# Mixed-Fidelity Prototyping of User Interfaces

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By

Jennifer Petrie

©Jennifer Petrie, February 2006. All rights reserved.

# Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

# Abstract

This research presents a new technique for user interface prototyping, called mixed-fidelity prototyping. Mixed-fidelity prototyping combines low-, medium-, and high-fidelity interface elements within a single prototype in a lightweight manner, supporting independent refinement of individual elements. The approach allows designers to investigate alternate designs, including more innovative designs, and elicit feedback from stakeholders without having to commit too early in the process. As well, the approach encourages collaboration among a diverse group of stakeholders throughout the design process. For example, individuals who specialize in specific fidelities, such as high-fidelity components, are able to become involved earlier on in the process.

We developed a conceptual model called the Region Model and implemented a proof-of-concept system called ProtoMixer. We demonstrated the mixed-fidelity approach by using ProtoMixer to design an example application.

ProtoMixer has several benefits over other existing prototyping tools. With ProtoMixer, prototypes can be composed of multiple fidelities, and elements are easily refined and transitioned between different fidelities. Individual elements can be tied into data and functionality, and can be executed inside prototypes. As well, traditional informal practices such as sketching and storyboarding are supported. Furthermore, ProtoMixer is designed for collaborative use on a high-resolution, large display workspace.

# Acknowledgements

I would like to acknowledge my supervisor, Dr. Kevin Schneider, for all of his guidance throughout my graduate studies. His suggestions were invaluable to my thesis work. I would also like to thank the members of the Software Research Lab, in particular David Paquette, Nicole Stavness, Andrew Sutherland, and Mark Watson, for their friendship, encouragement, and many interesting discussions. Special thanks to David Noete for his help with implementing ProtoMixer, saving me from many frustrations. Finally, I would like to thank my parents for their tremendous support over the many years of my university studies.

For my best friend and supportive husband, David.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

This research focuses on user interface prototyping, an important activity in software development. Prototyping involves creating mock-ups representing the user interface of the final software system. Prototypes serve as a common language with users, software engineers, and other stakeholders, offering a way for designers to explore design ideas and elicit feedback from stakeholders prior to committing to designs. Since prototyping helps flesh out requirements, prototypes may be used as a specification for developers. Prototyping is important in arriving at a well-designed user interface, and from many users' perspective the user interface is the software.

Prototypes are categorized according to three types or fidelities: low-, medium-, and high-fidelity, where fidelity refers to how accurately the prototypes resemble the final software in terms of visual appearance, interaction style, and level of detail [34]. Low-fidelity prototypes are paper-based sketches or images and only resemble the final software in terms of general appearance. High-fidelity prototypes most closely resemble the final software; they are computer-based designs that represent exact appearance and interactivity as well as have some functionality implemented. Medium-fidelity prototypes lie on the continuum between low and high; they are

computer-based refined versions of low-fidelity prototypes with additional detail, interactivity, and/or functionality. Commonly accepted best practice encourages starting with low-fidelity prototypes then moving to medium- and finally to high-fidelity, refining prototypes at each fidelity prior to advancing to a higher-fidelity.

Typically, a multidisciplinary team, often including end users, is involved in prototyping as well as in the overall software design. Collaboration among the team is important for brainstorming alternate design ideas and then refining the ideas through prototyping to arrive at a successful design in a timely manner. Low-fidelity prototyping is particularly conducive to collaboration since it involves large workspaces, such as whiteboards, chalkboards, and large sheets of paper, which are beneficial for expressing ideas for all team members to see as well as for allowing multiple people to work on a prototype simultaneously.

This research addresses the prototyping of a specific style of user interface that will be referred to as a true direct manipulation user interface. As the name suggests, this style of interface provides the user with visual objects to manipulate on the screen and these visual objects are usually more than simply form-based widgets. This style of interface is considered highly graphics-based and may include novel or non-standard interface elements and interactions. Furthermore, this style of interface is intended for traditional desktop machines, not embedded or other devices. While we may find that other styles of user interfaces may also benefit from the practices and techniques set out by this research, it is not our focus.

## 1.1 Motivation

The current practice for user interface prototyping has several shortcomings:

- it only addresses one fidelity at a time,
- it limits iteration to only occur within the current fidelity,
- it lacks support for collaboration between user interface designers and other stakeholders, and
- it does not encourage innovative user interfaces.

Under current practice, user interface designers work on one fidelity at a time. They do not advance to a higher-fidelity until the current fidelity prototypes have been extensively refined. This forces designers to make decisions on issues earlier than desired. For example, layout is commonly decided at the low-fidelity stage because it carries through to higher-fidelities and would be too much work to change later on, even though layout may not always be a high-priority issue in initial stages. This also undesirably delays investigating other issues such as complex functionality. As a result, key or at least more concrete design ideas are unable to be presented to stakeholders quickly and some design ideas are committed to prematurely.

While prototyping is often termed 'iterative' design, designers only iterate within a specific fidelity at best. Iteration does not occur between the different fidelities. For example, designers may iterate between different versions of a medium-fidelity prototype but they rarely iterate back to the low-fidelity version. Under current

practice, making a significant change that involves refining an earlier fidelity often requires starting over from scratch at that lower-fidelity. Because different fidelities are performed on different mediums and tools, there is significant overhead in transitioning between fidelities.

Current prototyping practice does not encourage collaboration between user interface designers and various stakeholder groups. For example, end users are often only involved in initial requirements gathering and sometimes in final design decisions. However, involving end users for regular feedback throughout the design process may be more effective in achieving the best designs. Another group, the software engineers, work independently from user interface designers, coming together only once respective designs have been finalized. There is a clear lack of coordination and communication while design artifacts are being developed, which may result in software that is not as attractive, usable, or accurate in meeting requirements. In recent years, researchers have shown considerable interest in attempting to bridge the gap between interface and software design through a series of workshops [10, 12, 13]. Gurantene et al. [7] argue for using high-fidelity prototypes as a bridging artifact.

Current design practice does not encourage innovative interfaces. Designers use standard toolkit components, standard layouts, and mouse-based interactions when developing user interfaces. This approach is reinforced in the tool support that is available. While standards are important for providing reuse as well as familiarity for the user, there are cases when new or non-standard designs may be more appropriate. Unfortunately, current tools make it impossible or at least infeasible to be creative

in terms of interaction styles, techniques, and devices. As a result, more creative or novel interface designs and interactions are rarely explored.

The above issues with current practice are evident in and often are reinforced by the prototyping tools that exist today. Let us consider interface builders, such as Visual Basic®, NetBeans™, and CodeWarrior™, as an example since they are the most widely used prototyping tools. Interface builders are commonly used for creating high- and sometimes medium-fidelity prototypes. When using an interface builder, designers are limited to exploring one fidelity at a time, are unable to iterate between fidelities when refinements at a lower-fidelity are needed, and are largely restricted to design layouts, components, and interactions that are built into the tool. Furthermore, designers are not able to easily collaborate because interface builders are single-user applications intended for traditional desktop displays. Interface builders do not allow opportunity for designers to defer decisions; rather they force designers to make immediate decisions on issues such as layout and types of components. On top of all these weaknesses, interface builders are fairly time-consuming to use and learn.

The user interface research community has been investigating a variety of tools to address some drawbacks of interface builders and other commercial tools. However, most research-based tools have focused on supporting one specific fidelity, most notably low-fidelity sketching [17, 18], which has largely been ignored at the commercial level. One of the most promising tools is DENIM [18], which supports low- and medium-fidelity prototyping of website design. In a limited way, DENIM allows for more than one fidelity at once; images and basic functionality (hyperlinks) may

be combined in a sketched page. DENIM attempts to encourage some innovation by allowing users to create reusable interface components. DENIM itself is not collaborative but has been used collaboratively through integration in Designers' Outpost [15], a tangible, large display system for structuring websites. DENIM does not offer much in terms of iterating between fidelities. Overall, DENIM addresses some of the issues presented in this section, but differs significantly from this research because DENIM is intended as a single-user website design tool that aims to support current practices rather than improving upon practices.

## 1.2 Thesis Statement

By mixing prototype fidelities in a lightweight manner, user interface designers will be able to collaboratively explore design issues earlier in the process while deferring design decisions. Composing a single prototype using any combination of low-, medium-, and high-fidelity elements will allow more flexibility in what design issues are explored at any given time. Some issues can be explored sooner in the prototyping process than possible under current practice. Also, committing to important design decisions can be deferred to a later point in time, once more urgent issues have been investigated. This approach will allow the opportunity to investigate alternate designs, including more innovative designs, as well as to involve various stakeholders prior to committing to a design.

## 1.3 Approach

This research is investigating a new approach to user interface prototyping that we refer to as Mixed-Fidelity Prototyping. By mixing fidelities, we mean combining elements of multiple fidelities into a single prototype in such a way as to allow independent refinement of individual elements. We aim for these mixed-fidelity prototypes to be composed in a lightweight manner, that is, in a flexible, informal, and easy to perform manner, requiring little or no complex initialization and coding steps.

The aim of mixing fidelities is to allow designers the opportunity to focus on a specific interface issue, refining it and moving it to lower- or higher-fidelities as needed. At the same time, design decisions on less pressing issues can be deferred. Also, the issue of interest can still be explored in context of a complete screen design since sketches and higher-fidelities can be composed together.

Through this research, we also aim to enhance the collaborative nature of prototyping. Mixing of fidelities allows individuals who have specific skills in higher-fidelities, such as in widget implementation, to get involved earlier in the design process. Also, users and other stakeholders can be elicited for feedback earlier on before final decisions must be made.

To enhance collaboration, we take advantage of a large display workspace. Previous research projects involving large display workspaces indicate benefits may exist in using large displays to support general collaborative design tasks such as brainstorming sessions [25, 22, 9, 30]. Two projects in particular, Portfolio Wall [5] and

Designers' Outpost [15], emphasize the potential of large display workspaces in supporting domain-specific design decisions. This research's large display prototyping workspace aims to leverage successful ideas from the other design workspaces.

## 1.4 Contributions

This research provides the following contributions:

- A technique for composing user interface prototypes that involves mixing multiple fidelities within a single prototype
- A conceptual model that allows for creating user interface prototypes of multiple fidelities
- A proof-of-concept system for supporting the mixed-fidelity prototyping technique

## 1.5 Outline

The remainder of this document is organized as follows. Chapter 2 discusses background and related work on prototyping techniques and tools, collaborative design, as well as large display workspace projects. The collaborative mixed-fidelity prototyping approach is described in Chapter 3, and includes a discussion of our design goals, example scenarios of designers using our approach to perform practices that are not currently possible, and facilities needed in a system to support the approach.

Chapter 4 presents a conceptual model as well as supporting notation for specifying mixing of fidelities. Then a proof-of-concept system is discussed in Chapter 5. An example design session is illustrated in Chapter 6 where the system is used to accomplish the scenarios. Finally, the document concludes with a summary of this research and areas for future work.

# Chapter 2

# Background and Related Work

This chapter discusses background and previous work relating to this research. First, the traditional user interface prototyping process is discussed, including descriptions of the different prototype fidelities. Next, tools for supporting user interface prototyping are presented, with emphasis on the strengths and weaknesses of each tool. Then, a discussion of collaborative design follows. Finally, research projects involving large displays are presented to show the benefits of large displays, particularly in supporting design decisions.

## 2.1 User Interface Prototyping Process

Under best practice, the user interface prototyping process has many steps. The first step in prototyping is to come up with a conceptual design or redesign of the interface. The second step is to develop a prototype of the new design. The third step is to evaluate the prototype, either formally or informally, depending on the situation. Based on the evaluation, these three steps should be repeated until a desirable design is achieved. Rapid prototyping encourages repeating these steps quickly, performing only small design changes at each iteration. Prototyping is very

important for ensuring the most usable, accurate, and attractive design is found for the final product.

Different fidelities of prototypes can be explored during the prototyping process: low-, medium-, and high-fidelity prototypes. Fidelity refers to how closely the prototypes resemble the final product in terms of visual appearance, interaction style, and level of detail [34]; low-fidelity prototypes differ most from the final product whereas high-fidelity ones most closely resemble the final product. Each fidelity of prototype uses different techniques and mediums and as such each is important at specific stages in the design process [29]. Commonly accepted best practice starts off with low-fidelity prototyping, moves to medium, and then to high, assuming that time and cost constraints permit.

### 2.1.1 Low-Fidelity Prototypes

Low-fidelity prototypes depict rough conceptual interface design ideas. They are traditionally paper-based prototypes, making them quick, easy, and low-cost to create and modify. Low-fidelity prototypes are sketches of static screens, presented either separately or in a series to tell a specific story, which is called storyboarding. These prototypes convey the general look and feel of the user interface as well as basic functionality. Low-fidelity prototypes are particularly well suited for understanding screen layout issues but not for navigation and interaction issues. The purpose of low-fidelity prototypes is to try out alternative design ideas while seeking frequent feedback from users and other stakeholders. Low-fidelity prototypes are best used

early in the design process when trying to understand basic user requirements and expectations [29].

### Sketching

Sketching is one of the most common techniques used in creating low-fidelity prototypes. It is a natural and low effort technique that allows for abstract ideas to be rapidly translated from a designer's conception onto a more permanent medium. Sketching is beneficial to the design process because it encourages thinking [32] and, ultimately, creativity. Sketches are also important to design because they are intentionally vague and informal, which allows for details to be later worked out without hindering the creative flow of the moment [19]. This technique also encourages contributions from users and other stakeholders since it is in a visual form that everyone is familiar with.

### PICTIVE

The other commonly used low-fidelity prototyping technique is the PICTIVE technique [20]. PICTIVE, or Plastic Interface for Collaborative Technology Initiatives through Video Exploration, is a technique for creating and modifying prototypes in real-time with users. The PICTIVE technique involves using standard office supplies such as sticky notes, labels, and plastic overlays as well as paper copies of pre-built interface components such as buttons, icons, and menus [20]. Materials are transformed through cutting and labeling to represent desired interface elements. Multiple layers of these elements are attached to the paper prototype, as needed, to

demonstrate interaction to the users. PICTIVE is a flexible technique that encourages active user participation. As the name suggests, PICTIVE prototyping sessions with the users may be videotaped and later analyzed by designers to see how the prototypes evolved and how users responded to different designs.

### Paper Versus Computer

Paper is the most common medium used in low-fidelity prototyping. Paper is an intuitive and natural medium for sketching on. Paper is cheap, easily accessible, and does not require special skills to use. Furthermore, because paper is a medium that everyone is comfortable with, it facilitates participation from users and other stakeholders. On the other hand, paper-based prototypes are hard to modify as a design evolves; designers often find themselves starting over from scratch and frequently having to re-sketch many of the same features [34]. Also, paper-based prototypes often have many smaller pieces of paper such as sticky notes and labels for representing interface components or for annotating design ideas; over time the relationships between the different pieces of paper are difficult to maintain [14]. Furthermore, storage and retrieval of paper-based designs is not very convenient or feasible. Perhaps one of the biggest drawbacks of paper is the lack of interaction that can be explored since humans have to mock up the interactivity [17].

Designers also regularly use other non-digital mediums such as whiteboards and chalkboards. Whiteboards and chalkboards share many of the same advantages and drawbacks of paper. An additional benefit is that designs on these mediums are more easily modified and reused than with paper because certain features can easily

be erased and extended upon. On the negative side, these mediums result in even less permanent prototypes because they are frequently erased. Also, whiteboards and chalkboards are less available than paper.

Using a computer for low-fidelity prototyping alleviates many of the drawbacks of using paper or other non-digital mediums. Computer-based prototypes may be much easier to evolve without requiring the designer to start over from scratch. As well, using a computer may allow for relationships between a prototype, its components, and annotations to be more easily maintained. Having prototypes on computer provides for easier storage and retrieval of prototypes, which also helps in documenting the design process and in electronic distribution to stakeholders in remote locations [17] [34]. Furthermore, using a computer for prototyping facilitates interactivity, giving users an experience that feels much closer to that of the final product. Some potential drawbacks of using a computer for low-fidelity prototyping exist: it may slow the design process, interrupt creative flow, limit designers to pre-built interface components and interactions, or force too much detail to be specified [34].

### 2.1.2 Medium-Fidelity Prototypes

Medium-fidelity prototypes lie on the continuum between low- and high-fidelity prototypes, thus sharing some of the advantages and disadvantages of the other two fidelities. Medium-fidelity prototypes are refined versions of the low-fidelity prototypes and are created on computer. They are best used after low-fidelity prototyping once only a small number of alternative designs remain under consideration and require further refinement. They resemble the end product more than low-fidelity

prototypes and require less effort than high-fidelity prototypes. Medium-fidelity prototypes are commonly created using multimedia design tools, interface builders, or certain scripting languages.

### 2.1.3 High-Fidelity Prototypes

High-fidelity prototypes allow users to interact with the prototypes as though they are the end product. High-fidelity prototypes strongly represent the final product in terms of visual appearance, interactivity, and navigation. As well, high-fidelity prototypes usually have some level of functionality implemented and may link to some sample data. High-fidelity prototypes are computer-based prototypes that are often developed using interface builders or scripting languages such as tcl/tk [24] to speed up the process. High-fidelity prototypes are particularly useful for performing user evaluations as well as for serving as a specification for developers and as a tool for marketing and stakeholder buy-in [29]. On the negative side, these prototypes are time-consuming and costly to develop. As such, high-fidelity prototypes are best used near the end of the design phase once the user interface requirements have been fully understood and a single design has been agreed upon.

### 2.1.4 Summary

User interface prototyping is a complex process with multiple fidelities of prototypes as well as several techniques and mediums for creating prototypes. This section showed the importance of using all three fidelities in prototyping since each fidelity

has its own purpose in the interface design process. Furthermore, this section emphasized the importance of maintaining low-fidelity prototyping as an informal and flexible process.

## 2.2 User Interface Prototyping Tools

Today the majority of applications are developed using some type of user interface prototyping tool such as an interface builder or a multimedia design tool. However, there are several limitations in using these types of tools that hinder the design process. In attempts to overcome some of these limitations, researchers in the academic community have been investigating informal prototyping tools such as SILK [17], DENIM [18], and Freeform [26]. While useful, these tools are not without drawbacks. The different types of interface tools will be discussed in this section, with focus being on the advantages and disadvantages of each type of tool.

### 2.2.1 Interface Builders

Interface builders are tools for creating and laying out user interfaces by allowing interface components to be dragged and placed into position on the desired window or dialog box. Some commercial examples include Microsoft®Visual Basic®, Java's NetBeans™, Borland® Delphi™, and Metrowerks™ CodeWarrior™. While interface builders are primarily intended for final product implementation, they are useful for medium- and high-fidelity prototyping. Myers [21] states that interface builders are commonly used for the following reasons: they visually represent visual

concepts such as layout, they speed up implementation by auto-generating certain code, and they are generally easy to use even for non-programmers. On the other hand, interface builders are restrictive in terms of what designs designers can build and the order in which designers have to build it. Also, interface builders require significant time and effort to create a prototype. Thus they are not suitable for early stages of prototyping when many alternate and ill-defined design concepts need to be explored.

### 2.2.2 Multimedia Design Tools

Multimedia design tools are often used in designing user interfaces, not because they are particularly well suited for software interfaces, but rather because of the lack of better prototyping-specific tools. Multimedia tools are useful in creating and demonstrating storyboards in medium-fidelity prototyping. Specifically, multimedia tools allow for creation of images that can represent user interface screens and components. They also allow for playing out transitions from one screen to the next that can convey the general picture of a user navigating through the interface screens. On the negative side, the interactivity supported by multimedia design tools is very limited, usually to only basic mouse clicks, and so is support for creating functionality and tying in data. Examples of commonly used commercial multimedia design tools are Macromedia® Director® and Flash®. Apple® HyperCard® is another commercial tool that has been widely used in the past. There have also been several multimedia tools to come out of the user interface research community including DEMAIS [1] and Anecdote [8].

### 2.2.3 SILK

SILK [17] was one of the first tools to support informal sketching of user interfaces. The purpose of SILK is to preserve the benefits of paper-based sketching while taking advantage of computerized design tools' features. The main features of SILK include support for stylus-based freehand sketching, annotation layers that allow for notes to be created and associated with specific user interface elements, and a run mode to show screen transitions. Also, SILK attempts to provide support for transitioning to higher-fidelity prototyping through automatic interface component recognition and transformation to real components; however, this feature is not suitable for low-fidelity prototyping and restricts the designer to existing toolkit components and interactions.

### 2.2.4 DENIM

DENIM [18], an extension of SILK, is a tool aimed at supporting the early stages of web design through informal sketching (see Figure 2.1). While DENIM is intended for website design, many of the features and concepts are applicable to the design of most graphical user interfaces. DENIM provides the following design features: informal pen-based sketching, zooming to different levels of granularity (from sitemap to storyboards to individual pages), linking of pages to create storyboards, a run mode to preview and interact with a prototype site, as well as supporting the creation of reusable components [18] [19]. These features make DENIM appropriate for low-

fidelity and medium-fidelity prototyping. On the other hand, DENIM provides little support for transitioning to high-fidelity prototyping.

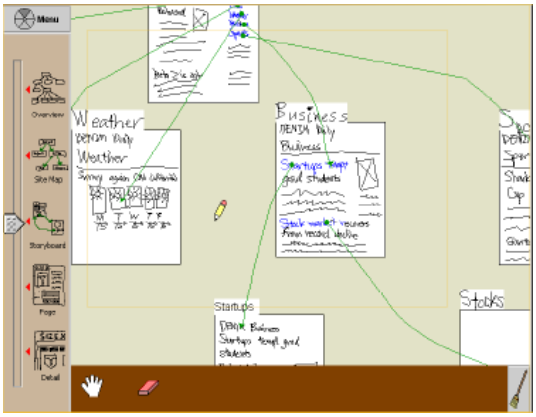


**Figure 2.1:** A snapshot of DENIM [18]

### 2.2.5 Freeform

Freeform [26] is another tool that supports sketching of user interfaces. Freeform also aims to support high-fidelity prototyping. Specifically, Freeform is a Visual Basic® add-in that translates the recognized sketched interface components and text into VB code (see Figure 2.2). However, this feature restricts the interface designer to simple forms-based interfaces that use standard Visual Basic® components. Also, the interfaces generated from sketches are not very visually appealing. Freeform is intended for use on large displays; however, there are no unique features in Freeform that make it better suited for use on large displays versus traditional desktop displays.

### 2.2.6 Summary

Each type of user interface prototyping tool discussed in this section offers some unique features or advantages to designers. However, most tools are designed for very

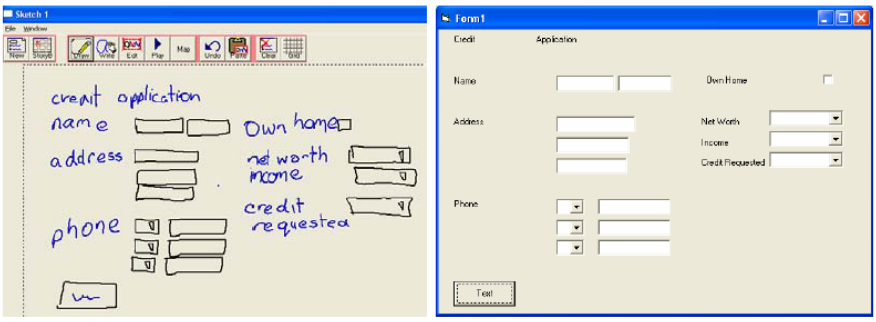


**Figure 2.2:** A snapshot of Freeform [26]

specific purposes and have very little or no support for transitioning between each of the different fidelities. None of the tools support mixing of all three fidelities within a single prototype. Also, none of the tools are specifically designed for collaborative use on a large display; however, simply using tools such as DENIM on a large display may offer some benefits over use on a traditional desktop display.

## 2.3 Collaborative Design

Software development is usually very much a collaborative process. Typically, a multidisciplinary team is involved throughout the software development process, including members such as user interface designers, software engineers, a project manager, and domain experts, who may be the end users. Core team members spend 70% of their time working with others and these team-based tasks account for approximately 85% of project costs [33].

Not surprisingly, the user interface prototyping process is also a team-based process. User interface designers work together, often with users, to come up with a desired design [20]. The team works together by brainstorming ideas and by try-

ing out the different design ideas through prototyping. The team collaborates by communicating their ideas verbally as well as through the use of workspaces such as whiteboards. The team also collaborates by coordinating efforts in building a prototype. For example, one designer may sketch parts of a prototype and another designer may expand on it, either jointly or by taking turns depending on the size of the work surfaces. Collaborative design is important in arriving at a successful design in a timely manner.

