern, he did not improve his performance from task-to-task. Upon closer inspection, we found that by performing a color balance prior to shifting the hue, we created a very difficult end-state to replicate. We briefly describe why this combination is especially difficult.

Applying color balance first selectively changes the relative proportions of a particular color channel in the image. A subsequent hue operation shifts all colors, creating a color distortion that can no longer be corrected by the color balance filter. Thus, when the subject first views the start and goal states, her inclination invariably is to first shift the hue, but after doing so, portions of the image remain incorrect, since the hue shift was not preceded by the color balance operation (Figure 31). The incorrect elements of the image *cannot* be corrected by applying the color balance *after* the hue shift, because the remaining differences no longer align with the RGB channels of the color balance filter. Consequently, subjects must experiment to fix these problem spots, though few discern that they must back up to the original state to first perform a color balance to truly correct the problem.

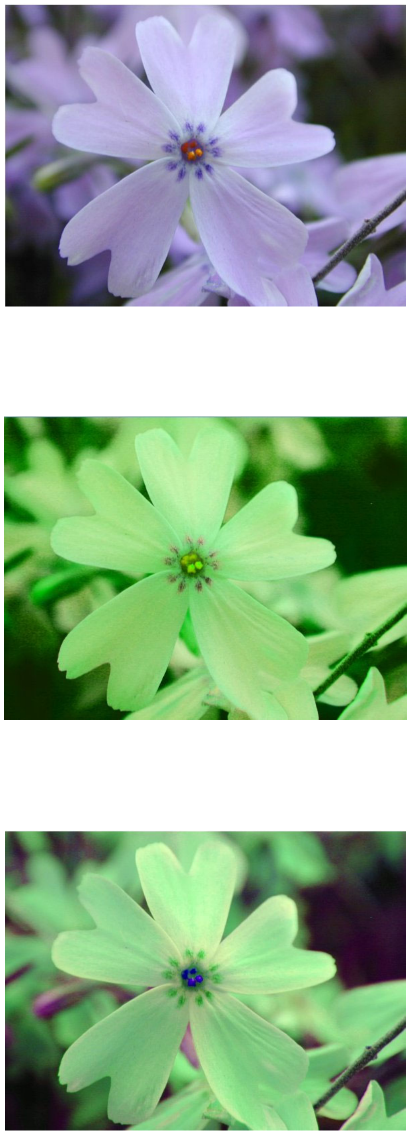


Figure 31. Common approaches to solving the color correction task. Start state (top), target state (middle), and a common first attempt at reaching the target state (bottom). While the overall hue is better matched, interior portions of the flower are off (the middle blue portions in the bottom image), requiring further experimentation.

To assess solution quality, all images are first converted into the LUV color space [FOL96], a color space designed to match the human perceptual system. After converting the images into this color space, an RMS difference is performed between all pixels in the images. We derive RMS differences for every state visited by a subject, in addition to the final solution state chosen by the subject. By computing the RMS difference of every state visited, we can find the best state developed, even if it is not the same as the one chosen by the subject.

8.1.4.2 Color Scheme Study

The task for the color scheme study is to transform the initial color scheme of a watch (Figure 32) to make it evocative of one of the four seasons (winter, spring, summer, and fall). For each of the tasks, the same start state is supplied. Since the task is open to interpretation, there is no target state, but rather, a window that provides a textual description of the desired the target state. All other characteristics of the study software are identical.

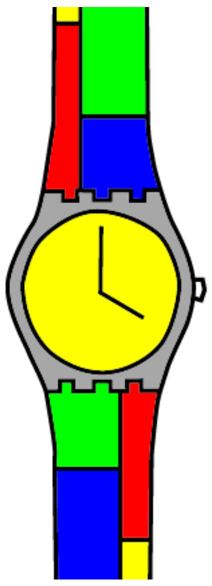


Figure 32. Start state for the color scheme study.

As can be seen in the figure above, the start state for the watch is composed mostly of the primary colors of red, green, and blue, with a yellow face and a few small yellow areas. A neutral gray forms the bevel. The primary colors are pure, having no other color channels mixed in. Thus, when the user adjusts one of the color channels (red, green, or blue) using the color balance filter, it does not affect any of the other primary colors, though yellow is affected when red or green is adjusted.

Assessment of subjects' results was obtained through a set of four independent judges. Two rounds of evaluations were conducted. First, each judge independently gave each solution state a rating from 1-5 based on how well it solved the particular problem. Second, judges collectively rank-ordered the solutions for each season. This second evaluation required the group to come to a consensus on what the best solution was for each task.

8.2 Research Questions

Through these studies, we sought to answer the following research questions, through the following means.

Table 3. Studies' research questions.

Research Question	**Color Correction Study Measure**	**Color Scheme Study Measure**
Do Side Views and/or Parallel Pies yield better solutions?	RMS differences between subject-derived image states and the target state	Independent judges' ratings
Do subjects find better solutions faster?	All states visited are timestamped according to when created, and compared to the goal state	Not applicable
Are desired settings found faster using Side Views?	Distance of slider travel[1]	Distance of slider travel[1]
Do Side Views and/or Parallel Pies affect cognitive load?	NASA TLX self-report	NASA TLX self-report
Do Side Views and/or Parallel Pies affect the overall problem solving process?	Point- and set-based measures (from Chapter 7)	Point- and set-based measures (from Chapter 7)
What are subjects' opinions of the utility of Side Views and Parallel Pies?	Exit questionnaire	Exit questionnaire

These research questions translate into the following set of hypotheses. When an experimental tool is available (Side Views or Parallel Pies), we expect:

1. Subjects' final solutions will be of higher quality

[1] Temporal measures are not used since one cannot cleanly segment when a subject starts and stops considering a parameter value.

2. Subjects will converge upon high quality solutions *faster*
3. Subjects will report lower cognitive load

Additionally, we hypothesize that subjects will rate the tools as useful.

Specific to Side Views, we hypothesize that:

1. Subjects will move parameters' sliders significantly less when Side Views is available

Specific to Parallel Pies, we hypothesize that:

1. Subjects will generate more alternatives when Parallel Pies is available
2. Subjects will explore more broadly when Parallel Pies is available

8.3 Results

8.3.1 Subjects

24 subjects were recruited for the color correction study. 10 of these subjects came back at a later date to participate in the color scheme study. Subjects ranged from students to professionals. Average Photoshop use reported was 10-15 hours per week, indicating a high degree of familiarity with image manipulation tasks.

8.3.2 Results Summary: Both Studies

Table 4 summarizes the overall results with respect to each hypothesis. We provide a brief summary here in addition to the table, then describe individual results in greater detail.

In the color correction study, no significant differences were found in solution quality, but in the color scheme study, Parallel Pies led to solutions that were rated *worse*. No differences were found in the speed with which solutions were developed – good solutions appeared at the same time, no matter the experimental condition. However, Parallel Pies led to more optimal solutions in the color correction study, as measured by the number of operations required to derive the chosen solution state. In the color correction study, no differences were found in any measure of cognitive load for either tool, though the color scheme study did yield results indicating that Parallel Pies increased perceived mental demand while lowering frustration levels.

In both studies, there is considerable evidence that the tools led to subjects adopting strategies more characteristic of set-based problem solving when Parallel Pies was present. Also, in both studies, the availability of Side Views resulted in 50% less slider usage, but otherwise had no *measurable* effect on solution quality or development. Other influences of this tool became apparent through the third study.

We discuss these results in greater detail next and interpret reasons why they arose.

Table 4. Summary of results per each hypothesis.

Hypothesis	Color Correction Study	Color Scheme Study
Subjects' final solutions will be of higher quality when an experimental tool is available	No significant differences were found in the presence of either tool	**Side Views**: No effect on solution quality **Parallel Pies**: Evidence that tool led to **lower**-ranked solutions
Subjects will converge upon high quality solutions faster when an experimental tool is available	**Side Views**: No evidence found by any measure **Parallel Pies**: No differences as measured by time. Evidence for more optimal solutions, as measured by proximity to root node	Not applicable
Subjects will report lower cognitive load when an experimental tool is available	No significant differences found in the presence of either tool	**Side Views**: No effect on cognitive load **Parallel Pies**: Found to increase perceived mental demand, while lowering frustration
Subjects will rate experimental tool as useful	**Side Views**: Highly rated in both studies **Parallel Pies**: Create New Version highly rated. Ability to embed versions in same window also highly rated in both studies	
Subjects will use sliders less when Side Views are available	Subjects moved slider 50% less when Side Views were available in both studies	
Subjects will generate more alternatives when Parallel Pies are available	Overall number of states visited was not significantly different between conditions, in both studies The number of final states to choose from was significantly greater when Parallel Pies was present, in both studies. Thus, we find partial support for this hypothesis	
Subjects will explore more broadly when Parallel Pies are available	By several measures (number of end states, number of leaf nodes in process diagram, and average number of children), subjects explored significantly more broadly in both studies	

8.3.3 Color Correction Study Results

In the first two minutes of the color correction study, subjects were able to make rapid progress towards the solution state, as illustrated in Figure 33. Across the top of the graph, the average performance is shown for each tool condition, while the bottom lines chart the best performances observed across all subjects (better solutions have lower values). After the initial leap towards the goal, the rest of the time was spent on refining the solution. As can be seen from the figure, no subject reached the goal state (which would be noticeable should one of the best performances hit zero on the y-axis), so nearly every subject spent the full five minutes experimenting with potential solutions.

Table 5 presents the results of measures of solution fitness (how well subjects did in solving the problem), measures of their problem solving process, and any statistically significant differences. These data were analyzed using a mixed model analysis of variance controlling for subject, trial number, and the image to color-correct.

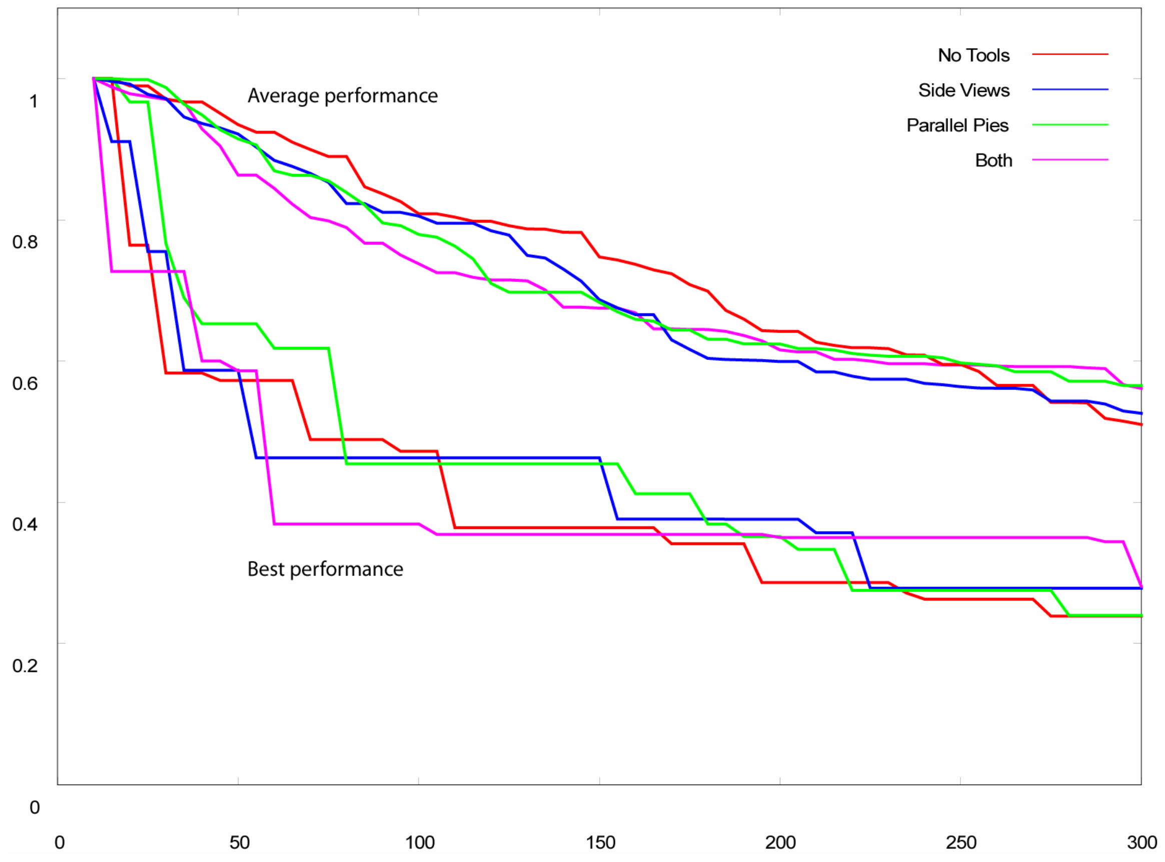


Figure 33. Average and best solution fitness across the four different test conditions. This graph emphasizes the ability to make a quick leap towards the solution state, with the subsequent need to refine. The y-axis corresponds to the distance to the target solution, while the x-axis is time, in seconds.

Table 5. Solution fitness and problem solving process measures (color correction study). All counts represent averages.

Measure	No Side Views	Side Views	Significance	No Parallel Pies	Parallel Pies	Significance
Normalized solution fitness, **subject's chosen state**. Lower numbers are better	0.62	0.68	None	0.64	0.67	None
Normalized solution fitness, **algorithmically chosen best state**. Lower numbers are better	0.58	0.65	None	0.60	0.63	None
Total number of states visited	9.0	10.2	None	9.7	9.4	None
Number of states available to choose from at the end	3.0	2.8	None	**2.1**	**3.6**	**p < 0.0001**
Number of leaf nodes in tree	2.2	2.3	None	**2.0**	**2.5**	**p < 0.05**
Height of tree (in nodes)	7.3	7.8	None	**8.3**	**6.8**	**p < 0.01**
Average number of children per node	1.26	1.22	None	**1.16**	**1.32**	**p < 0.01**
Chosen state birth time, in seconds	229	223	None	**248**	**205**	**p < 0.01**
Chosen state birth order	6.8	7.5	None	7.7	6.6	None
Chosen state depth in tree (number of nodes from root node)	5.5	5.7	None	**6.6**	**4.6**	**p < 0.001**
Best state birth time, in seconds	207	200	None	218	189	None

(Continued on next page…)

Table 5 (continued).

