Expanding Interface Design Capabilities through Semantic and Data-Driven Analyses

Amanda Swearngin

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Amy J. Ko, Chair

James Fogarty, Chair

Rastislav Bodik

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

Expanding Interface Design Capabilities
through Semantic and Data-Driven Analyses

Amanda Swearngin

Co-Chairs of the Supervisory Committee:
Professor Amy J. Ko
The Information School

Professor James Fogarty
Paul G. Allen School of Computer Science & Engineering

The design of an interface can have a huge impact on human productivity, creativity, safety, and satisfaction. Therefore, it is crucial that we provide user interface designers with the tools to make them efficient, more creative, and better understand their users. However, designers face key challenges in their tools throughout the design process. Designers explore alternatives of their interface layouts when prototyping. However, they are limited to exploring the layouts they can ideate and sketch, create in their prototyping tools, or find in other examples not containing their interface elements. In usability testing, designers can conduct large-scale studies and deploy their interfaces to gather data from crowdworkers, however, such studies can be expensive, time consuming, and difficult to conduct iteratively throughout the design process. Finally, designers often find that existing interfaces can be a platform for prototyping and enabling new forms of interaction, but existing interfaces are often rigid and difficult to modify at runtime. In this dissertation, I explore how we can use advanced technologies from program analysis and synthesis and machine learning, to enable semantic and data-driven analyses of interfaces. If we augment interface design tools with the capabilities of understanding, transforming, augmenting, and analyzing an interface design, we can advance designers' capabilities. Through semantic analysis of interfaces, we can help designers ideate more rapidly, prototype more

efficiently, and more iteratively and cheaply evaluate the usability of their interface designs. I demonstrate this through four systems that (1) let designers rapidly ideate alternative layouts through mixed-initiative interaction with high-level constraints and feedback, (2) help designers adapt examples more efficiently by inferring semantic vector representation from an example screenshot, (3) enable designers to quickly and cheaply analyze a key aspect of the usability of their interfaces through a machine learning approach for modeling mobile interface tappability, and (4) prototype new modalities for existing web interfaces through applying program analysis to infer an abstract model of interface commands.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to thank my advisors James Fogarty and Amy Ko who have taught me so much throughout this process. Amy taught me how to transform a disorganized mess of research ideas into a crisp and clear direction. She also taught me how to distinguish between engineering and research problems, enabling me to make quicker progress in my research over getting bogged down in engineering details. She provided so much encouragement and support I needed to gain confidence in myself and my abilities when I was just getting started in my Ph.D. James provided me with many valuable insights that transformed my research, and could always come up with just the right piece of literature to move the research forward. I enjoyed the many hours spent in his office brainstorming how to move my research forward. Both advisors provided me detailed critiques of my research papers that I built upon to develop my skills to write an engaging and high quality research paper. They provided me with valuable connections to industry researchers in order to find internships and full-time opportunities after I graduate. They both supported me immensely throughout the Ph.D. process, and targeted much of their advising toward my specific industry career goals. Without them, this dissertation would not have been possible.

I am also very thankful for my committee members, Ras and Audrey. Ras provided me with several key insights for how to move Scout forward and provided valuable feedback during my examples. Audrey, my GSR, helped me think about my work from a designer's perspective, and also provided valuable feedback.

I would also like to give a huge thanks to all of my wonderful mentors and friends from my industry internships, without which much of the work in this thesis would not have been possible. Joel Brandt gave me my first industry research internship opportunity and was an outstanding mentor. Together, we came up with the initial ideas for Rewire, which started me on the focus in interface design tools. Morgan Dixon was an awesome mentor on this project, and helped me find technical direction. In fact, it was Morgan's work that initially drew me to apply to UW. Mira Dontcheva and Wilmot Li taught me so much about industry research, and gave me great guidance during my second internship at Adobe. The Miracles at Adobe during summer 2017 helped me make it through the CHI deadline push, including Daniel, Ailie, Jasper, and Yeasul. It was a pleasure to have hands-on experience learning machine learning and data-driven research from Yang Li at Google during my internship. I also had the pleasure of working with Shamsi Iqbal at Microsoft Research who was incredible to work with, and helped me find organization and direction on a disarray of project ideas and supported me in learning how to navigate collaboration and communication between multiple product teams.

Many thanks to my labmates in James' lab "Fogies" including Annie, Ravi, Jessie, Jina, Alex, and all new Fogies. I would also like to thank some of the the "Old Fogies" who gave feedback on my work and presentations including Daniel, who was an awesome labmate, and Xiaoyi, who has also generously helped me with my industry job search. I would also like to thank my friends and fellow grad students in HCI at UW who provided me social connection and gave valuable feedback on my work.

During my Ph.D., I had the awesome pleasure to be a member of Amy's Code & Cognition lab, and I thank them immensely for being there for me throughout my Ph.D. process, listening

to my "Woos and Boos", giving feedback on my practice talks, and providing critiques on my research. I would specifically like to thank fellow Ph.D. students Dastyni, Benji, Yim, Greg, Kyle, Mina, Neil Ryan, and Alannah Oleson, who was also an awesome collaborator on Scout. I would also like to thank former member of Amy's lab Brian Burg, who met with me for lunch a couple times to give awesome advice on internships and Ph.D. life. Among this group, I would also like to thank the iSchool cohort of friends I made in my first-year in Jake's class who helped me make it through this process and feel a part of their community.

I am quite certain that I could not have made it through my Ph.D. without my awesome fellow PLSE labmates as well including Chandra, Eunice, Jared, Max, Martin, Remy, Pavel, Talia, Sam, James, John, Doug, Sarah, and all other current and former labmates and professors for facilitating an awesome lab culture. Without them, I would not have felt like such a member of the UW CSE community. With the PLSE group, I had many discussions, fun social hangouts, and a community of friends. Specifically, I would like to thank Chandra for being my closest confidant and friend at UW throughout my Ph.D. and for joining me on countless movie and dinner outings. I would like to also thank Chenglong, who was a fantastic collaborator for Scout and contributed significantly to that project's success.

I would also like to thank my Seattle family, friends, and board game crew, without which I would not have had a way to escape the stresses of Ph.D. life and integrate myself into the Seattle community. I would like to thank my church friends who supported me in this process sand welcomed me into their communities. I would also like to thank my fellow choir members of Joyful! Noise in Seattle, who supported me, and its directors, who have built an awesome community of singers.

Finally, I would like to thank my family, without whom none of this work would have been possible. Thank-you immensely to my two best friends and sisters, Stephanie and Elizabeth, who provided me with lots of emotional support, laughs, and encouragement to make it through my Ph.D, and even in some cases, house cleaning and meals to get me through paper deadlines.

The biggest thanks go to my mom and dad who gave me the independence to discover my passions in life. They provided me with a strong foundation to develop my curiosity, dedication, and a strong will without which I could not have made it through my Ph.D.

This work was financially supported by the National Science Foundation through the Graduate Research Fellowship Program, and through research grants CCF-1153625, IIS-1053868, IIS-1314399, and IIS-1702751. This work was also supported through two internships at Adobe Research, and an internship at Google Research.

Dedication

To my mom, my dad, my favorite big sister, and my favorite little sister.

Chapter 1

Introduction

Interface design can have a huge impact on human productivity, satisfaction, and numerous other factors [194]. Good interface design can help business make more money [174], keep people safe [194], and can empower people to be creative. In contrast, bad user interface design can cause user frustration, dissatisfaction, and errors [194]. Due to the importance of user interface design, it is crucial that we provide the designers of user interfaces with the tools to make them more efficient, creative, and better understand their users. There are an estimated 238,000 user interface designers in the United States alone [6], and companies rely on these trained professionals for improving their interfaces. Giving designers the tools they need to succeed can help them make their software useful, usable, and enjoyable.

1.1 Design Activities

In this dissertation, I explore how we can build better software tools for interface designers to accelerate their capabilities in key design activities. To understand the context of these

activities, I first describe a process that designers follow of *design thinking*[1]. This model of design consists of five stages including 1) *emphasize*, where designers observe, understand, and engage with their target audience, 2) *define the problem*, where the designer states the users core problems based on findings from the emphasize stage, 3) *ideation*, where the designer ideates and explores a large set of alternate solutions, 4) *prototyping*, where the designer creates one or more digital or physical prototypes of their ideas (e.g., sketch, paper prototype, digital mockup), and 5) *testing* the solution with one or more potential users. Design thinking is not necessarily linear as designers will often return to previous design stages to iterate and improve upon their design. In this dissertation, I consider three design activities designers use during *ideation*, *prototyping*, and *testing* to create better interface designs.

During *ideation*, interface designers *explore alternative* designs. Exploring alternatives is highly valuable because it can result in higher-quality outcomes and more diverse solutions [47, 66]. Designers can ideate by sketching [47], or they can *look for examples* in online design galleries [120], existing interfaces, or other sources. Viewing and using examples during design can make designers more creative [112].

Prototyping is a key part of the design process that interface designers use to demonstrate and evaluate their ideas. Prototyping not only helps designers concretely evaluate their designs, but can also be a powerful tool for idea generation and design exploration [127]. During prototyping, interface designers create low-fidelity paper prototypes [188] or high-fidelity prototypes in tools like Sketch [19] or Adobe XD [98] that demonstrate the visual design and layout of one or more interface screens in detail. Designers can also create software prototypes in code [158] (e.g., HTML/CSS) to get feedback on early versions of an interface, or they can prototype new interactive features on top of existing interfaces to explore new interactions [63].

Testing is a crucial design activity that designers use to discover problems in their interfaces, and to collect user interaction data. In *usability testing* [158], designers can evaluate low-fidelity, high-fidelity, or software prototypes. Designers can conduct these evaluations through the use

[1] http://dschool.stanford.edu/resources/getting-started-with-design-thinking/

of heuristics [159], in a lab where they have users complete tasks using a paper or digital prototype [188], or through crowdsourcing [60, 106] or remote usability testing where users complete tasks or provide first impressions through online services [16, 17].

1.2 Key Challenges in Design Activities

In this dissertation, I explore a number of challenges designers face during ideation, prototyping, and testing of user interfaces. Exploring alternatives is a key ideation activity [66, 87]. However, interface designers face several challenges when exploring alternatives and examples. First, designers frequently sketch on paper or whiteboards to visualize multiple alternative designs [47]. However, they are limited in how many alternatives they can explore by how many they ideate and sketch by hand. Second, designers gain inspiration for new design alternatives by browsing examples, yet it can be difficult to visualize these examples with the designers own set of interface elements. Finally, designers may "fixate" [99] on a single alternative, finding it difficult to ideate alternative designs outside of what they have already seen.

For prototyping, research and industry have developed sophisticated tools (e.g., Adobe XD [98], Sketch [19], Figma[2]) to ease the prototyping process. However, interface designers still face a number of challenges in prototyping. First, designers need to decide which interface elements to use and where to place them, which requires a deep understanding of usability (e.g., Nielsen's Principles [158]) and visual design (e.g., Gestalt [109]) principles. Additionally, designers may find an inspiring example in an online example gallery, however, these designs are typically shared as screenshots. If a designer wants to adapt elements of the example into their own design, they need manually recreate the shapes and properties of interface elements they want to modify. Some commercial tools for vectorizing examples into editable documents exist [3], however, they represent their output as vector paths rather than *semantic shapes* (e.g., rectangles, text boxes), which can be difficult to edit.

During usability testing, designers can conduct large-scale studies where they can deploy

[2]https://www.figma.com/

[3]https://www.adobe.com/products/illustrator.html

their interfaces to gather data from crowd workers [106]. While crowdsourcing such studies can be cheaper and quicker than lab studies, they can still be expensive and time consuming to conduct. These type of studies typically require having a full interface prototype, which means that any design problems found at this stage will be more difficult to fix. This may also make it challenging for designers to understand which properties of a design are leading to a usability issue [180]. Additionally, designers do not have a common infrastructure for collecting and sharing usability data across interface designs and companies. Such a dataset could help designers discover high-level design insights beyond the design languages of their own organizations.

Finally, designers must *consider the needs of people with diverse abilities* and expertise throughout the design process. People who are blind or low-vision may need to use screen readers to interact with interfaces [40]. People with motor impairments may need to use alternate input devices to interact with an interface [70]. While it is ideal to develop interfaces with these needs in mind, many interfaces released into the world, especially web interfaces [22], do not support these needs. Thus, researchers have developed techniques to enable designers to prototype new interactions for existing interfaces to modify them at runtime [51, 62, 221, 229]. However, such frameworks still rely on understanding and interacting with an application's visible features (i.e., interface elements) or information exposed through accessibility APIs. Web interfaces contain many non-visible behaviors (i.e., keyboard events, touch gestures) that can make predicting the behavior of interactive commands difficult. Second, web interface customization scripts and frameworks [39, 41] typically modify a specific website or aspect of behavior. Designers do not have a generic way to create application-agnostic prototypes to enable new interactions and input modalities for existing web applications.