The design workspaces and, more specifically, the work surfaces influence collaboration [3] [31]. For instance, work surfaces help focus designers attention and aid in expressing creativity. Furthermore, workspaces provide a medium for designers to communicate through with actions such as drawing, writing notes, and gesturing to emphasize or reference previously made points. For these reasons, design teams must have suitable workspaces.

## 2.4 Large Display Workspaces

Large displays have become prevalent in many research projects as they show promise for supporting traditional whiteboard activities, such as brainstorming and design sessions, and they are conducive to collaboration. This section surveys some of the key research projects, with particular attention on the project goals as well as the applications, input devices, and interaction techniques being investigated.

### 2.4.1 Tivoli at Xerox PARC

The Tivoli [25] project started in the early 90's at Xerox® PARC as one of the first research projects to explore applications of large interactive displays. Tivoli is the software designed to run on Liveboard, a 46" x 32" electronic whiteboard, which was one of the first large display systems to consider input devices beyond the typical keyboard and mouse.

Specifically, Tivoli is software for supporting informal, small group meetings. It is intended to provide the functionality of traditional whiteboards while using computer technology to augment meeting practices. The project looked at supporting various scenarios of different types of meetings, from brainstorming sessions to administrative meetings.

Aside from implementing functionality to support meetings, the other main goal of Tivoli is to investigate interaction techniques for large displays. Tivoli uses a pen on the touch-sensitive surface as the input device and allows for up to three pens at a time. Tivoli supports free-hand sketching on the display. Originally, Tivoli had on-screen control panels around the edges of the display but these were later eliminated because desired buttons were often inconveniently out of the user's reach, which disrupted the sketching flow, and they did not easily support multi-user interaction. As a result, Tivoli was converted to a gesture-based command system.

### 2.4.2 iRoom's Interactive Mural at Stanford

Interactive Mural [9], a four-projector tiled display, is part of Stanford University's iRoom project. The iRoom project [9] is investigating ways of integrating various hardware including a tabletop display, ceiling-mounted scanners, laptops, and PDAs with large displays. Aside from Interactive Mural, iRoom is also set up with three 6' x 3.5' rear-projected SMART Board™ displays. The purpose of the iRoom project is to investigate team-based collaboration in a technology-enhanced environment and as such Interactive Mural is being used for collaborative applications such as brainstorming sessions and design reviews. Specifically, Interactive Mural has been experimented with in the domain of construction project management.

Various interaction techniques are being explored with Interactive Mural, all of which use a pen device for input. Interactive Mural makes use of freehand sketching for the majority of user input. It also has a handwriting recognition feature that is applied to any text-like strokes whenever the user pauses after making a stroke. Interactive Mural uses a specialized gesture-based menu system called FlowMenu [6] to allow users to fluidly invoke commands. Beyond pen-based input, these researchers plan to explore voice and vision as methods of input to Interactive Mural.

### 2.4.3 i-LAND's DynaWall at Darmstadt

DynaWall [30] is a 4.5m x 1.1m wall display, which is part of Darmstadt University's i-LAND project. i-LAND is exploring the office of the future by augmenting objects in a room; beyond the augmented wall, i-Land has computer-enhanced chairs called

CommChairs® and a table called InteracTable®. Similar to Stanford's iRoom, the i-LAND project is investigating ways of supporting collaborative meeting scenarios. i-LAND is also looking at ways of supporting individual and subgroup activities during a meeting in the same workspace in order to achieve better productivity results [30]. Overall, the goal of i-LAND is to support creativity in a dynamic and collaborative setting by taking advantage of physical objects such as chairs and their placement in a room.

Different novel interaction techniques are explored with DynaWall. Three techniques – shuffle, throw, and take – are designed as means for moving objects around on the display space. Interaction with DynaWall is mainly done using either a finger or a pen on the touch-sensitive surface (see Figure 2.3).

**Figure 2.3:** A scene from DynaWall [30]

### 2.4.4 PortfolioWall at Alias|wavefront

Alias|wavefront™, in partnership with General Motors®, is investigating the application of large displays in automotive design. The automotive design process is collaborative, creative, and traditionally involves the use of large non-electronic

workspaces. The nature of this design process gives opportunity for large interactive displays to augment the process.

PortfolioWall® is one of the large display systems Alias|wavefront™ has developed [5] (see Figure 2.4). It is designed to replace the traditional wall-mounted corkboard, where automobile designers post dozens of concept sketches for collaborative reviewing and, ultimately, for making design decisions. PortfolioWall® is a 50" plasma panel with a touch screen. It is typically interacted with through finger-based gestures.

**Figure 2.4:** A scene from PortfolioWall® (Adapted from [5])

### 2.4.5 Designers' Outpost at Berkeley

Designers' Outpost [15] is a tangible user interface for supporting early stages of website development in a collaborative setting. The web design technique that Designers' Outpost focuses on is called affinity diagramming, which is collaborative diagramming done to work out the site structure. Other stages of website design are also possible because the system is integrated with DENIM, which was discussed in Section 2.2. Website design is done using sticky notes to represent pages, as com-

monly done in practice, and these sticky notes are arranged on the large display workspace to create a site structure (see Figure 2.5).

**Figure 2.5:** A scene from Designers' Outpost [15]

Designers can interact with this system by placing actual sticky notes on the board to add pages, taking sticky notes off the board to remove pages, moving sticky notes to move pages, or sketching with a stylus to link pages or make annotations; hence it is a tangible interface. Furthermore, digital representations of these pages can be created by selecting an option on the context menu and then the digital representations may be manipulated by using physical eraser and move tools.

Designers' Outpost also has a history recording feature that allows designers to revert back to some earlier design state [16] (see Figure 2.6). It uses a graphical timeline of thumbnails of the display content at specific points in time. Designers may bookmark desired states throughout the process and then return to these states later on.

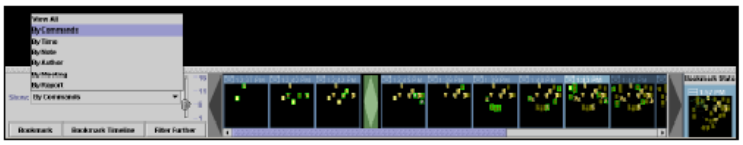

**Figure 2.6:** Recording design activity in Designers' Outpost [16]

### 2.4.6 LIDS Project at Waikato

LIDS [23], which stands for Large Interactive Display Surfaces, is a research project at University of Waikato. Some of the goals of the LIDS project is to find inexpensive display technologies and to investigate interaction issues associated with the use of such display technologies. Another goal of LIDS is to investigate specific uses for large displays such as in software development as well as in speech recognition and lecture capturing.

An example application of large displays being explored at Waikato is user interface design, which is relevant to this research. However, the tool they have created, Freeform (discussed in Section 2.2.5), does not have any design considerations that make it specialized for use on a large display as opposed to a traditional desktop display. Specifically, user interface sketching on a large display and automatic translation into Visual Basic® is being investigated. Freeform is run on a rear-projected display system, which is interacted with by sketching and gesturing using an ultrasonic pen.

### 2.4.7 Summary

Most research involving large displays has focused on investigating hardware integration issues, interaction techniques, or input devices. Little research has actually looked at practical uses for these large displays. Those that are looking at specific applications or uses are typically focused on augmenting traditional whiteboard and meeting tasks, like Tivoli and Interactive Mural. Two projects, PortfolioWall® and

Designers' Outpost, are investigating supporting domain-specific design tasks and these projects have motivated some of the ideas in this large display workspace, including recording history and comparative analysis. Perhaps one of the projects most closely related to this research is LIDS/Freeform since it is investigating user interface sketching on a large display; however, Freeform does not have any design considerations that make it better suited for a large display than a regular-sized display, it only supports the standard forms-based interface components and interactions, it is not much more collaborative than a desktop application other than more people can watch the designer work, and it does not allow for mixing fidelities or iterative refinements.

## 2.5 Conclusion

All three prototype fidelities have unique benefits and are important to include in the process. Existing tools inadequately support traditional prototyping practices, as they only fully support one or two fidelities, do not support refinement back and forth between fidelities, do not allow for multiple alternative designs to be compared at once, and most do not support easy tie in to data and functionality. Also, existing tools lack support for collaboration even though collaboration is important to the process. One tool, Freeform, claims to be collaborative since it may be run on a large display, but it offers no collaborative features other than simply enlarging a design for more people to watch; only one user can use the tool and only one design can be displayed at once. Also, Freeform makes no attempts to bring together

other groups working on the software. Large display research projects show promise for supporting collaborative activities as well as specific design activities such as comparing alternatives (PortfolioWall®) and high-level storyboarding (Designers' Outpost).

The next chapter describes an approach that addresses integrating multiple fidelities in a collaborative setting.

# Chapter 3

# Mixed-Fidelity Prototyping

We developed a new technique for user interface prototyping called mixed-fidelity prototyping and this technique will be described throughout this chapter. Mixed-fidelity prototyping involves combining multiple fidelities within a single prototype. As an example, consider having a sketched screen design that contains various sketched elements and images of elements. The sketch may also not only contain images, but could also have one or more interface elements presented as high-fidelity working components. The sketched elements may also be given some form of behavior similar to what they would possess at traditional higher-fidelities.

The idea of mixed-fidelity prototyping allows the designers opportunity to focus on a specific interface issue, by exploring it at higher-fidelities and making refinements as needed. In the mean time, other aspects of the prototype may be left at a lower-fidelity, delaying decisions while allowing designers to redirect their time and efforts to the more pressing design issue(s). Also, by leaving other elements at lower-fidelities, designers are able to explore the higher-fidelity elements while keeping them within the context of more complete screen designs.

Prototyping is collaborative by nature, ideally involving individuals from diverse backgrounds such as user interface designers, graphic artists, software developers, and end users. Mixing fidelities allows various stakeholder groups to come together earlier on and allows for their more active participation. For example, an individual with specialized skills in creating high-fidelity prototypes, such as a software developer, can begin implementing a high-fidelity component for inclusion in the prototype at an early stage.

A secondary aspect of this research is to further enhance the collaborative nature of user interface prototyping by utilizing a large interactive display workspace. Large display workspaces are conducive to collaboration as the large surface helps focus designers' attention, aids in expressing creativity, as well as provides a medium for designers to communicate through actions such as drawing, annotating, and gesturing [3, 31]. Also, such displays are large enough for multiple people to simultaneously work directly at the surface while allowing for everyone in the room to be aware of the workspace content.

Through this research, a conceptual model is developed to specify how prototypes can be composed of multiple fidelities. This is the topic of Chapter 4. Then a proof-of-concept system is implemented based on the conceptual model to demonstrate the idea of mixed-fidelity prototyping. This proof-of-concept system is presented in Chapter 5. Then in Chapter 6, an example design session is presented where the proof-of-concept system is used to better illustrate the mixed-fidelity technique.

Throughout this chapter, the novel technique of mixed-fidelity prototyping is further discussed. Specifically, major design goals for the mixed-fidelity technique

are presented. As well, example design scenarios are presented to illustrate what mixed-fidelity prototyping is and how it can aid designers beyond traditional practice. Finally, features, or more generally, facilities, that are needed to support mixing of fidelities are presented.

## 3.1 Design Goals

Several major design goals for the mixed-fidelity approach are identified. Each design goal is listed and described in detail in this section. The scenarios and supporting facilities in the following sections were then developed with these goals in mind.

These are the main design goals:

- Support traditional informal design activities
- Aid in transitioning between the different fidelities
- Enhance the design process by recording history
- Support collaborative design across diverse stakeholder groups
- Provide a lightweight environment

### 3.1.1 Support traditional informal design activities

Current practice often involves informal whiteboard or paper-based prototyping activities in the early stages, such as sketching and PICTIVE [20]. Such informal design techniques should be encouraged as designers are comfortable using them and these techniques have been very successful in practice. Also, these techniques are

very important for quickly narrowing down alternative designs and gaining a better understanding of user requirements.

### 3.1.2 Aid in transitioning between the different fidelities

Each prototype fidelity has its purpose and place in the design process so all of the fidelities should be encouraged. Current user interface prototyping tools rarely support transitioning from one fidelity to the next, so much time and effort is lost in porting each fidelity of prototype to a different medium or tool. This often results in certain prototyping stages being skipped, which may sacrifice quality of the final product. Also, current tools fail at supporting backward transitioning from a higher-fidelity to a lower-fidelity design; when changes are needed, designers are often forced to start from scratch rather than refining that design. Smooth transitioning between the fidelities promotes changeability and traceability and ultimately allows designers to move between fidelities without loss of productivity.

### 3.1.3 Enhance the design process by recording history

Early prototyping stages are traditionally performed on non-digital media such as whiteboards and paper while later stages take place on various computers and/or software tools. Under current practice, there is no way to track the progression of the design process or to revert back in history to a specific point in time. By using a large digital display throughout the prototyping process, the history of the process can easily be recorded. Recording history would allow for easy storage and retrieval

of previous prototypes and prototype elements at any point in time as well as for design rationale and other annotations to be permanently linked to prototypes for later viewing.

### 3.1.4 Support collaborative design across diverse stakeholder groups

The nature of prototyping is very much a collaborative process that should involve user interface designers as well as other stakeholders, like users and software engineers, working together at various stages throughout the process. However, current practice has user interface designers working largely independently, with little communication or coordination between stakeholders until the design process is completed or at least only the final decisions remain. Also, within the groups, people commonly work in isolation on their individual machines, only coming together for scheduled project meetings. This lack of collaboration is not in the best interest of the final product, as collaboration is important to generating more creative designs as well as designs that are more consistent and accurate in meeting the requirements. At a very minimum, there needs to be awareness and communication of the major design concepts across the different groups. Large displays are conducive to collaboration and may help augment user interface prototyping by allowing multiple users to watch or participate in the design process simultaneously.

### 3.1.5 Provide a lightweight environment

For a new workspace or system to be considered useful and eventually be adopted as part of regular practice in an organization, it needs to be easy to use. This includes being easy to learn and to get started with each session. Designers need to be able to walk into the room and quickly start using the workspace, without inconvenient technical or cognitive challenges. Also, the workspace needs to be able to maintain the state of recent uses, so that each time designers begin a new session, there is not significant overhead in continuing to work where they left off. Furthermore, the workspace needs to offer an easy way to access all available designs and design elements, including ones from past projects.

## 3.2 An Example Application and Its Domain

This section introduces a business forecasting tool that will be used as an example application that designers are working on. Specifically, this is the software that will be referred to throughout the scenarios in this chapter and will also be used as the example application in Chapter 6. Businesses have the important task of forecasting their business, that is, making best estimates of their revenue and expenses and ultimately profit for some future period. Accurate forecasting is important since key business decisions are made based on the expected future conditions. Well-managed businesses perform forecasting at some regular interval, such as annually or quarterly,

and whenever a factor affecting the business changes, such as when a major supplier raises prices.

Forecasting involves estimating a number of factors that affect the business. For example, factors that affect revenue include selling price and unit sales, which is affected by many other sub factors like market size and advertising success rates. Expenses are affected by factors such as advertising costs, payroll, and raw material costs. One major factor that directly affects profit is tax rates. The exact factors that affect a business depend on the type of business and the environment that it operates within.

With so many factors and so many possible values for each factor, business forecasting is a complicated task that is infeasible without software support. Many software packages exist on the market to support forecasting efforts. Better designed user interfaces for forecasting offer the user a more visual approach; for example, showing charts is more effective than only showing tables of data. Also, the best designed software will allow for users to experiment with different situations and have the effects visualized on the fly; that is what this example forecasting tool offers.

Figure 3.1 shows a sketch of a possible interface design for the example business forecasting tool. Let us walk through how one would use this software. When the user starts the forecasting tool, the first step would be to select a desired forecasting period. Then the user would select the desired level of detail from these view options: high-level 'Chart View', medium-level 'Financial Statement View', or finer-grained 'Accounting View'. The interface for both the 'Chart View' and 'Financial Statement View' are shown in Figure 3.1. In the 'Financial Statement View' screen, the user

selects the type of statement to view and the screen simply displays the desired statement for that period using the factors' values specified in the 'Chart View' or from the underlying database. Using the 'Chart View' is more complicated and requires a detailed explanation.

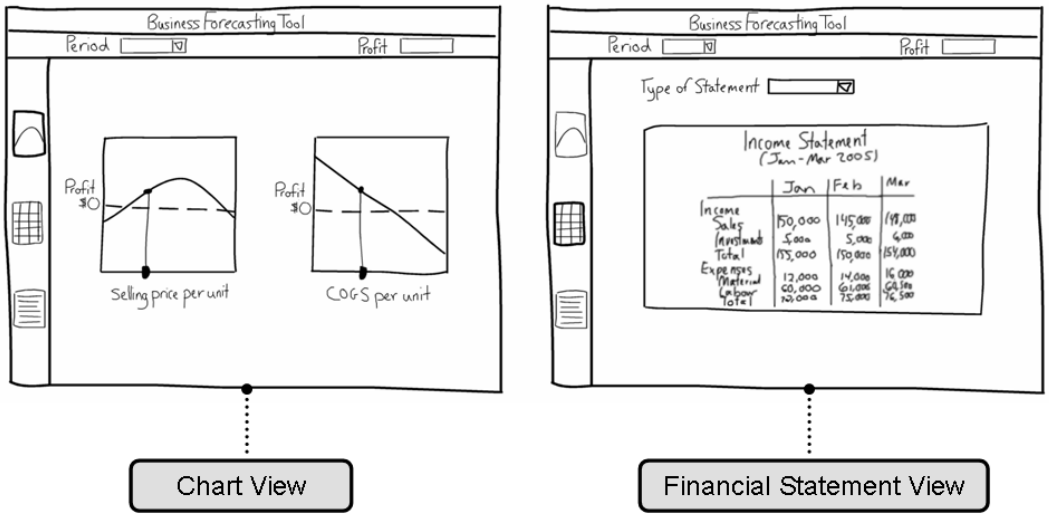


**Figure 3.1:** A sketch of the business forecasting tool

The 'Chart View' is where the user can play through different scenarios to see the effects different factors have on the overall business profitability. This interface design allows the user to adjust two key factors: selling price per unit, which affects total revenues, and cost of good sold (COGS) per unit, which affects total expenses. This design has been simplified for demonstrational purposes so it only allows for two factors to be considered; however, the design could easily be extended by including a list of factors to select from for both charts or by having a chart for each possible factor.

Manipulating one chart affects the visualization of the other chart, as well as the data contained in each financial statement. For example, assume the user is inter-

ested in visualizing how setting different selling prices affect profitability. The user would move the slider along the x-axis labeled 'selling price per unit' to try different prices. Moving the slider also moves a vertical guide bar left or right accordingly and highlights the point on the chart that the slider value corresponds to. Furthermore, this interaction affects the other chart; changing the 'selling price per unit' affects the profit and thus shifts the 'COGS' chart's curve up or down accordingly. Adjusting the slider also affects the underlying financial data in some database; the new values will be used in calculating information that is then displayed in the 'Financial Statement View' and in the 'profit' field in the top right corner of the screen.

This particular user interface for the example business forecasting tool was used because it encompasses many interesting concepts that are important in the scenarios and ultimately in demonstrating this research. It has novel interactions – the charts' axes have sliders to specify values of interest as well as a vertical guide bar to emphasize the currently selected value. It links to underlying financial data. It has domain-specific functionality, such as functionality for calculating the business's profitability and the demand elasticity function (that is, a function of the number of unit sales versus selling price). Furthermore, it has visual elements that are linked; changing one chart affects the other in real-time. The business forecasting application was chosen for illustrative purposes only and our technique is not intended to be specific to this domain.

## 3.3 Scenarios

This section will walk the reader through six collaborative design scenarios that are envisioned to be possible through this research. Each of the scenarios depicts a novel or key concept and attention is drawn to the advantages of supporting such a concept. While some of these scenarios may not be consistent with all current practices, these scenarios suggest potentially more effective ways of performing collaborative prototyping.

For all scenarios, assume that a large display workspace is being used. Also, assume that the design team is working on prototyping the business forecasting software described in the previous section. Illustrations of key concepts will be referred to throughout the scenario discussions to clearly demonstrate the new design practices envisioned with this research.

### 3.3.1 Scenario 1: Composing lightweight prototypes by mixing fidelities

Figure 3.2 shows the prototypes that designers are composing by combining medium- and high-fidelity elements with low-fidelity sketches. As with current best practice, user interface designers started by sketching low-fidelity designs either on paper that is then scanned in or by sketching directly on the large display. The designers are confident that aspects of this design are worth pursuing so they prototype these aspects at higher-fidelities.

They have a software developer implement a high-fidelity table component since they are confident it will be used in the final user interface. The table component is then linked to an existing database to display real data; using real or at least realistic data gives the user a more accurate picture of the final software and ensures that the software functions as desired by the user.

Also, the designers quickly create images of the two charts; these images give a more concrete idea of the final design than the sketched charts. The designers need to refine the design of the interface components, specifically by exploring novel interactions, before going ahead with high-fidelity implementations. Since these charts do not exist in standard toolkits there is significant overhead associated with developing these components.

The elements of different fidelities may then be used to compose a single prototype. The designers do this by placing the medium-fidelity charts on top of the sketched charts. They also place the high-fidelity table on top of the sketched version. Figure 3.2 illustrates the resulting prototype. At this point, they are ready to show the design concepts to stakeholders and elicit feedback for further design refinements.

This scenario presents a style of prototyping that is not possible under current practice with today's tools. Specifically, this design style supported by our approach allows for design decisions to be deferred, such as layout, while allowing the designers to focus on more high-risk issues such as designing new components and interactions that do not exist in standard toolkits, or to focus on design concepts that are more pressing from the user's perspective. Ultimately, by mixing fidelities, complex or

urgent design issues may be explored and refined sooner, while leaving other issues until later.

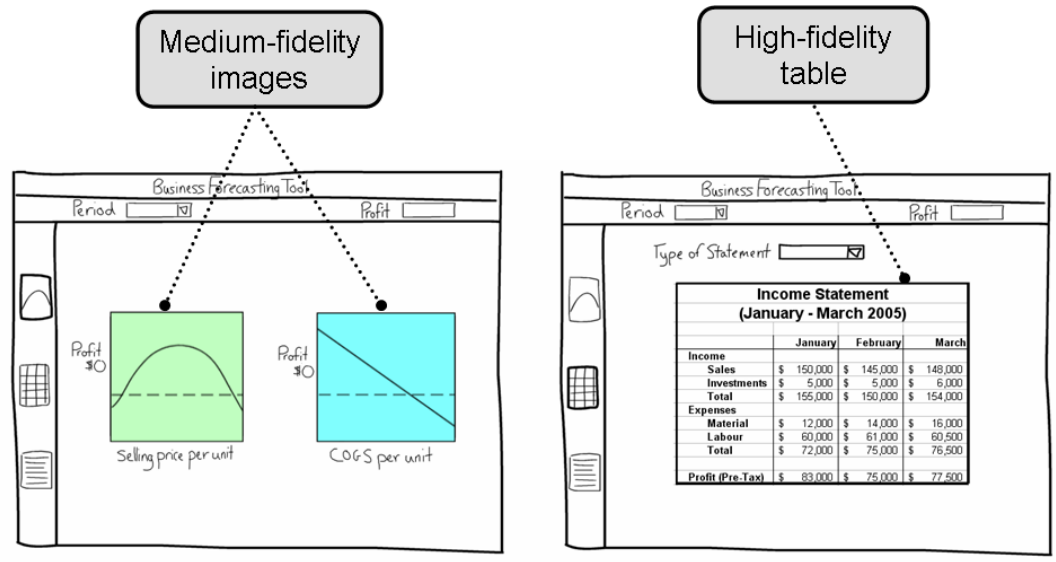


**Figure 3.2:** Mixing fidelities through layering of elements

### 3.3.2 Scenario 2: Transitioning between fidelities as ideas are refined

Continuing on where Scenario 1 left off, assume that the prototypes composed of multiple fidelities were presented to users for feedback. The users have made some valuable suggestions that require modification to the current prototype. One suggestion involves changing the appearance of the charts. Rather than starting from scratch, the designers can hide or turn off the medium-fidelity images of the charts, showing only the original sketched charts. The designers now erase parts of the sketched charts and make revisions to the charts by drawing directly on the display. They then modify the images to correspond to the new sketch and replace the old

images associated with the charts. The designers continue to refine the rest of the design by placing higher-fidelity elements on top of previous lower-fidelity elements.