Measure	No Side Views	Side Views	Significance	No Parallel Pies	Parallel Pies	Significance
Best state birth order	5.4	6.3	None	6.0	5.6	None
Best state depth in tree (number of nodes from root node)	4.4	4.7	None	**5.2**	**3.8**	**p < 0.01**
Number of backtracking events	0.4	0.5	None	**0.2**	**0.6**	**p < 0.01**
Number of backtracking events normalized to number of sibling pairs	0.2	0.2	None	0.1	0.3	None
Number of backtracking events normalized to tree size	0.05	0.04	None	**0.02**	**0.06**	**p < 0.01**
Average backtracking magnitude	1.0	1.3	None	0.8	1.4	None

As we can see, neither Side Views nor Parallel Pies had a significant effect on solution quality. This is true whether we consider end states chosen by the subject or by the computer. Similarly, neither tool led subjects to develop better solutions faster, as measured in terms of absolute time or in terms of how many states were visited prior to arriving at the best state, for computer-chosen best states.

That neither solution quality nor speed of development were influenced by the tools was somewhat surprising, since it was expected that Side Views would enable subjects to develop better solutions faster by virtue of displaying a broader range of options simultaneously. One likely explanation for this negative finding is that the application’s commands and their parameters are relatively straightforward, and previews (whether one or many) are rendered fast enough to create a tight feedback loop. Thus, in the absence of multiple previews, one can still quickly find values of interest because of the responsiveness of the user interface. Another potential factor could be that the parameters for the filters are also conceptually orthogonal to one another: It is easy to separate out the effects of hue, saturation, and lightness on an image. If the interactions between parameter settings had been more complex, we might have seen more pronounced effects on solution quality and speed of development related to the presence of Side Views.

Subjects visited the same number of states regardless of condition. On average, 1.5 more states were available to choose from at the conclusion of each task when Parallel Pies was available. In Table 6 below, we see that we can attribute this effect to the Create New Version operation, which was used more often than Duplicate to derive new, active states. These results provide partial support for the hypothesis that Parallel Pies would result in subjects developing more solutions. However, we are hesitant to consider it full support for this hypothesis simply because it is unclear how many of these end states were truly perceived to be viable solution alternatives, and how many served as “back-ups” during the problem solving process (more on creating back-ups below).

There is also considerable evidence that Parallel Pies influenced problem solving strategies, leading to behavior more characteristic of set-based problem solving. Specifically, subjects explored more broadly (as measured by the number of leaf nodes in the process diagrams and by the average number of children per node) and explored less deeply (as measured by the height of the process diagrams). Prototypical process diagrams highlight these differences in Figure 34, which displays the process diagrams for one subject’s four tasks. As mentioned, the differences in problem solving strategies can be attributed to the Create New Version capability, which made it easy to spawn new alternatives. When present, subjects used this capability more often than Duplicate (1.7x more, on average).

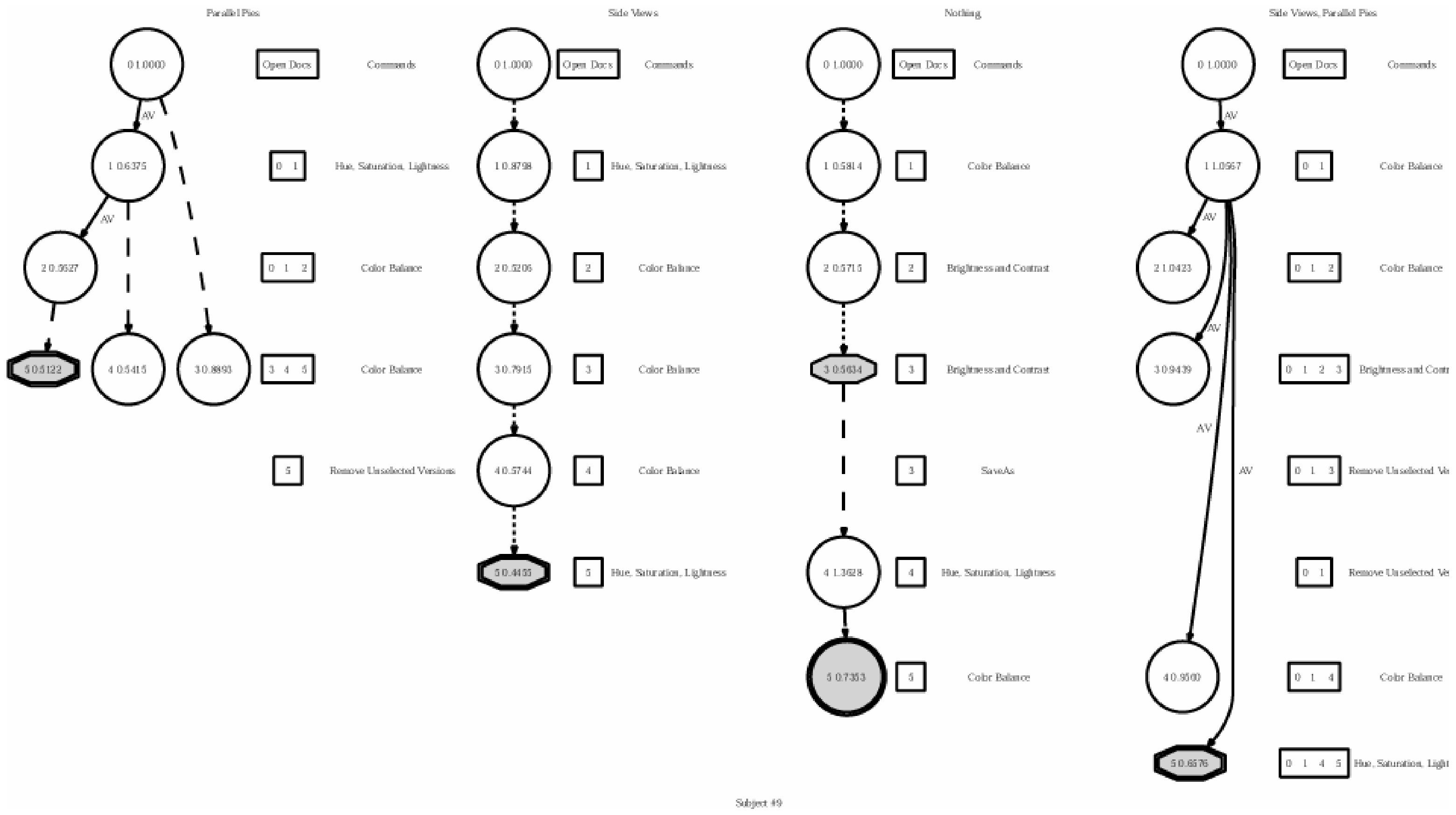


Figure 34. Example process diagrams for the color correction study. Tasks in which Parallel Pies were present yielded broader exploration with more states accumulating (visible in active state timeline).

Create New Version not only made it easier to branch out and explore, but it also made it easier to accumulate possibilities to choose from at the end of the task. When this capability was present, subjects would sometimes still engage in highly linear, point-based problem solving practices with little exploration, but use the Create New Version feature to accumulate all states visited. This strategy may be partially a result of the study's design: Since any open, active state is available to choose from at the end, creating a new version for each new step provides a range of options to choose from at the conclusion of each trial. This practice is visible in the process diagram for the second task in Figure 35.

Because our measures consider several attributes of the subject's tree, we can easily distinguish between this snapshotting use of Parallel Pies and its use to more broadly explore: Snapshotting leads to a high number of active states at the end of the task, but yields trees with few leaf nodes and 1 child per node, on average.

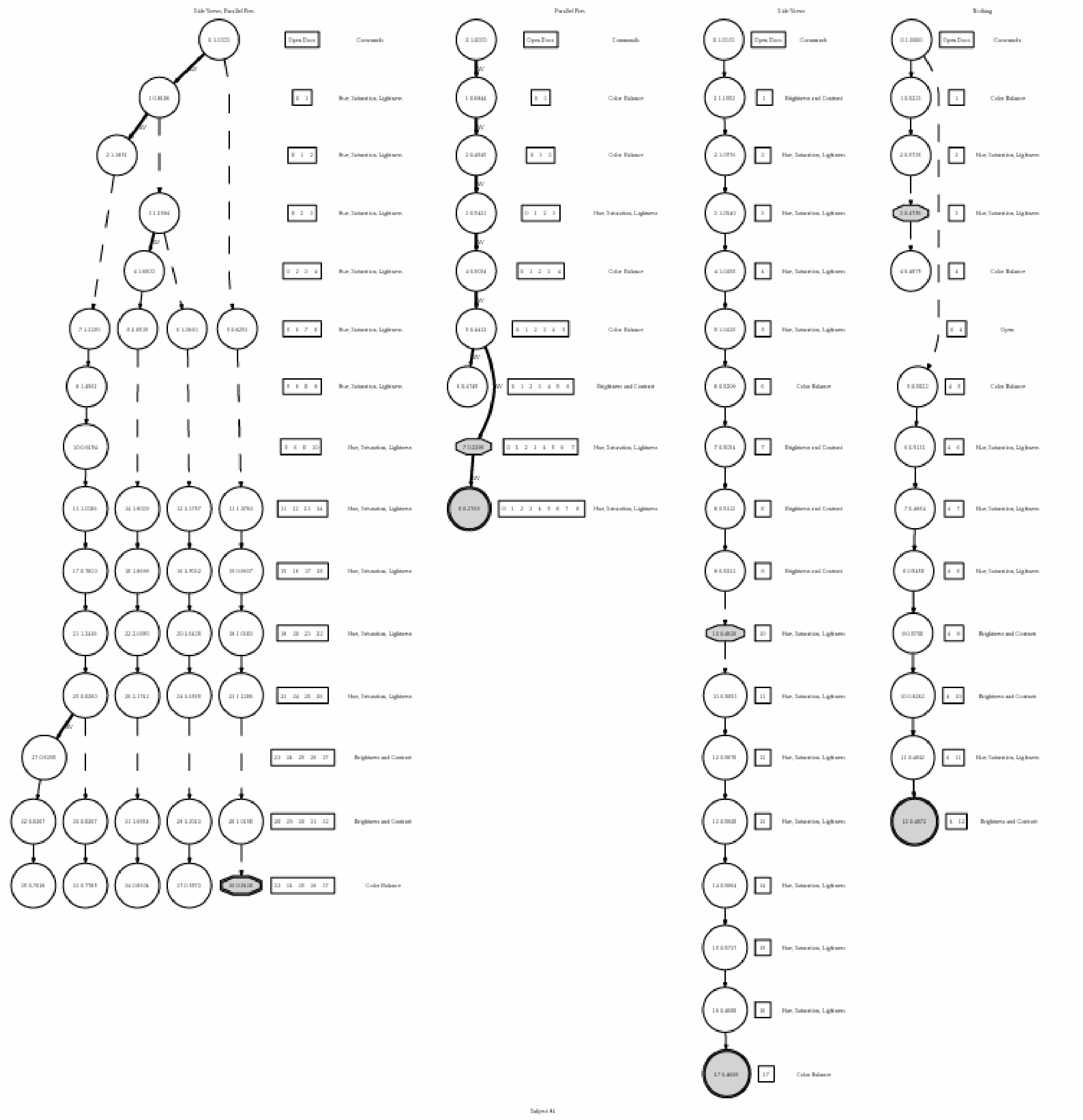


Figure 35. Example illustrating Parallel Pies' use to accumulate variations. In this example, Parallel Pies was present in the first two trials (text can otherwise be ignored in this process diagram). The use of the Apply to All Selected Versions command is visible in the first task (leftmost process diagram), apparent wherever more than one node appears on a row (indicating they were all created at the same time). Use of Parallel Pies as a method to accumulate all states visited is visible in the second task.

When Parallel Pies was present, we found subjects' *chosen* states were created significantly earlier in the task. This should not be interpreted as subjects developing better solutions faster. Rather, when Parallel Pies was used to accumulate states visited, earlier states were more likely to be available at the end of the task. If an optimal state was found early on, and subsequent operations veered away from the ideal solution, subjects could always pick the earlier, better example if it was still available. Figure 36 shows an example of chosen states appearing early in time and closer to the root node in tasks in which Parallel Pies is present.