Common across all of these challenges is that for interface designers, an interface design (i.e., sketch or prototype) is an artifact. A designer may need to do some manual work to transform it into another medium (e.g., a prototype into an alternative, an example screenshot into a design prototype). Current design prototyping tools do not let a designer easily make this transformation. To aid this transition, we can augment interface design tools with **semantic**

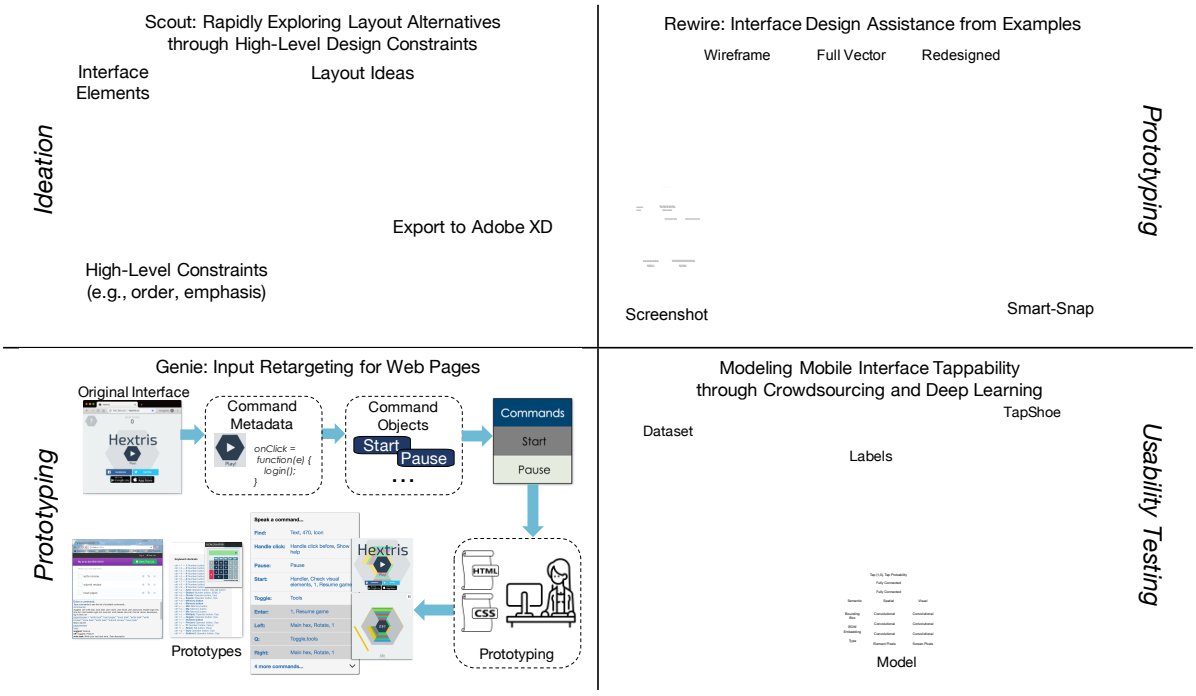


Figure 1.1: During ideation, Scout (top-left) enables designers to explore alternative layouts using high-level design constraints. During prototyping, Rewire (top-right) enables designers to adapt example screenshots by generating three modes of vectorized output from a screenshot. During prototyping, Genie (bottom-left) lets designers prototype new forms of interaction on top of existing web interfaces. During usability testing, TapShoe (bottom-right) enables designers to analyze the tappability of mobile interface designs.

analysis capabilities to understand, analyze, transform, and augment an interface design. Enhancing design tools with such capabilities can aid the design process by making it more efficient, more creative, and less rigid. Through this dissertation, I demonstrate the following thesis statement:

Augmenting interface design tools with high-level semantic knowledge gained through semantic and data-driven analyses can help designers more easily analyze, transform, and augment a design. This can enable them to ideate and prototype more efficiently, and more thoroughly analyze the usability of their interface designs.

To demonstrate this thesis, I present four systems, shown in Figure 1.1, that aid interface designers in ideating, prototyping, and testing. Each system expands upon designers' capabilities to automatically analyze higher-level interface semantics that make an interface designs more

malleable, more easily transformed, and annotated with interaction data. In these systems, I apply recent technological advances from program synthesis, machine learning, and data-driven design to enable these transformations. These systems consist of:

- Scout - a system that supports mixed-initiative interaction with high-level constraints and design feedback to help designers *rapidly visualize layout alternatives*.
- Rewire - a system that infers a vector representation from an interface screenshot where each UI component is a separate object with editable shape and style properties, which can help designers *avoid manual recreation of design prototypes from example screenshots*.
- TapShoe - an approach on gathering usability data on mobile interface tappability at scale, and a machine learning model to help designers explore the tappability of their interfaces without the need to collect any data. With this approach, designers can *avoid the time and cost* of a usability study and can *rapidly evaluate the usability of their designs during prototyping*.
- Genie - a system that reverse engineers an abstract model of web interface commands and provides a prototyping framework to enable designers to *quickly prototype existing interfaces under new input modalities* (e.g., speech, keyboard, command line input) and presentations.

1.3 Outline

Chapter 2 gives an overview of related work. First, I review a set of design principles that I formally model in projects throughout this dissertation. Then, I discuss constraints and their applications in user interface layout tools. I also define *semantic analysis* within the context of this dissertation, and I review how I have applied it to discover interface structure and computationally examine interface usability and aesthetics. Later, I describe key approaches in tools to aid designers in exploring alternatives. Finally, I review the work in data-driven design and describe opportunities that I see to apply these techniques within interface design prototyping tools.

Chapter 3 describes a mixed-initiative system, Scout, that lets designers define high-level

constraints to explore alternative interface layouts. Scout formalizes a set of design principles into high-level constraints that designers can use to specify the high-level layout of alternatives that Scout then translates into low-level constraints to rapidly generate alternate layouts. Designers can also give feedback to Scout to explore and refine the set of alternatives.

Chapter 4 presents a system, Rewire, that helps designers leverage example screenshots through automatically inferring a vector representation that represents each UI component as an object with editable shape and style properties. Rewire provides three modes of design assistance that help designers reuse or redraw components of the example design.

Chapter 5 presents TapShoe, a crowdsourcing approach for gathering usability data at-scale on mobile app tappability. I analyzed a set of signifiers people use to distinguish tappable and non-tappable elements, and I built a deep neural network model that predicts how likely a user will be to perceive an interface element as tappable. Such a model can help designers avoid the time and cost of conducting this type of usability study, and it can potentially help designers understand the effect on tappability of their design choices if integrated into their design workflow during prototyping.

Chapter 6 presents Genie, a system that automatically reverse engineers an abstract model of the underlying commands in a web application and provides a framework that enables designers to prototype new interactions with that functionality through alternative interfaces and other input modalities (e.g., speech, keyboard, command line input).

Finally, Chapter 7 summarizes future directions in integrating computational techniques into design prototyping tools including 1) enabling better prototyping with existing interfaces, 2) advancing mixed-initiative exploration of alternatives, 2) modeling and predicting human perception of usability, 3) building a dataset of design documents to support data-driven design interactions, and 4) integrating semantic analysis and data-driven design features into interface prototyping workflows.

Chapter 2

Background & Related Work

In this chapter, I review areas of past work that provide definitions, theories, and tools for this work, and areas of past work that I am building upon in my research. This chapter is divided into 7 sections.

- In section 2.1, I review and give examples for a set of *design principles*, and describe how we can use them to encode formal and informal design knowledge into high-level constraints, cost functions, and models that can aid designers in applying them in interactive design tools.
- In section 2.2, I review and give examples of *constraints* and describe their scope within modern interface layout and design tools. I then review the key research challenges with constraints in interactive constraint-based layout systems that my work aims to advance the research in or address.
- In section 2.3, I give an overview of past work that applies semantic analysis techniques to discover interface structure from incomplete inputs (e.g., interface screenshots, code) that can be used to aid designers in adapting examples, and in adapting interfaces to

alternate input modalities and presentations.

- In section 2.4, I review related work in semantic analysis of interface usability and visual design. Researchers have applied a variety of approaches to automatically or rapidly evaluate interface usability and visual design through formalized cost models of design principles, crowdsourcing, and machine learning based approaches. There is a huge opportunity to scale these approaches and build techniques for automatically evaluating different aspects of usability and visual design across a diverse set of interface designs.
- In section 2.5, I describe interactive systems and techniques for aiding designers in exploring alternatives. There are two key approaches described in previous work. Designers can define high-level rules and behaviors, and a system generates alternatives automatically, or designers can explore alternatives through examples of other designers work. Few techniques presented in past systems for exploring alternatives have been actively explored within interface design prototyping tools.
- In section 2.6, I review data-driven interface design and describe opportunities for applying these techniques within interface design tools, specifically in using machine learning models to automatically provide usability evaluations.
- In section 2.7, I review the history and current state of digital interface design prototyping tools, and I describe gaps in design support where the work in this dissertation can contribute.

2.1 Interface Design Principles

In this section, I define and give examples for a set of *design principles*, frequently used by interface designers to improve their designs, that I will refer to throughout this dissertation. I will describe in later chapters how we can use them to encode formal and informal design knowledge into high-level constraints, cost functions, and models that can aid designers in applying them in interactive design tools.

According to Lidwell, et al. [126], a *design principle* is a law, guideline, heuristic, theory of human bias, or a general design consideration that can be used to enhance usability, influence

perception, increase appeal, teach, and make better design decisions. Perhaps the most commonly taught set of principles by Nielsen and Molich [145] include heuristics such as *Visibility of System Status* which states that "A system should keep users informed of what is going on, through appropriate feedback within reasonable time", and *Consistency and Standards* which states that "Users should not have to wonder if different words, situations, or actions mean the same thing. Use platform conventions." Other design principles include theories of human perception like Gestalt Theory [109] which is a set of theories for how people visually organize information, or The Big 4 [177] (i.e., alignment, contrast, repetition, and proximity) which are a set of graphic design principles frequently applied in user interface design.

Throughout the projects I introduce in this dissertation, I apply, formalize, model, predict, and design interactions around a set of interface design principles. My goal is to encode both formal and informal design knowledge and guidelines into systems that can automatically evaluate or aid designers, and especially novices in applying them. In Chapter 3, I apply or model design principles in two ways. First, I develop a cost function based on graphic design principles (e.g., alignment, balance) to rank Scout's generated layout alternatives by their layout quality. Second, I use design principles to develop high-level constraints that a designer can use to rapidly explore layout alternatives. In Chapter 4, I apply graphic design principles (i.e., alignment, distribution) through a constraint optimization problem to beautify the layout of designs that have been inferred from a screenshot. In Chapter 5, I build a deep learning model to predict human perception of mobile interface tappability which can help designers more quickly understand peoples' perceptions of which elements are interactive in their designs. In the following sections, I detail the design principles underpinning each of these chapters.

2.1.1 High-Level Constraints Based on Design Principles

In Chapter 3, I use a set of common usability and visual design principles to develop a language of high-level constraints a designer can use to rapidly explore layout alternatives. In this section, I describe each of the design principles I used to develop Scout's high-level constraints. Chapter 3 details how I used these design principles to develop high-level constraints a designer

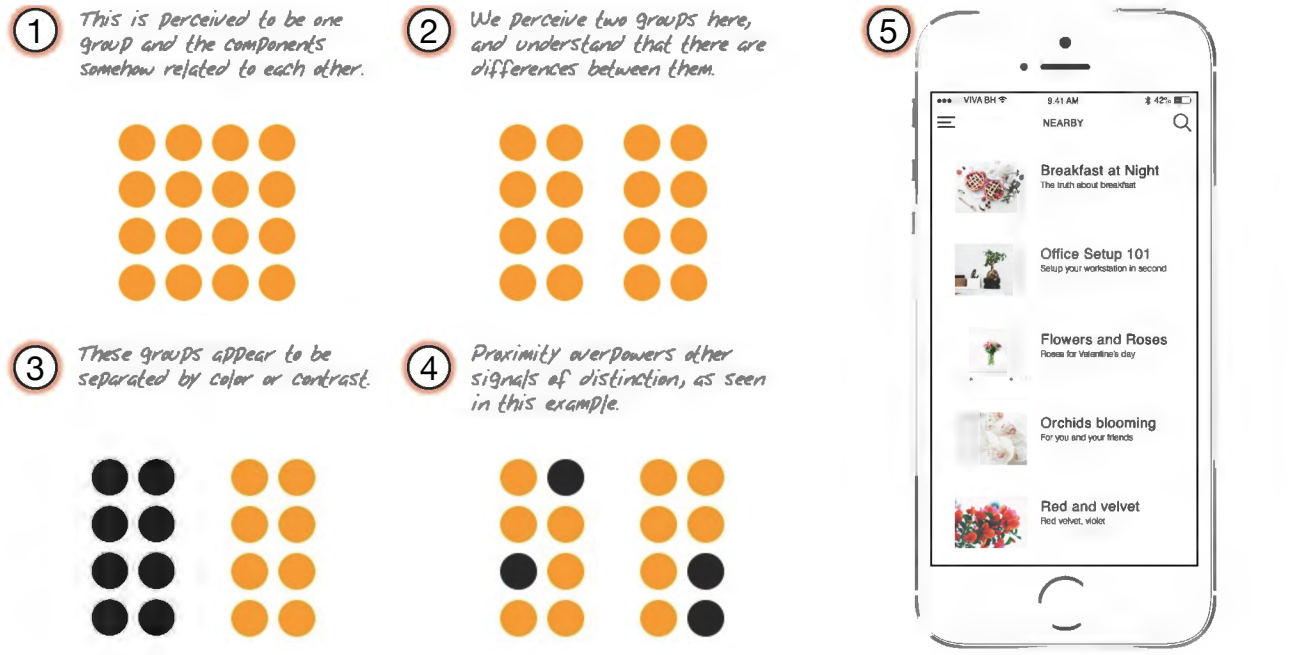


Figure 2.1: The proximity principle relates to human perception of grouped elements. (1) These circles are perceived to be a similar group. (2) People perceive these circles as two separate groups, as the groups are separated by whitespace that is larger than the separation between individual circles within each group. (3) Color can be used with proximity to visually separate elements, however (4) proximity overpowers other signals like color (1-4, Andy Rutledge, 2009 [1]). Here, the interface designer uses proximity to group together the elements of each list item (i.e., image, label subtext) while using a larger amount of whitespace to separate them from other list items.

can specify to describe the space of layout alternatives to explore, and how a system can use these constraints to generate layout alternatives automatically.