Current practice does not allow for easy transitioning between designs. In particular it is very difficult to move to an earlier fidelity prototype without starting completely from scratch, since different fidelities are usually created using different tools and/or mediums. This scenario shows that by having all designs stored in a single system, smooth transitioning between fidelities is possible. Ultimately, prototypes can be refined in a more productive manner.

### 3.3.3 Scenario 3: Integrating domain-specific data and functionality

Continuing where the previous scenario left off, the user interface design team has composed a lightweight prototype consisting of sketches with images of the charts as well as a high-fidelity table component displaying real data (as shown in Figure 3.2). The designers are now ready to add some functionality to the prototype. They want to display the calculated profit in the top right text field. Since this design team specializes in business information systems for the financial and accounting domains, they already have a function to calculate profitability built-in to their design environment. They simply open the appropriate function in the 'Functionality Window' and then modify it to link into this application's data and set the result to be displayed in the 'Profit' text field (see Figure 3.3). They have other built-in

domain specific functionality that will come in handy for this project such as the demand-elasticity function and chart components.

Integrating pre-built domain-specific functionality benefits the prototyping process. It helps speed up the process since designers do not having to wait on developers to implement a piece of functionality; instead designers can add in the functionality quickly and easily with only minor modifications to code. It also allows stakeholders to get a better appreciation for what the final software will be; rather than just showing sketched functionality, the prototypes can display accurate, calculated values based on real data.

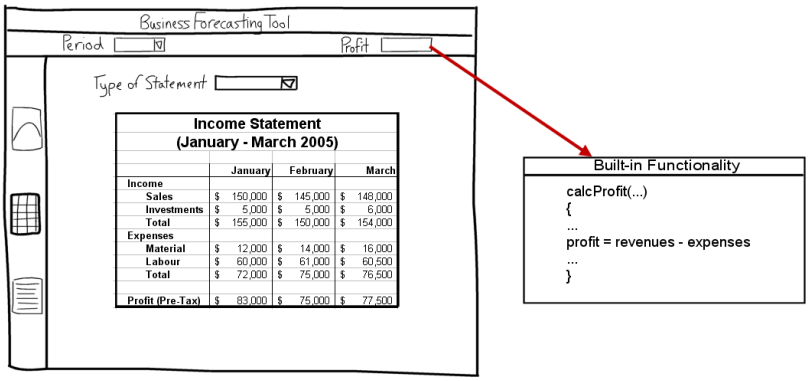


**Figure 3.3:** Prototype with built-in domain-specific functionality and real data

### 3.3.4 Scenario 4: Trying out novel interactive elements

After meeting with the users (in Scenario 2), the user interface design team gained a better understanding for the type of interactivity the users would like to experience with the charts. The users would like the x-axis of the chart to behave like a slider bar; the user can then slide through different x-values and have the effects on the

other chart visualized immediately by shifting the curve. They also want additional visual feedback to emphasize the current x-value such as having a vertical bar. Such a design for charts is unique and is not implemented in any standard component toolkit.

The designers start out prototyping the chart component by creating images that properly illustrate a snapshot of each chart at some instance, with selling price at X1 for example (shown in Figure 3.4). Then the designers create another image for each chart illustrating the resulting effects of moving a slider to a new value, X2 (shown in Figure 3.4). They also create a few pairs of images of intermediate charts, that is, what the charts look like as the user adjusts the slider at intermediate points between X1 and X2. They place all these images in sequential order on top of the sketched charts. The prototyping system then has a sequencing feature that allows for these images to be flashed in order over a specified interval of time, emulating the effects of a user actually moving the slider from X1 to X2.

The designers now want to transition the chart into a higher-fidelity but in a manner more lightweight than implementing a final product interface component. The designers mock up the chart by using pre-built standard toolkit line-chart and slider components. They place the chart in position and then lay the slider on top of the chart's x-axis. Then they create a graphical line object to represent the slider's vertical guide bar, as depicted in Figure 3.4. The design system allows for behavior to be added to objects, so the designers specify for the graphical line to redraw itself at the same x-coordinate as the slider is positioned at. The other chart's curve is linked to the slider's current value so it updates when the slider is adjusted. The

combination of these three elements generates a lightweight working version of the novel chart component.

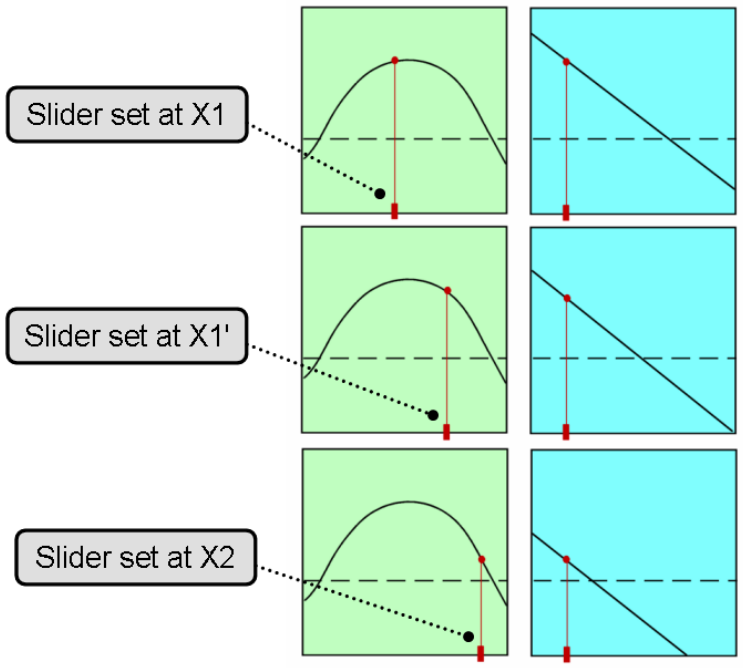


**Figure 3.4:** Prototyping a novel chart element

The lightweight high-fidelity approach presented here is not easily possible under current practice with current tools. This scenario presented a unique way for exploring new interactive elements that is much quicker than implementing a fully functional interface component and allows for new ideas to be experimented with and shown to users more quickly without over-committing to the design. Such a lightweight approach fosters the creative flow and is conducive to exploring new interactions, unlike current tools that restrict interface designers to using existing toolkit components in traditional ways.

### 3.3.5 Scenario 5: Comparative analysis of alternative designs

In this scenario, the design team is performing comparative analysis among multiple design alternatives. The designers have narrowed down alternative design layouts to the two shown in Figure 3.5. Alternative #1 has the 'Chart View' and the 'Financial Statement View' on the same screen, while Alternative #2 has each of the views separated into their own screens with a narrow vertical navigational bar to switch between the views. The designers are discussing the advantages and disadvantages of each design; Alternative #1 has the two views visible at once so navigation is not required, yet Alternative #2 separates the views which abstracts the financial statement details away and lets the user work with high-level charts. The designers make annotations directly on the designs to keep record of the stated advantages and disadvantages.

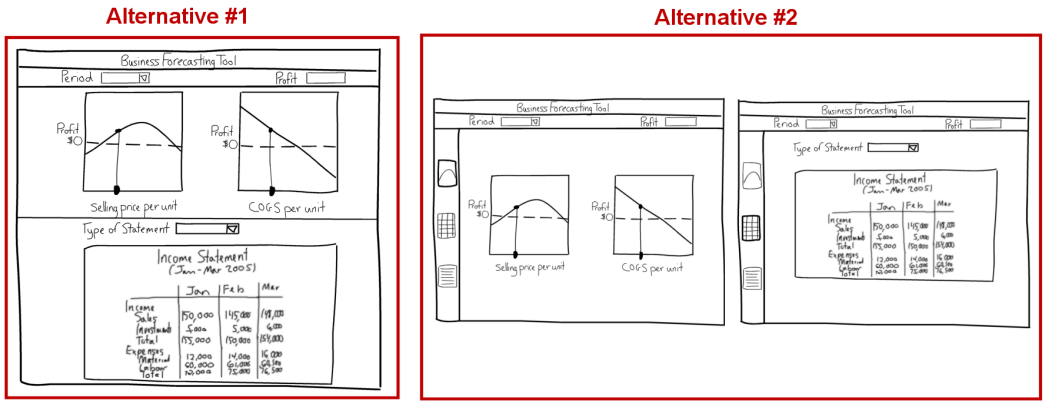


**Figure 3.5:** Exploring and comparing two alternate designs

Current practice involves the use of regular-sized displays, where it is very hard to lay out multiple designs simultaneously. Thus, this scenario cannot be performed effectively under current practice. Using a large display allows for multiple designs to be arranged on the display at the same time, which aids in the decision making, as Buxton [5] found in comparing and selecting among automotive designs. Also, having a large interactive display allows designers to annotate the designs and store the annotations with the designs for future reference.

### 3.3.6 Scenario 6: Recording collaborative design efforts

Later, the project is at the stage of implementing the final version of the user interface. A software developer is working on the full implementation of the chart components, which have a slider-like feature as well as a vertical guide bar. Again, no such standard component exists already. In order to get a thorough understanding of what this chart does and how it evolved into the current chosen design, the developer finds it very useful to look back at all of the versions of the chart. The developer uses a historical timeline feature to look back at previous versions of the chart over the projects' entire history.

Reverting back to previous prototypes is only possible with a mechanism for recording history of designer activities. Such recording does not exist in current practice since designers use multiple design tools and mediums throughout the prototyping process. Recording activities is important to save time and effort whenever a previous design needs to be reverted back to. It is also important when designers want to look at a previous project for design motivations. Furthermore, recording

is important for introducing better traceability into the prototyping and final implementation processes; under current practice there is a lack of traceability among designs, again because of the various tools and mediums that are used.

## 3.4 Supporting Facilities

In order to support the design goals and new design styles and practices set out in the scenarios in the previous sections, software tool support is necessary. The tool must offer certain built-in facilities to the design team. This section presents such facilities, with focus placed on the motivation or importance of each facility. Many of these facilities are implemented into a proof-of-concept system, which is the topic of Chapter 5.

The supporting facilities have been grouped according to these categories:

- Drawing Editor Facilities
- Organizational Facilities
- Prototyping Domain Facilities
- Multi-User Support Facilities
- Large Display-Specific Facilities
- Process/Productivity Enhancing Facilities

### 3.4.1 Drawing Editor Facilities

***Scaling*** is a basic facility built into all drawing editors. Being able to resize objects on the display is important for two reasons. In creating prototypes, designers need to be able to size interface components and regions relative to the rest of the prototype. Ideally, a smart scaling facility is needed to size interface elements to fit within a predefined region. Secondly, all objects displayed on the screen need to be able to be zoomed to different sizes depending on users' proximity to the display.

***Positioning*** is another facility fundamental to drawing editors. Users of the prototyping workspace need the ability to position objects wherever they desire on the screen, whether it be positioned some distance away from other objects or overlapping with other objects. Also, a smart positioning facility would be useful to layout interface elements accurately within a predefined region. X and Y coordinates are relevant as well as is the concept of a Z coordinate, since layering or overlapping of objects is possible.

A ***cloning*** facility is generally built into drawing editors. A clone is a duplicate copy of another object. In this large display workspace, modifying the original object could affect the cloned objects as well. However, deleting the original would not delete any clones. Cloning is particularly relevant to the domain of prototyping when several interface screens need to be created that share many of the same elements and layouts.

***Annotating***, or freehand sketching of text or markings, is also commonly found in drawing editors. This research recognizes the importance of traditional informal

whiteboard and paper-based activities, with one of the key advantages of these media being that content can easily be annotated. For example, an initial prototype can be created and then key elements of it may be easily annotated with textual justification for why it is a good design and what needs to be improved upon.

### 3.4.2 Organizational Facilities

***Grouping*** offers an easy way for users to keep related content together on the display. This facility is the same as the "group" feature built into most Microsoft® products such as Microsoft® PowerPoint®. Specifically, grouping is important for composing interface elements to create a prototype; grouping holds the elements together in the same relative positions and sizes whenever the user repositions or scales the whole prototype.

***Structuring*** allows for objects to be arranged in a structured or predefined manner often according to semantics. For example, objects may need to be arranged in a list, a hierarchy, a chart, or a storyboard sequence. The arrangement of objects is visual and indicates to the user a certain relationship between the objects.

***Relating objects*** is important for the designers to quickly be able to understand what objects are related and why. For example, one element on a screen may be connected to a different element on another screen because interacting with one element affects the other. By allowing for relationship arrows or some other visual feedback (such as highlighting) to be created to relate the objects, designers may be better able to understand the design concepts and improve upon them where needed.

Visually depicting relationships is particularly useful when using a large display since a large number of objects are being viewed at once.

***Highlighting and selection*** is important for designers to be able to quickly locate all objects of interest on the display. Having highlighting or some other visual feedback to emphasize specific objects is crucial on a large display since it is possible for dozens or even hundreds of objects to be displayed at once. Without highlighting, users may get lost in the space. It is necessary to draw designers' attention to particular objects when one designer has selected an object and other designers need to see what object is being referred to, when a designer is interested in finding all cloned versions or objects related to a particular object, or when a designer performs a query and wants the results presented visually through highlighting the relevant objects on the display.

### 3.4.3 Prototyping Domain Facilities

***Layering and transparency*** is key to composing interface elements of different fidelities within a single prototype. Layering allows for elements to be positioned over top of other objects. Being able to change the order of layers allows for a desired layer to be made visible. Setting a transparency value for each layer allows the designer to control what layers are fully or partially visible. For example, a designer may be interested in seeing the faint low-fidelity sketch under the higher-fidelity component to ensure no key ideas were omitted in moving to a higher-fidelity.

***Running executable components*** is very relevant to the domain of prototyping. When designers are creating prototypes they need to be able to work with

high-fidelity components and this includes being able to execute the components. Ideally, designers should be able to execute components without frames or other obtrusive borders so that it looks unified within the prototype.

***Connecting to real data*** is also important in the prototyping domain. When designers are creating higher-fidelity interface elements, it is useful to put data into the elements. Using existing or at least realistic data gives stakeholders a better idea of what the final software will look like and be capable of. The data may be in a database or some other storage such as XML files.

***Setting properties and behaviors of objects*** is very important to the prototyping domain. Designers need to be able to create interface elements that are easily customizable by simply changing properties' values. Designers also need to be able to give elements behavior in a lightweight manner. Behavior may be needed in the form of animation, that is having the object move or change forms at specific time periods, or performing calculations. Furthermore, properties or behaviors of objects may need to be able to change in relation to another object's properties or behaviors as desired.

### 3.4.4 Multi-User Support Facilities

***Multiple inputs*** is required in order for more than one user to interact with the system simultaneously. Multiple inputs means providing a unique pointer for each user who is currently interacting with the system. This is necessary so that one user's pointer does not interfere with another's.

Having a ***control panel for each user*** is useful in allowing multiple users to interact with the system. With separate control panels that appear near each user's point of focus, that is in their area of work, each user can work separately and simultaneously. This is a metaphor for a worker's toolbox that the worker can carry around the work site. Having individual control panels adds to the usability of the system. Also, using control panels at the point of focus rather than a menu bar at one edge of the screen as in standard desktop applications reduces the reaching required and allows the users to fluidly select a command from the control panel.

***Integration of other computing devices*** with the large display system allows for multiple users to work on the workspace in a productive manner. Integrating devices such as laptops, PDAs, and cameras allows for users to work with a variety of devices at once. For example, a couple of designers can be composing prototypes standing up at the large display using the touch-sensitive surface, while other designers are working at a table creating interface elements on their laptop or PDAs while others sketch on paper and take a digital picture of the sketch. The workspace should offer smooth transitioning of designs between devices, perhaps through overview windows.

### 3.4.5 Large Display-Specific Facilities

The ***Pick-and-Drop technique***[27] improves interaction with a large display in terms of moving objects around on the display across large distances. In particular, Pick-and-Drop is designed to replace traditional drag-and-drop by allowing the user to simply select an object and walk to the target location without any drag motion,

rather than forcing the user to carefully drag the object with a constant pressure while walking to the other side of the display. Pick-and-Drop also supports the use of multiple tiled displays whereas traditional drag-and-drop cannot.

***Other interaction techniques*** such as ***gestures*** and ***specialized menus*** are needed for issuing commands to the system. These techniques are well suited for a large display for these primary reasons: they allow for fluid interaction since they can be interacted with at the user's point of focus rather than at a hard to reach location, and there is no wasting of display real estate because they only appear upon user-specified invocation. The standard desktop way of issuing commands is by pressing a button on a menu bar or palette; this approach would not be usable on a large surface.

***Flexible space management*** is necessary to take advantage of the display real-estate in a productive manner, while realizing the limitations that no display will ever have enough space. To deal with the limited space, the system could implement a concept of UNIX's Virtual Rooms [11] that would allow for multiple workspaces but only one full-sized at once. Also, a layout algorithm is needed to effectively and automatically arrange design objects on the display, particularly when a portfolio of objects is first loaded. Furthermore, to allow users to more easily arrange objects as they work, grid guidelines may be necessary to improve manual layout efforts, perhaps with a snap-to-grid feature. Another idea for helping to manage space is to have regions or containers, much like folders in the desktop metaphor, that automatically perform different layouts, scalings, or other functions depending on which region or container the objects were put into.

### 3.4.6 Process/Productivity Enhancing Facilities

***Logging design activities*** provides enhancement over the traditional design practice, where there is very little traceability between designs and reverting back to a previous design is a difficult task. By logging all activities on the large display, designers will then be able to revert back in time to a particular design state. This allows for designers to review previous designs to inspire new design ideas and more importantly, allows opportunity to modify previous prototypes without having to start from scratch. Offering a visual timeline, like in Designers' Outpost [16], will improve the design process and ultimately designers' productivity.

A ***querying library*** will improve the process by reducing the time users currently have to spend looking for existing design artifacts while offering an easy to use query facility. Currently, previous designs, particularly from previous projects, are rarely referenced because of poor storage and retrieval practices. Paper-based designs are difficult to index and so are electronic files that are used across multiple computers by multiple users. The system can be enhanced by offering an easy way of indexing design artifacts and by offering an effective querying facility. Since interface design artifacts are visual by nature, it would be interesting to build in a visual querying facility, to allow searching for all prototypes that have sliders for example.

***Importing and exporting resources*** is a basic facility for the prototyping system to include. Allowing users to import resources from other projects, from other computing devices, or from other mediums such as paper is key to enhancing the process. Requiring all designs to be started from scratch on the large display

system is time-consuming and costly, whereas allowing designs to be reused or started on other devices or mediums is much more productive. Many more designers can work on a project than are able to fit standing at the large display. Also, with an easy-to-use import facility, sketching-support is no longer an essential feature of the prototyping system, allowing users to continue to use the natural feeling paper medium. Furthermore, the system needs to allow for designs to be exported for future reference.

## 3.5 Mapping Facilities to Scenarios

Table 3.1 shows which facilities help support each scenario. The facilities that are essential to achieving a particular scenario are indicated with a black circle. The facilities that are helpful in a particular scenario, but are not key to its success, are indicated with a white circle. Facilities without any circle are not considered particularly advantageous to that scenario but may be important to improving the overall usability and usefulness of the system.

## 3.6 Conclusion

Mixed-fidelity prototyping is a new approach to user interface prototyping. It offers enhancements over the current process by allowing designers opportunity to explore and refine certain issues earlier on as well as to involve individuals of different backgrounds earlier in the process. The five design goals discussed in this chapter are important aspects of the new approach. The design scenarios presented new styles

or practices envisioned as possible with mixed-fidelity prototyping. A software tool is necessary to support the mixed-fidelity approach and it should implement the facilities set out in this chapter, as each facility helps to achieve the design goals and scenarios.

Prior to implementing a software tool, it is necessary to formulate a conceptual model to specify how fidelities can be mixed. This conceptual model is the topic of the following chapter.

**Table 3.1:** Mapping facilities to scenarios

| | 1. Combining Fidelities | 2. Iterating Between Fidelities | 3. Domain-Specific Functionality | 4. Explore Novel Design Concepts | 5. Comparative Analysis | 6. Recording History |
|---|---|---|---|---|---|---|
| **Drawing Editor Facilities** | | | | | | |
| Scaling | ○ | | | ○ | ○ | |
| Positioning | ○ | | | ○ | ○ | |
| Cloning | ○ | | | | | |
| Annotating | | | | | ● | |
| **Organizational Facilities** | | | | | | |
| Grouping | ○ | ○ | | | | |
| Structuring | ○ | | | ○ | ○ | |
| Relating objects | | | | | | |
| Highlighting and selection | | | | ● | ● | |
| **Prototyping Domain Facilities** | | | | | | |
| Layering and transparency | ● | ● | | | | |
| Running components | ● | | ● | ● | | |
| Connecting to data | ● | | ● | ○ | | |
| Setting properties and behavior | ● | | ● | ● | | |
| **Multi-User Support Facilities** | | | | | | |
| Multiple inputs | | | | | | |
| User control panels | | | | | | |
| Integration of devices | | | | | | |
| **Large Display-Specific Facilities** | | | | | | |
| Pick-and-drop | | | | | | |
| Gestures and specialized menus | | | | | | |
| Space management | | | | | ○ | |
| **Process Enhancing Facilities** | | | | | | |
| Logging activities | | ○ | | | | ● |
| Querying library | | | | | | |
| Import and exporting | ○ | | ○ | | ○ | |

(Note: ● = key to achieving the scenario, ○ = helpful but not essential to the scenario)

# Chapter 4

# The Region Model for Mixed-Fidelity Prototyping

This chapter presents a conceptual model we developed to support mixed-fidelity prototyping. In creating such a model, it is important to identify the elements that are necessary for mixed-fidelity prototyping and how these elements are related. Designers need to be able to compose elements of different fidelities and these elements should be interchangeable as designs are refined or revisited. Also, this composition process needs to be as lightweight as possible in keeping with the traditional informal sketch-based approach. A model called the Region Model is developed to address these requirements.

The highest-level concept in the Region Model is *design space*. The design space represents the entire workspace that designers have to work with and it is where prototypes are composed. Prototypes are composed of multiple elements, often being of multiple fidelities. As these elements need to be interchangeable, all elements are represented as *regions*. Furthermore, the prototype screen designs are also represented as regions. In that way, prototypes can be as high-level as a single sketch

or as fine-grained as a prototype composed of several elements, with its elements also containing sub-elements. Thus, regions are used to represent every object in the design space.

To further build on the region concept, the overall design space is also represented as a region, and is referred to as the Root region. All other regions in the design space are then linked either directly or indirectly to the Root region in a parent-child relationship.

Prototypes are composed using the region metaphor by overlaying regions on other regions to arrive at a desired design. For example, the interface screen is a region and elements laid out within the interface screen such as widgets or images are also smaller regions overlaid within the screen region. Figure 4.1 illustrates how regions can be used to compose prototypes.

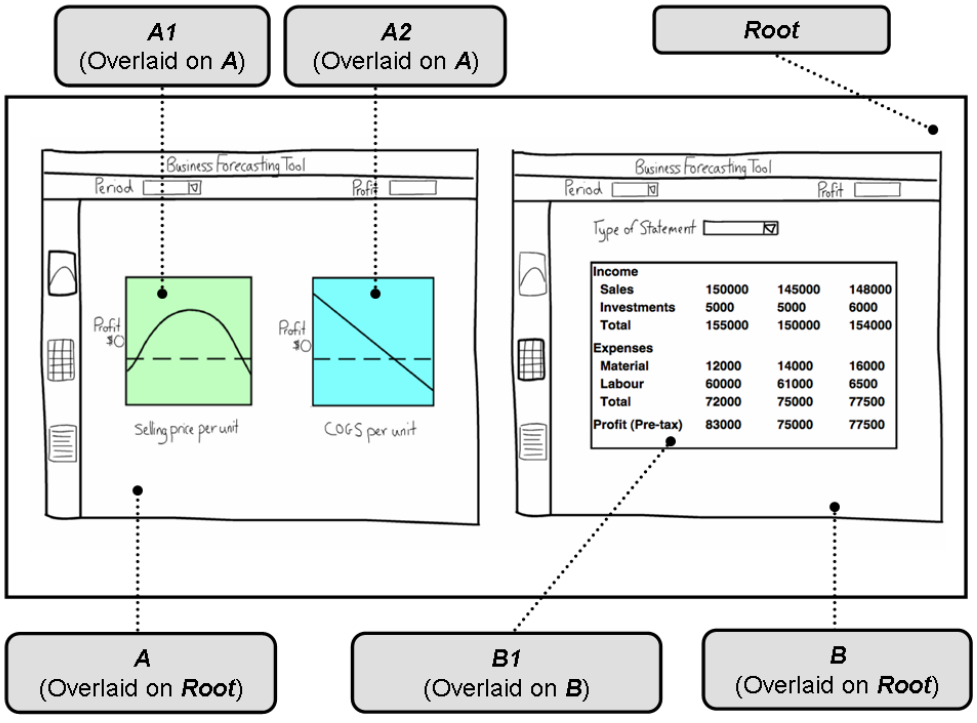


**Figure 4.1:** Using the concept of regions to compose prototypes

Regions have both visual properties as well as associated behaviors. The visual presentation of regions is represented as *assets*, where assets may be sketches, images, or high-fidelity widgets, for example. Behavior is represented as *scripts* that are bound to regions, where the scripts perform manipulations on regions. Scripts are made up of *commands*, which are used to perform basic manipulations on regions such as modifying their size or position. Also, scripts may contain a sequence of commands to perform more complex behavior, such as repositioning an image and shrinking its size as it moves into the new position. Finally, regions may be connected through *relationships*. Relationships specify that a change in one region's properties affect other regions.