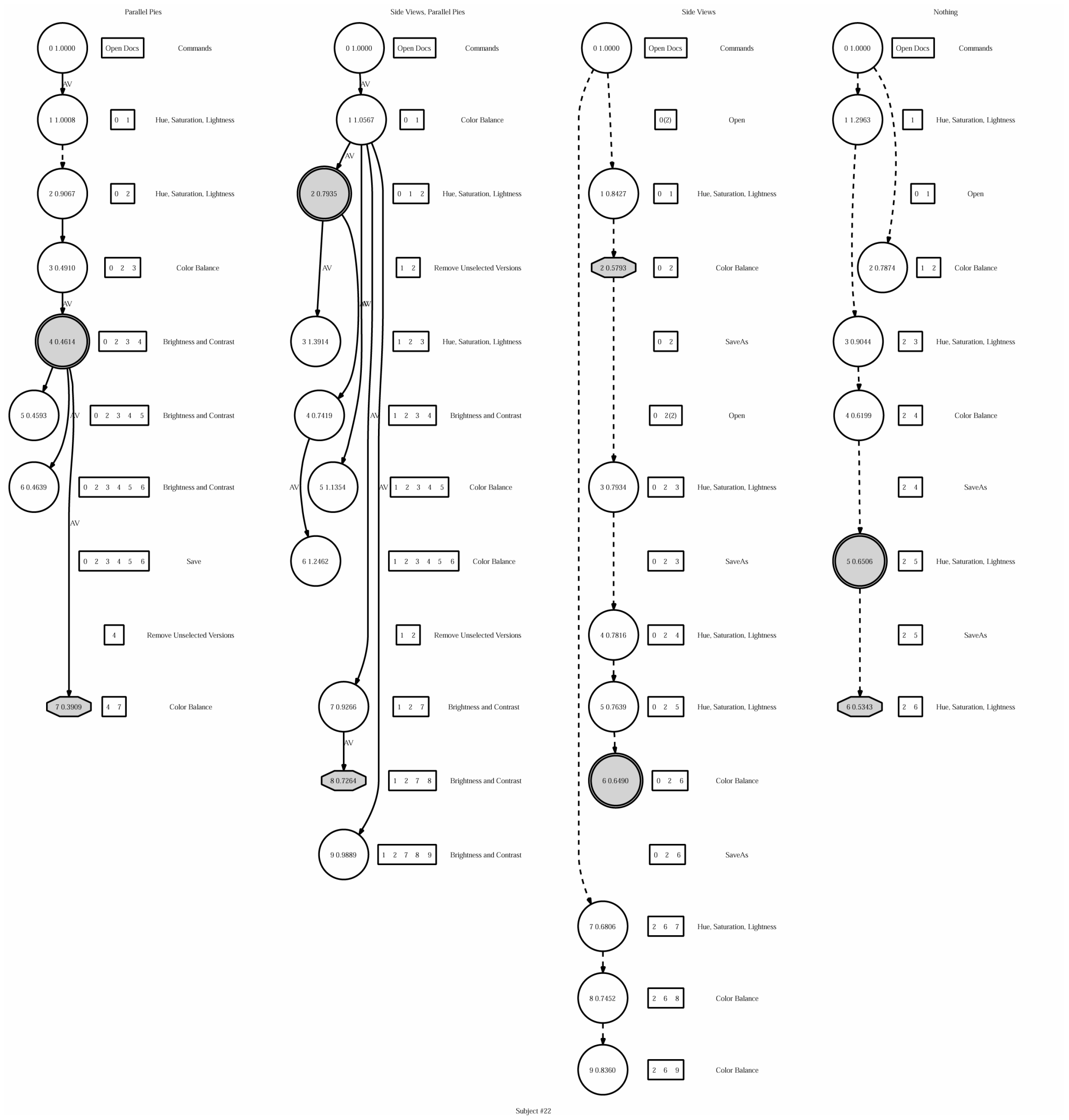


Figure 36. Example illustrating subjects' states chosen earlier when Parallel Pies is present. Parallel Pies was available in the first two trials (text can otherwise be ignored in this process diagram). Branching is clearly visible in Parallel Pie tasks, as is the accumulation of states. This subject also pruned states in the first task (visible via the shrinking active state boxes in the active state timelines). Also noticeable is that chosen states in Parallel Pies conditions are earlier in time and closer to the root node.

In contrast to our expectations, Parallel Pies led to significantly *more* backtracking overall (3x more, on average), though the *magnitude* of backtracking was not statistically different. These data indicate that people revisit past states after a similar amount of time in all cases (since the backtracking magnitudes are statistically equivalent), but they do so more frequently when Parallel Pies is present. This finding is again likely the result of more states being available as a result of the Create New Version command.

Another unexpected finding was subjects deriving more optimal states when using Parallel Pies, where optimality here is measured by proximity to the root node in the state tree. On average, the best states, as determined algorithmically from all states visited, were 1.4 nodes closer to the root node when Parallel Pies was present (meaning 1.4 fewer operations were required to get to the best state from the root node). This finding is noteworthy because it suggests that the broader exploration ultimately led to better solutions, as measured by the fewest number of operations required.

While we did find that Parallel Pies had some effect on the location of high quality solutions within the process diagram, we found no effect on solution quality or speed of development. However, it *could* have had an effect in this particular task: Because one must first perform a color balance prior to shifting the image's hue, the optimal way to approach this task is to generate a number of standalone alternatives, each with a different color balance applied to them. From these, one can then experiment with hue shifts to discover which of the previously generated variations gets closest to an optimal solution. However, no subjects discovered this strategy, leading to no significant effects of Parallel Pies on solution quality.

Table 6. Command usage (color correction study). All counts represent averages.

Measure	No Side Views	Side Views	Significance	No Parallel Pies	Parallel Pies	Significance
Number of undos	0.9	1.0	None	1.2	0.8	None
Number of redos	0.0	0.0	None	0.0	0.0	None
Number of duplicate operations	0.7	0.7	None	**1.0**	**0.4**	**p < 0.001**
Number of Save As operations	0.2	0.2	None	0.2	0.1	None
Slider distance traveled, in slider units	**2695**	**5847**	**p < 0.0001**	3909	4648	None

There were no measurable differences in the use of Undo, Redo, Save, or Save As operations across conditions. Apply to All Variations was used less than 10% of the time, suggesting subjects found little need to develop sets of alternatives in parallel. (However, the first trial shown in Figure 36 represents an instance where the Apply To All Variations command was used.)

While Side Views had no discernable effect on solution quality, it did have a significant effect on slider usage. When present, Side Views resulted in 46% less slider movement, on average. This effect can be accounted for by the fact that subjects did not need to move the slider in the wrong direction before discovering the correct area to move towards.

In this study, no significant differences were found in overall cognitive load, nor any of its constituent dimensions, for any condition in this study. We had hypothesized (and hoped) that these tools would result in lower cognitive loads, but were also concerned that they could increase cognitive load by virtue of adding more information to contemplate at any point in time. That they did not decrease cognitive load may again be partially due to the nature of the task and the operations available. For example, if more sophisticated filters were available (and required), we might have noticed a drop in reported cognitive load in the presence of Side Views.

Table 7. Rated utility of interface mechanisms (color correction study).

Feature	Useless (%)	Neutral (%)	Useful (%)
Undo	17	8	75
Save As	38	21	42
Multiple versions in same window	5	21	75
Just one preview	13	46	42
Create New Version	4	0	96
Multiple previews	8	13	79

Table 7 presents ratings of the utility of various features of the interface (values are aggregated into three groups). The Create New Version feature stands out as a highly rated feature, which is understandable given the effect it had on the problem solving process. Multiple previews, multiple versions embedded within the same window, and Undo were also all rated fairly highly.

8.3.4 Color Scheme Study

In contrast to the color correction study, this study does not allow us to objectively measure solution quality. Instead, we must rely on human judges. Because of the time it takes to rate solutions, and the number of states each subject typically visits, judges rated only subjects' *chosen* solutions, and not every state they visited. Consequently, we cannot consider the quality of states as a function of time, nor can we consider where the best solution state existed in the state trees. However, all other measures are applicable.

Table 8. Solution fitness and problem solving process measures (color scheme study). All counts represent averages.

Measure	No Side Views	Side Views	Significance	No Parallel Pies	Parallel Pies	Significance
Judged solution quality. Lower numbers are better	0.48	0.47	None	**0.43**	**0.53**	**$p < 0.05$**
Total number of states visited	9.5	9.6	None	9.7	9.5	None
Number of states available to choose from at the end	4.7	4.6	None	**3.4**	**5.9**	**$p < 0.0001$**
Number of leaf nodes in tree	2.0	2.2	None	**1.6**	**2.6**	**$p < 0.001$**
Height of tree	7.9	7.9	None	8.6	7.2	None
Average number of children per node	1.1	1.3	None	**1.1**	**1.3**	**$p < 0.05$**
Chosen state birth time, in seconds	232	186	None	226	193	None
Chosen state birth order	6.9	5.6	None	6.8	5.6	None
Chosen state depth in tree, as measured from root node	6.8	4.8	None	**6.3**	**4.3**	**$p < 0.05$**
Number of backtracking events	0.4	0.3	None	**0.2**	**0.6**	**$p < 0.01$**

(Continued on next page…)

Table 8 (continued).

Measure	**No Side Views**	**Side Views**	**Significance**	**No Parallel Pies**	**Parallel Pies**	**Significance**
Number of backtracking events normalized to number of sibling pairs	0.2	0.2	None	0.1	0.2	None
Number of backtracking events normalized to tree size	0.04	0.03	None	**0.02**	**0.05**	**p < 0.05**
Average backtracking magnitude	1.0	0.7	None	0.5	1.2	None

In contrast to the previous study, this study provides evidence that Parallel Pies can significantly affect solution quality, though not in the way expected. *Lower rated solutions resulted*, on average, when it was present. The data suggest that subjects spent more time exploring alternatives in breadth, and less time maturing a single solution in depth. In essence, the ability to spawn new alternatives was welcomed, but *overused*. Results from the follow-up think-aloud study (described in next section) provide evidence for these interpretations and further suggests that, in time, people will adjust their workflow to better integrate the tools into their everyday practices.

As in the previous study, subjects kept a similar pace, visiting the same number of states regardless of experimental condition, though, like the previous study, the number of end states available was significantly higher when Parallel Pies was present.

In contrast to the previous study, subjects' chosen solutions *were not* developed earlier in absolute time nor in terms of birth order when Parallel Pies was available.

These differences are likely a result of the differences in tasks: In the color correction task, one can quickly make a few changes to get a state closer to the target state (Figure 33). In the color scheme task, in contrast, it takes several operations and more time to explore before acceptable results can be found. Thus, acceptable solutions are likely to be discovered later in time.

Set-based strategies were again apparent in the results of this study: State trees were broader (as measured by the average number of children per node and the number of leaf nodes) but not significantly shorter. Subjects also heavily used the Create New Version feature as a mechanism to accumulate potential states. Thus, there were, on average, 1.7x more states to choose from in comparison to the color correction study.

As in the color correction study, we see about three times as much backtracking, overall, when Parallel Pies is present.

Table 9. Command usage (color scheme study). All counts represent averages.

Measure	No Side Views	Side Views	Significance	No Parallel Pies	Parallel Pies	Significance
Number of undos	0.2	0.3	None	0.4	0.1	None
Number of redos	0	0	None	0	0	None
Number of duplicates	1.5	1.5	None	**2.2**	**0.8**	**$p < 0.01$**
Number of Save As operations	0.2	0.1	None	**0.3**	**0.0**	**$p < 0.05$**
Slider distance traveled	**4284**	**9735**	**$p < 0.0001$**	6679	7340	None

As in the previous study, no significant effect of Side Views was found on solution quality, though subjects again used sliders less when Side Views was present (44% less, in this case). The Apply to All Variations was used less than 2% of the time in this study.

Table 10. Cognitive load (color scheme study). All values represent averages for the NASA TLX scales, with each measure out of a total of 100.

Measure	No Side Views	Side Views Available	Significance	No Parallel Pies	Parallel Pies Available	Significance
Total Cognitive Load	42	44	None	42	45	None
Mental Demand	25	28	None	**21**	**32**	**p < 0.05**
Physical Demand	11	13	None	11	13	None
Temporal Demand	18	27	None	20	25	None
Performance	22	21	None	18	24	None
Effort	32	30		32	30	None
Frustration	19	17	None	**24**	**12**	**p < 0.05**

In this study, an effect on some dimensions of cognitive load *was* observed due to Parallel Pies. Mental demand was rated higher, while frustration was found to be lower. Given the data collected, these results can be explained by Parallel Pies placing a higher mental demand on users as the number of embedded alternatives increases in the same document window. At the same time, the increased number of alternatives *lowers* frustration (described on the survey as being insecure or discouraged), since more alternatives are available to choose from at the conclusion of this task. This interpretation is not necessarily in conflict with the lack of similar findings in the previous study, for a number of reasons. First, in this color scheme study, subjects created more standalone, active alternatives than in the color correction study. Second, the nature of the task places different demands on the subject: In the color correction study, there was one goal to focus on, while in this study, the subject could continually reinterpret what their ideal solution may be. As such, they could potentially consider more goal states in their head than in the color correction study, which had a single visible goal. Our findings from the think-aloud study lend weight to this interpretation.

Table 11. Rated utility of interface mechanisms (color scheme study).

Feature	Useless (%)	Neutral (%)	Useful (%)
Undo	20	30	50
Save As	30	40	30
Multiple versions in same window	20	0	80
Just one preview	10	30	60
Create New Version	20	0	80
Multiple previews	0	0	100

We notice less distinction between the rated utility of the various features in this study, though the experimental features are again highly rated. Across the studies, we conclude that the experimental features are highly desirable, and consistently more so than current offerings.

8.4 Think-Aloud Study

Following data collection and analysis in the two controlled studies, several questions arose. First, we saw no measurable effect of Side Views on the problem solving process, despite it being highly rated (and leading to less slider usage). What, then, was its perceived value, if any? Second, what could explain Parallel Pies leading to lower-ranked solutions in the color scheme study? Finally, did subjects perceive changes in their problem solving strategies due to tool availability? That is, did they consciously modify their problem solving process?

To answer these questions, we conducted a think-aloud study using the watch color scheme problem as the task. In this study's design, inspired by a similar methodology employed by Miyake [MIY86], a pair of subjects work collaboratively on the same computer to develop the color schemes; one individual operates the computer for the first two tasks, then the subjects switch for the last two. Audio is recorded and subjects are encouraged to think aloud as they work.

Two pairs of subjects were recruited for this study. One pair of subjects had previously participated in the laboratory studies, while the other pair had assisted in piloting the laboratory studies. Subjects were not compensated.

The same study application was used as in the color scheme study, but time limits were extended to 10 minutes per task to allow subjects more time to develop solutions. No questionnaires nor workload assessments were administered. However, we conducted a semi-structured interview after all four tasks were completed.

In this study, the value of Side Views became apparent. Specifically, Side Views led to the following three behaviors:

1. Subjects quickly eliminating large portions of the solution space
2. Subjects serendipitously discovering viable alternatives, and
3. Use as a coordination mechanism when communicating.

We discuss each of these findings in turn.