Creating a Clean Layout and Hierarchy

Interfaces should have a *clear and organized hierarchy* [126]. The *structure principle* [55] states that interfaces should keep related things together and unrelated things separate. Interface elements that are close together on the screen will be seen as related [157]. This is motivated by the proximity principle of Gestalt theory [109] which defines a set of laws for how people organize visual information. Figure 2.1.1 shows a set of circles that people perceive as a single group, while Figure 2.1.2 shows the same set of circles separated into two groups of circles in

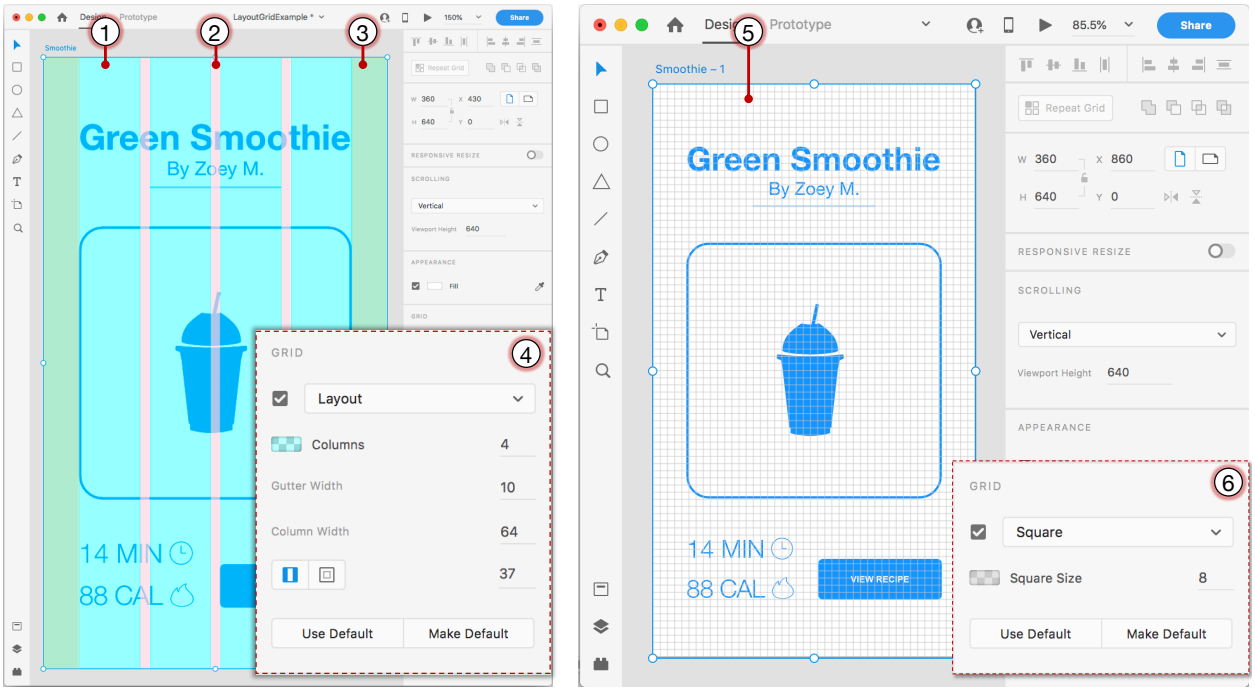


Figure 2.2: A layout grid, shown below in Adobe XD, divides a layout into regions including (1) columns (i.e., vertical containers for placing elements on the canvas, (2) gutters (i.e., spacing between columns where elements must not be placed, and (3) margins (i.e., spacing on the outside of the canvas that all elements must be placed inside). Designers can (4) specify which layout grid they want to use by providing the number of columns, gutter, and column width. A baseline grid divides a layout into horizontal sections, to guide horizontal spacing and alignment. In Adobe XD, this is actualized as a square grid of both horizontal and vertical lines (5). Designers can specify the square size (6) to set the vertical distance between the baseline grid lines.

close proximity visually separated by whitespace. Color can also be used in combination with proximity for visual separation (Figure 2.1.3), however, proximity overpowers color as a signal of distinction as seen in Figure 2.1.4. Interface designers can use the proximity principle to visually organize a hierarchy, as shown in Figure 2.1.5 where the mobile app design groups each list item close together while using a larger amount of whitespace to visually separate them from other list items.

Another way to create a clear and organized hierarchy is through the use of *layout grids* and *baseline grids*. A *layout grid* is a common method that designers use to place elements on a layout canvas. Using a layout grid can improve the alignment, balance, consistency, and

visual organization of a layout [212]. A layout grid, illustrated in Figure 2.2, consists of *margins* (i.e., spacing on the outside of the canvas that all elements must be placed inside), *columns* (i.e., vertical containers for placing elements on the canvas), and *gutters* (i.e., spacing between columns where elements must not be placed). A key requirement of a layout grid is that all elements should typically begin and end on the edge of column. There are many different types of layout grids (e.g., manuscript grid, modular grid), however, the layout grid shown in Figure 2.2 is an example of a column grid. Designers typically use multiple layout grids to adapt their designs across multiple device dimensions.

A *baseline grid* is another type of layout grid typically applied in combination with a column layout grid. A baseline grid, illustrated in Figure 2.2.5, defines the vertical spacing of a design, aids horizontal alignment, and creates hierarchy [28]. It consists of horizontal lines at even intervals down the layout to which all elements should align.

Layout grid features have become common in interface design prototyping tools like Adobe XD [98], shown in Figure 2.2.4, where a designer can specify the layout grid they want to use by providing a value for columns, gutters, and column width. In Adobe XD, a baseline grid is set using a square grid, shown in Figure 2.2.6, where designers can specify the vertical distance between baseline grid lines by specifying the square size.

Drawing the Person's Eye

Emphasis is a principle to enhance usability in interface design [213], stating that interfaces should have a main focal point to let a person know what to do next [5]. Emphasis is a strategy of drawing a person's eyes to a an important interface element (e.g., button, logo) or design element. To add emphasis to an element, designers can use colors, lines, size, shape, position of an element on the page, or other visual elements [5]. One way of adding emphasis or removing emphasis from an interface element is by changing the element's size or position in relation to other elements [213].

Creating a Usable and Accessible Layout

A key principle in creating a usable and effective interface is that elements should be placed in the *order* they are used for a task [159] (i.e., *Match Between the System and the Real World*). This principle is also expressed by the *Structure Principle* which states that "*Designers should organize the interface purposefully, in meaningful and useful ways, based on clear and consistent models that are apparent and recognizable to the users.*" [55].

Beyond the order interface elements are placed, the sizing of elements that are touch targets is also a key factor in good usability. When touch targets are too small, users take longer to tap them [71], especially those who have lower physical dexterity (e.g., children [10]). According to a study by Parhi, Karlson, and Bederson [169], touch targets in a mobile interface should be at least `1cm x 1cm` to prevent touch errors and support reasonable selection time. Because of this, industry companies have developed their own guidelines for the minimum sizing of touch targets. Apple's Human Interface Guidelines [14] state "*Provide ample touch targets for interactive elements.*" Try to maintain a minimum tappable area of `44pt` x `44pt`. Androids Accessibility Guidelines [13] state that touch targets should be at least `48 x 48` dp (density independent pixels) and provide a separate minimum size for pointer targets (`44 x 44 dp`).

Beyond the size of touch targets, minimum sizing guidelines should be followed for text elements. If the font size for text elements is too small, it could compromise readability for visually impaired users [37]. Apple's Human Interface Guidelines [20] specify a minimum font size of `17pt` for body text to ensure text is readable for mobile interface designs. Material Design Guidelines [11] specify the minimum size for body text should be `16pt`.

2.1.2 Computational Evaluation of Design Principles

In Chapter 4 and Chapter 3, I present systems that encode design principles into formalized cost functions that these systems use to beautify and rank interface layouts. These functions rely on the following 4 design principles: alignment, balance, consistency, and simplicity. Next, I define each principle. In Chapter 4 and Chapter 3, I describe how I formalize these principles into cost

functions for layout beautification and ranking to help designers prioritize layout alternatives.

The principle of *alignment* in design is defined as "*the placement of elements such that edges line up along common rows or columns, or their bodies align along a common center*" [126]. Alignment can create unity and cohesion across a design improving its aesthetic. It can also be used to indicate related sets of elements by aligning them across a similar axis (e.g., left, right, center). Elements can also be aligned to the edges of a design canvas. While most alignments are generally defined by rows and columns, other types of alignment can occur (e.g., diagonal, circular). However, in this dissertation all uses of alignment refer to the standard alignment of elements along a row (i.e., top, bottom, y-center) or column (i.e., left, right, x-center).

The principle of *balance* in design states that "*the aesthetics, stability, and unity of a design can be improved by placing elements and whitespace in such a way that no area of the design overpowers another*" [126]. Balance is defined as having an even distribution of visual weight and can be *symmetrical* or *asymmetrical*. With symmetrical balance, the visual weight is distributed evenly, either horizontally or vertically. With symmetrical balance, you should be able to draw a horizontal or vertical line through the center of the design and the weight of the elements should be equivalent on both sides. Balance can also be asymmetrical where the elements are placed to create tension, and are not distributed evenly across axes.

The principle of *consistency* states that the usability and aesthetics of a system can be improved when similar parts are expressed in similar ways [126, 145, 188]. There are four kinds of consistency including *aesthetic* (i.e., consistency of style and appearance), *functional* (i.e., consistency of meaning and action), *internal* (i.e., consistency with other elements in the system), and *external* (i.e., consistency of with other elements in the environment or across systems). In this dissertation, I focus on aesthetic and internal consistency.

The principle of *simplicity* states that interfaces should be as simple as possible, and remove unnecessary and extraneous information [145]. This is analagous to Occam's Razor [126] which states that given the choice between two functionally equivalent designs, the simplest one should be selected. Beyond functionality, interfaces can be simplified by reducing unnecessary visual clutter [177]. In the interaction design book About Face [56], Cooper states that "*Unnecessary*

variation is the enemy of a coherent, usable design. If the spacing between two elements is nearly the same, make that spacing exactly the same. If two typefaces are nearly the same size, adjust them to be the same size. Every visual element and every difference in color, size, or other visual property should be there for a reason.".

Another component of the simplicity principle is that a design should avoid being too cluttered. One way to avoid clutter is to use *whitespace* (i.e., the areas in a design that do not contain content) effectively [213]. Using whitespace effectively can enhance readability and simplify a design by breaking it down into discrete chunks that can help the user process the information more effectively [4].

2.1.3 Tappability, Signifiers & Affordances

Tapping is an extremely important gesture in mobile touchscreen interfaces, yet people learning to use an interface still may need to learn which elements are tappable (i.e., interactive) through trial and error. In Chapter 5, I present an approach to model interface element tappability that can automatically predict which interface elements a human will perceive as tappable. Here I describe the design principle of *discoverability* and related concepts of *signifiers* and *affordances* and describe how a lack of signifiers and affordances can lead to a lack of discoverability.

Fundamental design guidelines by Don Norman [160] include the principle of *discoverability* which states that "*Interfaces should make it possible to determine what actions are possible and the current state of the device*". Based on this principle, mobile interfaces should support discoverability by ensuring that users can quickly understand which elements are interactive.

One way that designers can support discoverability is through *signifiers* [162] which indicate to a user the *affordances* of an interface element. Affordances were originally described by Gibson [76] as the actionable properties between the world and actor (i.e., person). Don Norman [160, 161] popularized the idea of affordances of everyday objects, such as a door, and later introduced the concept of a "signifier" as it relates to user interfaces [161]. Gaver [75] described the use of graphical techniques to aid human perception (e.g., shadows or rounded corners) and showed how designers can use signifiers to convey an interface element's perceived

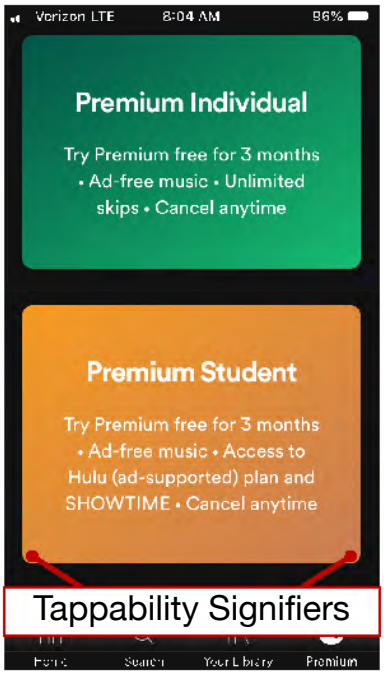


Figure 2.3: The current version of the Spotify iOS app, showing two cards with information on premium and student subscriptions. Designers frequently use rounded corners to indicate elements are tappable. However, these cards are not tappable, leading to a false signifier.

affordances.

Designers can use visual properties (e.g., color, depth) to signify an element's "clickability" [3] or "tappability" in mobile interfaces. Perhaps the most ubiquitous signifiers in today's interfaces are the blue color and underline of a link and the design of a button which both strongly signify to the user that they should be clicked. These common signifiers have been learned over time and are well understood to indicate clickability [161].

While some design guidelines for clickability and tappability exist [3], designers do not always use correct signifiers, or they may want to experiment with new signifiers. Users may still need to learn what is tappable in a new interface through trial and error. This can lead to users potentially not discovering a functionality that could be useful to them (i.e., poor discoverability [160]) or in a false signifier [75] where they tap on an element and receive no response [3, 7]. Figure 2.3 shows the Spotify iOS mobile app on a page for advertising premium services. The interface contains two cards advertising premium student and premium individual options. Designers frequently use rounded corners to make elements look tappable,

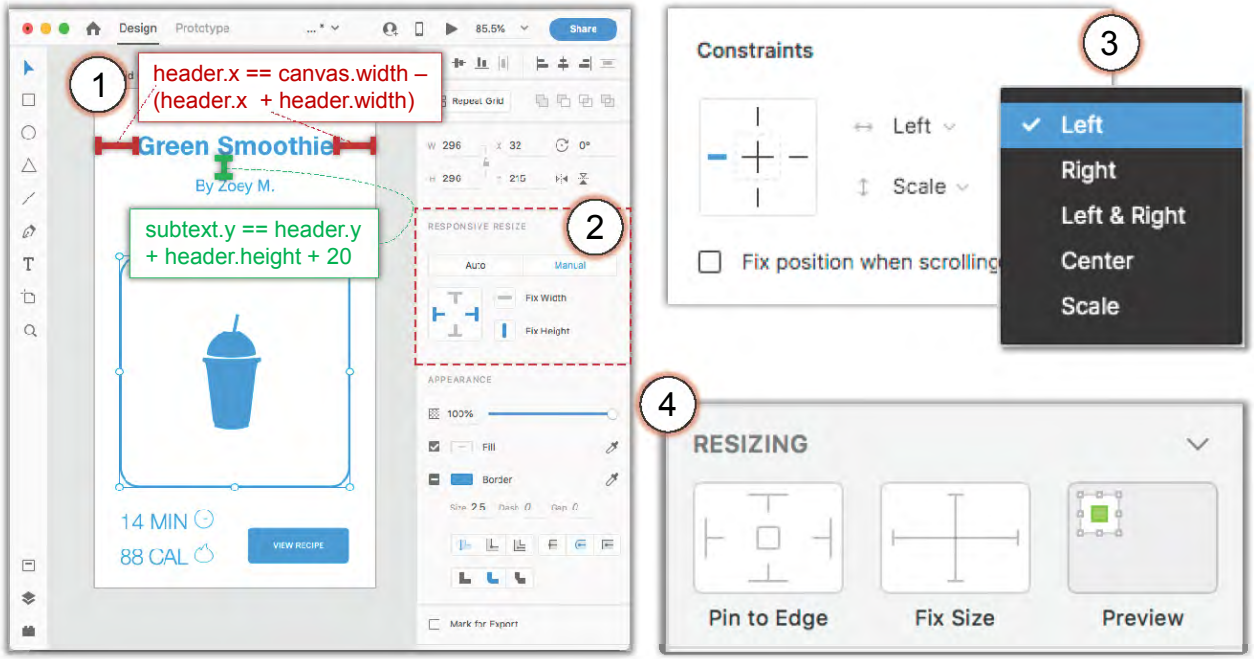


Figure 2.4: Constraints define spatial relationships between interface elements on a layout (1). Commercial prototyping tools include features for defining constraints (2-4) largely enabling responsive resizing of individual elements.

as shown. However, the cards in this Spotify page are not actually tappable. This is an example of a *false signifier* [75]. Chapter 5 describes an analysis of signifiers that are having an impact on tappability across a large dataset of mobile interfaces.