The remainder of this chapter further explains the Region Model. First, all concepts of the Region Model are defined. Second, the commands that are able to be performed on regions are discussed. Next, an XML-based notation is presented for specifying the different concepts within the Region Model. This chapter concludes by discussing the benefits of the model.

## 4.1 Defining the Region Model

This section defines each of the terms and concepts that the Region Model is composed of: design space, regions, commands, and relationships. All of the properties and behaviors of regions are also defined. UML diagrams as well as other illustrations are used as necessary to enhance the written definitions of the Region Model.

### 4.1.1 Elements of the Region Model

***Design Space***

*Design space* represents the entire workspace, that is, the overall display real estate that designers are working with. This parallels the concept of calling a whiteboard a design space in a traditional workspace. The design space consists of a reference to the Root region, that is, to the region encompassing the overall workspace. Since all regions are either directly or indirectly subregions of the Root region, all regions are located within the design space's boundaries, which is equivalent to the Root region's boundaries.

***Region***

A *region* is an area that represents a visual element in the design space. Regions may represent elements including interface screens of the system being prototyped as well as components being laid out on the prototype screen designs, such as widgets or images. As well, regions may represent the overall design space or even elements that compose the design space's interface to designers, like pie menus for manipulating the design space, for example.

A region has the following properties:

- $X$ is a value within the design space's coordinate system representing the horizontal position of a region in relation to the bottom left corner. X is some integer bounded by the design space's width.

- $Y$ is a value within the design space's coordinate system representing the verti-

cal position of a region in relation to the bottom left corner. Y is some integer bounded by the design space's height.

- *Z* is a value within the design space's coordinate system representing the depth displacement, that is the layer, of a region. Z is some integer between zero and infinity, where zero is the deepest layer and infinity is the uppermost layer.

- *Width* is a value representing the size of a region along the horizontal axis. Width is some integer between zero and infinity.

- *Height* is a value representing the size of a region along the vertical axis. Height is some integer between zero and infinity.

- *Color* is a set of three values representing some amount of Red, Green, and Blue from the RGB color model, respectively. Color's values are integers between zero and 255.

- *Alpha* is a value representing the degree of transparency of a region. Alpha ranges from zero to one, where zero makes the region completely transparent and one makes the region completely opaque.

- *Asset* represents the visual or presentation form that the region takes. For example, an asset may be in the form of a sketch, image, video, widget, or plain or textured graphical shape. An asset has two properties: the type of asset and the location of a file. Note that a region may have only one asset active at a given time, but may have a sequence of assets associated with it to be displayed at different times.

- *Script* is a sequence of commands associated with a region to give it basic behaviors. Scripts are used to manipulate one or more properties of a region, either immediately or upon the occurrence of a specified event.

- *Layout* is an algorithm associated with a region that is used for automatically positioning all of its subregions within itself. Examples of potentially useful layout algorithms include 'overlay', which positions regions over top of one another with a slight offset to allow for all regions to be partially visible, or 'tile', which positions regions evenly spaced side by side and row by row. Figure 4.2 illustrates the results these layouts may produce.

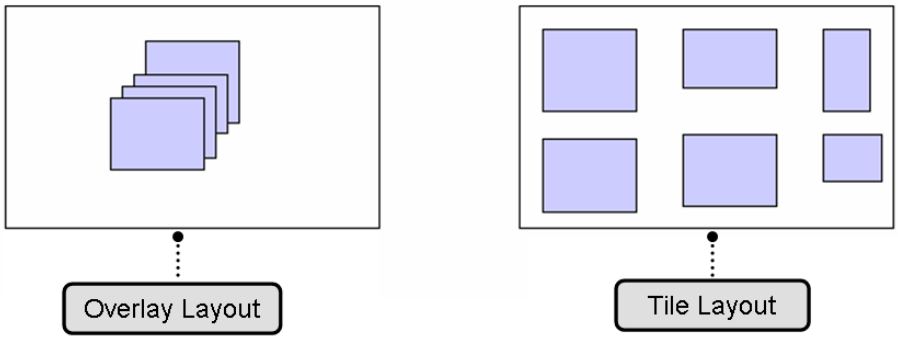


**Figure 4.2:** Examples of possible layouts for the Region Model

- *History List* is a set that stores clones of the region. Clones may be created and saved either at regular intervals or when some of a region's properties change. In other words, a History List maintains historical versions of the region so that a version may be referred to or reverted to at a later time.

- *Parent Region* refers to a region that the current region is positioned within. All regions, with the exception of the Root region, have one and only one Parent region with which they are positioned within and their coordinates are specified

relative to the Parent region's position. The parent-subregion relationship is used to compose interface prototypes of different elements.

- *Subregions* refers to a set of regions that are positioned within the current region. Every region is a direct or indirect subregion of the Root region and every region has zero or more subregions. Figure 4.3 illustrates the relationship between a Parent region and its subregions using a hierarchical tree structure.

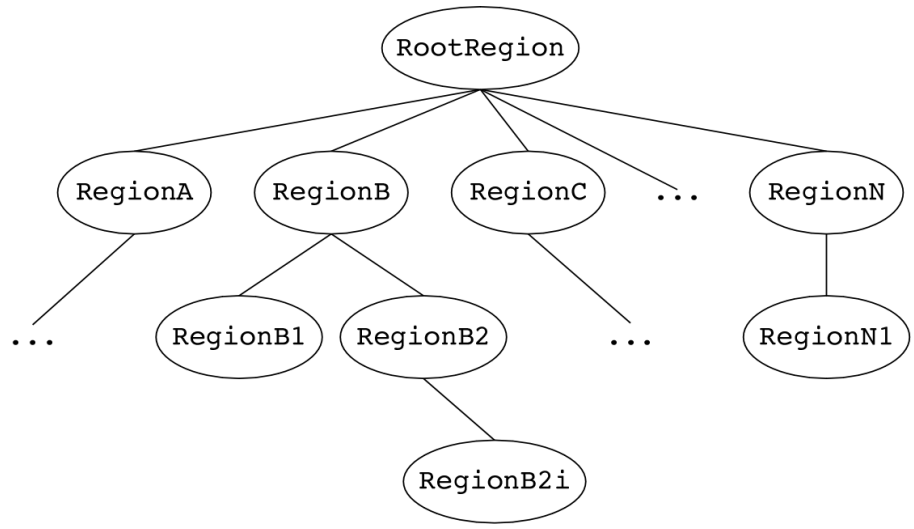


**Figure 4.3:** Every region has a Parent region as well as some number of subregions. For example, RegionB is a region with parent = RootRegion and subregions = {RegionB1, RegionB2}.

***Command***

A *command* is used to perform some basic operation on a region. Examples of commands include repositioning or scaling of regions. A complete list of commands that are useful within the context of the Region Model are discussed in Section 4.1.2.

***Relationship***

A *relationship* is used to represent a connection between a pair of regions. Relationships may exist for two main purposes: to bind behaviors and to indicate navigational

flow. Each type of relationship may have different properties, but all relationships have properties such as the "show" property to indicate that the relationship should be visually depicted.

The binding relationship may be used to specify the binding of behaviors to multiple regions, where manipulating one region results in an automatic manipulation of another region. For example, a relationship may exist between Region A and Region B where changing Region A's x value may result in a relative change in Region B's width and height properties.

The navigational relationship is used to indicate navigational flow between regions. For example, clicking on Region X may bring up the next screen, represented by Region Y; thus, a navigational relationship exists between Region X and Region Y.

Figure 4.4 is a UML Class Diagram illustrating the major concepts in the Region Model. Notice that region has a recursive definition, where the Parent property is also a region, and the subregions and History List properties are sets of zero or more regions.

### 4.1.2 Commands for Manipulating Regions

Commands are used to manipulate properties of regions. Manipulating properties is important in constructing prototypes as well as in managing the design space. This section presents several commands that are seen as fundamental to the Region Model. This section also provides a general notation for specifying each command, including required parameters as well as optional parameters (indicated with []).

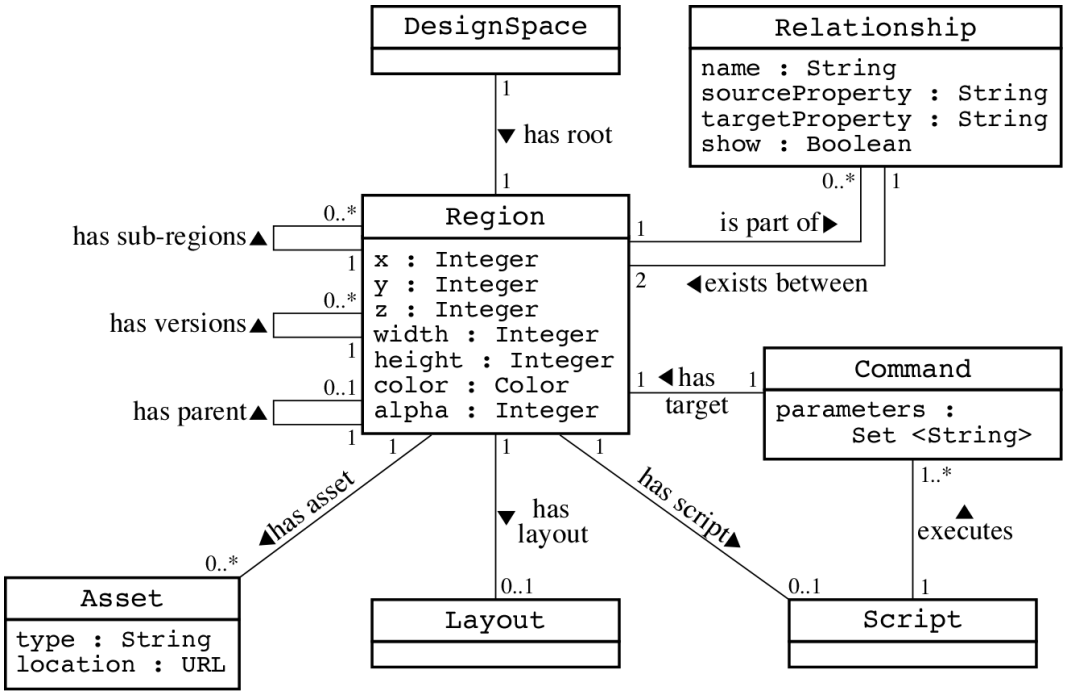


**Figure 4.4:** Region Model illustrated as a UML Diagram

Note that the bold font words are the command names, whereas the italicized words are parameters.

- Add Region - creates a new region named *regionName* in the design space and positions it at the specified $x$ and $y$ location, or else at 0 and 0 by default. This command is denoted as: **add** *regionName* [$x$ $y$].

- Delete Region - removes a region named *regionName* from the design space. The command is denoted as: **delete** *regionName*.

- Move - repositions a region named *regionName* to a new $x$ and $y$ location relative to its parent's origin; moving the current region also causes a relative move in all of its subregions. It is denoted as: **move** *regionName* $x$ $y$.

- Scale - modifies width and/or height properties of a region named *region-*

*Name* by some % of its current size; modifying the size of the current region may also modify the size of all its subregions if the flag ***applyToSubRegions*** is specified. This command is denoted in one of two ways: **scale** *regionName % [**applyToSubRegions**]* or **scale** *regionName width% height% [**applyToSubRegions**]* .

- Change Layer - repositions a region named *regionName* to a new $z$ value; by default, a parent region has a lower z value than its subregions and thus appears below its subregions. It is denoted as: **layer** *regionName z*.

- Change Transparency - modifies the *alpha* value of a region named *regionName* to make the region either more transparent or more opaque. This command is denoted as: **transparency** *regionName alpha*.

- Change Color - modifies the color of a region named *regionName* to have a new color specified by *red, green, blue*. The command is denoted as: **color** *regionName red green blue*.

- Select - specifies one or more regions named *regionName1* through *regionNameN* as active to perform an operation on. This command is denoted as: **select** *regionName1* [*regionName2*] [...] [*regionNameN*].

- Clone - creates a copy named *newRegionName* of the region, *regionName*, and either positions it on top of the original region or at the optionally specified $x$ and $y$ location; may also optionally copy all of the region's subregions using the

***applyToSubRegions*** flag. The command is denoted as: **clone** *regionName newRegionName* [*x y*] [***applyToSubRegions***].

- Replace - replaces a region named *targetRegionName* with the properties of a specific region named *sourceRegionName*; may also optionally replace all of *targetRegionName*'s clones by specifying *applyToClones*. This command is denoted as: **replace** *sourceRegionName targetRegionName* [***applyToClones***].

- Assign an Asset - associates an asset of type *assetType* at *assetLocation* with the region named *regionName*; this asset becomes the current presentation element of the region. This command is denoted as: **asset** *regionName assetType assetLocation*.

- Assign a Script - associates a script at *scriptLocation* with a region named *regionName* in order to give the region specific behaviors. This command is denoted as: **script** *regionName scriptLocation*.

- Record History - indicates that changes to a region's properties over time should be recorded so past versions of that region can be reviewed or reverted to. This command is denoted as: **record** *regionName*.

- View History - indicates that historical versions of a region named *regionName* should be shown for reviewing and for allowing the current version to be replaced with a previous version. This command is denoted as: **versions** *regionName*.

## 4.2 The Region Model Notation

XML syntax is used to describe the Region Model. XML (eXtensible Markup Language) is a widely used text markup language developed by W3C [4]. It was originally designed for organizing web data and has since evolved into a standard for electronic data exchange. XML language has three key components: a syntax for describing the structure of the data (Schemas), a syntax for specifying the actual data (XML Documents), and a syntax for specifying how the data should be displayed (XSL-Style Sheet Language).

Schemas and XML Document notations are used in specifying the Region Model; schemas specify how the regions and other elements in the design space are to be structured while the XML documents demonstrate examples of the design space and elements that occupy the space. A complete schema is presented in Appendix A.1 and a complete XML document example describing the design space shown in Figure 4.5 is given in Appendix A.2. This section provides parts of the schema as well as example XML documents for each element within the Region Model.

### 4.2.1 Specifying Regions

Regions are the fundamental concept in the Region Model. Regions must have the following properties specified: a name to refer to as well as x, y, width, and height properties to be able to display it at a certain location and of a certain size in the design space. Also, z, color, and/or alpha property values may optionally be specified

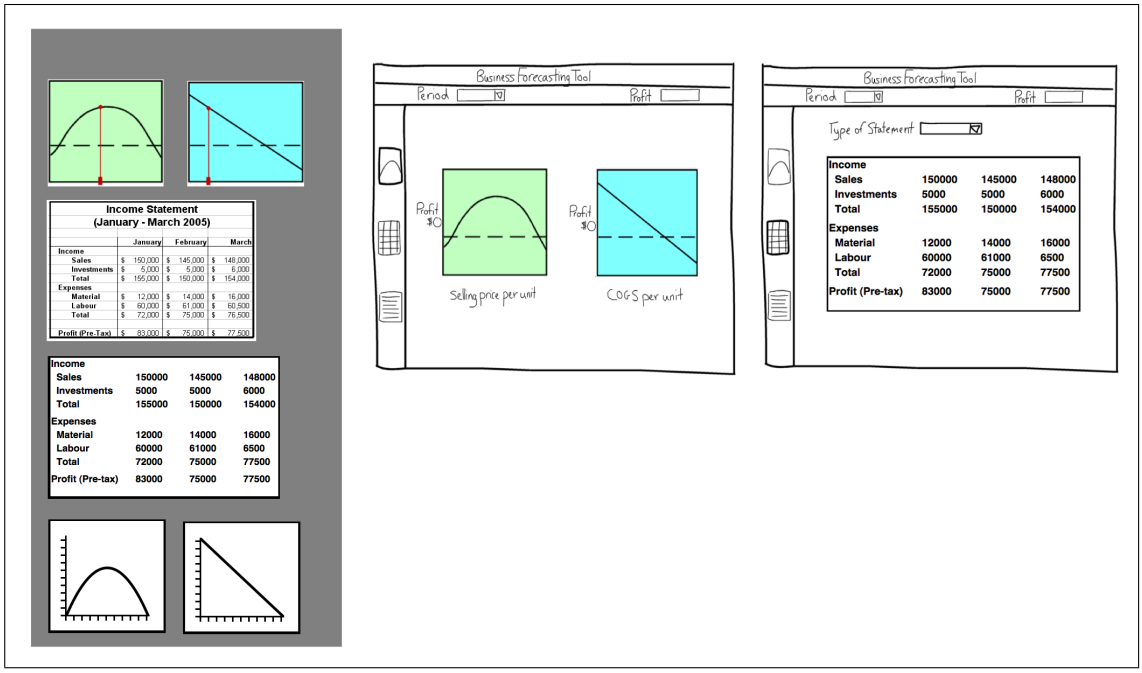


**Figure 4.5:** An example state of the design space

to further determine how the region is to appear. All regions must have a parent region specified, where the Root region has a null value for a parent. Subregions and historical versions in a History List may also be specified if any exist. Furthermore, assets, scripts, and layout properties may be specified for regions. The following portion of the schema outlines how a region and its properties are to be specified.

```
<xs:complexType name="RegionType">
     <xs:sequence>
          <xs:element name="Asset" type="AssetType" minOccurs="0"/>
          <xs:element name="Script" type="ScriptType" minOccurs="0"/>
          <xs:element name="Layout" type="LayoutType" minOccurs="0"/>
          <xs:element name="Region" type="RegionType" minOccurs="0"
          maxOccurs="unbounded"/>
     </xs:sequence>
     <xs:attribute name="name" type="xs:string" use="required"/>
     <xs:attribute name="x" type="xs:integer" use="required"/>
     <xs:attribute name="y" type="xs:integer" use="required"/>
     <xs:attribute name="z" type="xs:integer" use="optional"/>
     <xs:attribute name="width" type="xs:integer" use="required"/>
     <xs:attribute name="height" type="xs:integer" use="required"/>
     <xs:attribute name="color" type="xs:string" use="optional"/>
```

```
     <xs:attribute name="alpha" type="xs:integer" use="optional"/>
     <xs:attribute name="historyList" type="xs:string" use="optional"/>
     <xs:attribute name="parentRegion" type="xs:string" use="required"/>
     <xs:attribute name="subRegions" type="xs:string" use="optional"/>
</xs:complexType>
```

A region has some complex properties that require further specification beyond a single simple value. The complex properties of a region include asset, script, and layout. Notice that in the schema segment above, these three properties are defined as separate complex elements instead of as attributes like the rest of the properties.

The following piece of the schema defines how to specify an asset. An asset has type and location properties. The type of an asset is restricted to four values: Sketch, Image, Widget, and Shape. This restriction is able to be described within the schema.

```
<xs:simpleType name="AssetTypeOptions">
     <xs:restriction base="xs:string">
          <xs:enumeration value="Sketch"/>
          <xs:enumeration value="Image"/>
          <xs:enumeration value="Widget"/>
          <xs:enumeration value="Shape"/>
     </xs:restriction>
</xs:simpleType>

<xs:complexType name="AssetType">
     <xs:attribute name="type" type="AssetTypeOptions" use="required"/>
     <xs:attribute name="location" type="xs:anyURI" use="required"/>
</xs:complexType>
```

The following portion of the schema shows how to specify a script.

```
<xs:complexType name="ScriptType">
     <xs:attribute name="behavior" type="xs:string" use="required"/>
</xs:complexType>
```

Finally, the Layout element is specified as follows.

```
<xs:complexType name="LayoutType">
     <xs:attribute name="regions" type="xs:string" use="required"/>
     <xs:attribute name="algorithm" type="xs:string" use="required"/>
</xs:complexType>
```

An example XML Document adhering to the region schema is provided below. This example has a Root region with three immediate subregions: R2, R3, and R4, all of which are laid out in the design space using the tile algorithm. R2 also has a subregion named R2b.

```
<Region name="RootRegion" x="0" y="0" width="400" height="300"
color="255 255 255" parentRegion="null" subRegions="R2 R3">
     <Layout algorithm="tile" regions="R2 R3 R4"/>
     <Region name="R2" x="25" y="15" width="100" height="100"
     parentRegion="RootRegion" subRegions="R2b">
          <Asset type="Image" location="./image2.png"/>
          <Region name="R2b" x="1" y="1" width="45" height="45"
          color="255 0 0" alpha="1" parentRegion="R2"></Region>
     </Region>
     <Region name="R3" x="150" y="150" width="55" height="55"
     color="0 255 0" parentRegion="RootRegion"></Region>
     <Region name="R4" x="350" y="250" width="35" height="35"
     color="0 255 0" parentRegion="RootRegion"></Region>
</Region>
```

### 4.2.2 Specifying Relationships

Relationships exist between two regions. Relationships may exist for different reasons, such as to indicate a navigational relationship or a functional relationship. Not all relationships, such as navigational relationships, include source or target properties, and as such these are optional attributes in the XML schema. The following part of the schema shows how to specify a relationship.

```
<xs:complexType name="RelationshipType">
     <xs:attribute name="name" type="xs:string" use="required"/>
     <xs:attribute name="sourceRegion" type="xs:string" use="required"/>
     <xs:attribute name="targetRegion" type="xs:string" use="required"/>
     <xs:attribute name="sourceProperty" type="xs:string" use="optional"/>
     <xs:attribute name="targetProperty" type="xs:string" use="optional"/>
</xs:complexType>
```

The following is an example XML document describing a Relationship.

```
<Relationship name="R2affectsR3" sourceRegion="R2" targetRegion="R3"
sourceProperty="x" targetProperty="width"/>
```

### 4.2.3 Specifying Commands

Each command has these properties: the name of the command to be performed, the name of the region to perform the command on, as well as other parameters that vary based on the type of command. The following is a piece of the schema illustrating how generally to specify a command.

```
<xs:element name="Command">
     <xs:complexType>
          <xs:attribute name="name" type="xs:string" use="required"/>
          <xs:attribute name="regionToAffect" type="xs:string" use="optional"/>
          <xs:attribute name="parameters" type="xs:string" use="optional"/>
     </xs:complexType>
</xs:element>
```

As mentioned, each type of command has unique parameters. The generic notation for specifying specific commands was provided in Section 4.1.2. Below an example XML document is given, illustrating how to specify each command and its respective parameters in XML notation. Note that in this XML notation, all parameters beyond the name of the region to be manipulated are specified in a comma separated list.

```
<command name ="add" regionToAffect="R1" parameters="10, 500">
<command name ="remove" regionToAffect="R2">
<command name ="move" regionToAffect="R1" parameters="50, 250">
<command name ="scale" regionToAffect="R1" parameters="2.0, 1.0">
<command name ="layer" regionToAffect="R3" parameters="0">
<command name ="transparency" regionToAffect="R3" parameters="1">
<command name ="color" regionToAffect="R1" parameters="255 0 0">
<command name ="select" regionToAffect="R4">
<command name ="clone" regionToAffect="R1" parameters="R1clone, 15, 15">
<command name ="replace" regionToAffect="R3" parameters=" R1, applyToClones">
<command name ="setAsset" regionToAffect="R1" parameters="./newimage.png">
<command name ="setScript" regionToAffect="R1" parameters="./../script">
<command name ="record" regionToAffect="R3">
<command name ="view" regionToAffect="R3">
```

### 4.2.4 Specifying Design Space

The design space consists of a Root region that represents the overall workspace, as well as some number of direct and indirect subregions. These subregions are used to compose prototypes. Commands and relationships may also exist within the design space. The following XML schema shows how to specify the overall design space.

```
<xs:element name="DesignSpace">
     <xs:complexType>
          <xs:choice>
               <xs:element name="Region" type="RegionType" minOccurs="1"/>
               <xs:element name="Relationship" type="RelationshipType"
               minOccurs="0"/>
               <xs:element name="Command" type="CommandType" minOccurs="0"/>
          </xs:choice>
     </xs:complexType>
</xs:element>
```

## 4.3 Benefits of the Region Model

This research aims to utilize a large display workspace to enhance prototyping. The Region Model seems particularly well suited for large display workspaces that are

constructed of multiple, tiled displays as well as for single display systems. For multiple display systems, the design space could manage the process of assigning specific subregions of the Root region to specific displays. Each display would be responsible for rendering a "Display Root" region and all of its subregions. This should scale well as more displays need to be added.