On several occasions, subjects opened several filter windows at once, then, after reviewing the sets of previews for a particular filter, completely removed the filter from consideration. For example, in one trial, the Brightness and Contrast filter was opened, adjusted somewhat, but then closed after a subject said that the results were all "mud" and thus not worth considering. In this way, Side Views allowed subjects to contemplate a set of options then quickly dismiss an entire *class* of possibilities deemed unacceptable. In cases when Side Views was not available, subjects were not afforded this luxury and often asked each other to "see what happens if" a particular parameter is moved in a certain way. These findings indicate that Side Views was being used by subjects in the ways intended (i.e., to guide them in finding values of interest), and that its lack of apparent effect in the laboratory studies was likely due to subjects being able to compensate through quick, iterative cycles with the single preview and sliders.

We also found evidence of Side Views leading to the serendipitous discovery of new possibilities, a phenomenon our design intended to facilitate. On occasion, changes in one parameter led to unexpected changes in the previews of other parameters, leading subjects to shout and point at surprising results. These findings ultimately added to the repertoire of potential solutions under consideration.

Side Views also served as a task coordination tool in this cooperative task. In working through the problem, subjects described, in abstract terms, the effect they wanted, but then quickly grounded their thoughts by pointing to examples generated by Side Views. In its absence, subjects expressed what they wanted, but then needed to suggest ways they might be able to achieve the result. In interviews, subjects spontaneously reported using Side Views in this way (as a communication vehicle), and suggested it could also help ground conversations when consulting remotely, if both parties could see the same sets of previews simultaneously. This use of Side Views – as a mechanism to support collocated or remote collaboration – was unexpected, but suggests additional ways this tool could be extended and applied.

One area of improvement for Side Views became apparent in this study. In particular, subjects often narrowed the range of values shown in a parameter spectrum, but then forgot that the range of previews shown was a small subset of the overall possibilities. This typically happened after "drilling down" to a smaller range, switching to another filter, then returning to the original (still open) filter. Based on their conversation, it became clear that they thought they were analyzing the full range of possibilities when it was in fact a small subset. In interviews, subjects reported forgetting about the ability to reset the range of values shown, which suggests a need to reconsider this particular design choice.

As in the laboratory studies, Parallel Pies and its features were heavily used and appreciated. Several episodes illustrate that its use coincides with its intended uses, and that it was addressing the needs we had identified earlier (namely, quickly generating and comparing alternatives). However, we also learned that subjects sometimes felt overwhelmed with the amount of information on the screen at one time.

The Create New Version feature was designed to allow people to spawn new alternatives as they are found while using Side Views. There is evidence that subjects used this capability for precisely this purpose, and one case illustrates this well. In one instance *without* Parallel Pies available, a team adjusted parameter settings of a command until they were satisfied. However, they then realized that they wanted to create a new standalone version to hold this new state. At this point they stopped prior to applying the command, and said, "Oh, we need to duplicate the document first." Thus, they were taken off task to prepare the interface to create a new alternative. When Parallel Pies was present, on the other hand, they freely created new alternatives, and when results proved unsatisfactory, they made comments such as "no harm done," switched to another version, and moved on.

The freedom Parallel Pies grants users to broadly explore was a common theme in the interviews. Even when we suggested that this broad exploration could lead to worse solutions (as in the second controlled study), they insisted that it is better to have this large number of possibilities to choose from when solving these types of problems. Further, they indicated that they knowingly modified their problem solving process as a result, but were still learning how to best integrate the tool and its capabilities into their workflow. Thus, we believe that the lower rated solutions in the color scheme study were partially due to this unfamiliarity with the tool and expect that with increased proficiency, the differences in results would disappear.

Despite subjects' expressed desire to have multiple options to choose from, they did report feeling somewhat overwhelmed at times with the number of choices that Side Views and Parallel Pies generated. For example, one subject reported feeling that there was “too much color on the screen at once,” making it difficult for him to focus exclusively on the problem at hand. This finding lends support to our interpretation of the cognitive load results of the color scheme study: Subjects found multiple versions highly desirable for solving the problem (lower frustration), but, at the same time, a bit overwhelming (higher mental demand) because of the multitude of options simultaneously visible. Consequently, alternative schemes such as Elastic Windows [KAN97] may be necessary to help better manage the sets of previews that Side Views generates to avoid clutter.

8.5 Contrasting Results Between Studies

Between studies, we find fairly consistent results: Subjects explore more broadly with Parallel Pies, and Side Views leads to markedly less slider usage when adjusting parameter settings. However, some differences between study results suggest that the tools serve different purposes according to the nature of the task. We summarize salient differences then consider their implications for designing future user interface mechanisms intended to support the problem solving process.

In the color scheme (watch) study, subjects produced approximately 1.5x more solution alternatives than in the color correction task. The use of Parallel Pies also led to increases in perceived mental demand, whereas such increases were not observed in the color correction task. These differences indicate that subjects were attempting to simultaneously contemplate multiple ways of solving the problem in the watch study, while being much more focused on one solution alternative in the color correction study. The greater use of Undo in the color correction study (3x more than for the watch study) further indicates that subjects were not trying to generate sets of alternatives as much as optimize a single solution instance.

These differences point to an important distinction between the prerequisites to solving the problems. In the color scheme (watch) study, subjects had to spend more time defining the *form* of an ideal solution. In the color correction task, the ideal solution was given, so no time was needed to solve this sub-problem. In general, when solving ill-defined problems, one must initially devote time to discovering what forms a desirable solution can take. That is, there is a need to create and define the boundaries and parameters of an acceptable solution. Consequently, individuals explore alternatives in an attempt to define what the goal *can* be. In contrast, when the solution space is much more constrained, either because the problem is well-defined to begin with, or because a target has been established for an ill-defined problem, the problem solving process becomes an optimization task where one attempts to achieve the envisioned solution as best as possible. Across these stages of the problem solving process, some types of user interface support are more appropriate than others.

When one is attempting to establish the boundaries of the solution space for an ill-defined problem, there is less of a need for history tools, because one is attempting to generate sets of viable solutions rather than refine a single solution. Thus, tools that streamline the process of creating new standalone alternatives are most appropriate in these phases of the problem solving process. As it becomes clear what an ideal solution is, iterative refinement of a solution instance becomes essential to successfully reaching this ideal. At these stages, tools that facilitate backtracking or that streamline editing past actions (such as editable histories) are most appropriate.

8.6 Comparison to Other Experimental Findings

Lunzer’s subjunctive interface attempts to facilitate the simultaneous development and evaluation of sets of alternatives in ways similar to the set-based interface, but at the level of allowing individual user interface mechanisms to assume multiple states simultaneously. In a recent pair of studies, he compares performance of his toolset with conventional interface mechanisms in a task involving census data [LUN04]. Despite the differences between his study and ours (most notably, the nature and type of tasks), it is useful to compare study designs and outcomes.

In Lunzer’s study, subjects must answer questions using census data, both with and without use of subjunctive interface mechanisms. He measured solution correctness, time to find the right answer, cognitive load, and user satisfaction with the tools. Notably, the task has a right answer that the subject can easily verify, making it much more well-defined than either of the tasks in our studies.

Like our study, his interface mechanisms did not significantly affect task accuracy, but, unlike our study, his mechanisms did improve task completion time after an interface redesign. He also found his tools lowered cognitive load. However, a detailed analysis of how the tools influenced the overall problem solving process is not given.

The subjunctive interface and set-based interface both attempt to increase the ease with which people can generate and explore alternatives in parallel. Our respective studies confirm that this capability is desired and used, when available. They also provide some evidence that these tools can lower cognitive load under some conditions, though one of the questions that arises is whether cognitive load would be lowered under less stressful conditions (e.g., when time limits are less of a factor).

While both studies suggest the utility of providing capabilities that enable users to explore alternatives, neither research effort has arrived at tool designs that reliably lead to higher quality solutions. Thus, a remaining question is, given that people want these tools, how can they be better designed to truly increase solution quality?

8.7 Improving Side Views and Parallel Pies

In addition to the quantitative data from the controlled laboratory studies, we have also accumulated information suggesting ways of improving the design of Side Views and Parallel Pies. This information comes from early formative evaluations of the tools, the controlled studies, and the think-aloud study. In this section, we summarize the findings from these separate evaluations to develop a list of aspects that should be considered when building similar tools in the future.

8.7.1 Side Views

Our experience building and deploying Side Views suggests the following issues should be considered in future designs:

- Preview sizes
- Quick switching of preview source states when Parallel Pies is used to manage sets of alternatives
- Attention to use of screen real estate
- Avoidance of user interface clutter
- Tri-slider zoom-in/out issues
- Interactive exploration of a range of values

Subjects sometimes complained that previews were too small, and that they would prefer the preview modifying the actual document itself, as Adobe Photoshop does [ADO]. This request could easily be accommodated when only one document state is embedded within the same window, but otherwise breaks the current Parallel Pies model. One potential way around this is to use a device such as a fisheye view technique [FUR86] to temporarily push states other than the default state to the edges when previewing through the document. Or, one could simply hide the other states while actively modifying parameter values for the default state.

The converse of this problem is that subjects sometimes desired to see sets of previews for *all* the alternatives in a Parallel Pies visualization. Side Views does not currently support this, but it could be modified to add an additional, row of previews (similar to the parameter spectrums) where each preview corresponds to an alternative.

Displaying sets of previews necessarily takes additional screen real estate, making Side Views windows compete with the document window. Some subjects lamented the loss of screen real estate, though they admitted to the utility of the multiple previews. To alleviate this tension, it would be worthwhile to consider one more state a Side Views window could be in. In the original design and implementation of Side Views, a before and after preview is initially shown, with all parameters and parameter spectrum previews hidden. Opening up the Side Views reveals all of this additional information. The third state could be an intermediate state that corresponds to current dialog box designs that offer just a single preview and the controls for setting parameter values. This would yield a smaller area for the dialog box, since the space occupied by the parameter spectrums would be freed. Together, these three states would enable quick previews of the function (tool-tips), broad overviews conducive to exploration of the problem space (via parameter spectrums), and the focused refinement of parameter settings with a minimally sized dialog box (traditional dialog box layout).

Users often take advantage of Side Views' modelessness, leading to several filters being open simultaneously. To combat screen clutter, techniques such as Elastic Windows [KAN97] could provide a single window in which to house all Side Views.

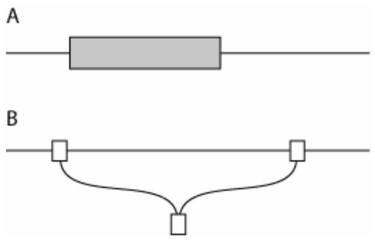


Figure 37. Two slider redesigns that allow scanning of ranges.

Two issues arose with respect to the parameter spectrum's slider and its depiction of the range of values on the control itself. First, users sometimes forgot that it was zoomed in to a very narrow range of values, with their attention focused on the previews themselves. Thus, they did not realize they were seeing only a small portion of the command's capabilities. Second, users expressed a desire to be able to interactively explore a function by moving the entire range at one time. That is, after narrowing down to a particular range, they wanted the ability to then move the entire range across the slider.

A redesign of the slider mechanism may be able to address both issues. Rather than provide three separate parts to the slider, it could be rendered as a single element whose width represents the range displayed. For example, it could be represented as a bar whose edges can be resized to modify the ranges (Figure 37A) or as a wireframe (Figure

37B). In this latter design, both the range's span and its location could be varied by interacting with the center piece (the box that joins the two sides). Vertical movement of the center piece could increase or decrease the range previewed, while horizontal movement could vary the position of the range on the slider. Either of these designs might make the ranges previewed more visible, while enabling scanning of ranges.

8.7.2 Parallel Pies

The primary issues with Parallel Pies' design center around management of the alternatives. In particular, these issues stood out:

- Slice sizes are fixed
- Thumbnail buttons poorly represent document states
- One cannot generate entire sets of new alternatives at one time

Currently, Parallel Pies creates a circular visualization whose wedges all share the same arc angle. As the number of alternatives grows beyond four, this even division of the space makes it increasingly difficult to completely see a single alternative, even if the tool is moved to the far side of the window. At present, the only way to deal with this issue is to turn off alternatives by depressing their corresponding thumbnails.

Subjects expressed a desire to manually modify the arc angle, but a more elegant solution may be to automatically increase the arc angle for the default state (i.e., the one most visible) as the visualization tool is pulled off to the side. As the tool is brought back into the center, it could return to its original, equal division of space.

Another solution is to allow at most four alternatives to be visualized at once within the document window, with no rotation of the visualization tool allowed. With this scheme, users could simply move the center hub to one corner of the document to completely display a particular variation. However, limiting the visualization to at most four alternatives would then require a secondary mechanism to choose which four, in the event that the user generates a large number of alternatives.

At times, subjects had some difficulty understanding whether a particular variation's thumbnail button was active or inactive. Stronger visual cues that do not embellish the scaled-down image, but rather the area around it on the button, would help. (The image's thumbnail cannot be embellished because there is no one scheme that works for *all* images. For example, if disabled variations are represented by being grayed out, this scheme would not be particularly effective for images that are largely gray.) Thus, if future designs retain thumbnail-sized toggle buttons as the convention for activating and deactivating variations, they may need to be further enhanced to convey state.

One design that potentially addresses both of these issues is to create a two-dimensional space for holding the variations, similar in spirit to the ART system [NAK00]. Within this space, users could arbitrarily arrange the thumbnails representing the alternatives, and use a lens to frame those they wish to consider at any point in time. For example, in Figure 38, the state space holds eight alternatives, though only four fall in the lens in the middle. Thus, only these four would be visible in the document window. With some care, users could physically arrange the alternatives to make it easy to switch between batches of variations that are closely related.