2.2 Constraints in User Interfaces

Chapter 3 and Chapter 4 present systems that use *constraints* as a tool to encode rules about user interface layouts and design principles into a layout specification. In the domain of user interface layouts, a *constraint* is a rule defining a spatial relationship between one or more interface elements or to the layout canvas that should hold across any potential layout configuration (e.g., across alternate device dimensions) [42]. Figure 2.4.1 shows an example constraint (red) stating that the left (i.e., `header.x`) and right margin (i.e., `canvas.width - (header.x + header.width)`) of the "Green Smoothie" header should be equal. A constraint below (green) states that the margin between the bottom of the header (i.e., `header.y + header.height`) and the top of the subtext (i.e., `subtext.y`) should be equal to 20. A developer or designer

expresses a constraint and a user interface management or prototyping tool uses a constraint solver (e.g., Cassowary [43], Z3 [57]) to maintain the constraint across alternate configurations of the layout. A constraint is a declarative statement of *what* the desired properties of the system should be rather than *how* they should be maintained [42].

2.2.1 Solvers and Constraint Priorities

Constraint solvers are a key part of constraint-based layout tools. After the designer produces a set of constraints, either through direct manipulation or as a spatial equations, the system converts them into a set of formalized equations and specifies them to a solver which will return whether the set of constraints is *satisfiable*. If so, the solver provides a solution or multiple solutions to the set of constraints. In the case of an interface layout, a solution typically consists of sizes (e.g., width, height) and positions (e.g., x, y) for interface elements. Many constraint-based layout systems use linear or quadratic programming [31, 92, 135], and some have recently used SMT solvers to support a wider variety of layout specifications [100].

Constraint solvers used in constraint-based layout typically support constraint *priorities* which they use to penalize layouts that deviate from desired aesthetics. Support for priorities consists of letting the user specify *hard* and *soft* constraints. Hard constraints must be satisfied by every solution. Soft constraints need not be satisfied by every solution but the solver will typically try to maximize the number of soft constraints through an objective function. Frequently, solvers let users specify weights on their constraints so that they can specify that some soft constraints should have a higher likelihood of being satisfied. In a *constrained optimization* problem, a solver uses an objective function to minimize the value of a cost function (e.g., human performance [74], layout quality [226]).

Z3 [57] is a powerful SMT (Satisfiability Modulo Theory) solver that I use in the systems I present in this dissertation. An SMT solver determines the satisfiability of a logical formula. Given the formulas `0 < x < 360` and `{0 < y < 640}`, solutions would include `{x = 44, y = 320}` and `{x = 1, y = 1}` but not `{x = -2, y = 700}`. Beyond linear constraints, Z3 supports encoding other types of constraints (e.g., boolean constraints, first-order logic,

functions) and supports constraint priorities (i.e., hard and soft constraints). Z3 has been used for a variety of recent applications including verifying webpage accessibility [168], generating a set of statistical tests to use from a study design and set of assumptions [101], and to enable specifying OR constraints in GUI layout managers [100].

2.2.2 Origins & Modern Usage

Initially introduced in Sketchpad [197], constraints became widely used to define interface element relationships in early GUI builders [49, 86, 96, 150, 191, 223, 224]. Constraints in these early interface builders are mainly specified by writing low-level spatial equations or through direct manipulation by dragging elements around on a canvas. Constraints were later adapted to web interfaces through a CSS based framework [30] and webpage design interface [42] for specifying constraints in webpage design.

Constraints have more recently become features in commercial interface builders including Apple's AutoLayout [9] (Figure 2.5) and Android Interface Builder [8]. Apple's AutoLayout [9] enables specifying constraints by dragging interface elements on an interface builder. When a designer drags elements near each other (Figure 2.5.2), the interface builder displays a drop-down menu (Figure 2.5.3) with a list of suggested constraints based on the location of the selected elements. These constraints manifest as a list of spatial equations (Figure 2.5.1) in the element view hierarchy panel. In Android Interface Builder [1], constraints are specified similarly to AutoLayout but the interface does not provide any suggestions of constraints to apply.

Constraint-based features have also been introduced recently in popular UI prototyping tools like Adobe XD [2] (Figure 2.4.2), Sketch [3] (Figure 2.4.4) and Figma [4] (Figure 2.4.3). Constraint support in these tools is currently limited to defining a limited set of resizing rules for elements when the size of the layout canvas changes.

---

[1] https://developer.android.com/training/constraint-layout/

[2] https://www.adobe.com/products/xd.html

[3] https://www.sketch.com/docs/layer-basics/constraints/

[4] https://help.figma.com/article/54-constraint

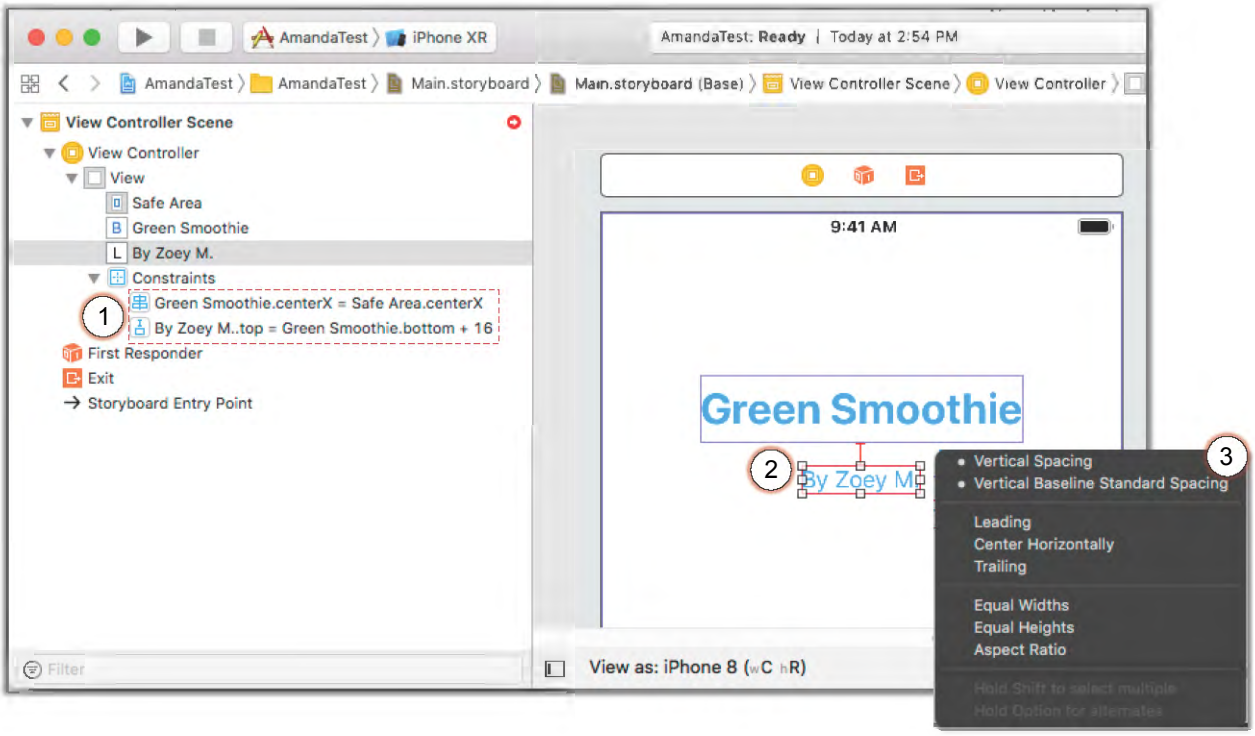


Figure 2.5: Apple's AutoLayout Interface Builder enables creating constraints by dragging arrows between elements to define relationships (2). When a designer defines a relationship, the interface builder displays a list of suggested constraints (3). The interface builder displays all of the active constraints in the view hierarchy panel (1).

2.2.3 Key Research Challenges

Recent research in constraint-based layout has focused on several core problems in constraint-based layout systems: (1) resolving constraint ambiguities, (2) prevention or repair of overconstrained or invalid layouts, and (3) the creation of high-level constraint abstractions and tools to make constraints easier to specify.

One key challenge in constraint-based layout systems is that *ambiguities* can occur in constraint specification. Ambiguities occur when the designer does not specify enough constraints on the goal layout. Therefore, there will be multiple solutions to a layout specification. Programming by Manipulation [93] presents a set of direct manipulation techniques to resolve ambiguities in constraint-based data visualization layouts. For graphic design layouts, Xu et al. [219] present an interface to facilitate ambiguity resolution within a constraint-based layout

beautification system. In contrast to previous works which have focused on resolving ambiguity in constraints, I present Scout in Chapter 3 which embraces ambiguities in constraints to let a designer explore many alternate layouts satisfying a single set of constraints.

Another key challenge in constraint-based layouts is that the layout can be overconstrained or conflicting (e.g., no single solution exists that satisfied all constraints). Resolving such conflicts can be challenging, even for experienced programmers [93]. The ALE [225] presents a GUI builder that helps designers create layouts with simple, direct manipulation operations, which do not require any direct constraint editing. ALE guarantees that each edit operation leads to a single, non-overlapping solution, with no *conflicts*. PBM [93] presents a set of techniques to prevent conflicts in data visualization layouts. Because Scout (Chapter 3) embraces ambiguities in constraints, conflicts in these constraints can arise. Scout adopts a multi-faceted approach to explain and highlight conflicts to help designers resolve them if they occur.

Finally, constraints in interface layout systems are often specified using low-level spatial constraints, like those seen in Figure 2.5. Some systems infer constraints to make them easier to specify through direct manipulation (e.g., Rockit [102], Peridot [150], ALE [225], PBM [93]). However, a recent thread of recent work has explored the creation of high-level constraints to ease the constraint specification burden. Many are specialized for domains outside of user interface layouts like visualization design [147], specifying graph layouts [88], and automating statistical analyses [101]. For GUI layout, the ALE presents some high-level constraints [135] (e.g., rows, columns), yet these are still based on concepts from GUI layout managers. In the ORC layout editor [100], a designer specifies high-level patterns of elements that should be maintained which limits the number of alternative layouts that can be explored. In Chapter 3, I present high-level constraints for interface layout, based on *interface design principles*. I developed these through the lens of visual design (e.g., layout grids, emphasis) and usability principles (e.g., order, structure principle) for developing clear and compelling layouts.

2.3 Semantic Analysis of Interface Structure & Presentation

Chapter 4 presents a system to generate vectorized design documents from screenshots of interfaces. Chapter 6 describes a system to enable prototyping new interactions for existing interfaces. Both projects apply a form of *semantic analysis* of interfaces. In this dissertation, I define *semantic analysis* as any form of automated analysis that can infer and understand interface structure from incomplete inputs (e.g., screenshots, web interface code) or encode constraints about interface structure that enable validating or generating interfaces automatically.

In this section, I review work related to Chapter 4 on methods to infer interface structure from a screenshot to generate interface code or mockups. This can bridge the gap between interface design screenshots and the code a developer or designer needs to modify them. I also review related work for Chapter 6 on methods to modify the behavior and inputs to interfaces at runtime, requiring semantic analysis of interface elements and source code.

2.3.1 Inferring Interface Structure from Screenshots

Currently a gap exists between pixel-based interface screenshots and the code developers or designers need to modify them. An important step in prototyping is in creating digital mockups in order to demonstrate design concepts. Designers frequently create these prototypes in tools like Photoshop, Adobe XD [98], or Sketch [19]. They may then share these in design galleries (e.g., Dribbble, Behance) for other designers to use for inspiration [47, 87]. Screenshots can be easy for designers to capture, however, they are unstructured and difficult to edit. Thus if a designer wants to edit or adapt the example design, they first need to recreate its shapes, properties, and structure by hand. Another challenge is that when a designer has created a design they are ready to move to production, designers may give these prototypes to developers as a screenshot to translate them into code. This process has been demonstrated by past studies to be time-consuming and error prone [148]. Recent work has shown that developers are still spending a lot of time re-creating design documents in a different tool [140].

Inferring Interface Code from Screenshots

A recent focus in semantic interface analysis is in reverse engineering screenshots of high-fidelity interface prototypes or implementations into interface code or screenshot mockups. A number of pixel-based methods have inferred interface structure from screenshot inputs [34, 50, 62, 221] to modify or control the behaviors of an interface at runtime.

Remaui [155] presented the first approach to transform mobile app screenshots into code through computer vision and optical character recognition (OCR) techniques. Recently, machine learning approaches have advanced the quality of the generated interface code from screenshots [35, 53, 146]. These approaches apply deep learning models trained on large-scale mobile app datasets to generate Android interface code [53, 146] and HTML webpages [35] with reasonable levels of accuracy (i.e., ~70 to 90 percent).