The Region Model is effective because of its expressiveness. This model can be used to describe both the current state of prototypes being designed as well as the design space itself and how the various prototypes are arranged within the design space. The range of its expressiveness will be demonstrated in the following two chapters.

# Chapter 5

# ProtoMixer: Software Support for Mixed Fidelity Prototyping

This chapter presents a software tool to support the mixed-fidelity approach to prototyping. The tool, called ProtoMixer, implements the Region Model from Chapter 4. This chapter describes what features are implemented, how they are to be used, and how they are implemented if relevant.

The aim of ProtoMixer is to be able to support the scenarios illustrated in Chapter 3. The next chapter shows how ProtoMixer successfully supports the scenarios by describing an actual design session of an example application Also, illustrations of ProtoMixer being used are provided in the next chapter.

## 5.1 Overview of ProtoMixer

ProtoMixer is a proof-of-concept system developed in Java. Refer to Figure 5.1 for a screenshot of the system. ProtoMixer is approximately 5,000 lines of Java code. There are two different implementations: one in JOGL, which is OpenGL for Java, and another in Java 2D. JOGL was initially used because it has better

graphics rendering performance; however, since Swing components cannot easily be integrated with JOGL, a second version using Java2D was built to integrate high-fidelity elements. Aside from incorporating high-fidelity elements, both implementations largely look and behave the same from the user's perspective.

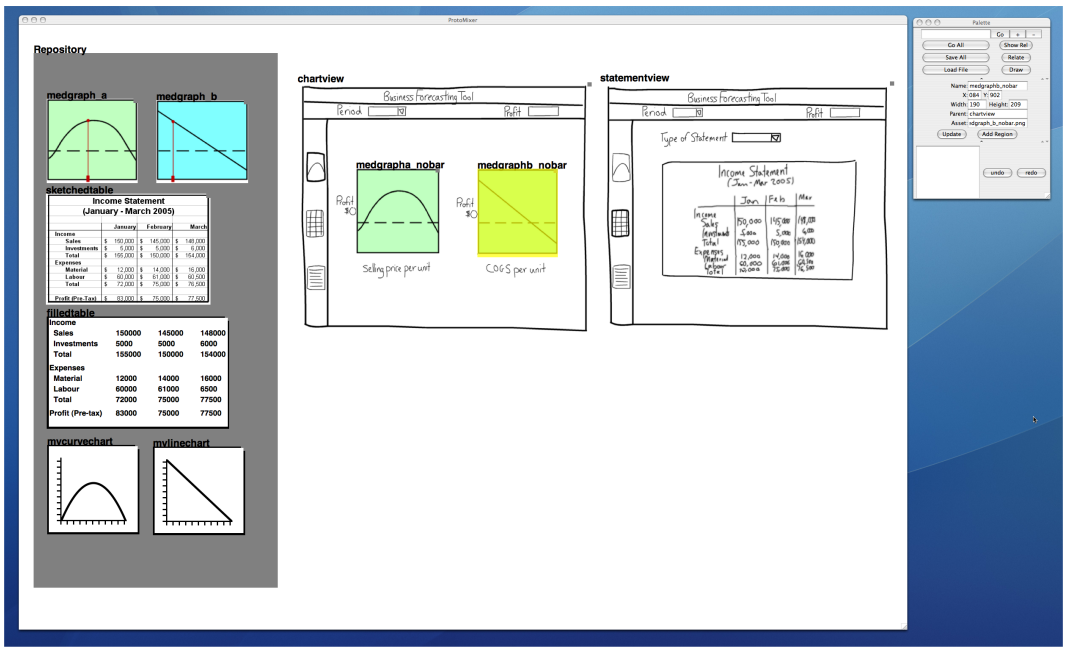


**Figure 5.1:** Screenshot of ProtoMixer

ProtoMixer is intended to be an easy to use, lightweight system, much in the same manner as a basic drawing editor. All objects may be positioned anywhere on the workspace, which is the same as the canvas in a drawing editor; placement of objects is not restricted by frames and borders as with other tools like Interface Builders. Also, there are no menu bars; all system operations are either performed directly on the object or through a simple command panel, much like the always visible color palette and property panels in drawing editors. For system development purposes, command line interactions are also possible.

ProtoMixer implements all of the region commands from Chapter 4. By implementing the commands as well as additional features, ProtoMixer satisfies the most important and relevant facilities discussed in Chapter 3. Refer to Table 5.1 for a list of all features ProtoMixer offers as well as how these features map to the facilities and commands discussed in earlier chapters.

## 5.2 Implementing the Region Model

ProtoMixer was implemented based on the Region Model presented in Chapter 4. Regions represent every prototyping element in the system, including the prototype's screen or components within that screen. Regions in ProtoMixer have all of the same properties discussed in the preceding chapter including the location and appearance related properties as well as a parent and subregions. The ProtoMixer system also incorporates all of the commands outlined in Chapter 4 to perform basic manipulations on regions.

**Table 5.1:** Mapping facilities to region commands and ProtoMixer features

| **Facilities** | **Region Model Commands** | **ProtoMixer Features** |
|---|---|---|
| **Drawing Editor Facilities** | | |
| Scaling | Scale | Scale |
| Positioning | Move | Move |
| Cloning | Clone | Clone |
| Annotating | —— | Sketch/Annotate |
| —— | Add, Delete | Add, Delete |
| —— | Change color | Change color |
| —— | —— | Cut out |
| | | |
| **Organizational Facilities** | | |
| Grouping | —— | Overlay (relative positioning of regions within regions) |
| Structuring | —— | Layout |
| Relating objects | —— | Relating (visually and by binding properties) |
| Highlighting and selection | Select | Select |
| —— | Replace | Replace |
| | | |
| **Prototyping Domain Facilities** | | |
| Layering and transparency | Change layer, Change transparency | Change layer, Change transparency |
| Running components | —— | Running high-fidelity Java components |
| Connecting to data | —— | In high-fidelity java components |
| Setting properties and behavior | Assign asset, Assign script | Assign asset, Assign script, Flashing through assets, Threaded tasks (for animating), Constrained movement, Binding properties |
| | | |
| **Multi-User Support Facilities** | | |
| Multiple inputs | —— | Future work |
| User control panels | —— | Future work |
| Integration of devices | —— | Future work |
| | | |
| **Large Display-Specific Facilities** | | |
| Pick-and-drop | —— | Future work |
| Gestures and specialized menus | —— | Future work |
| Space management | —— | Layout, Relative positioning of regions within regions |
| | | |
| **Process Enhancing Facilities** | | |
| Logging activities | Record history, View history | Record history, View history, Redo/undo |
| Querying library | —— | Future work |
| Import and exporting | —— | Import and exporting xml files |

The Root region represents the entire design space and every other object on the display is either a direct or indirect subregion of the Root region. This holds true in our implementation as well. The rendering of the design space happens recursively, starting with the Root region; each region renders itself and then renders all of its subregions. This rendering is fairly straightforward and based on the depth first search algorithm. Pseudo code illustrating the rendering process is shown below:

```
method main
{
     ...
      renderRegions(rootRegion)
     ...
}

method renderRegions(currentRegion)
{
     renderSelf(currentRegion)

     if currentRegion hasSubRegions then
          for each subRegion
               renderRegions(subRegion)
}

method renderSelf(currentRegion)
{
     ...
}
```

The coordinates of each region are specified relative to its parent region's origin. Thus, repositioning a parent region also repositions its subregions relative to the parent's new position. This relative coordinate system is used to hold together prototypes and the elements that they are composed of.

The same recursive depth first search algorithm presented above is used whenever the regions structure needs to be searched. For example, to identify which region a designer has selected, the regions must be searched to find the uppermost layered

region, contained within a specific location. The same algorithm is also used when the user moves a region off of its current parent to determine which region becomes its new parent.

The other elements of the Region Model are also incorporated into ProtoMixer. These elements will be discussed throughout the remainder of this chapter.

## 5.3 Integrating the Different Fidelities

The main goal of ProtoMixer is to allow users to compose prototypes of multiple fidelities. ProtoMixer supports the integration of elements of any fidelity through the use of assets. In the current version of ProtoMixer, assets may be of these types: sketch, image, and widget, where each of these types clearly corresponds to a fidelity. (Note that in Chapter 4, assets are discussed as being of other types like geometric shapes, but currently ProtoMixer only handles rectangular-shaped regions.) To incorporate a specific element in ProtoMixer, users set a new or existing region's asset to point to the location of the asset file as well as specify the type of the asset so the renderer can respond accordingly. The remainder of this section discusses how ProtoMixer successfully integrates the three different fidelities: low, medium, and high.

### 5.3.1 Low-Fidelity

Low-fidelity elements are typically sketched designs. They may be sketches of any part of a prototype, from the whole screen to pieces of an interface component.

ProtoMixer supports low-fidelity design by allowing for importing of sketches created externally as well as for sketching directly on the design space.

ProtoMixer is primarily developed to use sketches that are created outside of this system. For example, sketches may be images imported from scanned in paper-based sketches, photographed images of sketches done on whiteboards, or images from other computer-based sketching applications such as Adobe® Illustrator®. These images of sketches are then imported into the system as assets that are associated with particular regions. Specifically, sketches are imported by providing the system with a name and location of the image file.

The system also supports basic freehand sketching. When the designer is in the Sketch mode versus the regular interactive mode, all of the strokes are encapsulated into a region. The sketch can then be manipulated in the same manner as any other region. While the sketching capabilities are fairly limited at this point, there is nothing restricting us from adding further sketching features such as different stroke styles, widths, and colors, as well as an eraser and grid-based tracing feature. Sketching is not where our attention has been focused, as several commercial software packages already exist to do detailed sketching and designers who are accustomed to a current way of sketching, on paper for example, likely would prefer to continue with that approach and then import their designs into our system.

ProtoMixer also supports the PICTIVE technique, or specifically the integration of PICTIVE elements, which are annotated paper cutouts. As PICTIVE designs are also paper-based, they may be imported into the system in the same manner as

non-digital sketches – by scanning them or taking a photograph and then importing them as images.

### 5.3.2 Medium-Fidelity

Medium-fidelity elements are non-sketched, more refined images of design concepts. Again, these may be images of any part of the prototype, from skeleton screens to components. These images can be created using commercial multimedia tools such as those discussed in Section 2.2, as is traditionally done. Medium-fidelity elements may also be screenshot images of existing user interfaces or interface elements. These images are then able to be imported into ProtoMixer as assets, by providing the system with the name and location of the image file. Medium-fidelity elements may also have some limited behavior, as will be discussed in Section 5.4.3.

### 5.3.3 High-Fidelity

High-fidelity elements are actual programmed user interface components. ProtoMixer allows for working user interface components to be imported into the system as assets. These components may be standard toolkit components or custom-built components. Specifically, designers can import high-fidelity elements into ProtoMixer by providing the name and location of the component.

The current version of ProtoMixer allows for Java Swing or AWT components to be imported and used within the system. This is because the system is implemented in Java. It supports any pre-built standard components that inherit from

JComponent such as JTables and JTextFields. ProtoMixer also supports customized components that a software development team may have built specifically for the current project or may be reusing from past projects. Customized components must extend from Java's JComponent class or one of its children, because the rendering class needs to know the type of the component in order to be able to instantiate and render it.

The following block of Java code from ProtoMixer shows how a component is initialized, regardless of whether it is a standard Java component or some custom component. Specifically, reflection is used to create an instance of the component object. (Note that exception handling has been removed for improved readability.)

```
public JComponent createWidget(String widgetType)
{
  JComponent widget = (JComponent) Class.forName(widgetType).newInstance();
  return widget;
}
```

It is interesting to note that by allowing integration of any component that inherits from JComponent, ProtoMixer offers a very flexible and powerful feature. Firstly, ProtoMixer allows for standard or custom components to be imported into ProtoMixer with pre-loaded data. This is very useful in the domain of prototyping, where it is often important to convey the impression of a real, working system and using real data aids in that. Secondly, ProtoMixer not only supports integration of high-level control components (like JTables and JTextFields), but also supports lower-level general purpose container objects such as JPanel and JScrollPane. This allows for more complex, previously composed elements or entire prototypes to be

displayed and manipulated as any other region within ProtoMixer. Ultimately, this allows designers to integrate high-fidelity components from their previous projects or from existing specialized component libraries.

### Alternative Approaches to High-Fidelity

Integrating high-fidelity interface components could be accomplished using approaches other the one implemented and discussed above. One alternative approach is to implement our own user interface component toolkit in whatever language the system is implemented in: JOGL or Java2D in our case. This approach was not pursued because it requires a lot of extra coding and system users would typically not want to implement every component from scratch. Instead it is important to encourage reuse of existing components and lightweight development for components that do not already exist.

A second and rather unique alternative to integrating high-fidelity elements is to use VNC-based technology [28]. VNC or Virtual Network Computing allows for a client machine (the one running ProtoMixer) to have access to view and to interact with the screen display contents of a separate (server) machine. It should be possible to modify existing VNC applications, such as RealVNC and UltraVNC, to allow for viewing of only a specific area of the remote screen, that is, the area which contains the interface component to be integrated into ProtoMixer. This area containing only the component of interest could then be displayed on the machine running ProtoMixer. Of course, the area would need to be displayed without a

framed window, which is normally used in standard VNC applications, presenting a new implementation challenge of removing frames from windows.

This VNC approach encourages the use and reuse of existing components, unlike the first alternative approach that involves implementing our own toolkit. The VNC approach also allows for integrating any existing components or parts of interfaces regardless of whether or not they can run as stand-alone components within our system, which betters our implemented approach of tying in stand-alone Java components. The major disadvantage of the VNC approach is that it leaves little opportunity for tying in a data model and linking data between two or more running components that are not being run from within the same remote application.

## 5.4 Using the System

This section discusses how to use ProtoMixer. Specific issues discussed include how to interact with ProtoMixer, how to compose prototypes and give them behavior, how to create storyboards, how to organize the design space, as well as how to use other interesting features to enhance the design process. Implementation details are provided where appropriate.

### 5.4.1 Interacting with the System

ProtoMixer allows for designers to interact with the system via direct manipulation, a user interface form (referred to as the command panel and shown in Figure 5.2), and command line input. These three approaches of interacting were implemented

for different reasons. Direct manipulation combined with the command panel are likely the preferred interaction methods for designers using ProtoMixer in a real design session. Direct manipulation allows for designers to manipulate the regions on the screen, by scaling, moving, and selecting regions for example. These types of commands are conducive to direct manipulation, whereas other commands like change color or transparency or record history are not. Instead such commands are better performed by inputting instructions into a command panel. Finally, the command line interaction was created primarily for testing purposes at the development stage, but also may be found useful for designers who wish to extend the system by adding their own commands. Also, command line interaction is conducive to putting multiple basic commands together to perform a more complex manipulation on regions.

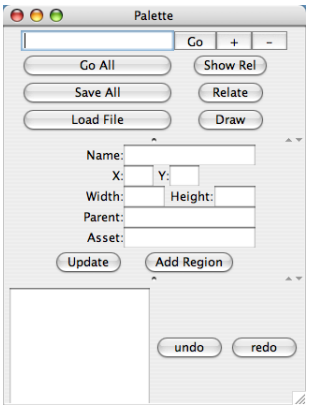


**Figure 5.2:** Screenshot of ProtoMixer's command panel

The system is designed to provide the user with visual feedback where possible, particularly for interactions that are specified via direct manipulation. For example, selecting an element causes that element to be highlighted with a yellow border. And starting to resize an element by directly manipulating the element's resize handle

causes a red rectangular outline to appear representing the currently specified new size of the element.

### 5.4.2 Composing Prototypes

ProtoMixer allows designers to create prototypes by importing previously designed prototype screens or elements, or by creating new ones within the system. A prototype or a prototype element is integrated by first creating a region and then assigning an asset to it. A region may only have one active asset at a time, but a sequence of assets may be assigned to be displayed at some instance or upon some event occurring. Also, a new region may be created by utilizing the built-in sketching feature.

Prototypes can then be composed in ProtoMixer by layering regions on top of other regions. By placing a region on top of another region, the parent and its subregions are effectively grouped together (called the Grouping feature), so moving the parent region keeps the subregions within it at the same relative positions.

Visual properties of prototypes can be modified and enhanced by taking advantage of other basic built-in features. For example, designers may scale elements or prototypes, where scale is implemented to allow for either proportionate scaling, that is, changing the width and height by equal amounts, or disproportionate scaling, where a percentage of change in the width differs from the height. Designers may also move elements or prototypes, or change their color or transparency. During runtime, designers may also change which assets are currently displayed.

ProtoMixer has features that support reuse. Regions may be cloned and used in more than one design, which is particularly useful when exploring alternative designs with some similarities. The Clone feature is implemented as a deep clone, creating a clone of all associated Java objects, particularly the region's subregions when specified. In ProtoMixer, regions keep track of all regions that are clones of themselves, which supports the Replace feature. The Replace feature allows regions that look the same to be replaced by either another region or, minimally, another asset. This is useful when a prototype or a piece of a prototype that appears in multiple places has been re-designed and needs to be updated to reflect these changes. Furthermore, through the Cutting feature, pieces of regions may be reused by allowing the designers to cut out an area of a region and encapsulate the specified area as a new region. This is also useful in creating multiple designs that have some elements in common.

Designers may annotate designs in ProtoMixer by using the Sketch feature. Designers may make notes or markings on the designs and once they are done, ProtoMixer encapsulates this into a region. The annotation region may then be associated with the region it is annotating by creating a relationship between the two regions. The annotation Region may then be hidden or made visible upon the designers' request. The annotation could also be shrunk and stored in the corner of the region it is annotating, and be enlarged when a designer is interested in reviewing it.

### 5.4.3 Adding Behavior to Prototypes

Aside from taking advantage of the built-in behavior offered by high-fidelity components, ProtoMixer also allows for certain behavior to be added to low- and medium-fidelity elements. Designers are able to add behavior to prototypes and/or their elements through a few different approaches: constraints, animation, and bindings. Using these approaches, ProtoMixer currently supports several different examples of behavior that designers may be interested in incorporating into their prototypes.

#### Constraints

One example of the type of behavior ProtoMixer is able to support is constrained movement. In our implementation, constrained movement restricts where a region can be moved within its parent region; that is, a region's position may be restricted to anywhere within its parent or be more heavily restricted to movement along some specified path. This may be useful in composing prototypes with lower-fidelity slider components, for example, where the movement of the slider can be restricted to a horizontal path along the slide bar. ProtoMixer allows for this type of constraint-based behavior to be extended to place constraints on other element properties than only position.

#### Animation

ProtoMixer also provides behavior to prototypes through the use of animation, a useful feature that allows for a wide range of example behaviors to be demonstrated

to stakeholders. Specifically, ProtoMixer allows for a series of tasks or manipulations on prototypes to be specified and then performed over a period of time. The manipulations on the prototype can be scheduled to start taking place immediately, at some delayed instance in time, or upon the occurrence of some user interaction.

One example of animation that ProtoMixer supports is flash sequence animation. Flash sequence animation allows for a series of assets associated with an element to be displayed one at a time over a specific period of time. Flash sequence animation could be used to demonstrate the effects of a video playing inside the prototype from a series of images, for example. As another example, flash sequence animation could be used to show navigation between screens.

A second example of animation that ProtoMixer supports is motion animation. Designers can use motion animation to demonstrate a prototype element moving across the screen, like a drag-and-drop interaction for example. Designers can also specify further animated behavior beyond simple motion. Elements could be resized as they are dragged, for example, having them expand to a certain size as they reach a target position.

### Binding Behaviors

Often in user interfaces, the behavior or appearance of an interface element may depend on properties or behaviors of some other element in the interface. ProtoMixer supports this by allowing for designers to be able to bind one element's (target element) behavior or appearance with the properties or actions of another element

(source element). Changing the source element causes some behavior to happen in the target element.

One example of binding behavior is when using a slider's position to determine the size of another element in the prototype screen. The designer would specify the slider's position as the source property and bind that to the other element's size (the target property). As another example, designers could use the slider as a scroll bar and have the slider's position update which area of the prototype is displayed by changing the current asset of a particular region.

Another example is binding the event of selecting one element (the source) with specifying which asset to be displayed in another region (the target) of the prototype. More concretely, the designer would want to specify this type of binding when clicking on a next arrow image to cause the image in an image previewer region to be changed to the next image in the asset list.

***Binding High-Fidelity Elements*** ProtoMixer supports a more complex binding involving high-fidelity widgets only, which allows for the value(s) in one high-fidelity widget to affect some value(s) in another high-fidelity widget. This binding feature is implemented in a generic manner, using the Publisher-Observer pattern. The observer widget, who depends on the value(s) from some other widget (the publisher), subscribes to the publisher to be notified when the specified value(s) have changed. The publisher widget then notifies all observer widgets upon the occurrence of the subscribed value(s) changing.

Three different categories of values, or more specifically, three common types of underlying data structures that contain values the widgets may be interested in

binding to are identified. These include: an array, a table (two-dimensional array), and a single value. A specific Publisher and Observer interface were implemented for each of these three types of data structures. Below is the code for the Array publisher and observer interfaces. (Note that the Value and Table publisher and observer interfaces are very similar to these, with only slight name changes as well as the obvious return and argument type changes.)

```
public interface ArrayPublisher extends Publisher {
     Object[] getArray();
     void setArray(Object[] array);
     void subscribeToArray(ArrayObserver aObserver);
}

public interface ArrayObserver {
     void arrayChanged(ArrayPublisher aPublisher, Object[] array);
}
```

Below is the code for the Publisher interface, which all three types of Publishers must extend. The purpose of this higher-level Publisher is to enforce the publisher to maintain a list of all subscribing (observer) widgets.

```
public interface Publisher{
     void addSubscriber(Object s);
     void removeSubscriber(Object s);
     Vector getSubscriberList();
}
```

In order to bind two high-fidelity widgets together within ProtoMixer, the appropriate widgets' Java classes must implement the Publisher and Observer interfaces. For example, assume a JTextField widget needs to display the value corresponding to the JSlider's current position. A new class would need to be implemented that extends from JTextField and implements all methods specified in the ValueObserver

interface. A new class would also be needed that extends from JSlider and implements all methods from the ValuePublisher (and indirectly, Publisher) interface. These two new classes could be incorporated into ProtoMixer by creating new regions or setting existing regions' assets to point to these widget classes. To actually bind the two widgets, the user would then have to use the Bind command that requires the observer region name, the publisher region name, and the data structure type to bind to. In this example, our bind command would look like this: "bind TextFieldRegion SliderRegion value".

### 5.4.4 Organizing the Design Space

ProtoMixer's design space can be organized by taking advantage of built-in layout algorithms. To organize the entire design space, designers can apply layout algorithms to the Root region. To organize a specific area of the design space, a layout can be applied to the region that occupies that specific area. ProtoMixer supports tiled layout which is particularly useful in organizing the overall design space. As discussed earlier, tiled layout arranges all of the display contents side by side and row by row. ProtoMixer can be easily extended to support further layout algorithms.

To use the automatic layout feature, designers must associate a particular layout algorithm with the region of interest. Designers then instruct the system to perform whichever layout has been associated with the region. This is useful for re-laying out regions after designers may have moved elements out of or placed new elements into the region.

ProtoMixer also supports further organization of the design space by allowing designers to create visual relationships between regions. Showing how regions are related should help create organization in the minds of the designers. These visual relationship lines may be turned on and off.

### 5.4.5 Storyboarding

ProtoMixer offers designers a storyboarding feature. First, designers can lay out the screens as desired either by taking advantage of built in layout algorithms or by laying out the screens on their own. Designers can then link the screens together to demonstrate navigational flow. ProtoMixer allows the designers to specify navigational flow between two prototypes either by drawing a line from one prototype to another while in the Create Navigation mode or by specifying it in the initial import file or via command line at run time.