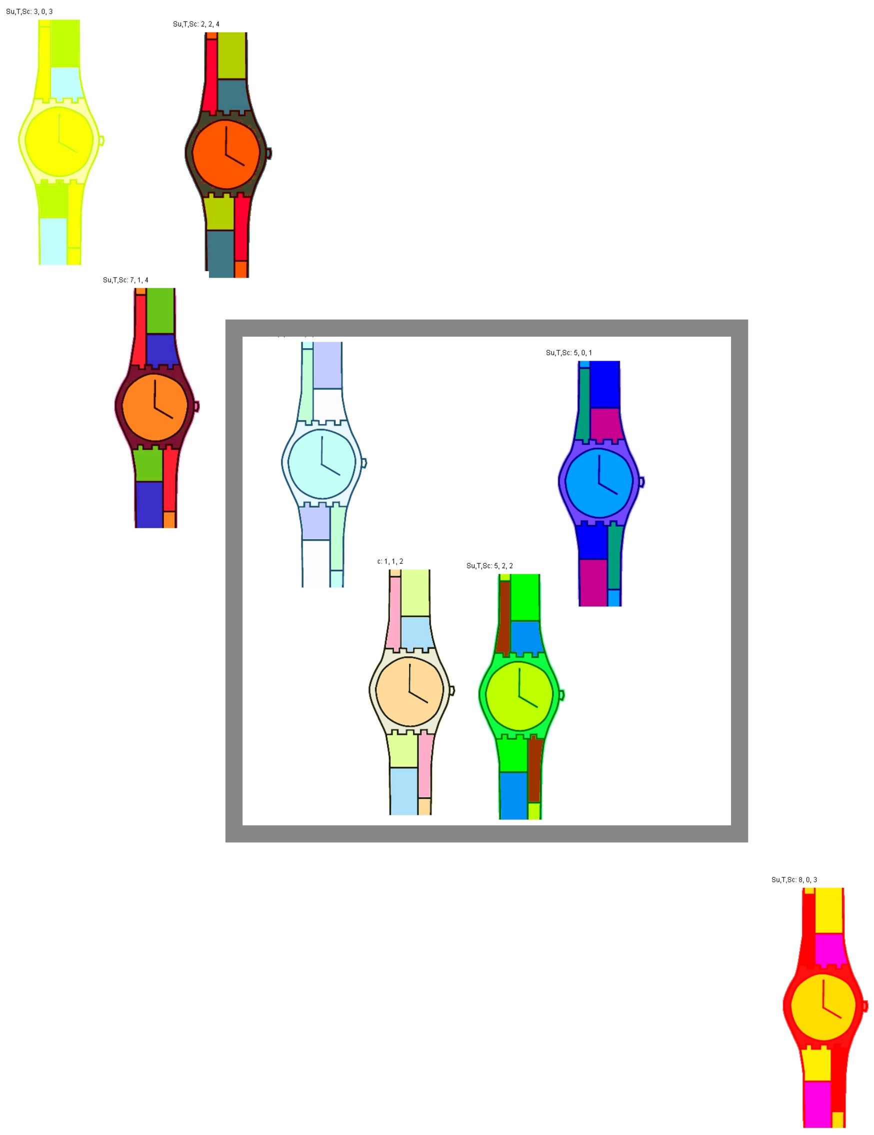


Figure 38. A state space for holding alternatives.

Finally, users sometimes asked whether they could create an entirely new set of alternatives with a filter, rather than just one at a time. That is, they wanted a "Create New *Set* of Versions" command in addition to the Create New Version command. How often such a command would be used is unclear, but is worth testing out.

8.8 Summary

Results from our studies indicate that tools designed to support set-based interaction can have a significant effect on the problem solving process, despite being entirely optional components of the user interface. In particular, Parallel Pies' ability to automatically duplicate and apply a command to a document resulted in subjects exploring more broadly and developing more solution alternatives. In a tightly constrained task, subjects also developed more optimal solutions with Parallel Pies, but in an open-ended task, subjects initially overused this feature, leading to lower-rated solutions. Side Views, on the other hand, led to reduced slider usage (about 50% less, on average), and also facilitated the serendipitous discovery of alternative solutions. In a collaborative setting, the tool facilitated communication while enabling subjects to quickly dismiss entire areas of the design space by virtue of its multiple previews.

CHAPTER 9

IMPACT AND OPPORTUNITIES FOR FUTURE WORK

This dissertation makes the following contributions:

1. We provide evidence that individuals often wish to explore alternative possibilities when solving complex problems (that is, engage in set-based problem solving practices), though few facilities exist in current user interfaces to expressly support this practice
2. We provide evidence that the lack of tools to explore alternatives can lead to a linear problem solving process marked by continual revision of a single solution instance (point-based problem solving). Evidence for this phenomenon can be found in our initial exploratory study as well as our laboratory studies, where linear problem solving processes are prominent when our tools are absent
3. To facilitate exploration of alternatives, we present the concept of a set-based interface. A set-based interface refers to an interface that provides services intended to facilitate the generation, manipulation, evaluation, and management of sets of alternatives. We contrast this concept with point-based interfaces, which we argue most accurately describe current interface designs
4. We provide specific means (visualizations and metrics) for distinguishing between point- and set-based problem solving
5. We present two tools designed to support set-based problem solving, Side Views and Parallel Pies

6. Based on results from two controlled laboratory studies and a think-aloud study, we conclude that set-based interface mechanisms, when available, are adopted and can influence an individual's problem solving process in the following ways:
    a. Users more broadly explore a problem space, sometimes at the cost of going in less depth
    b. Users can find more optimal solutions, where optimality is measured by the fewest number of operations required to derive a chosen solution from a given start state
7. With respect to Side Views, we find that the presence of Side Views leads to approximately 50% less slider usage when adjusting parameter settings
8. With respect to Parallel Pies, we find that this tool can:
    a. Lead to *worse* solution states if overused in time-constrained tasks
    b. Both positively and negatively impact cognitive load in open-ended tasks, while showing no effect in more tightly constrained tasks
9. With respect to Side Views and Parallel Pies, we find users highly rate the ability to quickly generate new standalone alternatives and the ability to broadly survey one's options through the guise of enhanced previewing mechanisms
10. We find no evidence that Parallel Pies or Side Views enable a user to develop a high quality solution faster, counter to expectations that multiple previews would lead to better solutions faster

9.1 Conclusions

Based on our research, we reach the following conclusions about set-based interfaces. Of the four proposed characteristics of a set-based interface (generation, manipulation, evaluation, and management), the capabilities to quickly spawn a new standalone, permanent alternative (generation) that is automatically managed by the interface (management) and easily comparable with other states (evaluation), constitute the most important and influential features of the tools we built. These capabilities are largely absent from current applications (especially the degree to which they are integrated in our application), but our results provide compelling evidence that they should be considered for future applications: Their presence consistently resulted in subjects exploring more broadly, a tactic they consciously embraced and prized, and also resulted in more optimal solutions for a tightly constrained task. Notably, this feature set is distinct from history and Undo mechanisms, in that it allows for the *proactive* generation of alternatives as data are actively manipulated, allowing for the inclusion of serendipitously discovered results.

From our study data, the ability to manipulate sets of alternatives simultaneously (i.e., apply an operation to all of them at the same time) seems less vital. Further research may demonstrate when and how this capability may prove most useful.

Multiple sets of previews, while highly prized, had no effect on solution quality, though they significantly reduced slider usage in our studies. However, our subject pool was composed of highly experienced, expert users. Thus, the utility of multiple previews may be much greater and more noticeable for novice users. Their utility may also be greater for functions whose parameters have complex interactions that cannot be easily predicted by users.

Our results suggest that future work should focus primarily on exploring ways of supporting the rapid generation, management, and evaluation of *standalone* alternatives in other task domains. In the next section, we consider open research problems, and suggest some additional studies that could be performed.

9.2 Future Work

Future work for set-based interaction research can be divided into three broad categories: Exploring additional problem domains, tool-level support for each aspect of a set-based interface, and further studies.

9.2.1 Problem Domains

In this body of work, we have focused primarily on the domain of image manipulation. Many opportunities remain to explore how these principles can be more fully realized in other domains. For example, how could one support experimentation with time-based media, such as audio or video, where parallel evaluations are not practical? How should these alternatives be represented? How could the interface represent alternative temporal sequences?

Experimenting with alternatives is vital to developing a good solution in any domain, but little explicit support has been developed for this process in current user interfaces. The primary challenge for user interface research is to develop a set of domain-independent services that a user can expect to exist in *any* application when they feel the need to explore. That is, just as services such as Undo or Cut/Copy/Paste are expected to exist in any application, a similar set of services should be available to support exploration.

9.2.2 Better Support for Generating Alternatives

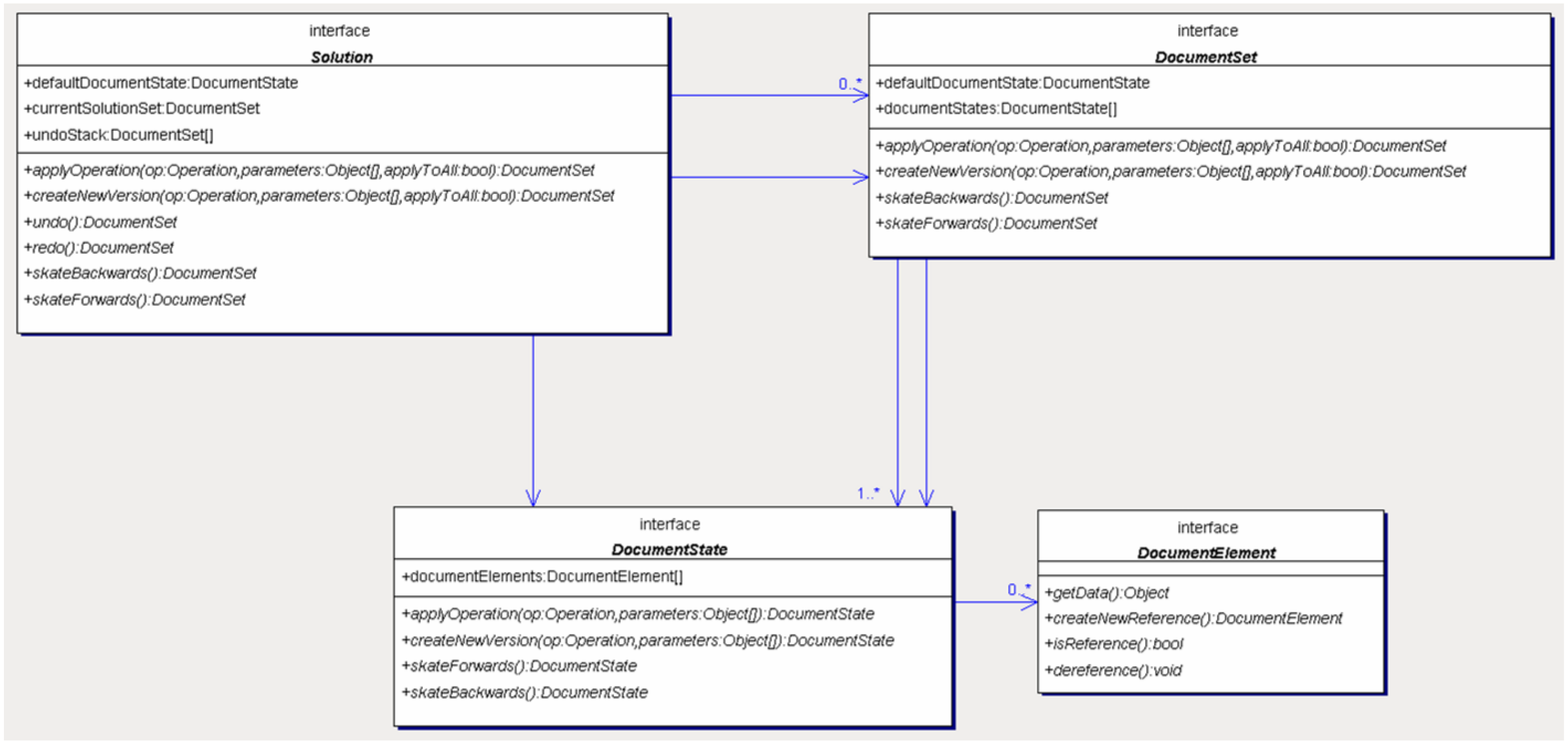
As we have argued, generating alternatives is one of the key aspects of a set-based interface. We can consider this process from both the user interface implementer's and user's perspective. From the implementer's perspective, we are interested in making it easy to create new alternatives without unnecessarily wasting memory. From the user's perspective, we wish to make it easy to designate and generate a new alternative with little effort. We discuss possibilities for future research from both perspectives.

As discussed in Chapter 6, Parallel Pies creates a copy of the entire document state when generating a new alternative. However, for some applications, this deep copy could be wasteful of resources, especially if only a small part of two or more solutions differ. Toolkits could provide useful abstractions to facilitate the developer's task of generating alternatives that are not wasteful of resources.

Figure 39 presents a UML class diagram [BOO99] that extends the Parallel Pies architecture shown in Figure 20. This architecture is virtually identical to the one discussed earlier in Chapter 6, with the exception of a new type, *DocumentElement*. A DocumentElement is intended to store meaningful segments of data in the document state. In an image editing application, these may correspond to layers in the image; in a text document, they could be paragraphs. The goal is to provide an abstraction for these chunks of data to make it easier to conserve memory when generating alternatives. Conservation of memory can thus occur in the following manner.

When creating an alternative to a given DocumentState, the new DocumentState can, at first, contain references to all of the DocumentElements of the original

DocumentState. When a DocumentElement in either the original or new state is modified, it can be converted from a reference to a true copy by calling its dereference() method.



interface
Solution
+defaultDocumentState:DocumentState
+currentSolutionSet:DocumentSet
+undoStack:DocumentSet[]
+applyOperation(op:Operation,parameters:Object[],applyToAll:bool):DocumentSet
+createNewVersion(op:Operation,parameters:Object[],applyToAll:bool):DocumentSet
+undo():DocumentSet
+redo():DocumentSet
+skateBackwards():DocumentSet
+skateForwards():DocumentSet
0..*
interface
DocumentSet
+defaultDocumentState:DocumentState
+documentStates:DocumentState[]
+applyOperation(op:Operation,parameters:Object[],applyToAll:bool):DocumentSet
+createNewVersion(op:Operation,parameters:Object[],applyToAll:bool):DocumentSet
+skateBackwards():DocumentSet
+skateForwards():DocumentSet
1..*
interface
DocumentState
+documentElements:DocumentElement[]
+applyOperation(op:Operation,parameters:Object[]):DocumentState
+createNewVersion(op:Operation,parameters:Object[]):DocumentState
+skateForwards():DocumentState
+skateBackwards():DocumentState
0..*
interface
DocumentElement
+getData():Object
+createNewReference():DocumentElement
+isReference():bool
+dereference():void

Figure 39. Outline of a basic toolkit architecture for a set-based interface.