Beyond digital mockups, researchers and industry companies have explored converting hand-drawn sketches into interface code. Sketching is a key part of the design process [47]. Researchers [115, 130] and more recently industry companies such as AirBnB[5] and Microsoft[6] have created systems to support rapid sketching into interface code to streamline the rapid prototyping process.

Inferring Design Documents from Screenshots

Designers frequently adapt interface screenshots, found from inspiring sources or design galleries, into their digital prototypes created in tools like Sketch [19]. Yet, most recent systems created by researchers and industry companies aim to automate the entire prototyping and interface implementation process, going from screenshots and sketches directly to interface code. This highlights an important gap in design tooling where interface designers could benefit, as they currently need to manually adapt examples in order to modify them. In Chapter 4, I present a system that automatically infers a vectorized design document from an interface screenshot.

[5] https://airbnb.design/sketching-interfaces/

[6] https://www.microsoft.com/en-us/ai/ai-lab-application-samples

Rather than inferring interface code like prior work [35, 146, 155], it infers semantic vector shapes (e.g., rectangle, text) and design properties (e.g., font, color). Semantic shape types (e.g., rectangles, text areas) are a more useful format to infer for designers as prototyping tools (e.g., Sketch, Adobe XD) represent designs with these shapes.

Another challenge with previous systems is that they analyze inputs with existing structure (e.g., Android app hierarchies [53, 146]) and require labeling training examples. They operate on extremely large datasets of app designs that have been collected and cleaned for training machine learning algorithms. In Chapter 4, I present a system that infers useful structure from flat screenshots of user interfaces, which are a convenient format for designers to collect and share. This approach does not require training on a large UI design dataset.

A challenge for future work is to explore creating a generalized interface design dataset that could be used to apply machine learning models within UI prototyping tools. Machine learning approaches that generate interface code [35, 53, 146] could potentially improve the accuracy of the approach I present in Chapter 4. However, a large-scale UI design dataset would be needed. Unfortunately, such a dataset does not yet exist for UI design prototyping documents. Such a dataset would be challenging to collect given the diversity in interface prototyping tools, their output formats, and the diversity of design formats shared in online design galleries. A challenge for future work is to explore creating a generalized interface design dataset that could be used to apply machine learning models within UI prototyping tools.

2.3.2 Runtime Modification and Control of Interfaces

A diverse set of techniques and systems have explored modifying interfaces at runtime. Key motivations of this work are for making an interface more accessible, and enabling developers, designers, and end users to prototype new interactions and behaviors for existing interfaces. Both of these areas provide related work and motivation for Chapter 6 which dynamically infers a web application's commands and their current state to enable designers to prototype new interactions that let users interact with them in alternate ways at runtime.

Making Interfaces More Accessible

Chapter 6 presents a method to discover commands and behaviors of web applications, enabling them to be controlled through alternate input modalities and presented in different ways in a webpage-agnostic manner. One motivation for the work presented in Chapter 6 is that it could improve the accessibility of an interface by enabling those who could not access the interface through its default input method (e.g., mouse) to use an alternate input method to access it (e.g., voice). Past research has explored methods to modify interfaces at runtime to make them more accessible [39, 44, 45, 95] and methods of automatically generating interfaces to customize them for a person's physical needs [24, 74].

To make a webpage accessible, web designers and developers should add support for ARIA[7] attributes. These attributes enable screen readers to interpret web content, and several prior works have dynamically analyzed and injected ARIA attributes into a website. One method describes dynamic updates by monitoring and dynamically injecting ARIA attributes onto the updated content [44]. Another method detects and makes static content accessible [45]. However, these methods primarily operate within the existing input modality of the page, improving the interaction for people who could already access the page via that modality, but not enabling access via entirely new modalities.

Other approaches have used crowdsourcing and collaboration to identify web accessibility issues and apply fixes, including AccessMonkey [39] and CAN [95]. These systems enable developers to write scripts to fix specific accessibility issues. However, these scripts are mostly written for a specific website or subset of websites. They typically modify a specific aspect of behavior or add a new functionality. One limitation of these systems is that they do not provide a generic way of discovering and describing existing functionality of an interface that can support providing alternate forms of access to that functionality across interfaces.

Prior work also explores automatically generating interfaces to make them more accessible or efficient to use. SUPPLE [74] had users take a one-time performance test that enabled

[7] https://www.w3.org/WAI/standards-guidelines/aria/

generation of a custom interface suited to personal abilities, improving efficiency of generated interfaces. The EKOGI system [24] accounted for a person's abilities and the interactions that best suit those abilities, generating a tailored interface. These systems as presented are limited to working for a single interface. Such systems could benefit from a more application-agnostic model of commands that would notify them when commands are available, and enable them to select and use an alternate input modality that is more appropriate to a person's abilities.

Only a few prior works have analyzed source code for enhancing accessibility or usability. Ko et al. [108] applied program analysis to detect paths in a web application that did not result in any feedback in the interface. Many works have analyzed source code and accessibility APIs to understand the structure of web applications for software testing [136, 154, 189]. There are numerous opportunities to use static and dynamic program analysis techniques to support automatic interface adaptation and analyses to improve usability.

Dynamically Modifying an Interface's Presentation, Inputs, and Behaviors

Chapter 6 presents a method of discovering and modifying the inputs and command behaviors of web interfaces at runtime to enable them to be controlled through alternate inputs and presented in alternate ways. Methods to modify interfaces at runtime have long been an active area of research for prototyping new visual effects or interactions on top of existing interfaces [62, 68, 69, 166, 228] and to let users customize their own applications [39, 41, 195, 204].

Edwards et al. [69] present a method to transform the output of graphical user interfaces. Olson et al. [165] introduce techniques to implement interface attachments that augment interface functionality. User Interface Facades [195] and WinCuts [204] enable users to modify interfaces themselves by cutting out and combining multiple screen regions. One limitation of these methods is that they require modifying existing applications, windowing systems, or toolkits limiting their applicability across platforms. In contrast, non-toolkit based approaches require inferring the location and structure of interface elements to control or modify them. These methods use pixels as input (i.e., Prefab [62]), combined with code based descriptions (i.e., Vismap [228]), screenshots (i.e., Sikuli [221]), software videos (i.e., Waken [34]), and

accessibility APIs (i.e., PAX [51]). Pixel-based methods [34, 62, 221, 228] can only infer an application's visible behaviors. In Chapter 6, I show that we can take advantage of the open access to source code and DOM structure in web pages to infer information about a web application's non-visible behaviors.

Several systems have both detected and enabled modifying the behavior of an application's interactive components. Runtime toolkit overloading in Scotty [68] is one approach, a technique for supporting manual program analysis for adding functionality to existing runtime behavior. Prefab [62], Sikuli [221], and Waken [34] all use pixel-based analyses to discover interface components through templates and machine vision techniques. These systems also enable modifying the behavior of the detected interface components. However, because these methods only have access to the pixel-level appearance of an application's interface, but not the application's source code, they can only understand visible behaviors. Pixel-based methods have no understanding about whether the components can currently be interacted with, nor any way of predicting their behavior. Using program analysis methods, as I present in Chapter 6, we can discover the current state of an interfaces input commands, in order to let users know when they are currently available.

Researchers have long studied methods of modifying the inputs of interfaces to let users control them through alternate modalities and interactions. Pixel-based methods have realized this through intermediate controllers (i.e., Sikuli [221], VisMap [228]), input and output redirection (i.e., Prefab [62]), and runtime toolkit overloading (i.e., Scotty [68]). In the web, input modification can be achieved with scripting languages such as Chickenfoot [41] and AccessMonkey [39]. However, these scripts typically modify a specific website, or a specific aspect of behavior. They do not provide a generic method of discovering and describing interface functionality to support adding alternate input modalities.

Recently, such alternate input modalities have been enabled in mobile phones to control them with gestures [118] and 3D printed tactile interfaces [231]. Gesture Avatar [118] allowed people to interact with an existing mobile interface through gestures. Their method operated on the pixel level, creating a mapping between gestures and their corresponding objects in

the interface. Their approach creates custom mappings from one input domain (i.e., gestures) to another. Chapter 6 presents a system that enables the creation of generic input mappings (e.g., speech, keyboard) through *input retargeting* which enables the control of web applications through alternate input modalities.

2.4 Semantic Analysis of Interface Usability & Visual Design

Semantic analysis approaches have been explored to automatically evaluate usability and visual design aspects of an interface. Past work has automatically evaluated designs through crowdsourcing and machine learning based approaches, and through formalized cost models and functions based on design principles. Researchers have also identified several key requirements to integrating these computational approaches into designers' workflows and tools. I identify gaps in recent systems where these requirements are not being satisfied.

2.4.1 Computational Measures of Usability

In Chapter 5, I present an approach and deep learning model to automatically measure the tappability of mobile apps. This project required collecting data on a large scale to enable the use of a machine learning approach. Data-driven design [113] is a large-scale approach to design mining of design data from user interfaces to support design interactions (e.g., design search). Recently, these approaches have been used to identify usability issues [60] and collect mobile app design data at scale [59, 61].

One such system, Zipt [60], shown in Figure 2.6, enables comparative user performance testing at scale. Zipt uses crowd workers to construct user flow visualizations through apps that can help designers visualize the paths users will take through their app for specific tasks. From these visualizations, designers can pinpoint where a usability issue might be occurring. With this approach, designers manually examine the visualizations for anomalies, and then examine the interface to determine what what specific issue might be causing the anomaly in user behavior. It can be challenging to diagnose these issues as there might be multiple factors impacting the usability on a single screen. In Chapter 5, I present a system that can help

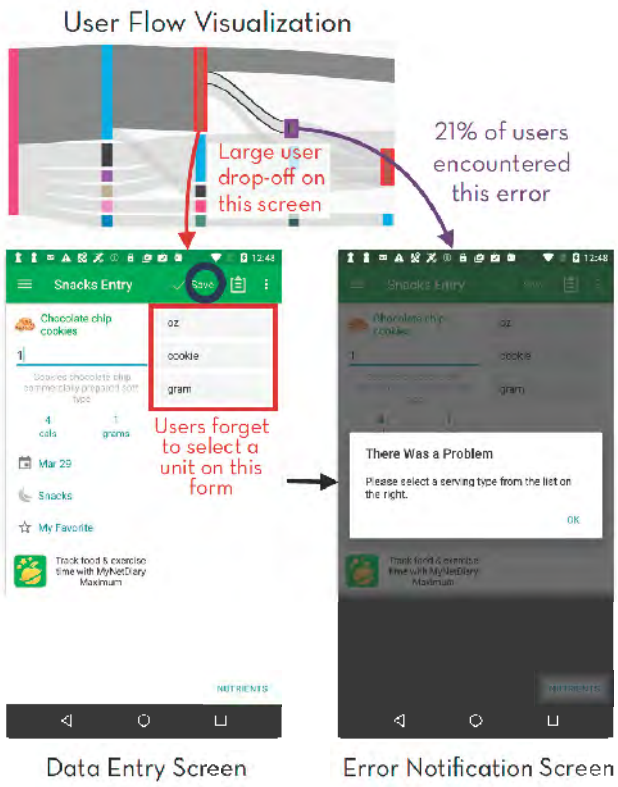


Figure 2.6: Zipt helps designers discover usability issues in their apps. A designer can use the flow visualization to pinpoint where a usability issue might be occurring, and then examine the associated app screens to determine the specific usability issue.

designers identify a specific usability issue – cases where false affordances or missing signifiers will cause a user to misidentify a tappable or not-tappable interface element.

Large-scale crowdsourcing approaches, like Zipt [60], can collect crowdsourced user data to aid the diagnosis of usability issues. Amazon's Mechanical Turk has previously provided a platform for large-scale usability [152] and human subjects experiments [106, 110, 186] and gathering data about the visual design of user interfaces [78, 134, 218]. Few of these previous works have used this data to build machine learning models that learn from the crowdsourced data. In Chapter 5, I describe a machine learning model that learns from this crowdsourced data to automatically examine a key aspect of usability (i.e., tappability).

Some previous work has built deep learning models to identify specific usability issues automatically. Deep learning [119] is an effective approach to learn from a large-scale dataset.

Recent work applies learning approaches to predict human performance on mobile apps for tasks such as grid [172] and menu selection [123]. These approaches rely on both lab-collected and crowdsourced datasets, via Mechanical Turk, of human grid and menu selection tasks. Bylinskii et al. [48] crowdsourced a dataset of human importance annotations, and built a deep learning model to identify salient elements in graphic designs and interfaces [48]. No work has yet applied deep learning models to predicting the tappability of interface elements, nor collected a large-scale dataset of tappability annotations across a diverse set of mobile interfaces. Using a deep learning approach can enable leveraging a rich set of features involving the semantic, spatial, and visual properties of an element without extensive feature engineering.

2.4.2 Computational Measures of Layout Quality and Aesthetics

In Chapter 3, I present Scout, which lets designers explore a set of layout alternatives. Scout ranks these alternatives using a cost function to help the designer prioritize them. This cost function builds on top of a large body of prior work that attempts to formally measure human perception of visual design properties (e.g., saliency, clutter, balance, alignment, symmetry).

Prior work has built computational models to measure *layout quality* through formalizing principles of layout (e.g., balance, symmetry). Ngo et al. [153] present a set of 14 aesthetic measures for graphical layouts (e.g., balance, symmetry, density, rhythm) that they compute based on the positions, sizes, and relative placement of elements in a layout. Zheng et al. [232] adopt a pixel-based approach to compute low-level image statistics to analyze the perceived layout structure of a website (e.g. symmetry, balance, equilibrium). O'Donovan et al. [163] use an energy based model to evaluate the quality of graphic design layouts based on a set of "energy" terms (e.g., alignment, balance, whitespace, flow). Their models take inspiration from formalized measures of aesthetics for document layouts [32, 82].

Other work has focused on measuring human perception of interfaces through saliency, visual clutter, and visual complexity models. Rozenholz et al. [181] present a model to predict the saliency of graphical design elements, and later developed a combined model [182] that measures both saliency and visual clutter. Reinecke et al. [176] developed a model of visual

complexity and colorfulness based on human annotations of the aesthetics of web pages. They then use their model to predict users impressions of the aesthetics of web page. Wu et al. [216] developed an approach to measure the visual complexity of webpages using web mining and machine learning.