ProtoMixer allows the designers to play through the storyboard they have constructed by animating the navigational flow between the prototype screens. Designers may choose to demonstrate the navigation by having the prototypes be highlighted one at a time in navigational sequence. Designers may also choose to have arrows drawn between the prototypes one at a time, again in sequence. Finally, designers may have the prototypes laid out on top of one another and flashed through in sequence to animate the screen changes.

### 5.4.6 Additional Process-Enhancing Features

Additional features of ProtoMixer contribute to process and productivity enhancements. ProtoMixer offers three features to support the Logging Activities Facility from Chapter 3: Record History, View History, and Undo/Redo. The Record History feature in ProtoMixer is implemented by having every region keep track of itself over time. Designers can then view the history of a region by using the View History command, where a new region pops up displaying all historical versions of the region. At this point, ProtoMixer can record a region at regular time intervals or upon certain changes to that region, such as when its asset changes. ProtoMixer also records a list of all user interactions with the design space and its contents, offering designers the ability to undo and redo actions.

ProtoMixer also supports Importing and Exporting of the design space contents. The files that are imported and exported are XML documents matching the Design Space Schema presented in Section 4.2. ProtoMixer can also be set up to export the design space at a regular interval.

Between recording each region's appearance, the user interactions performed, and the overall design space contents at regular intervals, ProtoMixer could be extended to have a fairly complex history recording mechanism. It would be interesting to provide the user with a visual timeline of how designs have evolved.

## 5.5 Utilizing a Large Display Workspace

The current version of ProtoMixer is set up on an 8 mega-pixel display space. Specifically, this workspace consists of two 30" Apple Cinema Displays®, both running at a resolution of 2560x1600 pixels. Having high resolution allows us to display multiple screen prototypes at once. It also allows multiple designers to gather around the workspace and participate in the design session. Originally our system was run on a SMART Board™ display; however, we found that the best resolution supported (1280x1024) was insufficient for our purposes. With ProtoMixer, it is essential to explore multiple prototypes or at least multiple prototype elements at once and in a collaborative setting.

To date, we have primarily used a standard keyboard and mouse to interact with the system. We also have explored using a sketch tablet for selecting and drawing objects in the system. The envisioned ideal workspace should consist of a variety of computing devices aside from the large displays, such as Tablet PCs, laptops, or PDAs for interacting with the display from anywhere in the room. The currently envisioned ideal techniques for directly interacting with the large display are pen-based interactions on a touch-sensitive surface, including sketching and gestures; however, it may be worthwhile to explore other novel interaction techniques.

## 5.6 Achieving the System Goals

Five system design goals were presented earlier in Chapter 3. We aimed to satisfy these goals in our proof-of-concept implementation. This section describes how progress was made towards satisfying these goals.

First, ProtoMixer encourages the use of traditional informal design activities by allowing designers to perform low-fidelity prototyping outside of ProtoMixer. These externally created designs can then easily be imported into ProtoMixer. Furthermore, ProtoMixer allows for sketching within it so traditional informal activities are not abandoned in our mixed-fidelity approach.

Second, ProtoMixer supports transitioning between fidelities in a couple of different ways. Assets of a region may be changed to move to a higher-fidelity asset or revert back to a lower-fidelity asset. Also, regions representing the same prototype element may be layered on top of one another, and then the layers may be adjusted to bring a region of a certain fidelity to the top.

Third, ProtoMixer offers history recording to enhance the design process. Specifically, ProtoMixer records copies of each region when it changes as well as records all user interactions with the system. ProtoMixer also allows previous versions to be reverted to and previous user interactions to be undone.

Fourth, while ProtoMixer does not yet allow for multiple users to interact with it, ProtoMixer does support collaboration. Multiple people can view the designs in ProtoMixer, as ProtoMixer is run on a large display workspace. And more importantly, different individuals, such as developers who implement components, are now

able to become involved early in the design process. Also, in supporting mixing of fidelities, ProtoMixer allows for rich feedback to be elicited from users and other stakeholders throughout the process.

Fifth, ProtoMixer provides a lightweight design environment. Specifically, it implements features for importing and exporting the design space so it is easy for designers to start where they left off in a previous session. Also, it is implemented with no hidden menus and most interactions are based on direct manipulation. Finally, behavior can be added to elements with little effort.

## 5.7 Future Implementations

As indicated in Figure 5.1, most of the facilities discussed in Chapter 3 were accomplished with this version of ProtoMixer. We envision an ideal system to have implemented all of the remaining facilities set out in Chapter 3. At this point, ProtoMixer does not support any multi-user features such as integration of multiple devices and multiple inputs. Also, ProtoMixer needs to have features to better facilitate the use of large display workspaces such as specialized interaction techniques. Furthermore, the ideal system should have a more complex versioning system to provide users with a visual timeline to show how designs have evolved. Finally, it would be beneficial to include a visual querying facility to allow designers to search for previous, relevant designs. Even with some limitations in the current implementation, a range of user interface designs and behaviors are able to be expressed, as will be shown in the following chapter.

# Chapter 6

# Mixing Fidelities In Action: An Example

In the previous chapter, the proof-of-concept system called ProtoMixer was presented. This chapter describes an example design session where we, as designers, are utilizing ProtoMixer to prototype the business forecasting application described in Chapter 3. The design scenarios from Chapter 3 are revisited, and it is illustrated how they are successfully accomplished using ProtoMixer. Note that this chapter offers an illustrative example and our intent is for the mixed-fidelity approach to be generalized beyond the business forecasting domain as well as these particular scenarios.

## 6.1 Leading up to the Scenarios

We start by sketching prototype screens as well as some elements using an external drawing application. Note that we also could have drawn the designs on paper and then scanned them in as images. We then import these sketches into ProtoMixer as images and specify them as regions' assets (done through an xml input file). All of the sketches are then displayed in ProtoMixer in the Repository Region. The rest of

the design space is empty as we are just beginning a new project. Figure 6.1 shows the current state of ProtoMixer.

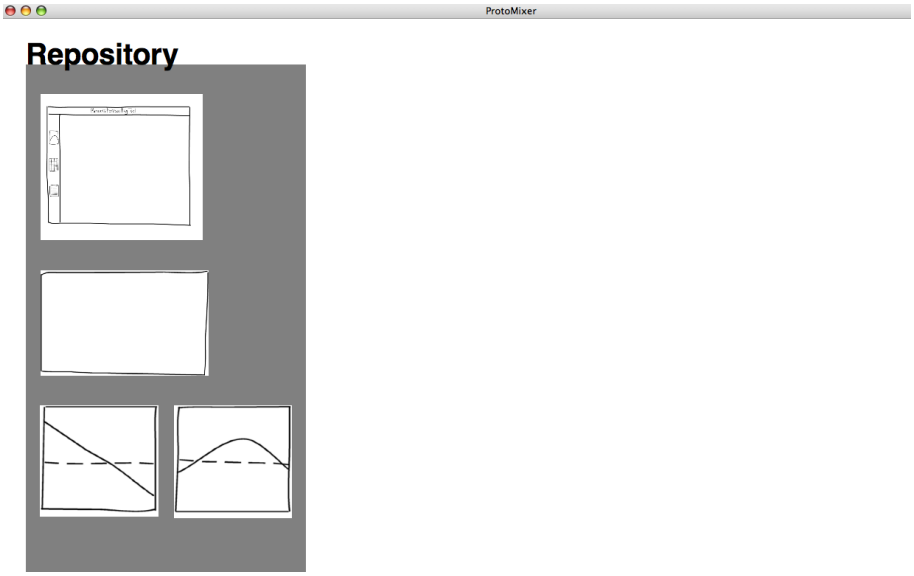


**Figure 6.1:** Initial state of ProtoMixer upon loading assets into the Repository

We then proceed with prototyping by laying out the screen designs on the design space. First we lay out the rough screen sketch that has three different view options: 'Chart', 'Financial Statement', and 'Accounting Details', as shown in Figure 6.2.

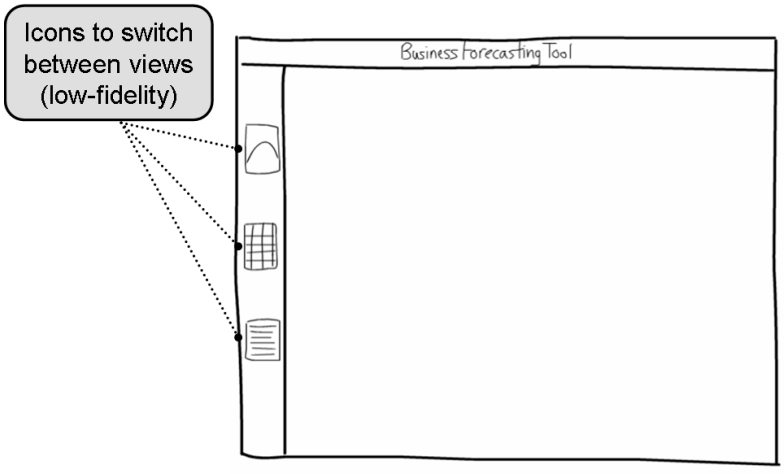


**Figure 6.2:** A sketched prototype of the screen

Next we make a clone of this screen sketch, and drag and drop the sketched elements to create the two different views. Specifically, we create the 'Chart View' by dragging on the two chart sketches. We also create the 'Financial Statement View' by dragging on the sketched table. The resulting designs of the two views are shown in ProtoMixer in Figure 6.3.

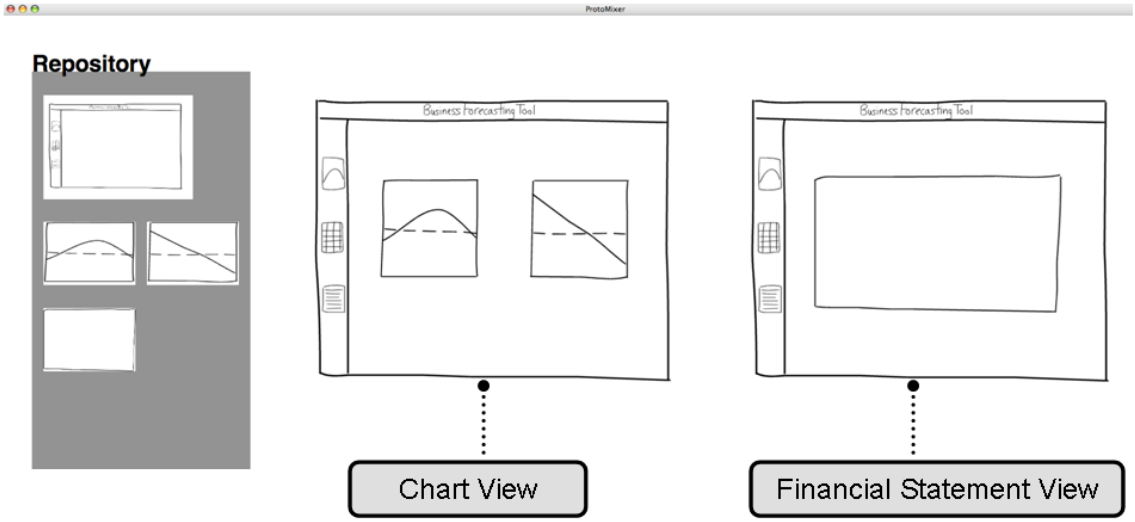


**Figure 6.3:** Leading Scenario created in ProtoMixer

## 6.2 Achieving the Scenarios

Now we are ready to proceed with the scenarios from Chapter 3, building from where we left off in the leading Scenario.

### 6.2.1 Achieving Scenario 1: Composing lightweight prototypes by mixing fidelities

We start off this Scenario by refining the 'Charts View' screen. First, we need to add fields for selecting the types of drivers to chart out as well as add labels categorizing the revenue versus cost drivers. We do this by sketching (in an external application) the new details into the current screen skeleton design. We then import this new sketch into our system and replace the previous skeleton sketch with the new one. This replacement is done by modifying the region's asset, or specifically the asset's file location, on the graphical command panel. Refer to Figure 6.4 for the resulting design.

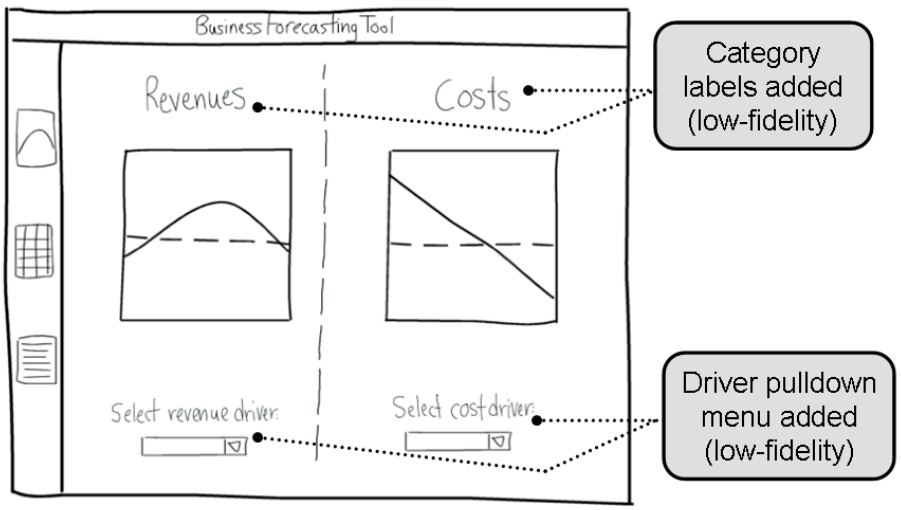


**Figure 6.4:** 'Chart View' after sketching labels and pulldown menus

Next we want to give the sketched charts a more refined look, so we import medium-fidelity images of the charts. We use the Fill-to-size command to automat-

ically place the images on top of the sketched versions and resize them to match the sketches. The result is shown in Figure 6.5.

Our final modification to the 'Charts View' in this scenario is to label the two charts' axes, as shown in Figure 6.6. We do this in the same manner as we added revenue and cost driver details, that is, by replacing the asset of the current sketch with a new externally modified sketch.

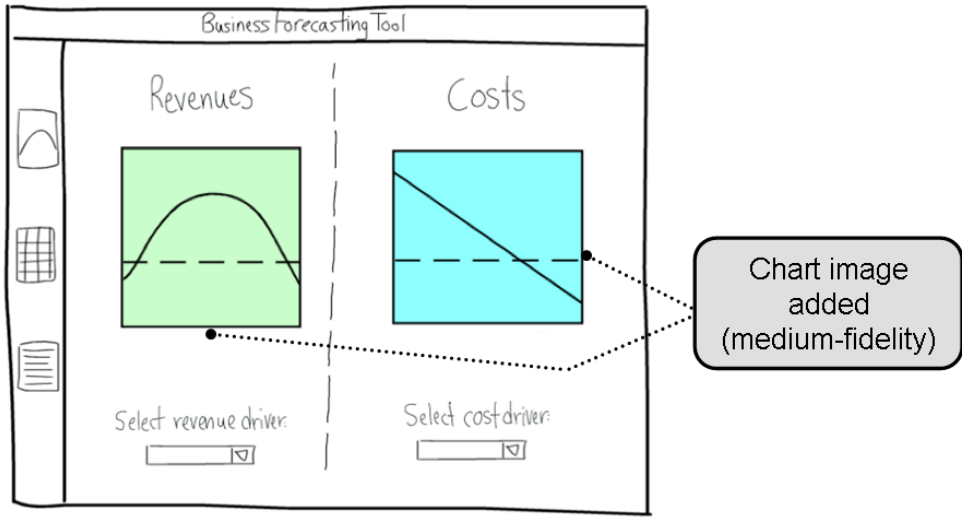


**Figure 6.5:** 'Chart View' after adding images of the charts

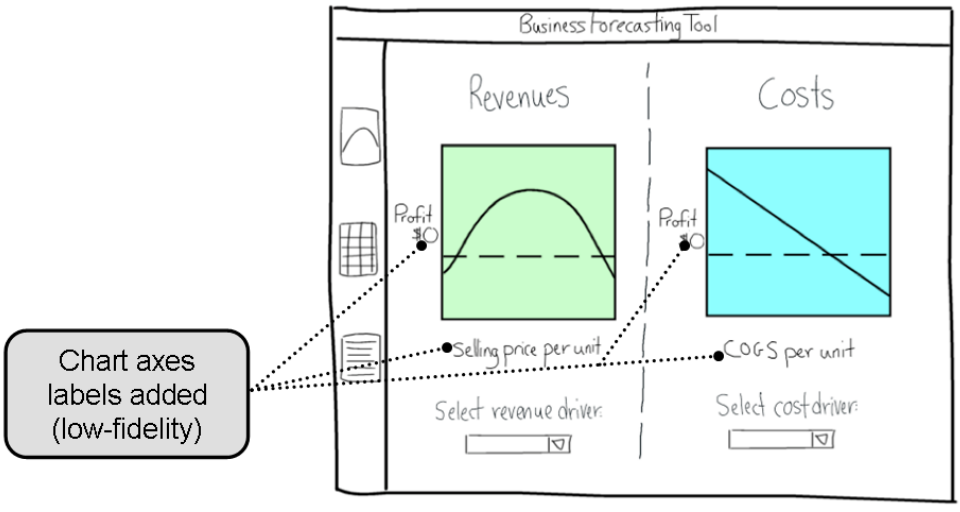


**Figure 6.6:** 'Chart View' after adding chart axes labels

We now turn to working on refining the table element in the 'Financial Statement View' screen. We sketch out example data to put in the table. We then import this new table element into the system and use the Fill-to-size command, automatically positioning and resizing the data-filled table sketch on top of the old sketched table outline. Next we add a field for the user to select the type of financial statement to display by importing the sketched pieces and dragging and dropping them into position on top of the screen's design. The resulting design is shown in Figure 6.7.

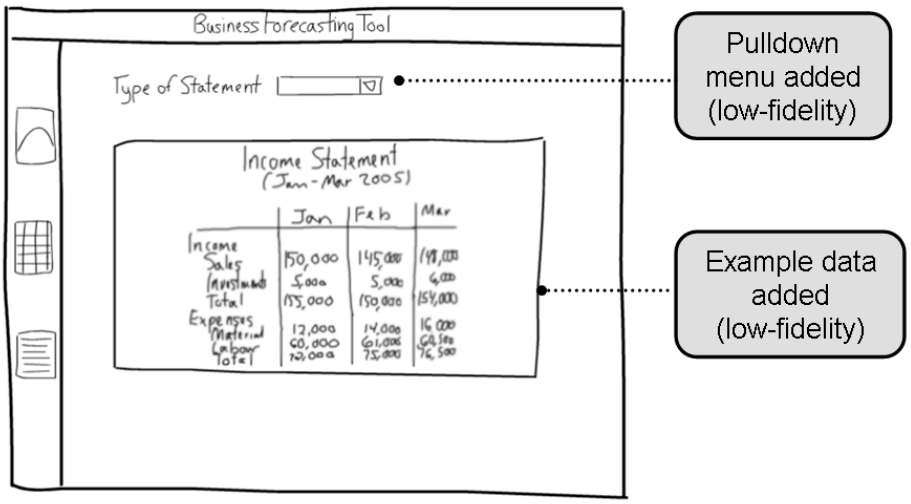


**Figure 6.7:** 'Financial Statement View' after adding sketched data and pulldown menu

Finally, we need to add a high-fidelity table widget to the design. We do this in ProtoMixer by creating a new region using the graphical command panel and setting its asset to be of type "Widget" and the widget object type to be "JTable". This high-fidelity table is then positioned and resized on top of the sketched version using the Fill-to-size command, as shown in Figure 6.8.

The designs resulting from the completion of Scenario 1 using ProtoMixer are shown in Figure 6.9.

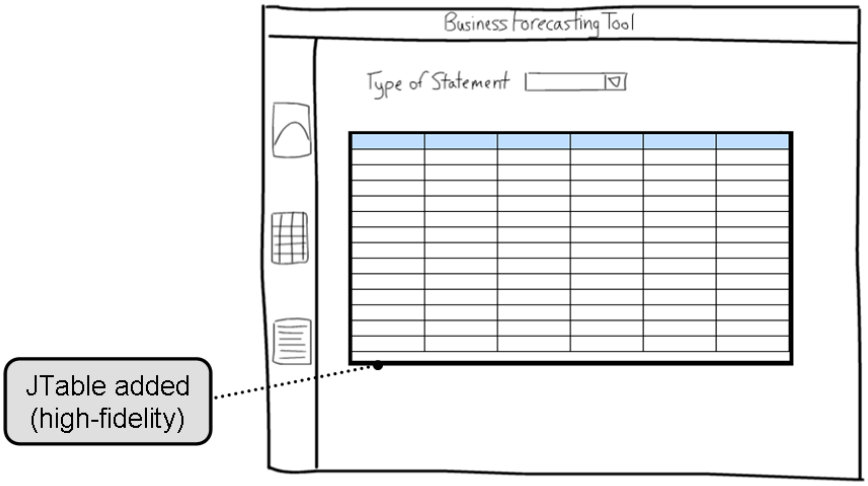


**Figure 6.8:** 'Financial Statement View' after adding a high-fidelity table

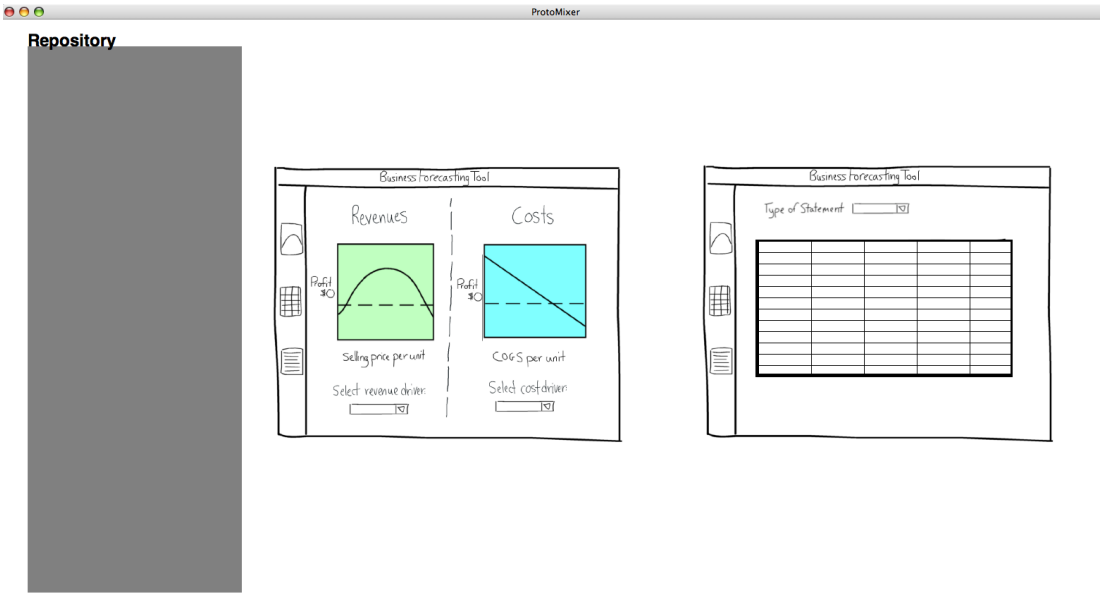


**Figure 6.9:** Scenario 1 completed using ProtoMixer: medium-fidelity images of charts and a high-fidelity table component have been mixed into refined sketch designs

### 6.2.2 Achieving Scenario 2: Transitioning between fidelities as ideas are refined

We continue to refine the design of the charts. First off, we need to try a modification to the charts at the sketched fidelity first before proceeding with higher fidelities. We revert back to the sketches using the Change Layer feature of ProtoMixer, which hides the top layered medium-fidelity images and brings the sketches back into view. We then are able to sketch the new design idea, a vertical guide bar, onto the charts by using the built-in Sketching feature. This is shown in Figure 6.10.