Ideally, these details – creating new references and dereferencing DocumentElements when modified – would be hidden from developers, so that they can simply supply an operation and its parameters to the Solution's applyOperation() or createNewVersion() methods (where applyOperation() replaces the existing state with the resultant state, and createNewVersion() applies the operation to a copy of the original state to produce a new, standalone state). For applications in which the DocumentElements logically correspond to objects within the document (e.g., layers in an image manipulation application, or objects within a vector-based drawing program) and operations are conceptually discrete and well-defined (i.e., there is a well-defined start and end to an operation), it is reasonable to assume that we can shield developers from these lower-level memory management details via this architecture. However, for problem domains such as text editing, where objects and operations are more fluid and less discrete, it is not clear if this design is optimal. Thus, one of the research threads to pursue in the future is to examine how appropriate this scheme is for problem domains with data objects and operations less discrete in nature.

From the user's perspective, we would like to support the creation of new standalone alternatives at any level of granularity. For example, in a word processor, we would like to allow a user to create alternatives at the level of an individual word, all the way up to alternative story flows. Supporting these various levels of generation may be as easy as allowing the user to select all or part of a document, then applying the Create New Version operation to that selection. However, how these alternatives are managed and presented to the user is an open research problem that we consider next.

9.2.3 Better Support for Managing Alternatives

One of the most critical components of a set-based interface is the infrastructure it provides to manage the alternatives – the facilities that allow variations to be stored, organized, and retrieved. This infrastructure creates a richer organizational structure than the notion of a document to offer explicit services to maintain all the solution variations.

In our work in the domain of image manipulation, Parallel Pies provided this infrastructure through the thumbnail representation of the variations and the visualization tool within the document window. However, this organizational scheme is obviously not appropriate for all task domains. In this section, we consider potential methods of visualizing and organizing alternatives based on where the user can expect to find them within the application. These can be summarized as placing the alternatives:

- Within the solution itself
- In an entirely separate space within the application

The first method of organizing alternatives is to place them directly within the solution itself, as Parallel Pies does. Microsoft Word [MIC] also takes this approach with its change-tracking mechanisms: Changes are displayed within the document itself and differentiable by virtue of the text's color. To remove clutter, these changes can be hidden; tool-tip windows provide a transient, on-demand view of the changes. The Fluid Documents interaction mechanism [BAY98] provides another mechanism by which alternatives can be embedded directly within the data itself.

Placing alternatives within the data has the advantage of making them readily accessible and directly associated with the data that is varied. This technique also works well for alternatives that are of very fine granularity. However, for variations that span large amounts of data (say, alternative story flows for a document), this method starts to break down because it becomes more difficult to place the alternatives side-by-side. This is a general problem with embedding alternatives within the data itself: This method can complicate the user's view of the overall solution by overloading the type and meanings of the data shown in the document window. The WYSIWYG metaphor is also compromised by virtue of embellishing the document with additional information, not all of which is desirable in the final version. Currently, this problem is partially handled through mode switches (e.g., turning "on" or "off" alternatives), though a better solution may be to move the variations outside the main data view, as we discuss next.

Alternatives can be placed outside the document data, in another, well-defined space. Parallel Pies does this via the thumbnails situated next to the data view. With this method, the alternatives are represented as separate, unique objects that are easily discernable, but still grouped according to the data that is varied. They are also tightly coupled to the document window itself. However, as currently implemented in Parallel Pies, this technique is most appropriate for variations of relatively coarse granularity since very small differences are difficult to notice in the thumbnails.

The ART system [NAK00] creates an entire space that can be used to experiment with segments of a text document. Next to a WYSIWYG view of the document, ART provides an ElementsMap that allows sections of text to be arranged in a 2D space. The vertical ordering of these text segments determines their ordering in the WYSIWYG view, but the user is otherwise free to interpret the spatial meanings of the text segments. With this scheme, one can create alternative text segments, then use spatial positioning along the x-axis to contemplate and compare alternatives.

The primary advantage of moving alternatives into their own, separate space is that they can more easily become first-class, manipulable objects. User interface designers are also freed from many of the constraints that arise when trying to represent the alternatives within the data itself (e.g., they do not need to worry about organizing the alternatives in a way that respects the WYSIWYG presentation of the data).

These methods of managing alternatives – placing them within the data or in their own separate, dedicated space – represent a continuum of possibilities. Moving forward, there are a number of opportunities to better scope out this design space and the affordances, strengths, and weaknesses of the various approaches. For example, do users prefer the alternatives within the data, or do they find it more helpful to have them isolated in separate windows? Does one method work better for one type of data than others? Which techniques work well for finely-grained alternatives, and which work well for alternatives that span the entire solution? The overall goal in answering these questions should be to develop a single, well-defined mechanism that is generally applicable across application domains so that users can have an expectation of this service being available in any application.

9.2.4 Better Support for Evaluation

In our work in the image manipulation domain, alternatives are physically laid out to enable side-by-side comparisons. However, this tactic does not work for time-based media, nor data such as text. This tactic also does not work well for extremely large documents or sets of data. Tools that summarize differences between two documents (e.g., [NEU92]) can assist in comparisons of larger documents by virtue of providing a very fine, localized view of the differences, but these types of tools lose their value when a user explores highly divergent alternatives.

There are thus a number of opportunities to investigate additional mechanisms for enabling the evaluation of sets of alternatives. Time-based media present one challenge, but the other primary challenge is in providing the ability to summarize and compare multiple alternatives at various levels of granularity. For example, for two extremely long documents, neither of which overlaps in any significant way, it may be more advantageous to perform semantic summarizations of the documents to reduce the amount of information to contemplate when making comparisons.

9.2.5 Better Support for Manipulation

Parallel Pies allows users to apply a command across all currently active document variations. Similarly, Lunzer has investigated the possibility of "multiplexing" user input so that the same input is replicated across multiple document instances [LUN99]. In both systems, the intent is to reduce the need for users to replicate actions when working with multiple alternatives. That is, these facilities attempt to increase the ease with which one can keep alternatives in sync. Apart from these research examples, other applications facilitate the process of keeping two or more document states in sync via mechanisms such as scripts or merge commands.

While a number of mechanisms exist to help keep different document versions in sync with one another, it is not clear which are the most useful. For example, do users prefer to proactively keep variations up-to-date using mechanisms such as those found in Parallel Pies or the subjunctive interface? Or do they prefer to focus on one alternative exclusively, then transfer the fruits of their labor to other alternatives (either by copying data or replicating command usage)? Studies are needed to answer these questions.

9.2.6 Further Studies

In our studies, we found that Parallel Pies' Create New Version command heavily influences subjects' problem solving process. One of the questions that remains unanswered is whether we would observe similar effects if we replaced Parallel Pies with branching histories or snapshot mechanisms. The intuition is that we would not, because neither provides the ability to quickly spawn a new alternative the moment it is found when interacting with a command. This, alone, seems to be the primary reason that people adopted more set-based practices in our studies, and neither of the aforementioned mechanisms provides this capability. However, further studies are needed to understand how all of these types of tools influence the problem solving process across a wider range of task domains.

It would also be beneficial to better understand how various design choices for these process support tools affect the problem solving process. For example, Dix describes a number of parameters to consider for implementations of Undo [DIX96]: The size of the undo stack, what actions are undoable, whether a series of actions get chunked together into one undoable operation, and so on. Each property of a tools' design has the potential to influence its usage. However, there is little understanding of what ranges of values are suitable or desirable for each of these properties. For example, do users need unlimited undo, or is it sufficient to provide only the last 100 actions? Would Side Views be just as effective with three previews per parameter? For functions that take a significant amount of time, at what point does waiting for Side Views' multiple previews become a liability? How many alternatives should be readily accessible and viewable at one time?

For over 20 years, process support tools such as Undo mechanisms or enhanced histories have been proposed and built. Intuitively, many of these tools are invaluable: It would be difficult to imagine using any application without Undo being available. However, few have studied the true impact these tools have on the problem solving process. This research provides one of the first investigations of how these types of process support tools can influence the problem solving process and provides a base from which others can continue to explore how to build and evaluate computational tools designed to complement human problem solving across problem domains.

APPENDIX A

TOP THREE RESULTS FOR EACH TASK IN THE COLOR SCHEME STUDY

This appendix shows the top three rated results for each season of the color scheme task. Subjects could only vary the hue, saturation, lightness; color balance; and contrast of the image. Furthermore, they could only modify the entire image – selective application of a function was not possible.

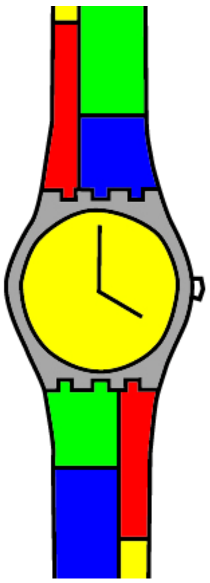


Figure 40. The initial state for the watch task.

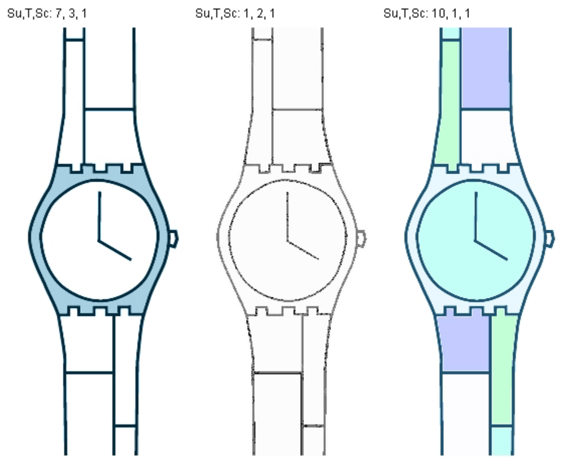


Figure 41. The top three rated solutions for winter.

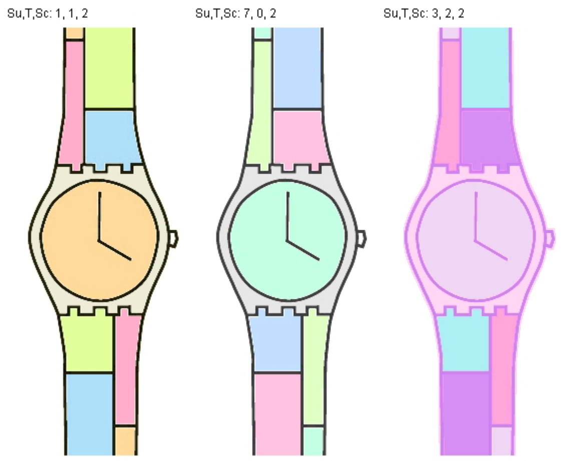


Figure 42. The top three rated solutions for spring.

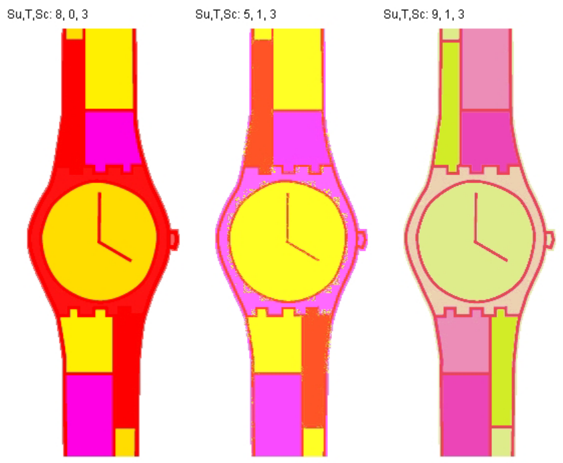


Figure 43. The top three rated solutions for summer.

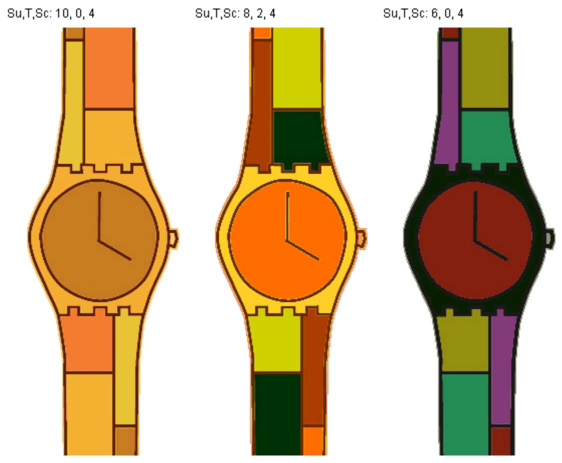


Figure 44. The top three rated solutions for fall.

APPENDIX B

PARTIALS

The focus of this dissertation has been on exploring the concept of a set-based interface in the domain of image manipulation. In this appendix, we broaden our scope to consider how these principles might apply to a completely different domain: programming.

As with any other ill-defined task, there is no one right way to solve a programming problem. Thus, programmers must experiment with their code at all levels of granularity – from individual expressions, to algorithms, to the overall architecture. Informally, we have observed individuals' experimentation supported through a number of devices: some principled, some ad-hoc; some clearly documented, and others less so.