Researchers are recently exploring how to combine these models and metrics into an overall "goodness" score of interface design aesthetics by computing both layout quality measures (e.g., balance, symmetry) and models of human perception (e.g., saliency) on the pixels and elements of an interface design. Minukovich et al. [142] introduce eight automatic measures of GUI interface aesthetics (e.g., visual clutter, symmetry, grid quality, whitespace) and tested the measures on both iPhone apps and webpages. Riegler et al. [178] present a element-based method specifically for analysis of mobile interfaces which includes metrics such as alignment, balance, density, color complexity, and typographical complexity.

2.4.3 Integrating Computations of Usability, and Aesthetics into Design Tools

Although researchers have developed many computational models of aesthetics and layout quality and validated them against human annotations of aesthetics, few have studied how to integrate these metrics into user interface design tools as part of the workflow of design creation and evaluation. With DesignEye [180], researchers studied the challenges of integrating computational models of aesthetics into designers workflows. One challenge they discovered is that designers struggled to *understand the model output and how they could improve it* (i.e., what design elements are causing the model to compute or predict a value and how can they improve or change the output). A second challenge was that designers wanted to be able to *easily compare multiple versions of a design* to see how various changes affect the model's output. Finally, designers wanted to *have these tools integrated into their prototyping workflow* so that they could have automatic "goodness" calculations of their designs.

Some recent research has explored the integration of computational models of aesthetics into design tools. One way is through a webpage that lets designers upload and evaluate their designs according to a variety of visual metrics (Zen et al. [227], AIM [167]). However,

designers cannot easily explore multiple variants of a design to see how it affects the metric, nor are these tools integrated into other design tools (e.g., Photoshop, Sketch) such that a designer does not need to use a separate tool to use the aesthetics models.

Bylinskii et al. [48] present a graphic design prototyping tool to evaluate and view real-time saliency heatmaps for design elements. Their system lets designers move elements around a design canvas in a prototyping tool to see real-time saliency heatmap, solving two of the key challenges summarized by DesignEye [180]. However, their system does not evaluate any other measures of aesthetics (e.g., layout quality, clutter), and does not give the designer a way to compare among alternatives to see how the model is affected.

DesignScape [164] and Sketchplore [209] present interactive graphic design and interface sketching tools that use aesthetic and layout quality models to automatically generate and rank layouts. However, these systems do not expose their results of these metrics to the designer.

2.5 User Interface Alternatives

Creating multiple alternative designs in parallel results in higher-quality and and more diverse solutions [47,67] and enables designers to make stronger critiques and better decisions [65,210]. Designers also frequently need to create alternatives to suit the needs of diverse users [74] or to design for diverse device dimensions or platforms [226].

Due to the benefits and needs for creating alternatives, research has explored systems and interaction techniques to help designers ideate and brainstorm [104, 111, 114, 120, 138, 139, 164, 209], visualize the outcome of a range of parameters [84, 133, 137, 205, 206], create cross device and cross platform user interface designs [54, 73, 129, 156, 156, 192, 220, 226], and improve the accessibility of their interfaces by creating alternative interfaces for those who need them [24, 74, 195, 211].

To address these motivations, researchers have explored (1) interaction techniques to define and create alternatives, (2) algorithms to automatically generate alternatives, (3) tools to manage alternatives once created, and (4) systems to explore and adapt example designs from other designers. I discuss related work in each of these areas in the following sections.

2.5.1 Defining Requirements for Alternatives

In order to help a system create alternatives, designers need to define the rules and models that those alternatives should satisfy. Perhaps one of the earliest methods introduced for defining these rules was model-based user interfaces [196,203,223] where a designer or developer defines a high-level model specification their interface should satisfy and a system generates an interface or alternative interfaces that maintain the specification in the model. A more recent application of this approach was in creating cross-platform and cross-device user interfaces, exemplified by Smart Templates [156] and Damask [129] which present parameterized templates and high-level patterns to specify when interface conventions should be applied across platforms.

A similar concept to model-based user interfaces is *high-level constraints* which are constraints a user specifies that must be maintained across alternatives that a system, typically a constraint solver, generates. High-level constraints are typically more abstract and than low-level constraints, which are typically in the form of an equation or mathematical expression that get encoded into a constraint solver. In one recent work [135], high-level constraints consist of GUI layout concepts (e.g., rows, columns); a designer uses them to specify a set of elements that should be contained in a particular row or column. Recent applications of high-level constraints have been presented in diverse domains to encode best practices into formalized constraints to enable automatic generation of alternatives. Draco [147] formalizes visualization design principles and uses a solver to synthesize visualizations satisfying these principles. SetCola [88] provides a high-level language for specifying graph layout. Tea [101] presents a high-level language for specifying study design, assumptions, and hypotheses and generates a set of valid statistical tests for an experiment.

For ideation and brainstorming in 2D and 3D modeling, generative design tools like DreamLens [139] and GEM-NI [222] enable specifying high-level constraints through an interface [139, 222] or through sketching [104]. These systems generate a set of 2D graphic designs [222] or 3D models [104, 139] satisfying the designers' constraints.

For user interface layout, many systems rely on a developer or designer to specify low-level

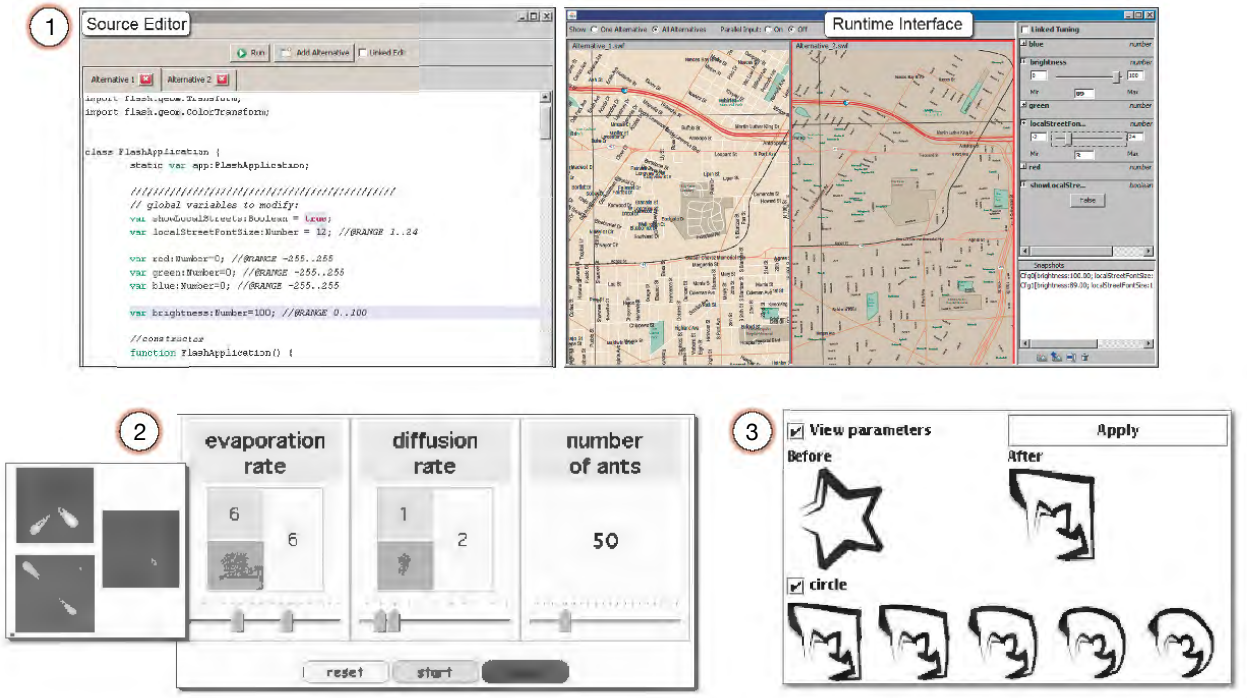


Figure 2.7: In Juxtapose [85] (1), designers can create and view side-by-side alternatives and dynamically adjust them through code-based tuning of design parameters. Subjunctive Interfaces [133] (2) lets users set up, view, and control alternate scenarios within a single interface. An example is this simulation of ant foraging behavior where a user can simultaneously view and update the parameters for multiple scenarios. Parameter Spectrums [205] (3) lets a user preview the effects of editing commands with a range of a parameters.

equations and spatial relationships [9], either through direct manipulation operations or through editing low-level constraints. Several systems systems infer constraints automatically (e.g., Rockit [102], Peridot [150], ALE [225]) through a GUI layout builder. None of these systems have explored supporting high-level constraints based on design principles and best practices, so that designers can use familiar concepts at a high level of abstraction to explore alternatives, an approach I explore in Chapter 3. Additionally, these systems are typically limited to the creation of alternatives for alternate device dimensions [225], and they are not typically used for ideation and brainstorming as I explore in Chapter 3.

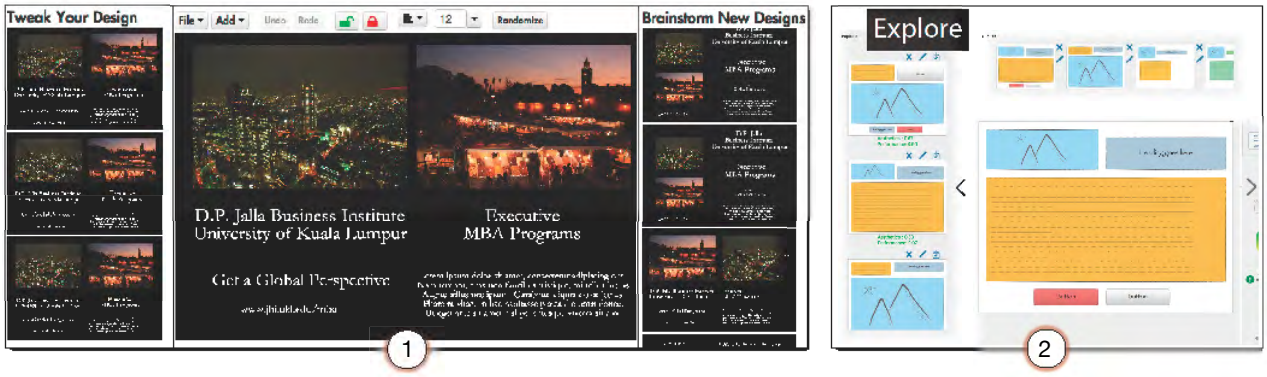


Figure 2.8: DesignScape [164] (1) provides designers interactive layout suggestions for graphic designs, consisting of refinement ("Tweaking") and brainstorming suggestions that a designer can directly apply to their design canvas. Sketchplore (2) provides a designer realtime layout suggestions it infers by predictive models of usability and aesthetics as they sketch on a canvas.

2.5.2 Visualizing & Creating Alternatives

A key interaction technique for creating alternatives for user interfaces, visualizations, and graphic designs is through parameter previews [85, 133, 137, 205, 206]. Marks [137] presents Design Galleries, a graphical design tool that automatically generates and presents a range of visual outcomes for different input parameters (e.g., to help users select illumination values for an image). Side Views [205] presents an interface mechanism (i.e., Parameter Spectrums, shown in Figure 2.7.3) to preview the effects of editing commands. However, a user could not ultimately instantiate more than one of these alternatives. Parallel Paths [206] enables an interaction technique, based on Side Views [205], that lets users instantiate multiple command previews within a single interface to enable comparing and manipulating multiple alternative solutions simultaneously. Subjunctive interfaces [133] (Figure 2.7.2) provides mechanisms for the setup, viewing, and controlling of alternate scenarios for information processing tasks.

Juxtapose, shown in Figure 2.7.1, [85] was the first to extend these ideas to interface design. In Juxtapose, a designer can create and view side-by-side interface alternatives through code-based tuning of design parameters. They can compare, combine, and manipulate multiple alternatives at once through linked editing. One thing in common with these previous systems is that they focus on low-level parameter tuning or the previewing of alternative outcomes for commands rather than helping designers ideate and brainstorm. Supporting designers

in exploring alternatives for ideation and brainstorming imposes additional challenges for automatic alternative generation (i.e., generating diverse, high quality alternatives). I explore solutions to these challenges in Chapter 3.

Two recent systems have enabled designers to explore alternatives in design prototyping. DesignScape [164], shown in Figure 2.8.1, provides designers automatically generated alternative suggestions for graphic design layouts using an energy-based model based on design properties (e.g., alignment, whitespace). Sketchsplore [209], shown in Figure 2.8.2, is an interface sketching tool that provides alternatives generated by human performance models (e.g., visual search, clutter). One limitation of these two systems is that a designer does not have any direct control over the space of alternatives the system explores and cannot give feedback on design alternatives to refine and explore different sets of alternatives. In Chapter 3, I present techniques that let designers give feedback on properties of alternatives they like and don't like, which can give them control over the space of alternatives the system generates. Sketchplore and DesignScape also do not let designers define high-level constraints to describe the semantics (e.g., ordering) and emphasis of their interfaces, described in Section 2.5.1, which can enable these systems to violate the high-level properties a designer desires. I present high-level constraints for interface layout alternatives in Chapter 3.

2.5.3 Algorithms to Generate Alternatives

Chapter 3 presents a system that enables rapid generation of alternatives through constraint solving techniques [30, 42, 102, 219, 224, 225]. Constraint solving has been used to generate alternative interfaces by past work. In *constrained optimization*, an a solver uses an objective function to produce higher quality alternatives according to some criteria. One example of this is Supple [74] which generates alternative interfaces through an optimization function based on human motor capabilities. This approach has also been applied to generate alternative interfaces for alternate devices and screen orientations. Decor [192] presents a system that recommends multi-device responsive interfaces that it ranks through an objective function based on design heuristics. Zeidler, et al. [226] generate alternative layouts for different screen orientations

through an objective function that measures layout quality as determined by aesthetic criteria. For interface design, Sketchplore [209] provides alternate layout suggestions for designs as a designer sketches them on a canvas and ranks them with an objective function based on predictive models of human performance and visual perception.