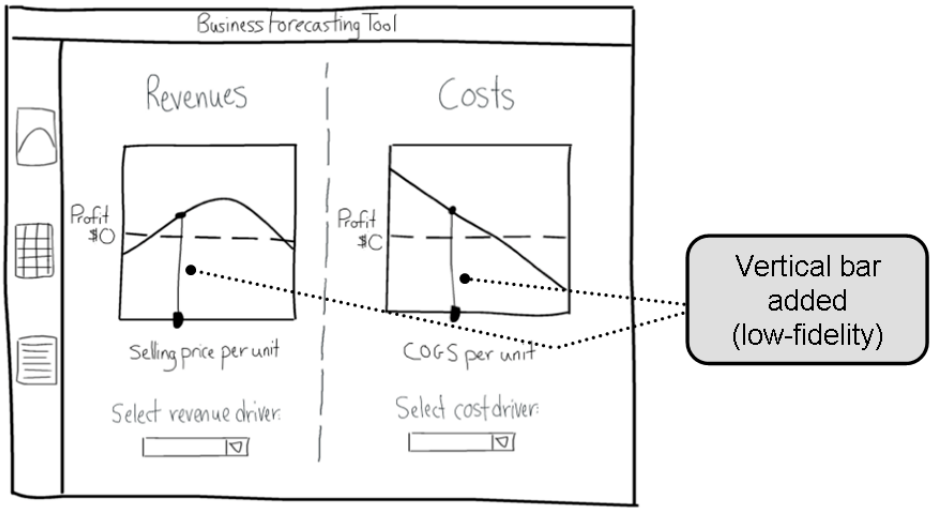


**Figure 6.10:** 'Chart View' after sketching vertical guide bar

Next we create medium-fidelity images of these refined charts externally and import them into ProtoMixer as new regions, with these images as their assets. These new charts are placed exactly on top of the sketched versions using the Fill-to-size command, with the results shown in Figure 6.11.

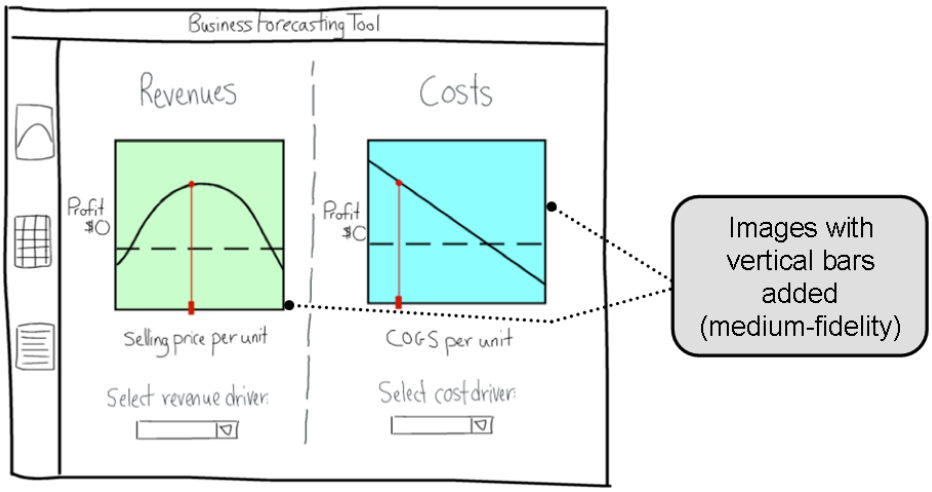


**Figure 6.11:** 'Chart View' after adding updated images of charts

We also modify the appearance of the skeleton screen design by adding fields to select a time period and to display calculated profit. This is done by externally modifying the sketch. Then we update the skeleton screen in our system by replacing the image asset of the 'Chart View' prototype and update all of its cloned regions to match the new version – in this case the only clone is 'Financial Statement View' prototype. The updated sketches are shown in Figure 6.12.

### 6.2.3 Achieving Scenario 3: Integrating domain-specific data and functionality

To accomplish this scenario, we need to put real data into the high-fidelity table as well as use this data to calculate a value for displaying in the profit field. We start off with the design where we have an empty high-fidelity table. In order to add data to the prototype's table, we implement a table widget that we call "FilledJTable". In FilledJTable, we inherit from JTable and initialize the table with arrays filled with

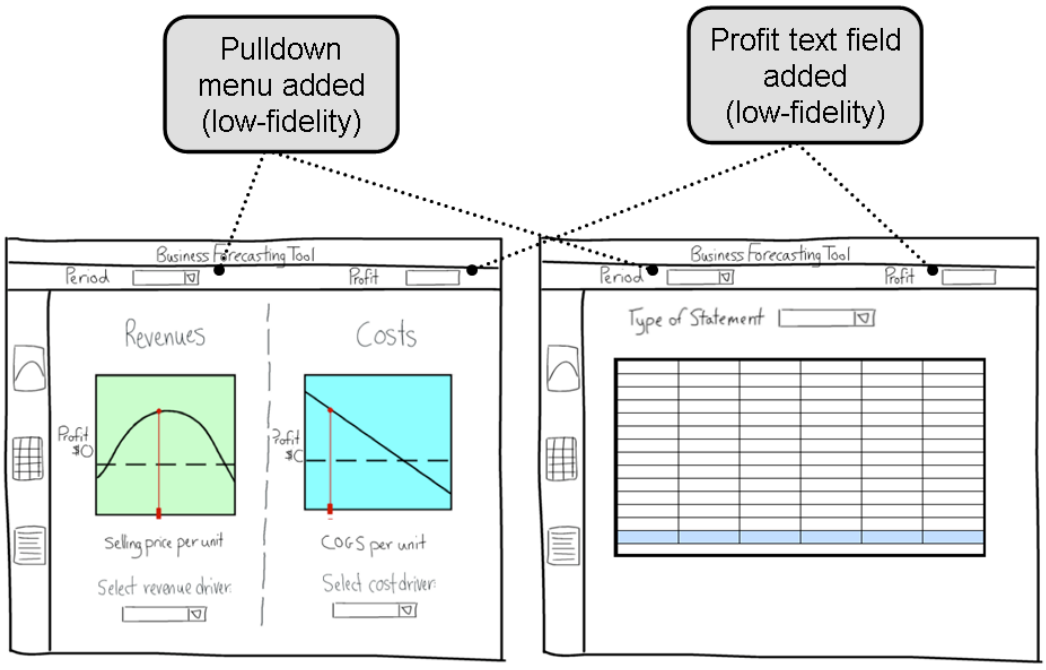


**Figure 6.12:** Both views after adding period pulldown menu and profit field

example data to match the data from the sketch. On the graphical command panel, we then change the asset of the region containing the empty high-fidelity table to instead have FilledJTable as its widget object type (refer to Figure 6.13 for result).

Next we need to display a value in the profit field so we make use one of Java's pre-built text components, JTextField, where we can easily set a value. We create a new region with a JTextField widget as its asset and place it over top of the sketched profit field (refer to Figure 6.13 for result).

Next we need to connect the high-fidelity table and text field widgets together, as they share a common data model. We do this using ProtoMixer's high-fidelity Binding feature. Specifically, we further extend FilledJTable to implement the TablePublisher interface, as described in Section 5.4.3, and we call the new class "LinkedJTable". We also implement a class called "LinkedJTextField" that extends

JTextField and implements the TableObserver interface, again as described in the previous chapter. Because LinkedJTextField's profit value is derived from multiple specific values in the table, we also implement a specialized function for calculating the profit.

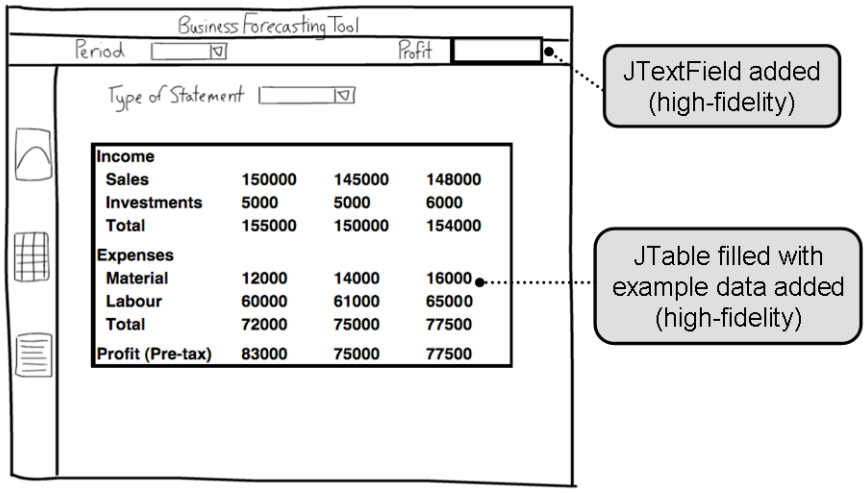


**Figure 6.13:** 'Financial Statement View' after adding the high-fidelity text field and table with data

We have included our specialized function that calculates after tax-profit based on this table's format. This function, called calculateNetProfit, is called by the LinkJTextField's tableChanged method, after tableChanged is invoked by the LinkedJTable class upon its data changing (as discussed in Section 5.4.3).

```
public void calculateNetProfit(Object[][] table)
{
    int totalPreTaxProfit = table[10][1] + table[10][2] + table[10][3];
    int taxes = totalPreTaxProfit * curTaxRate;
    int afterTaxProfit = totalPreTaxProfit - taxes;
    this.setText("\$ "+ afterTaxProfit);
}
```

Once the new widgets are implemented, we compile the new classes and then update the prototype's table and text field regions' assets to map to the new Linked-

JTable and LinkedJTextField classes, respectively. We then specify a connection between these two widgets at run time by using the Bind command, which looks like this in our case: "bind LinkedJTextField LinkedJTable table". The widgets are now connected so whenever the table's data is modified, the text field is notified of the change and is passed the updated data. The text field then uses our built-in function to calculate the profit and displays the resulting value (shown in Figure 6.14).

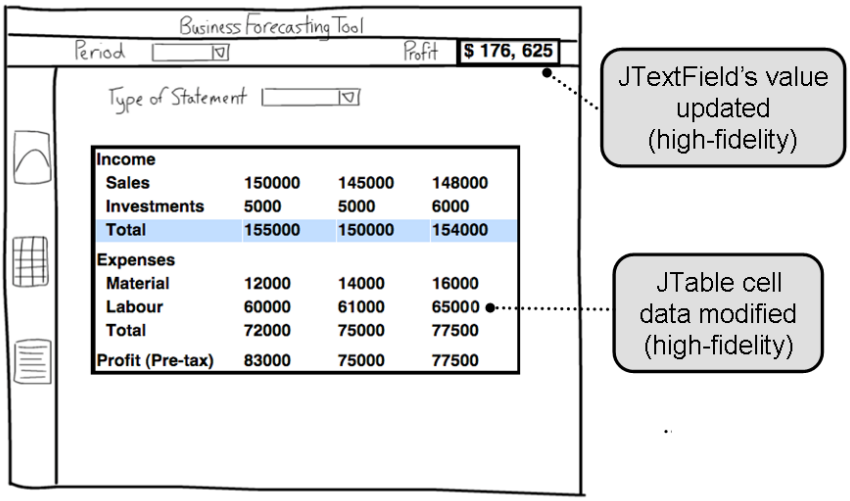


**Figure 6.14:** 'Financial Statement View' after binding text field's value to the table data

The end result of this scenario is shown in Figure 6.15. Note that the text field's value of $176,625 is calculated using the above calculateNetProfit function based on the table's current example data.

### 6.2.4 Achieving Scenario 4: Trying out novel interactive elements

We now work on further refinements to the charts, specifically with regards to the charts' behavior and interactivity. We start off by creating a series of medium-fidelity

images of the charts that illustrate their appearance at some instance in time (as illustrated earlier in Figure 3.4). We then load these charts into the Repository Region of ProtoMixer.

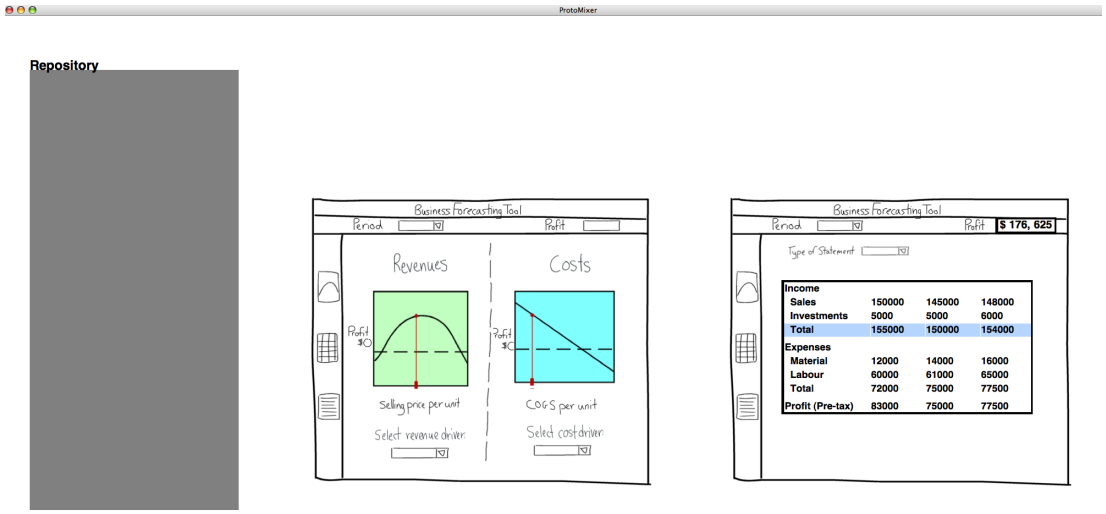


**Figure 6.15:** Scenario 3 completed in ProtoMixer: the high-fidelity table now has real data that is bound to the high-fidelity text field's value

Next we set these new charts as assets of the most recent charts, by updating their asset location in the graphical command panel. We are careful to add the charts in sequential order to allow for animation of the charts' behavior. We now animate the charts by using the Flash Animation feature. We can set the charts to flash through in sequence any number of times.

Now we need to move to a higher-fidelity design with these chart components by giving them some actual behavior as desired in the final design. Because this component is fairly novel and does not exist in the Java toolkit, or any other toolkit for that matter, we compose a component by utilizing and connecting multiple elements.

First we work on the main chart area. As Java's standard toolkit does not offer a chart widget, we implement a simple one on our own and implement it to look like

the sketched versions. Also, when implementing the chart component, we keep in mind that we need to be able to create both line and curve charts as well as need to be able to shift the line and curve based on an inputted shift value. Then we need to use a slider widget for specifying an active x-axis value, so we make use of the JSlider from Java's standard toolkit.

Now that we have the key elements to create our novel chart component, we need to connect them together, to create some behavior based on other element's properties. Specifically, we need the line chart's slider to shift the curved chart's curve and vice versa. We incorporate this into our prototype by further extending the chart widgets to implement the Observer interface, and more specifically, the ValueObserver interface because each chart is interested in receiving a single value from the slider. Next we implement a slider class that extends JSlider and implements the ValuePublisher interface.

We are ready to start composing the component. We import these high-fidelity widgets and lay them out nicely to produce a unified looking novel chart component, as shown in Figure 6.16. We then bind the widgets together using the Bind feature, as described in the previous subsection.

Now we can use our newly composed chart element to demonstrate the design concepts. Dragging the slider below the line chart results the other curve chart shifting upwards or downwards. Conversely, dragging the slider below the curve chart results in the line chart shifting upwards or downwards. The final result of this scenario is shown in Figure 6.17.

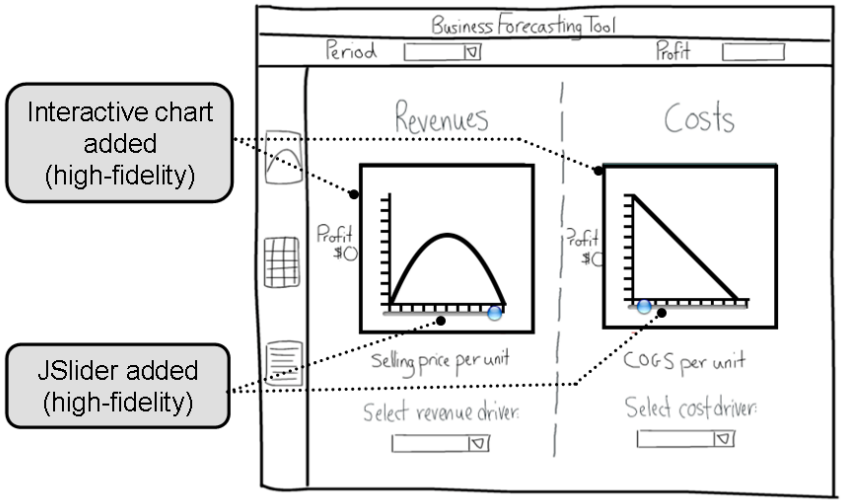


**Figure 6.16:** 'Chart View' after composing a lightweight novel chart element

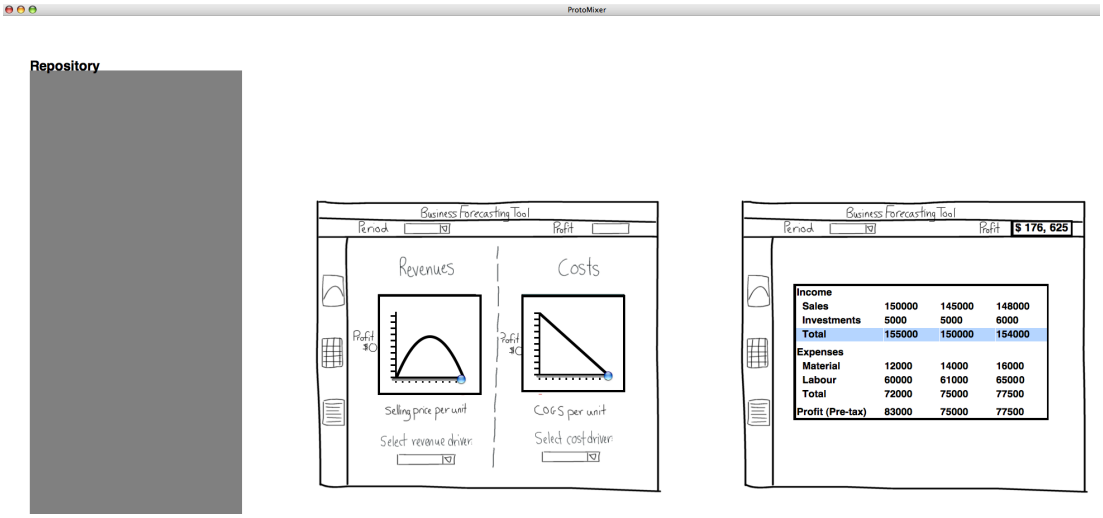


**Figure 6.17:** Scenario 4 completed using ProtoMixer: a rather novel chart component has been composed of a chart widget and a slider widget

### 6.2.5 Achieving Scenario 5: Comparative analysis of alternative designs

We now explore an alternative layout for the prototype. We sketch out an alternate layout idea, which tries to incorporate the charts and financial statements in the same view. We then import the two alternate sketches into ProtoMixer. We use the automatic layout feature to position the two designs side-by-side, as shown in Figure 6.18.

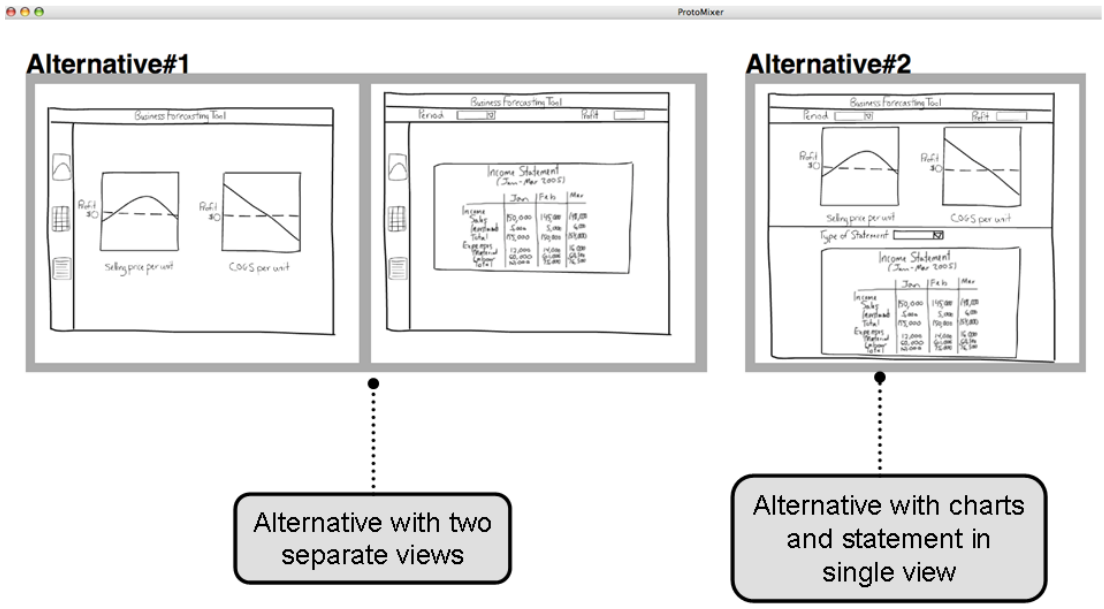


**Figure 6.18:** Comparing two alternate prototype layouts using ProtoMixer

We then annotate the two alternative sketches with the advantages and disadvantages of each. We annotate them by creating two new regions with JTextArea widgets as their assets. We then type in the annotations we wish to make about each design in their respective annotation regions, resulting in the state in Figure 6.19.

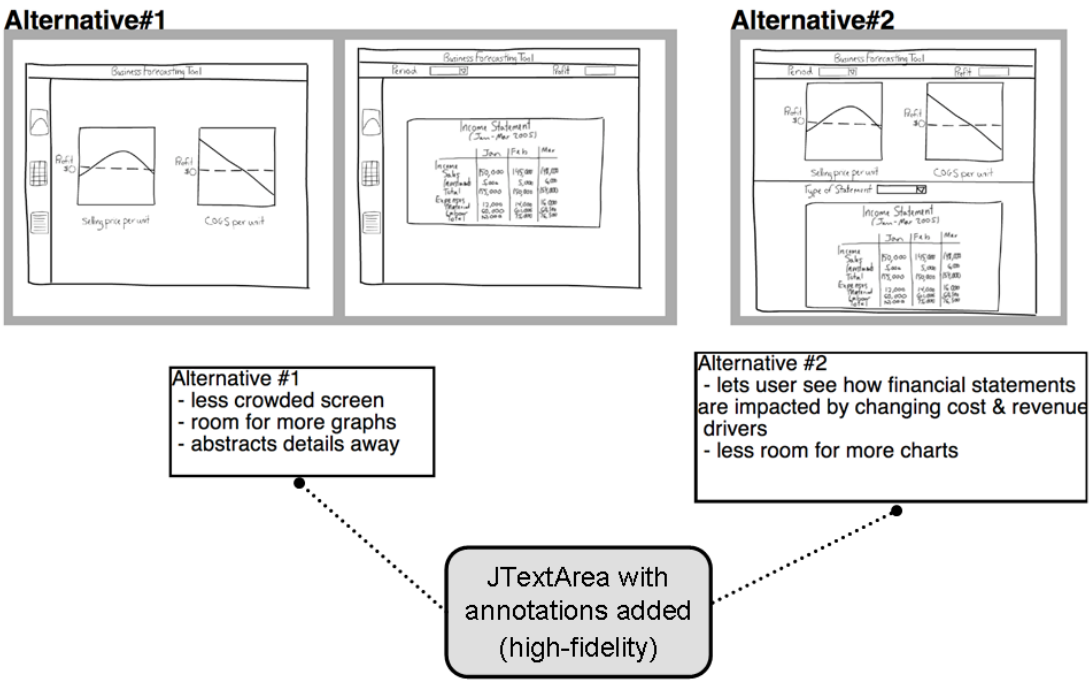


**Figure 6.19:** Annotating the two alternate layouts with pros and cons

We are then able to hide and unhide our annotations for future reference. Specifically, we hide them by shrinking the annotation regions and positioning them into the corner of the sketches they are annotating. We can later unhide them by enlarging them using Fill-to-size or the basic Scale command.

### 6.2.6 Achieving Scenario 6: Recording collaborative design efforts

We now need to look back at how the complex charts have evolved and how the final versions are to be designed in terms of appearance and functionality. We use the View Versions feature in ProtoMixer and specify to see versions of the region representing the revenue driver chart. ProtoMixer then pops up a new region next

to the specified chart, and this new region is displaying all of the previous versions of all fidelities. This state is shown in Figure 6.20.