At the most principled level, the use of globally-accessible variables provides a way to factor out elements that are uncertain and likely to change due to experimentation. For example, when developing a network server, the programmer may be faced with several implementation choices that affect the overall performance of the server, such as the maximum number of clients to service at one time, how large to make read/write buffers, and so on. In this case, these unknown quantities represent atomic values that can be factored out into globally accessible and configurable variables. At least three approaches are possible for setting these variables: They can be directly set within the code, requiring a recompile and re-link each time they are changed; they can be read in from a configuration file; or they can be set by prompting the user. The latter two avoid the need to recompile when changing values, but require significantly more effort to enact since both require additional support code (code to read a file, or code to interact with a user). However, once in place, all three methods make it somewhat easier to experiment with quantities whose values may change. But these methods do not, by themselves explicitly document the likely *alternatives* for a given quantity. That is, one must manually document that a value such as the maximum number of threads could be any number between 20 and 30.

The C language's preprocessor facility (as realized through *#define*, *#ifdef*, and *#else*) affords the explicit expression of alternatives by enabling entire blocks of code to be conditionally included or excluded. Software developers can experiment by developing several sets of alternatives in code (either contiguous on non-contiguous) that are demarcated via these preprocessor directives. However, testing each alternative requires a recompilation and re-linking of the executable.

Another method of supporting experimentation is to save a copy of the code before developing an alternative. This copy could be a snapshot in a revision control system (e.g., RCS [TIC82]), or simply a copy of the file in the file system. Both tactics have the advantage of fully preserving a previous state, but only the revision control system provides a well-defined mechanism for documenting and swapping alternatives in and out. However, with either of these two mechanisms, the differences are not visible in the code itself – one must compare and contrast two or more separate files using tools such as diff [DIFF]. Additionally, like the preprocessor facility, one must recompile and re-link after developing an alternative.

A more ad-hoc practice of generating and storing alternatives is to comment code in and out. This method of experimentation is universally available in every language and readily accessible without the need for any additional tools. However, it is prone to error since one must carefully remove and re-introduce complete blocks of code when switching between alternatives. Furthermore, unless the developer explicitly states her intentions, the commented-out code does not necessarily communicate to others that it is an alternative. Finally, as with other methods, a recompilation and re-linking is required to test out different versions.

Given this need to experiment and these current methods, we set out to devise a scheme that allowed one to explicitly express uncertainty within the language itself. The intent was to formalize this process of experimentation, require as little effort as possible to experiment (e.g., no need to create support code), and enable the developer to delay commitment to a final value until runtime. In this spirit, we developed a language enhancement for the Java language called a *partial*.

A partial describes a portion of code that is uncertain. A partial expression has the following syntax:

```
partial(type : alternative1 [, alternative2]* )
```

Where *type* represents the type of the expression, and *alternative* represents one or more expressions that evaluate to that type. For example, if the value for an int is unknown, we could create the following expression:

```
int myValue = partial(int : 1, 5, 8, 100);
```

At runtime, a user can choose between assigning myValue the value 1, 5, 8, 100, or a custom value.

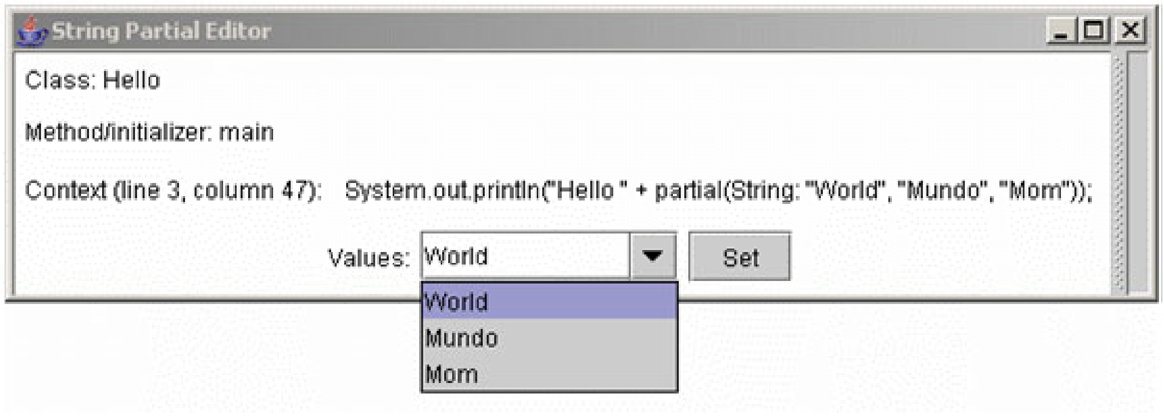
Similarly, we could create a partial for a String:

```
     System.out.println("Hello " + partial(String: "World",
"Mundo", "Mom"));
```

Arbitrary objects are supported as well:

```
     Vector a = getSomeVector();
     Vector b = getSomeOtherVector();
     Vector c = partial(Vector : a, b);
```

Using a modified version of IBM's jikes compiler [JIK], this construct is translated into code that invokes a function in a runtime library. When the resultant application is executed, a dialog box appears at the point in execution when a partial line of code is encountered. For example, for "Hello World" example above, the following dialog box displays:


String Partial Editor

Class: Hello

Method/initializer: main

Context (line 3, column 47): System.out.println("Hello " + partial(String: "World", "Mundo", "Mom"));

Figure 45. A partials dialog box.

At runtime, the user receives context about the line executing (the actual code at that line, the method that is executing, and the class of the object), and a drop-down list of the potential values specified. Importantly, the user can also manually input a new value for expressions that evaluate to numbers or Strings (we discuss how to deal with manually creating arbitrary objects in the future work portion of this section). Once a value has been chosen, pressing the Set button assigns the chosen value to the expression, and execution resumes.

9.2.7 Affordances

Partials provides an explicit mechanism for specifying (generating and managing) sets of alternatives at the level of the programming language itself. At runtime, evaluation can be performed serially each time a partial is executed. (In our current implementation, *parallel* evaluation is not supported, though one could potentially fork the process [SIL02] to derive a separate standalone application instance. However, this method would require careful consideration of how to handle code that affects open resources such as network sockets or files.) Currently, we have focused on grammar-level support for experimentation, and have not investigated interface-level support for specifying or simultaneously manipulating the alternatives in an editor.

Compared to other mechanisms in use to support experimentation, partials provides a clearly identifiable mechanism for designating uncertainty. Furthermore, because the compiler "knows" that this expression represents a set of alternatives, it serves as a hook to provide explicit support for exploration at this point. With the current strategies, this intent is missing in the representations of the alternatives, as are mechanisms designed to help people evaluate these different paths.

9.2.8 Related Work

We use the term "partial" to describe this technique to evoke the sense that the software is partially constructed, and that details still need to be worked out. Most related to this concept in the field of HCI are user interface continuations [QUA03]. User interface continuations allow a user to partially specify the information for a dialog box or form, and continue working. Each incomplete piece of information is collected and made available to be completed at a later time.

Like user interface continuations, partials allows one to delay commitment to particular values. However, continuations primarily address those circumstances when a single value is unknown, whereas partials is designed to allow sets of alternatives to be specified.

9.2.9 Implementation

Partials has been implemented by modifying the Java grammar and Jikes compiler, and by providing a new run-time library. The canonical JavaCC Java grammar [JAV] was modified to include two new constructs: *partial* and *partialBlock* (described in future work below). A custom preprocessor (a parser derived from the modified grammar) converts instances of these constructs into executable code that calls our runtime support library. The modified Jikes compiler automatically loads and calls this preprocessor when reading in files. The resultant bytecode is compatible with any JVM, but requires the runtime library at execution time.

9.2.10 Evaluation and Discussion

We have had the opportunity to evaluate partials in the context of developing a Java-based chat client [VOI05]. As part of the testing of this client, we found that a remote user was encountering a bug we could not reproduce on our own machines. Based on the description of the problem, we were able to identify the probable cause, and wrote several partials for the expressions we suspected were causing the bug. After compiling the application with the partials, we shipped it to the remote user and, via a phone conversation, were able to guide her through trying the different alternatives until we discovered the problem and its fix. In this case, partials served as an embedded “debugging environment” that took very little effort to create on the part of the developer.

While we can relate this one success story, our other experiences suggest that partials, as currently implemented, is most useful for debugging individual API calls that are poorly documented or not working as expected. Exploration of alternatives at the level of classes or the overall software architecture are not well supported by partials, and would be better served by other means.

9.2.11 Future Work

The partial feature, as described, could be implemented without modifying the Java grammar: A normal function call and a custom library could be created that enabled one to achieve similar functionality. However, providing this service at the level of the language provides us with capabilities not available otherwise, helping to increase the ability to experiment at runtime.

Since partials is implemented at the level of the grammar itself, it can receive, at compile-time, the complete state of the parse tree. Thus, at the point it executes, it can “know” all variables accessible to it in its context (i.e., all local variables, all parameters, all member variables, etc). Our preprocessor currently collects all this information so that it can pass these data on to a scripting environment, such as Jython [JYT]. For example, for each partial, a new Jython environment could be created and attached to the process, and variables could be set within that environment to correspond to all variables available in the current executing context. Through this mechanism, users could write custom code at runtime to *derive* the value for the partial expression. Additionally, users could instantiate new objects rather than limiting themselves to the object choices specified at compile time. For example, for a partial that expects a Vector, custom Jython code could be written at runtime to create a new Vector to set the partial to.

The partial construct was designed to work at the level of individual expressions. To achieve exploration at a slightly higher level, we also devised a second, untested construct for expressing uncertainty. A *partial block* provides developers the capability to specify blocks of code that can be executed 0 or more times in succession. The basic syntax of a partial block is as follows:

```
     partialBlock(
        [type identifier : alternative1 [, alternative2]*]
        [, type identifier2 : alternative1 [,
alternative2]* ]*
     ) {
           // Code to execute
     }
```

In this expression, zero or more variables and their alternatives can be specified in the specification of a partialBlock. For example:

```
     partialBlock (int aValue: 1, 3, 10) {
          System.out.println("Value: " + aValue);
     }
```

This expression creates a block of code in which aValue can assume the values 1, 3, 10, or a custom value prior to the block's execution. This subsequent block of code can execute zero or more times, and each time the block is executed, users can re-specify the values for the variables specified. For example, in the example above, this partial block allows one to specify the value for aValue each time the proceeding block of code is executed.

This construct was designed to facilitate fine-tuning the details of entire algorithms, rather than individual expressions. At present, the entire preprocessor has been implemented, though the runtime support is currently absent.

REFERENCES

ABO92 Abowd, G., and Dix, A. J. Giving undo attention. Journal of Interactive Computing, v. 4, no. 3, 1992. pp. 317-342.

ADO Adobe Photoshop. http://www.adobe.com

ALD94 Aldous, D., and Vazirani, U. "Go with the winners" algorithms. In Proceedings of the Symposium on Foundations of Computer Science, pp. 492-501. 1994.

ARC84 Archer, J.E.Jr., Conway, R., and Schneider, F.B. User Recovery and Reversal in Interactive Systems. ACM Trans. Programming Language Systems, v6, number 1, 1984. pp. 1-19.

AYE99 Ayers, Eric, and Stasko, John. Using graphic history in browsing the World Wide Web. In Proceedings of the 4th International World Wide Web Conference. 1999.

BAE91 Baecker, R., Small, I., and Mander, R. Bringing icons to life. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '91), 1991. pp. 1-6.

BAY98 Bay-wei, C., Mackinlay, J., Zellweger, P., and Igarashi, T. A negotiation architecture for fluid documents. In Proceedings of the 11th annual ACM symposium on User interface software and technology, 1998 (UIST '98). pp. 123-132.

BER94 Berlage, T. A selective undo mechanism for graphical user interfaces based on command objects. ACM Transactions on Computer-Human Interaction, 1(3):269-294, 1994.

BIE93 Bier, E.A., Stone, M.C., Pier, K., Buxton, W., and DeRose, T.D. Toolglass and Magic Lenses: The see-through interface. In Computer Graphics, 27 (Annual Conference Series), 1993, 73-80.

BIE94 Bier, E.A., Stone, M.C., Fishkin, K., Buxton, W., and Baudel, T. A Taxonomy of See-Through Tools. In CHI, pages 358-364. Addison-Wesley, April 1994.

BOO99 Booch G., Rumbaugh J., and Jacobson I. The Unified Modeling Language User Guide. Addison-Wessley 1999.

CHI94 Chiueh, T., and Katz, R. Papyrus: A history-based VLSI design process management system. In Proceedings of the Tenth International Conference on Data Engineering, 1994. pp. 385-392.

CHI98a Chi, E.H., Riedl, J., Barry, P., and Konstan, J. Principles for information visualization spreadsheets. In IEEE Computer Graphics & Applications, vol. 18, no. 4, pp. 30–38, July-August 1998.

CHI98b Chiueh, T., Mitra, T., Neogi, A., and Yang, C.K. Zodiac: a history-based interactive video authoring system. In Proceedings of the sixth ACM international conference on Multimedia, 1998 (Multimedia ’98). pp. 435-444.

COC96 Cockburn, A. and Jones, S. Which way now? Analysing and easing inadequacies in WWW navigation. International Journal of Human Computer Studies, 44 (1996)

CRO01 Cross, N. Design cognition: Results from protocol and other empirical studies of design activity. Chapter 5 in Design Knowing and Learning: Cognition in Design Education. Eastman, C., McCracken, M., and Newstetter, M. (eds.). Elsevier Science, 2001, 79-103.

DER00 Derthick, M., and Roth, S. Data exploration across contexts. In Proceedings of the 5th international conference on Intelligent user interfaces, 2000 (IUI ’00). pp. 60-67.

DIFF Diffutils. http://www.gnu.org/software/diffutils/diffutils.html

DIX96 Dix, A., Mancini, R., and Levialdi, S. Alas I am undone - Reducing the risk of interaction? In HCI '96 Adjunct Proceedings (Imperial College, London, UK, 1996), 51-56. Available as http://www.soc.staffs.ac.uk/~cmtajd/talks/risk/

EDW00 Edwards, W.K., Igarashi, T., LaMarca, A., and Mynatt, E.D. A temporal model for multi-level undo and redo. In Proceedings of UIST 2000, 31-40.

EDW97 Edwards, W.K., and Mynatt, E.D. Timewarp: Techniques for autonomous collaboration. In Conference Proceedings on Human Factors in Computing Systems (CHI 1997), 218-225.