One limitation of past constraint-based layout systems [30, 42, 102, 219, 224, 225] is that they often produce only a single solution. Hotellier, et al. [93], in Programming by Manipulation, exploit ambiguities in constraints to generate alternative data visualizations. Jiang, et al. [100] also embrace such ambiguities to generate adaptive GUI layouts. However, their approach relies on specifying high-level patterns, and thus cannot generate layouts that do not exist in an existing pattern. Magellan [138] uses a genetic algorithm to generate alternatives. However, they support only a limited set of mutations which can limit the set range of alternatives their system can generate. In Chapter 3, I explore an approach that can generate a significantly larger set of alternatives by not relying on patterns or a fixed set of mutations.

Machine learning has also been applied to explore alternatives by transforming the content of an interface into the style and layout of another [114]. DesignScape [164] uses an energy-based model with encoded design principles to generate alternate graphic design layouts. LayoutGAN [122] synthesizes alternative layouts using a generative adversarial network based on the modeling of geometric relations of 2D elements. Key challenges with machine learning based approaches are that a designer can have no control over the attributes of the alternatives that the algorithms explore, and they require a large dataset of pre-existing designs. In Chapter 3, I present an approach that utilizes constraint solving which does not require a design dataset to generate alternatives. It also enables a designer to have more control over the space of alternatives the system generates.

Beyond these approaches, algorithms based on generative design [104, 139, 222] have been used to generate alternatives for 3D modeling [104, 139] and 2D graphic designs [222]. These algorithms can be computationally unfeasible for interactive systems. In some cases [139], it make take more than a day to create a generative design dataset that a designer can then explore through an interactive system. My approach, in Chapter 3, can be used interactively

Figure 2.9: Adaptive Ideas [120] (1) is a design tool for webpages that lets designers search and adapt stylistic elements directly from examples (bottom) into their own designs in a design canvas (top). D.Tour [179] lets designers search for examples using stylistic keywords (e.g., colorful image-heavy).

and does not require the alternatives to be pre-generated.

2.5.4 Exploring Alternatives through Examples

Designers seeking to explore alternatives also frequently look for examples to gain inspiration from and adapt ideas from the work of other designers [87,120]. An extremely common practice among today's interface designers is to explore examples shared online through design galleries like Behance[8] and Dribble[9]. Designers can find examples of interaction and design patterns through pattern repositories like UI Patterns[10] or gain inspiration from image sharing sites like Flickr[11] or Pinterest[12].

Tools for exploring and adapting examples from design galleries have not yet become

[8]https://www.behance.net/

[9]https://dribbble.com/

[10]http://ui-patterns.com/

[11]https://www.flickr.com/

[12]https://www.pinterest.com/

commonplace as features in interaction design tools, yet some research has explored interaction techniques to support these scenarios. D.Tour [179] allows designers to search for examples by color and style, but they cannot adapt them into their own designs. WebCrystal [50] lets designers extract and reuse design and structural information from webpage examples. Lee et al. [120] present Adaptive Ideas, a system that lets designers search and adapt stylistic elements directly from examples into their own designs in a design canvas. Bricolage [114] lets designers transform the layout of a webpage into the content and style of another.

One challenge with popular design galleries is that examples are frequently shared as screenshots. This means that designers must spend some time re-creating properties and shapes in the screenshot to modify them in a prototyping tool. In Chapter 4, I present a system that helps designers adapt example screenshots with less manual effort in re-creation, taking inspiration from tools and interaction techniques for example adaptation.

Another challenge with example galleries is that a designer cannot easily visualize the example designs with their own elements. In Scout (Chapter 3), designers can quickly explore a large number of layout alternatives, all based on their own elements and content, without needing to manually rearrange or restyle other examples.

2.6 Data-Driven User Interface Design

Data-driven design [113] is an approach to mining large-scale design data from the web and mobile apps to support design search, design curation, and data-driven interactions. This approach frequently relies on large-scale collection of design data (i.e., screenshots, structural layout metadata) from user interfaces such as webpages [113] and mobile apps [185].

2.6.1 Origins

For mobile applications, Shirazi et al. [185] analyzed the top 400 Android applications to yield design insights. They found that the complexity of mobile interfaces differs across application categories (e.g., shopping, social networks) and identified commonly used interface widgets and combinations that can be used to develop optimized widgets and tools for recommendation.

Beyond these insights, they were the first to present a *static* design mining approach to automatically download (i.e., from the Google Play store) and disassemble Android applications from their APKs (application programming kits) to extract screenshots and layout structure metadata (XML) to enable automated analyses of Android app design at a large scale.

Shirazi's approach [185] enabled several large-scale *static* data-driven design analyses such as that of Alharbi et al. [25] who present a framework for studying design pattern adoption which can be used to yield high-level insights (e.g., there are 5 common patterns used for navigation across apps) on the patterns are being used across apps at scale. Such insights can yield answers to questions such as: Which design patterns are used for which types of apps? and Why are some design patterns rarely adopted? Tian et al. [207] also adopted a static design mining approach to analyze the characteristics that distinguish high and low-rated apps (e.g., app size, code complexity).

While static design mining approaches can yield useful insights, a static approach cannot capture dynamic data and thus cannot be used to yield any insights to aid in the dynamic components of an app's design. The Erica system [61] presents *interaction mining* which is an approach that uses a web interface and app crawlers to enable capturing both static (e.g., UI and layout) and dynamic (e.g., interaction traces, motion details) of an app's design. In Erica, crowd workers interact with an app through a web interface, connected to a physical Android device, while Erica captures screenshots, view hierarchies (i.e., structural representation of interfaces), and user interactions. Erica's data was used to build machine learning classifiers to detect 23 common user flows (i.e., sequences of user states making up a semantically meaningful task such as searching, logging in, or composing).

2.6.2 Large-Scale Design Analysis & Insights

After demonstrating the benefits of an interaction mining approach with Erica [61], the authors used a platform similar to Erica's to create Rico [59], a large repository of mobile app design data, containing visual, structural, and interactive design properties of more than 72 thousand unique app screens. Rico was created with the goal of supporting a variety of data-driven applications

such as design search, UI layout and code generation, and user interaction modeling and prediction.

The Rico dataset has since been used to discover large-scale UI patterns for design analytics and design search. Doosti et al. used Rico to analyze the impact of Google's Material Design [11] finding that the use of Material Design was positively correlated with app quality as measured by user ratings. Such large-scale analyses can yield useful insights for debates over common patterns such as the Floating Action Button and Navigation Drawer [11]. Wu et al. [217] used Rico to build a model to predict human perception of brand personality, defined as "a set of human characteristics associated to a brand".

Rico has also been used develop classifiers to semantically annotate UI elements and icons automatically [132]. These semantic annotations can potentially enable design-based search interactions (e.g. allowing a designer to search examples by components, interactions, or user flows). Swire [94] introduces a system for retrieving relevant design examples from a hand drawn interface sketch, using Rico to develop the sketch dataset and to train a model for searching relevant interface design examples.

Beyond applications in design analytics and search, the Rico dataset has been used to better understand a key aspect of app privacy through studying the use of login features [141]. Another promising use of this dataset is in improving app accessibility through analysis of accessibility features [183] and through building a framework to support developers in annotation and repair of accessibility issues [230].

2.6.3 Adapting Analysis & Insights into Design Prototyping

Most recent work reviewed here in data-driven design has focused on large-scale insights while little has been integrated into design prototyping tools (e.g., Sketch, Adobe XD) directly. Although high level insights can be useful to designers, they do not have much impact on the daily work of designers when they are using their prototyping tools. In contrast to previous work, I seek to integrate data-driven design ideas directly into design prototyping tools. In Chapter 4, I present a system that lets designers adapt example screenshots into their designs.

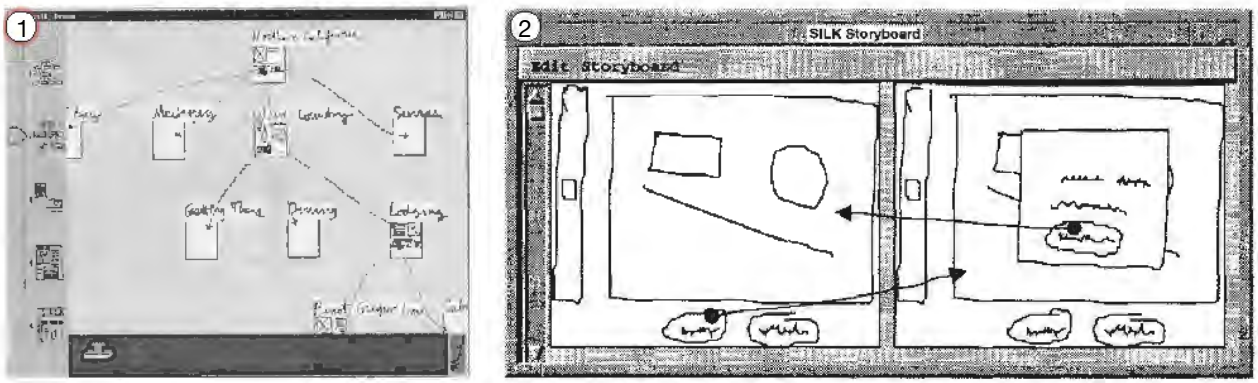


Figure 2.10: SILK (1) is an early tool for sketching graphical user interfaces on a tablet. DENIM (2) is a tool for prototyping website design at multiple levels of abstraction.

In Chapter 5, I use Rico [61] to collect human perception of tappability data to help designers discover false affordances and missing signifiers for tappable elements in mobile interfaces. Such an interface can be integrated directly into the app prototyping process so that an app designer can understand tappability, and potentially other key aspects of visual design and usability, while they are prototyping. This can potentially help them better understand how a visual design change they are making can affect human perception and usability of their design.

2.7 Digital Interface Design Prototyping Tools

In addition to the research my work directly advances upon, understanding the background that led to current digital interface design prototyping tool is important. Interface designers use digital prototyping tools to create low and high fidelity prototypes. Digital prototyping tools have a long history in research and industry. This dissertation can motivate features that can be made a part of future interface prototyping tools.

While there is no universally agreed upon definition for a prototype, Moggridge defines a prototype as a "*representation of a design before the final solution exists*" [144]. According to Lim et al. [127], prototypes are "*the means by which designers organically and evolutionarily learn, discover, generate, and refine designs.*" In the software engineering literature, prototypes are commonly manifested as working software prototypes that enable experimentation [125]. In interface design, designers create both low-fidelity (e.g., paper prototypes, wireframes) and

high-fidelity (e.g., digital mockups) prototypes to demonstrate their designs [144]. These prototypes are valuable tools for designers to get feedback, iterate, and refine their designs.

Digital prototyping tools enable designers to prototype their interface designs at both a low and high-fidelity. These tools have a rich history in user interface design. Early research in introduced the idea of interactive sketching to create user interface design mockups. The SILK system [115] (Figure 2.10.1) introduced a tool for sketching graphical user interfaces on a tablet. Designers sketch the interface while stroke recognizer recognizes the interface elements they are drawing. They can assemble multiple screens together to create storyboards and link them together by drawing transitions from interface elements. DENIM [130] (Figure 2.10.2) extends these ideas to the domain of web interface design, enabling designers to zoom based on the level of detail (e.g., overview, sitemap, storyboard, sketch). The Designer's Outpost [107] enables team collaboration on website prototyping. Designers can affix post-it notes to a digital display and link them through a digital pen to capture the hierarchy. For mobile interfaces, DeSa's mixed-fidelity prototyping tool [58] enables designers to create prototypes across multiple levels of resolution (e.g., wireframes, code).

Since these works, research has explored how to go beyond prototyping a single interface through enabling alternative prototyping and suggestion of alternatives. For creating multiple versions of a design, Damask [128] facilitates designers in prototyping cross-device user interfaces. Juxtapose [85] lets designers create multiple alternative prototypes, edit them through parameter tuning, and view them side by side. D.Note [83] gives designers the ability to track, visualize, and create revisions of design prototypes. Rather that having the designer create each alternative, or generate them through parameters, recent tools provide designers with brainstorming alternatives suggestions automatically [164, 209]. Research has also explored how to enable crowds to work together to create a design. With Apparition [117], a designer sketches and describes an interface, and crowd workers translate the designers input into user interface elements. In SketchExpress [121] designers describe their prototype while crowd workers create replayable animations through the SketchExpress interface.

Since the early digital prototyping tools (i.e., Denim [130], Silk [115]), the landscape of

digital prototyping tools has vastly changed. In 2008, Myers et al. [149] conducted a survey and found the most common digital prototyping tools were Adobe Photoshop, Adobe Dreamweaver, and Microsoft Powerpoint. According to a recent survey [13], the most popular digital wireframing and interface design tools were Sketch, Figma, Adobe Photoshop, and Adobe XD. Designers use digital tools for wireframes and high-fidelity design, however, the most popular tool for brainstorming and ideation is still pen and paper. While these tools have been enabling many advanced features (e.g., layout grid support, resizing constraints, cross-device prototyping), there is still little support for brainstorming and ideation within these tools, either with an interface designer's own interface elements, or through exploring examples of other designers' work. Support for modifying and adapting examples is limited. The work in this dissertation can inspire ways to adapt these features into prototyping tools.

[13] https://uxtools.co/survey-2018/

Chapter 3

Mixed-Initiative Exploration of Design Alternatives

During ideation, designers *explore alternatives* which results in higher-quality outcomes and more diverse solutions [47, 66]. When designers explicitly compare these alternatives, it can enable them to make stronger critiques and better decisions [65, 210]. Designers can explore examples through design galleries [120] (e.g., Behance) to help them ideate or sketch out their alternative design ideas to visualize them on paper or a whiteboard. Exploring alternatives is a key part of the *ideation* stage of design, although it does frequently appear throughout other stages of the design process (e.g., prototyping).