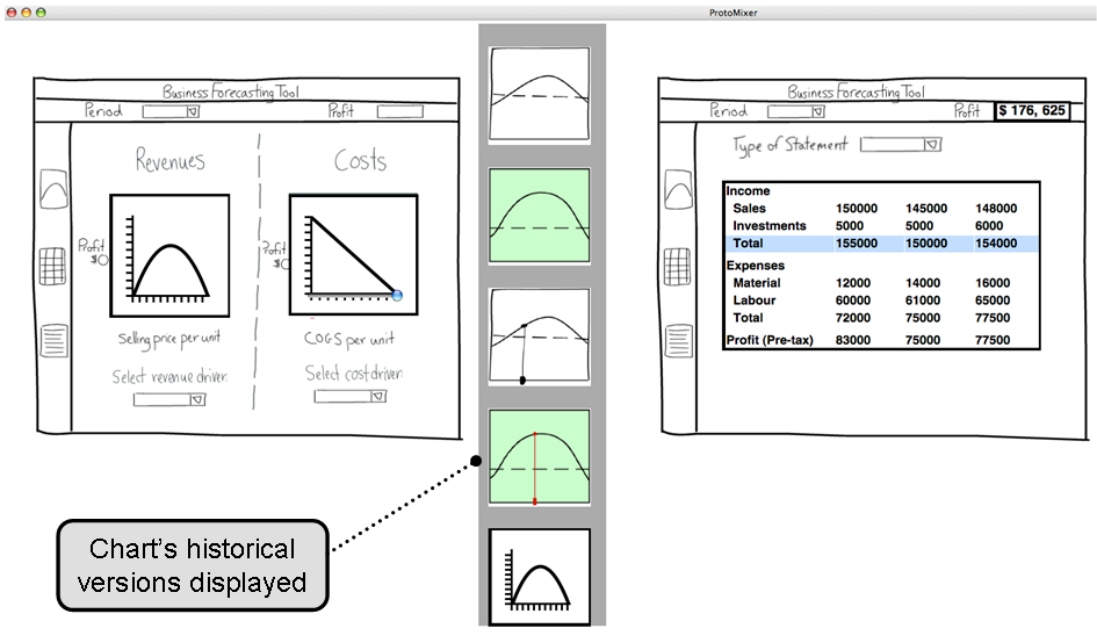


**Figure 6.20:** Scenario 6 completed using ProtoMixer: a new region appears displaying all historical versions of the Revenue Chart

## 6.3 Discussion

The goal in this chapter was to demonstrate how ProtoMixer supports the completion of the scenarios from Chapter 3 and, ultimately, supports mixed fidelity prototyping. Throughout this chapter, it was shown how the scenarios were able to be successfully completed by using ProtoMixer.

While performing the scenarios, some improvements were identified that could be made to ProtoMixer in order to enhance the prototyping capabilities. Ideally, it would be valuable to incorporate the easier tie in of data into ProtoMixer. Currently,

ProtoMixer only supports data through the use of high-fidelity components, which may be pre-loaded with data. It would be beneficial to smoothly integrate interface elements with existing databases. It also seems useful to allow for data models to be associated with lower-fidelity elements instead of only high-fidelity components and ultimately give low-fidelity elements behavior in the same manner as our high-fidelity publish-subscribe Binding feature. Finally, an ideal system would utilize a custom scripting language to allow for easier binding of elements with less coding burden placed on the designers.

Also, in using ProtoMixer to perform the scenarios, it was clear that a better way of interacting with the system is needed. The current command panel is not very intuitive and requires the user to memorize command names and parameters. It would be useful to explore other interfaces to ProtoMixer, such as pie menus or other graphical menu systems that better lay out the available commands and their parameters.

# Chapter 7

# Conclusion

In this chapter, the motivation for this research as well as the contributions of the research are reiterated. Also, areas for possible future extensions and improvements to this research are discussed.

## 7.1 Summary

Under current best practice, user interface prototyping involves performing the three different fidelities of prototyping, but involves refining prototypes at each fidelity prior to advancing to a higher-fidelity. It also involves using different mediums and tools for each fidelity, which results in lack of continuity and traceability between fidelities. Existing tools restrict designers to standard, non-novel design elements and interactions. Also, current practice does not encourage collaboration between user interface designers and the various stakeholders throughout the process. The current approach to prototyping forces designers to prematurely commit to certain design decisions in some cases and does not allow them to explore other issues soon enough in other cases.

This research presented a new approach to user interface prototyping called mixed-fidelity prototyping. Mixing fidelities allows designers the flexibility to focus on one specific aspect of a prototype at a time, by exploring that aspect in the various fidelities. In turn, our approach allows for designers to defer the exploration of less urgent issues, unlike current techniques and tools that heavily restrict designers in their workflow. For example, with Interface Builders designers are immediately forced to choose a layout and specific component types when composing a prototype. Our approach also allows individuals with expertise in a specific fidelity to be involved in that fidelity earlier on. For example, a software developer can begin implementing a high-fidelity component for the prototype at an early stage.

Our approach also addresses some of the issues or shortcomings with the current prototyping practice: multiple fidelities may be explored at any given time, iteration may occur between the different fidelities, user interface designers may better collaborate with each other and with other stakeholders, and potentially more innovative user interface designs may follow. Also, our approach adds continuity and traceability to the process by offering a single unified tool for prototyping in while allowing for designers to still take advantage of paper and whiteboard-based designs that they are familiar with.

A secondary aspect of this research was to enhance the collaborative nature of prototyping by utilizing a large display workspace. Previous work has shown the benefits of using large displays in collaborative settings and in design settings in particular. Large displays seem ideal for this domain of prototyping, a visually rich, team-based domain. While we did not formally evaluate the benefits of using

a large display workspace, it is evident that exploring multiple designs at once, whether it be exploring alternatives or exploring more than one screen design at once, requires more resolution than available on traditional-sized displays. Using tiled high-resolution displays to create a large-display workspace is conducive to our Region Model and supports our approach well.

### 7.1.1 Contributions

This research provides the following contributions:

- A technique for composing user interface prototypes that involves mixing multiple fidelities within a single prototype, which was presented in Chapter 3
- A conceptual model that allows for creating user interface prototypes of multiple fidelities, as presented in Chapter 4
- A proof-of-concept system for supporting the mixed-fidelity prototyping technique, as was described in Chapter 5 and then later illustrated being used in Chapter 6

## 7.2 Future Work

This research could be further explored and extended in several areas. In this section, the following areas for possible future work are discussed:

- Perform a detailed field study with expert as well as beginner designers
- Integrate software engineering models with user interface prototypes

- Explore the benefits of our prototyping approach for different styles of interfaces
- Investigate the use of large display workspaces, including novel interaction techniques
- Implement new features and improve existing features in ProtoMixer

### 7.2.1 Perform a Field Study

It would be useful to perform a field study to further evaluate mixed-fidelity prototyping using ProtoMixer. This field study would be most beneficial if performed using professional designers as the subjects. It would also be interesting to compare the results of expert designers to that of more beginner designers who have yet to become reliant on a particular process or set of tools. As well, it would be interesting to evaluate the usefulness of ProtoMixer as a single-user versus collaborative tool. This field study would be primarily qualitative by nature and should be performed by observing designers at work either in person or through video recordings, conducting interviews or focus groups, and using questionnaires. Such formal user studies have not yet been performed as they require significant time and resources and we have little access to professional designers at this point in time.

### 7.2.2 Integrate Software Engineering Models

It would be worthwhile to explore the integration of software engineering models with user interface prototypes. Software engineering models that may be useful to tie in include but are not limited to domain models, task models, state models, architecture

models, and object-oriented models such as class diagrams or Class-Responsibility-Collaboration(CRC) cards [2]. These software models may share some dependency with the end user interface so it makes sense to evolve the two types of artifacts in parallel. Also, to further tie in with our mixed-fidelity approach, it is reasonable to think of software engineering models also as being different fidelities; for example, CRC cards can be considered low-fidelity models whereas concrete architecture models are higher-fidelity models.

Integrating software engineering and user interface artifacts allows opportunity for a cross-functional design and development team to work together, helping them better meet user and project requirements and adding more continuity and traceability into the design process. The research community has shown considerable interest in attempting to bridge the gap between user interface and software design. Gurantene et al. [7] argue for using high-fidelity prototypes as a bridging artifact and we also recognize the potential in using prototypes to bring the two sides together.

### 7.2.3 Explore Different Styles of User Interfaces

In Chapter 1, we defined a specific style of user interface we aimed to address in our research: a highly graphics-based, direct manipulation interface for traditional desktop machines. It would be interesting to explore our mixed-fidelity lightweight approach for other styles of interfaces. Specifically, it seems worthwhile to explore the design of graphical user interfaces intended for other devices such as embedded devices or PDAs as well as for various sizes of displays such as large display workspaces.

Our approach utilizes a large display workspace and the Region Model, both of which seem conducive to exploring designs for alternative devices and displays.

### 7.2.4 Further Investigate Using Large Displays

It would be useful to further investigate the use of large display workspaces in this domain. Specifically, it would be interesting to explore different existing and novel interaction techniques and devices. Potentially useful techniques and devices may already exist to enhance the designers' experience in using ProtoMixer, but at this point, we have only interacted with ProtoMixer using a standard keyboard and mouse, as well as a sketching tablet. Also, we would be interested in exploring whether the mixed-fidelity prototyping approach would be effective on a traditional desktop display versus a large display workspace, and which features are only possible with a large display.

### 7.2.5 Further Implement Features in ProtoMixer

It would be worthwhile to further extend ProtoMixer. As areas for extensions and improvements were set out in Sections 5.7 and 6.3, they are only briefly mentioned here. Multi-user support and a more complex versioning system could be added. Also, better techniques for interacting with the large display as well as a better interface for specifying commands could be implemented. Furthermore, extensions could be made to allow for easier tie in of data and binding of elements, perhaps

for elements of all fidelities rather than only high-fidelity. These improved or new features could help enhance the usability as well as usefulness of ProtoMixer.

## 7.3 Conclusion

Through this research, we investigated a new approach to prototyping called mixed-fidelity prototyping. The idea of mixing fidelities allows designers the opportunity to focus on a specific interface issue, by exploring it at higher-fidelities and making refinements as needed, while leaving other aspects of the prototype at a lower-fidelity for the time being. By mixing fidelities, designers are able to defer design decisions, while exploring pressing issues, including alternative and potentially more innovative designs. Mixing-fidelities on a large display promotes collaboration, by bringing together individuals with different skill sets that may otherwise work in isolation from one another as well as by eliciting stakeholders for rich feedback earlier on. For these reasons, mixed-fidelity prototyping offers an interesting alternative to the traditional best practice and we believe it is worthwhile to pursue future work in the area.

# Bibliography


[1] Brian P. Bailey, Joseph A. Konstan, and John V. Carlis. Demais: designing multimedia applications with interactive storyboards. In *Proceedings of the Ninth ACM International Conference on Multimedia*, pages 241–250. ACM Press, 2001.

[2] Kent Beck and Ward Cunningham. A laboratory for teaching object oriented thinking. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 1–6. ACM Press, 1989.

[3] Sara A. Bly. A use of drawing surfaces in different collaborative settings. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 250–256. ACM Press, 1988.

[4] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible markup language (XML) 1.0 (third edition). W3 Recommendation available at: http://www.w3.org/TR/REC-xml - Accessed on February 6, 2006.

[5] William Buxton, George Fitzmaurice, Ravin Balakrishnan, and Gordon Kurtenbach. Large displays in automotive design. *IEEE Computer Graphics Applications*, 20(4):68–75, 2000.

[6] Francois Guimbretière, Maureen Stone, and Terry Winograd. Fluid interaction with high-resolution wall-size displays. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 21–30. ACM Press, 2001.

[7] Junius Gunaratne, Beatrice Hwong, Christopher Nelson, and Arnold Rudorfer. Using evolutionary prototypes to formalize product requirements. In *Bridging the Gap II Workshop at ICSE*, 2004.

[8] Komei Harada, Eiichiro Tanaka, Ryuichi Ogawa, and Yoshinori Hara. Anecdote: a multimedia storyboarding system with seamless authoring support. In *Proceedings of the Fourth ACM International Conference on Multimedia*, pages 341–351. ACM Press, 1996.

[9] Brad Johanson, Armando Fox, and Terry Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing Magazine*, 1(2):67–74, 2002.

[10] Bonnie E. John, Len Bass, Rick Kazman, and Eugene Chen. Identifying gaps between hci, software engineering, and design, and boundary objects to bridge

them. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems*, pages 1723–1724, New York, NY, USA, 2004. ACM Press.

[11] D. Austin Henderson Jr. and Stuart Card. Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Trans. Graph.*, 5(3):211–243, 1986.

[12] Rick Kazman, Len Bass, and Jan Bosch. Bridging the gaps between software engineering and human-computer interaction. In *Proceedings of the 25th International Conference on Software Engineering*, pages 777–778, Washington, DC, USA, 2003. IEEE Computer Society.

[13] Rick Kazman, Len Bass, and Bonnie John. Bridging the gaps ii: Bridging the gaps between software engineering and human-computer interaction. In *Proceedings of the 26th International Conference on Software Engineering*, pages 773–774, Washington, DC, USA, 2004. IEEE Computer Society.

[14] Scott Klemmer, Mark Newman, Ryan Farrell, Raecine Meza, and James A. Landay. A tangible difference: Participatory design studies informing a designers outpost. In *Workshop on Shared Environments to Support Face-to-Face Collaboration*. ACM Press, 2000.

[15] Scott R. Klemmer, Mark W. Newman, Ryan Farrell, Mark Bilezikjian, and James A. Landay. The designers' outpost: a tangible interface for collaborative web site. In *Proceedings of ACM Symposium on User Interface Software and Technology*, pages 1–10. ACM Press, 2001.

[16] Scott R. Klemmer, Michael Thomsen, Ethan Phelps-Goodman, Robert Lee, and James A. Landay. Where do web sites come from?: capturing and interacting with design history. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1–8. ACM Press, 2002.

[17] James A. Landay and Brad A. Myers. Interactive sketching for the early stages of user interface design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 43–50. ACM Press/Addison-Wesley Publishing Co., 1995.

[18] James Lin, Mark W. Newman, Jason I. Hong, and James A. Landay. Denim: finding a tighter fit between tools and practice for web site design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 510–517. ACM Press, 2000.

[19] James Lin, Michael Thomsen, and James A. Landay. A visual language for sketching large and complex interactive designs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 307–314. ACM Press, 2002.

[20] Michael J. Muller. Pictive-an exploration in participatory design. In *Proceedings of the SIGCHI Conference on Human factors in Computing Systems*, pages 225–231. ACM Press, 1991.

[21] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction*, 7(1):3–28, 2000.

[22] Elizabeth D. Mynatt, Takeo Igarashi, W. Keith Edwards, and Anthony LaMarca. Flatland: new dimensions in office whiteboards. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 346–353. ACM Press, 1999.

[23] University of Waikato HCI Group. Large interactive display surfaces (lids). Available at http://www.cs.waikato.ac.nz/hci/lids.html - Accessed on February 6, 2006.

[24] John K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[25] Elin R. Pedersen, Kim McCall, Thomas P. Moran, and Frank G. Halasz. Tivoli: an electronic whiteboard for informal workgroup meetings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 391–398. ACM Press, 1993.

[26] Beryl Plimmer and Mark Apperley. Freeform: A tool for sketching form designs. In *Proceedings of 17th Annual Human-Computer Interaction Conference (BHCI)*, pages 183–186, 2003.

[27] Jun Rekimoto. Pick-and-drop: a direct manipulation technique for multiple computer environments. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 31–39. ACM Press, 1997.

[28] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[29] Jim Rudd, Ken Stern, and Scott Isensee. Low vs. high-fidelity prototyping debate. *Interactions*, 3(1):76–85, 1996.

[30] Norbert A. Streitz, Jörg Geißler, Torsten Holmer, Shin'ichi Konomi, Christian Müller-Tomfelde, Wolfgang Reischl, Petra Rexroth, Peter Seitz, and Ralf Steinmetz. i-land: an interactive landscape for creativity and innovation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 120–127. ACM Press, 1999.

[31] John C. Tang and Larry J. Leifer. A framework for understanding the workspace activity of design teams. In *Proceedings of Conference on Computer-Supported Cooperative Work*, pages 244–249. ACM Press, 1988.

[32] Barbara Tversky. What does drawing reveal about thinking? In *Proceedings of Visual and Spatial Reasoning in Design*, pages 93–101, 1999.

[33] Iris Vessey and Ajay Paul Sravanapudi. Case tools as collaborative support technologies. *Communications of the ACM*, 38(1):83–95, 1995.

[34] Miriam Walker, Leila Takayama, and James A. Landay. High-fidelity or low-fidelity, paper or computer medium? In *Proceedings of the Human Factors and Ergonomics Society 46th Annual Meeting*, pages 661–665, 2002.

# Appendix A

# Region Model Notation

## A.1 Complete XML Schema

Below is the complete XML schema for the Region Model, where Design Space is the root element.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

   <xs:simpleType name="AssetTypeOptions">
      <xs:restriction base="xs:string">
         <xs:enumeration value="Sketch"/>
         <xs:enumeration value="Image"/>
         <xs:enumeration value="Widget"/>
         <xs:enumeration value="Shape"/>
      </xs:restriction>
   </xs:simpleType>

   <xs:complexType name="AssetType">
      <xs:attribute name="type" type="AssetTypeOptions" use="required"/>
      <xs:attribute name="location" type="xs:anyURI" use="required"/>
   </xs:complexType>

   <xs:complexType name="ScriptType">
      <xs:attribute name="behavior" type="xs:string" use="required"/>
   </xs:complexType>

   <xs:complexType name="LayoutType">
      <xs:attribute name="regions" type="xs:string" use="required"/>
      <xs:attribute name="algorithm" type="xs:string" use="required"/>
   </xs:complexType>

   <xs:complexType name="RegionType">
      <xs:sequence>
         <xs:element name="Asset" type="AssetType" minOccurs="0"/>
         <xs:element name="Script" type="ScriptType" minOccurs="0"/>
         <xs:element name="Layout" type="LayoutType" minOccurs="0"/>
```

```
            <xs:element name="Region" type="RegionType" minOccurs="0"
            maxOccurs="unbounded"/>
         </xs:sequence>
         <xs:attribute name="name" type="xs:string" use="required"/>
         <xs:attribute name="x" type="xs:integer" use="required"/>
         <xs:attribute name="y" type="xs:integer" use="required"/>
         <xs:attribute name="z" type="xs:integer" use="optional"/>
         <xs:attribute name="width" type="xs:integer" use="required"/>
         <xs:attribute name="height" type="xs:integer" use="required"/>
         <xs:attribute name="color" type="xs:string" use="optional"/>
         <xs:attribute name="alpha" type="xs:integer" use="optional"/>
         <xs:attribute name="historylist" type="xs:string" use="optional"/>
         <xs:attribute name="parentRegion" type="xs:string" use="required"/>
         <xs:attribute name="subRegions" type="xs:string" use="optional"/>
      </xs:complexType>

      <xs:complexType name="RelationshipType">
         <xs:attribute name="name" type="xs:string" use="required"/>
         <xs:attribute name="sourceRegion" type="xs:string" use="required"/>
         <xs:attribute name="targetRegion" type="xs:string" use="required"/>
         <xs:attribute name="sourceProperty" type="xs:string" use="optional"/>
         <xs:attribute name="targetProperty" type="xs:string" use="optional"/>
      </xs:complexType>

      <xs:complexType name="CommandType">
         <xs:attribute name="name" type="xs:string" use="required"/>
         <xs:attribute name="regionToAffect" type="xs:string" use="optional"/>
         <xs:attribute name="miscParameters" type="xs:string" use="optional"/>
      </xs:complexType>

      <xs:element name="DesignSpace">
         <xs:complexType>
            <xs:choice>
               <xs:element name="Region" type="RegionType" minOccurs="1"/>
               <xs:element name="Relationship" type="RelationshipType"
               minOccurs="0"/>
               <xs:element name="Command" type="CommandType" minOccurs="0"/>
            </xs:choice>
         </xs:complexType>
      </xs:element>

   </xs:schema>
```

## A.2 Example XML Document

Below is the complete XML document describing the Design Space illustrated in Figure 4.5.

```
<DesignSpace>
<Region name="Root" x="0" y="0" width="1000" height="800"
color="255, 255, 255" parentRegion="null">
   <Region name="Repository" x="36" y="121" width="577" height="1262"
   color="128, 128, 128" parentRegion="Root">
      <Region name="medgraph_a" x="31" y="956" width="215" height="199"
      parentRegion="Repository">
         <Asset type="Image" location="./images/medgraph_a.png"/>
      </Region>
      <Region name="filledtable" x="31" y="376" width="431" height="265"
      parentRegion="Repository">
         <Asset type="Widget" location="FilledJTable"/>
      </Region>
      <Region name="medgraph_b" x="290" y="955" width="217" height="196"
      parentRegion="Repository">
         <Asset type="Image" location="./images/medgraph_b.png"/>
      </Region>
      <Region name="sketchedtable" x="29" y="668" width="389" height="262"
      parentRegion="Repository">
         <Asset type="Image" location="./images/chart.png"/>
      </Region>
      <Region name="mycurvechart" x="31" y="127" width="218" height="211"
      parentRegion="Repository">
         <Asset type="Widget" location="MyChart"/>
      </Region>
      <Region name="mylinechart" x="281" y="125" width="218" height="209"
      parentRegion="Repository">
         <Asset type="Widget" location="MyLineChart"/>
      </Region>
   </Region>
   <Region name="chartview" x="660" y="716" width="695" height="598"
   parentRegion="Root">
      <Asset type="Image" location="./images/Sketchv1.png"/>
      <Region name="medgraphb_nobar" x="424" y="186" width="190" height="209"
      parentRegion="chartview">
         <Asset type="Image" location="./images/medgraph_b_nobar.png"/>
      </Region>
      <Region name="medgrapha_nobar" x="137" y="187" width="198" height="208"
      parentRegion="chartview">
         <Asset type="Image" location="./images/medgraph_a_nobar.png"/>
```

```
      </Region>
   </Region>
   <Region name="statementview" x="1375" y="718" width="695" height="598"
   parentRegion="Root">
      <Asset type="Image" location="./images/Sketchv2.png"/>
      <Region name="table" x="136" y="124" z="1" width="473" height="288"
      color="128, 128, 255" alpha="255" parentRegion="statementview">
         <Asset type="Widget" location="FilledJTable"/>
      </Region>
   </Region>
</Region>
</DesignSpace>
```

# Appendix B

# Commands Implemented in ProtoMixer

**Table B.1:** ProtoMixer Commands

| Command | Description | Notation |
|---|---|---|
| addRegion | create a region and add it to the display as a child of parent | add regionName x y w h resource parent |
| deleteRegion | delete a region from the system | delete regionName |
| select | indicate selected region with yellow highlighting | select regionName |
| scale | resize image (and have it scale slowly for visual feedback) | scale regionName % or %w %h |
| move | reposition a region from current position to target position (have the movement animated slowly for visual feedback) | move regionName targetX targetY |
| clone | create a copy of a region and all its subregions(give clone a new name of originalNameClone) and optionally reposition the clone at targetX and targetY | clone regionName [targetX targetY] |
| changeTransparency | change the transparency of the specified region to be the specified float value | alpha regionName alpha |
| cutOutPiece | encapsulate a specified area of a region(s) as a new region (user defines area by drawing a line from one corner to the opposite corner of the hotspot rectangle) | ALT Key + draw hotspot boundary with mouse |
| fillToSize | move a region (source) to overlay within another region (target) and resize it to fit | fill sourceRegionName targetRegionName |
| performLayout | apply the specified layout or default layout to the region | layout groupName [layoutType] or layout parentName layoutType |
| restrictWithin | restrict a region to move within its parent only | within regionName |
| alongPath | restrict a region to move along the specified path only | path regionName x1 y1 x2 y2 |
| record | record history for a specific region, creating cloned regions of itself on a regular interval for some number of cycles | record regionName numberOfCycles |
| | | Continued On Next Page ... |

| Command | Description | Notation |
|---|---|---|
| versions | show the historical versions recorded for the specified region (appears in a new region beside the current region) | versions regionName |
| import | load the specified design space file into the system | import xmlFile |
| export | export the current design space to export.xml or another file | export [xmlFile] |
| storyboard | show navigational flow of the storyboard, using either animated highlighting of regions or animated arrows between regions | storyboard arrows or highlight |
| showClones | show lines illustrating all clones of the specified original region | showclones regionName |
| unshowClones | hide lines of current clones being illustrated | unshowclones |
| updateClones | updates all clones to look like the original regions asset | updateclones regionName |
| removeExtras | removes region name labels, resize boxes to clean up presentation | removeextras |
| flashAnimation | flash through a sequence of assets associated with regionName, some specified number of cycles | flash regionName numberOfCycles |
| snapshot | take a snapshot of the specified region and save it as a png file | snapshot regionName |
| switch | switch the runtime version to either java2d or jogl | switch |