FOL96 Foley, J.D., van Dam, A., Feiner, S.K., and Hughes, J.F. Computer Graphics – Principles and Practice. Addison Wesley, 1996.

FOX98 Fox, D. Composing magic lenses. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI ’98), 1998. pp. 519-525.

FRA87 Fraser, C.W., and Myers, E.W. An editor for revision control. In ACM Trans. Program. Lang. System., v.9, number 2, 1987. 277-295.

FUR86 Furnas, G.W. Generalized fisheye views. In Proceedings of the SIGCHI conference on Human factors in computing systems, 1986 (CHI ’86). pp. 16-23.

GAM94 Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.

GEN95 Genau, A., and Kramer, A. Translucent History. In Conference companion on Human factors in computing systems, 1995, (CHI '95). pp. 250-251

GIMP The GNU Image Manipulation Program (GIMP). http://www.gimp.org

GOE92 Goel, V., and Pirolli, P. The structure of design problem spaces. Cognitive Science, 16(3), 1992, 395-429.

GOR85 Gordon, R.F., Leeman, G.B. Jr., and Lewis, C.H. Concepts and implications of undo for interactive recovery. In Proceedings of the 1985 ACM annual conference on The range of computing : mid-80's perspective, 1985. pp. 150-157.

GRE Green, T.R., and Blackwell, A.F. A tutorial on cognitive dimensions. Available at: http://www.thomas-green.ndtilda.co.uk/workStuff/Papers

GRE88 Greenberg, S., and Witten, I.H. How users repeat their actions on computers: principles for design of history mechanisms. In Proceedings of the SIGCHI conference on Human factors in computing systems, 1988 (CHI '88). pp. 171-178.

GRO96 Gross, M.D., and Do, E.Y.L. Ambiguous intentions: A paper-like interface for creative design. In Proceedings of the 9th annual ACM symposium on User interface software and technology (UIST '96), 1996. pp. 183-192

GUN99 Gunther, J., and Ehrlenspiel, K.. Comparing designers from practice and designers with systematic design education. In Design Studies, 20 (1999), pp. 439-451.

HAR88 Hart, S.G. and Staveland, L. E. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. Chapter in Human mental workload. North-Holland, 1988.

HEP02 Hepting, D.H. Towards a Visual Interface for Information Visualization. In Proceedings of the 6th International Conference on Information Visualization, IEEE Computer Society, 2002. pp. 295-302.

HEP03 Hepting, D.H. Interactive evolution for systematic exploration of a parameter space. In Intelligent Engineering Systems through Artificial Neural Networks, Volume 13, 2003. pp. 125-131.

HUD97 Hudson, S., Rodenstein, R., and Smith, I. Debugging lenses: A new class of transparent tools for user interface debugging. In Proceedings of the 10th Annual Symposium on User Interface Software and Technology (UIST 97), 1997. pp. 179-187.

IGA01 Igarashi, T. and Hughes, J.F. A suggestive interface for 3D drawing. In Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology, pages 173-181. ACM Press, 2001.

IPP Intel Performance Primitives (IPP). http://www.intel.com/software/products/ipp

JAN01 Jankun-Kelly, T.J., and Ma, K.L. Visualization exploration and encapsulation via a spreadsheet-like interface. IEEE Transactions on Visualization and Computer Graphics, 7(3), 2001, 275-287.

JAV JavaCC. https://javacc.dev.java.net

JGI JGimp. http://jgimp.sourceforge.net

JIK IBM Jikes compiler. http://www.research.ibm.com/jikes

JIP JIPP Java Interface and Wrappers to IPP. http://www.cssip.uq.edu.au/staff/bamford/JIPP

JYT Jython. http://www.jython.org

KAN97 Kandogan, E, and Shneiderman, B. Elastic Windows: Evaluation of multi-window operations. In Proceedings of the SIGCHI conference on Human factors in computing systems, 1997 (CHI '97). pp. 250-257.

KAW04 Kawasaki, Y., Igarashi, T. Regional undo for spreadsheets. Part of the demo presentations at the Symposium on User Interface Software and Technology, 2004 (UIST '04). Available at: http://www-ui.is.s.u-tokyo.ac.jp/~kwsk/undo/kawasaki_uist04_regional.pdf

KHA04 Khandelwal, M., Kerne, A., and Mistrot, J.M. Manipulating history in generative hypermedia. In Proceedings of the Fifteenth ACM Conference on Hypertext & Hypermedia, 2004. pp. 139-140.

KLE01 Klemmer, S.R., Newman, M.W., Farrell, R., Bilezikjian, M., and Landay, J. The designer's outpost: A tangible interface for collaborative web site design. In Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST 2001). pp. 1-10.

KLE02 Klemmer, S.R., Thomsen, M., Phelps-Goodman, E., Lee, R., and Landay, J.A. Where do web sites come from? Capturing and interacting with design history. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI 2002), 1-8.

KRA88 Krasner, G.E., and Pope, S.T. A cookbook for using the model view controller user interface paradigm in Smalltalk-80. In Journal of Object-Orientated Programming, 1(3):26-49, August/September 1988.

KRE04 Kreusler, M., Nocke, T., and Schumann, H. A History Mechanism for Visual Data Mining. In Proceedings of Information Visualization, 2004 (NFOVIS 2004). Pp. 49-56.

KUR90 Kurlander, D., and Feiner, S. A visual language for browsing, undoing, and redoing graphical interface commands. In Visual Languages and Visual Programming. S.K. Chang (ed.). Plenum Press, New York, NY, 1990, 257-275.

LAN01 Landay, J., and Myers, B. Sketching Interfaces: Toward More Human Interface Design. In IEEE Computer, 34(3), March 2001, 56-64

LEE86 Leeman, G.B. Jr. A formal approach to undo operations in programming languages. In ACM Trans. Program. Lang. Syst., vol. 8, no. 1, 1986. pp. 50-87.

LEE92 Lee, A. Investigations into history tools for user support. Ph.D. thesis, University of Toronto, Ontario, Canada. 1992.

LI04 Li, Y., Hong, J., and Landay, J. Topiary: A tool for prototyping location-enhanced applications. In Proceedings of the 17th Annual Symposium on User Interface Software and Technology (UIST 2004), 2004. pp. 217-226.

LUN94 Lunzer, A. Reconnaissance support for juggling multiple processing options. In Proceedings of the Symposium on User Interface Software and Tools, 1994 (UIST '94). pp. 27-28.

LUN98 Lunzer, A. Towards the subjunctive interface: General support for parameter exploration by overlaying alternative application states. In Late Breaking Hot Topics, IEEE Visualization 1998, 45-48.

LUN99 Lunzer, A. Choice and comparison where the user wants them: Subjunctive interfaces for computer-supported exploration. In Proceedings of IFIP TC, 13 International Conference on Human-Computer Interaction (INTERACT '99), 474-472.

LUN01 Lunzer, A. Subjunctive Interface Support for Combining Context-Dependent Semi-Structured Resources. In Proceedings of IFIP TC. 13 International Conference on Human-Computer Interaction (INTERACT '01), Tokyo, Japan, Jul 2001, pp. 761-762.

LUN04 Lunzer, A., and Hornbaek, K. Usability studies on a visualization for parallel display and control of alternative scenarios. In Proceedings of the Working Conference on Advanced Visual Interfaces, pp. 125-132. ACM Press, 2004.

MAM01 Mamykina, L, Mynatt, E, and Terry, M. Time Aura: Interfaces for pacing. In Proceedings of the SIGCHI Conference on Human Factors in Computing. 2001, pp 144-151.

MAR97 Marks, J., Andalman, B., Beardsley, P. A., Freeman, W., Gibson, S., Hodgins, J., Kang, T., Mirtich, B., Pfister, H., Ruml, W., Ryall, K., Seims, J., and Shieber, S. Design galleries: A general approach to setting parameters for computer graphics and animation. In Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, 1997, 389-400.

MIC Microsoft Word. http://www.microsoft.com

MDI Multiple Document Interface. Available as: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/windowing/multipledocumentinterface/aboutthemultipledocumentinterface.asp

MIY86 Miyake, N. Constructive Interaction and the Iterative Process of Understanding. Cognitive Science, Vol. 10, 1986, pp. 151-177.

NAK00 Nakakoji, K., Yamamoto, Y., Reeves, B.N., and Takada, S. Two-Dimensional Positioning as a Means for Reflection in Design. Design of Interactive Systems (DIS 2000), 145-154.

NAK02 Nakakoji, K., Yamamoto, Y., and Aoki, A. Interaction design as a collective creative process. In Proceedings of the 4th Conference on Creativity and Cognition, 2002. pp. 103-110.

NEU92 Neuwirth, C.M., Chandhok, R., Kaufer, D.S., Erion, P., Morris, J., and Miller D. Flexible Diff'ing in a Collaborative Writing System. In Proceedings of the 1992 ACM conference on Computer-supported cooperative work (CSCW '92). 147-154.

NEW72 Newell, A., and Simon, H.A. Human Problem Solving. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972.

NEW00 Newman, M., & Landay, J. Sitemaps, Storyboards, and Specifications: A Sketch of Website Design Practice. In Proceedings of Designing Interactive Systems (DIS 2000), 263-274.

QUA03 Quan, D., Huynh, D., Karger, D., and Miller, R. User Interface Continuations. In Proceedings of the Symposium on User Interface Software and Technology, 2003 (UIST '03). pp. 145-148.

QUI02 Quigley, A., Leigh, D.L., Lesh, N.B., Marks, J.W., Ryall, K., and Wittenburg, K. Semi-automatic antenna design via sampling and visualization. In IEEE AP-S International Symposium and USNC/URSI National Radio Science Meeting (APS/URSI), 2002.

REI65 Reitman, W.R. Cognition and Thought. John Wiley & Sons, Inc., 1965.

REK99	Rekimoto, J. TimeScape: a time machine for the desktop environment. In Extended Abstracts on Human Factors in Computing Systems, 1999 (CHI '99). pp. 180-181
RIC97	Rich, C., and Sidner, C.L. Segmented interaction history in a collaborative interface agent. In Proceedings of the 2nd international conference on Intelligent user interfaces, 1997 (IUI '97). pp. 23-30.
RIT84	Rittel, H.W.J., and Webber, M.M. Planning Problems are Wicked Problems. Chapter in Developments in Design Methodology, 1984, 135-144.
SAL97	Salvendy, G. (Ed.). Handbook of Human Factors and Ergonomics. John Wiley & Sons, Inc., Second Edition, 1997.
SCH83	Schön, D. The Reflective Practioner: How Professionals Think in Action. Basic Books, New York, NY, 1983.
SEI98	Seidman, I. Interviewing as Qualitative Research. Teachers College Press, 1998.
SHN99	Shneiderman, B. User interfaces for creativity support tools. In Proceedings of the 3rd Conference on Creativity & Cognition, 1999. pp.15-22.
SHN00	Shneiderman, B. Creating creativity: User interfaces for supporting innovation. ACM Transactions on Computer-Human Interaction, 7(1):114-138, March 2000.
SIL02	Silberschatz, A., Gagne, G., Galvin, P.B. Operating System Concepts. Wiley, 2002.
SIM73	Simon, H. The structure of ill-structured problems. Artificial Intelligence, 4:181-203, 1973.
SOB99	Sobek II, D.K., Ward, A.C., Liker, J.K. Toyota's principles of set-based concurrent engineering. Sloan Management Review, Winter 1999, pp. 67-83.
TER02a	Terry, M. and Mynatt, E.D. Recognizing creative needs in user interface design. In Proceedings of the Fourth Conference on Creativity & Cognition, 2002, 38-44.
TER02b	Terry, M. and Mynatt, E.D. Side Views: Persistent, on-demand previews for open-ended tasks. In Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology (UIST 2002), 71-80.
TER04	Terry, M., and Mynatt, E.D. Variation in Element and Action: Supporting Simultaneous Development of Alternative Solutions. To appear in Proceedings of CHI 2004.

TIC82 Tichy, W. Design, implementation, and evaluation of a revision control system. In Proceedings of the 6th International Conference on Software Engineering, 1982. pp. 58-67.

TSA04 Tsang, S., Balakrishnan, R., Singh, K., and Ranjan, A. A suggestive interface for image guided 3D sketching. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI04), 2004. pp. 591-598.

VAR03 Vargas, I., Borgres, J., and Pérez-Quiñones, M.A. A Usability Study of an Object-Based Undo Facility. In Proceedings of HCI International, 2003.

VER02 Verlinden, J.C., Igarashi, T., and Vergeest, J.S.M. Snapshots and Bookmarks as a Graphical Design History. In Proceedings of International Design Conference 2002, Dubrovnik, Kroatia, 567-572.

VIT84 Vitter, J.S. US&R: A new framework for redoing. In Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 1984. pp. 168-176.

VOI05 Voida, A., and Mynatt, E.D. Six themes of the communicative appropriation of photographic images. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2005 (CHI ’05). pp. 171-180.

WAR95 Ward, A., Liker, J.K., Cristiano, J.J., Sobek II, D.K. The second Toyota paradox: How delaying decisions can make better cars faster. Sloan Management Review, Spring 1995, pp. 43-61.

WAS02 Washizaki, H. and Fukazawa, Y. Dynamic hierarchical undo facility in a fine-grained component environment. In Proceedings of the Fortieth International Confernece on Tools Pacific, 2002 (CRPITS '02). pp. 191-199.

WEI94 Weiss, R.S. Learning from Strangers. Free Press, 1994.

YAM01 Yamamoto, Y., Aoki, A., and Nakakoji, K. Time-ART: A tool for segmenting and annotating multimedia data in early stages of exploratory analysis. In CHI ’01 Extended Abstracts on Human Factors in Computing Systems, 2001. pp.113-114.

ZHO97 Zhou, C., Imamiya, A. Object-based nonlinear undo model. In Proceedings of 21st International Computer Software and Applications Conference, 1997 (COMPSAC '97). pp. 50-55.