Currently, designers face many barriers in creating high-quality and diverse alternatives. First, it is difficult to overcome fixation to think of completely new ideas [99]. Designers often sketch alternatives on paper [47], but such sketches can be difficult to change and a designer is still limited by the ability to envision new ideas to sketch. Additionally, some designers may brainstorm new layout ideas by moving elements around on a prototyping canvas which may make it even more difficult to avoid fixation. Example galleries [87] (e.g., Behance, Dribble), can help designers find inspiration from other design examples. However, a designer still needs

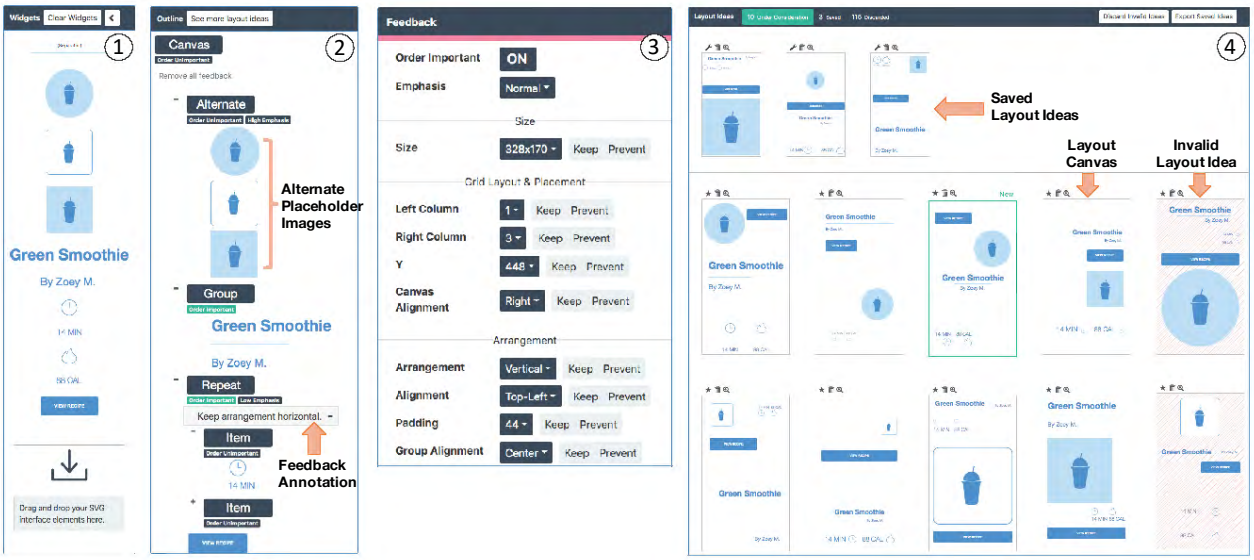


Figure 3.1: The Scout interface has four main panels: (1) Designers import their interface elements by dragging their SVGs into the *Widgets* panel. (2) Designers create hierarchy and high-level constraints (e.g., grouping, order, emphasis) in the *Outline* panel. (3) Designers control generation of alternatives through the *Feedback* panel, which they can activate by clicking an element in the Outline panel or on an element in the Layout Ideas panel. (4) The *Layout Ideas* panel presents alternative layouts, which a designer can save, discard, or zoom in on.

to manually adapt the example into a design alternative. This may require low-level resizing, restyling, and relocating of interface elements. This can be particularly challenging for novice designers, as they need to have knowledge of usability and visual design principles [126, 159] to maintain quality across alternatives.

To aid designers in exploring and creating alternatives, I present Scout, shown in Figure 3.1, a mixed-initiative system to help designers rapidly explore layout alternatives. A designer can use Scout to express their interface elements and constraints at a high level (e.g., grouping, order, emphasis), and Scout then generates multiple alternative layouts satisfying those constraints to augment the designer's ideation.

Scout applies constraint solving techniques to automatically generate alternatives. Constraints have a rich history in interface design and visualization [42, 93, 102, 219, 224, 226]. However, such research has generally focused on reducing ambiguity in constraints to produce a single design. In contrast, my goal with Scout is to leverage a constraint solver to generate

many diverse designs.

Additionally, constraint-based systems have generally focused on low-level spatial constraints (e.g., see Chapter 2, Section 2.2 for an example in Apple's AutoLayout [9] that looks like mathematical equation). Such constraints can be confusing and difficult for designers to construct. Scout lets designers specify high-level constraints based on *usability and visual design principles* like emphasis [5] and clear hierarchies [109, 213]. Scout translates a designer's high-level constraints into low-level spatial constraints that a constraint solver uses to generate alternatives satisfying the designer's constraints.

Through this work, I aim to further my goal of formalizing *interface design principles* into rules that can enable applying them automatically in interactive design tools. If we augment interface design tools with high-level semantic knowledge of interface design concepts, we can help designers more *rapidly visualize interface layout alternatives* by transforming their high-level design goals into flexible layout specifications that enable generating a large set of concrete layout ideas. Scout adopts a mixed-initiative approach [26, 91] which is a method of interaction where a person and an intelligent system collaborate to achieve a person's goals. Scout relies on the designer to define high-level goals, generates layouts satisfying them, and enables the designer to control the exploration by updating high-level goals and giving granular feedback. The key contributions of this chapter are are:

- Scout, a system to help designers rapidly visualize many layout alternatives through interaction with high-level constraints and feedback on alternatives.
- A set of constraint encodings based on design principles, with solving algorithms that enable generating a range of diverse layouts for a set of interface elements.
- An evaluation with 18 interface designers, finding: (1) that Scout can help them create more spatially diverse designs with similar quality to those created with paper and a baseline prototyping tool, and (2) qualitative feedback demonstrating Scout's potential as a tool for early ideation and breaking out of a linear design process.

3.1 Motivating Example

To describe and motivate Scout, consider an example scenario in which Eunice, a UX designer, is redesigning a landing page for a recipe app. Eunice has conducted a desirability study [36] of the current page. In such a study, participants assign emotional and descriptive keywords to a design (e.g., “creative”, “simple”). The top keywords assigned to Eunice’s current design were “dull” and “unrefined”. Eunice would like to change reactions to her landing page by using Scout to explore alternatives. First, Eunice imports a set of interface components from her company’s design library into Scout’s widgets panel (Figure 3.1.1). Then, Eunice clicks on elements in the widgets panel to add instances to her design’s outline panel. Specifically, she adds 3 alternatives for a smoothie placeholder image, a header and subtext, calorie and time labels and icons, and a “View Recipe” button. Her elements appear in Scout’s outline panel (Figure 3.1.2).

3.1.1 Specifying Hierarchy and High-Level Constraints

Eunice next specifies high-level constraints on her elements. Scout lets designers group related elements, specify a relative order, and give elements high, normal, or low emphasis. I designed Scout’s high-level constraints from common design principles for clear and usable layouts (e.g., [55, 109, 126]). For a review of the design principles I apply in Scout, see Chapter 2, Section 2.1.

Eunice’s first goal is to create a hierarchy. A key design principle is that interfaces should have a clear and organized hierarchy [126]. Similarly, the structure principle [55] states that interfaces should keep related things together and unrelated things separate, motivated by Gestalt theory [109]. In Scout’s Outline panel (Figure 3.1.2), Eunice creates a group for the “Green Smoothie” and “By Zoey M.” labels. For each group, Scout creates constraints to ensure these elements appear as visually distinct groups in layouts Scout generates.

Eunice next wants to specify that the "Green Smoothie" label should always appear before the "By Zoey M." label. A usability principle is that elements should appear in the order they are used for a task [159]. Scout lets Eunice specify the order is *important* or *unimportant* for each

group. When order is *important*, Scout encodes a constraint to maintain the spatial reading order of elements in a group (i.e., left to right and top to bottom). Additionally, Scout lets Eunice specify an element should appear first (e.g., a header) or last (e.g., a footer) on a canvas.

Many interfaces include repeating patterns of elements (e.g., a list, a grid). Scout supports *repeat groups* to ensure the layout of subgroups is consistent. Eunice creates a repeat group for the calories and minutes labels and icons (Figure 3.1.2). When Scout generates layouts (Figure 3.1.3), the layout of these subgroups is kept consistent (i.e., alignment, arrangement, order, padding). Scout also infers repeating patterns of elements within a group to suggest when this constraint can be applied.

Finally, Eunice wants to see layouts that use alternate versions of the smoothie image placeholder. She creates an *alternate group* with 3 different placeholders (Figure 3.1.2, "Alternate"). When Scout creates layouts, it uses only one of the three placeholders in each layout.

Eunice has created her initial high-level constraints, so she clicks “See more layout ideas” at the top of Scout’s Outline panel (Figure 3.1.2). Scout displays an initial set of 20 canvases satisfying Eunice’s high-level constraints in the Layout Ideas panel (Figure 3.1.4). Eunice sees that some layouts show the smoothie image too small and the calorie icon pairs too large in relation to other elements. She decides to set *emphasis* levels for these elements. Emphasis is a principle in interface design [213], stating that interfaces should have a main focal point to let a person know what to do next [5]. Scout allows specifying *High*, *Normal*, or *Low* emphasis. Eunice specifies in the Feedback panel that the smoothie placeholder should have *High Emphasis* and that the minutes and calories repeat group should have *Low Emphasis*. Scout will therefore adjust the size and position of her elements to make them more or less visually prominent.

3.1.2 Feedback & Layout Curation

After reviewing Scout’s generated layouts, Eunice decides to use a horizontal layout for the minutes and calories group. She clicks a layout with a horizontal group, and Scout displays a pink outline around the selected element (Figure 3.2.1) to indicate the Feedback panel is active for that element. That element is now the *primary selection*. Scout highlights the corresponding

element in other layout idea canvases in below to set them as the *secondary selection*.

The Feedback panel displays feedback properties that let Eunice "Keep" or "Prevent" specific property values in the alternatives (e.g., "Keep alignment left"). Eunice clicks the "Keep" button next to the arrangement dropdown for the minutes and calories group to tell Scout to use a horizontal arrangement for the group in future alternatives. Eunice's feedback appears in Scout's Outline panel as a *feedback annotation* (i.e., "Keep arrangement horizontal"). Eunice can also activate the Feedback panel by clicking an element in the Outline panel. In that case, Scout will set each feedback property dropdown to "Vary" until a "Keep" or "Prevent" feedback is applied. Scout supports multiple "Keep" and "Prevent" values for a property (e.g., "Keep arrangement horizontal OR vertical"). Scout lets Eunice give several types of feedback, including on the top-level canvas (e.g., "Keep layout grid 4 columns"), groups (e.g., "Keep arrangement

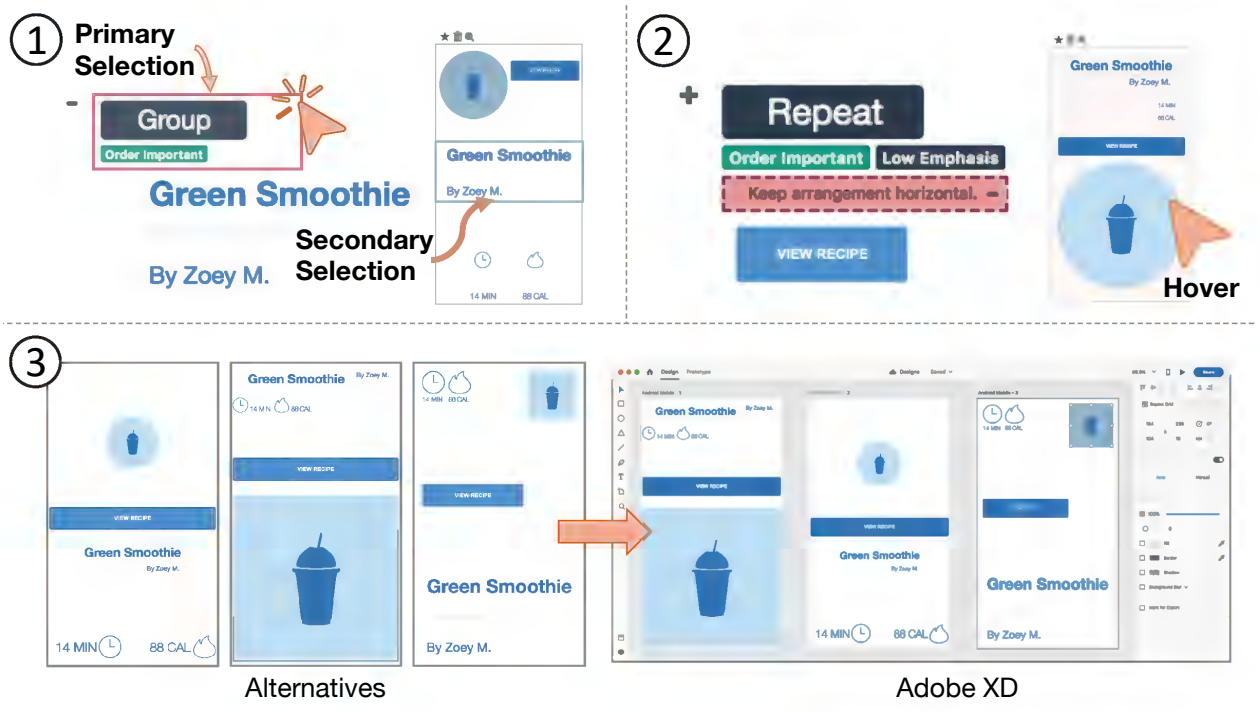


Figure 3.2: (1) Designers can click on nodes in Scout's Outline panel to make them the primary selection, which highlights corresponding elements in each canvas on the Layout Ideas panel. (2) Designers can hover their mouse over a layout canvas, and Scout highlights conflicting feedback annotations. (3) Designers can export their saved layouts into SVG canvases which they can import into their prototyping tools, such as Adobe XD.

horizontal”), and elements (e.g., “Keep location here”).

Values that a designer “Keeps” or “Prevents” can cause a conflict in existing layouts. Eunice sees that Scout has put red diagonal stripes over two of her layouts. She hovers her mouse over one of the canvases, and Scout highlights the conflicting feedback, labeled with "Keep arrangement horizontal.", in red (Figure 3.2.2). This means that layout has a conflict because the minutes and calories repeat group does not have a horizontal arrangement. When Scout detects such conflicts, Scout tries to repair the layout to match Eunice’s feedback. If it cannot repair the layout, Scout retrieves a new layout to replace it, ensuring that Eunice’s Layout Ideas panel is continually filled with new layouts as she applies her feedback.

Using Scout, Eunice explores over 100 layouts. She discards several by clicking the *trashcan* icon above each layout. As she finds layouts she likes, she saves them for export by clicking the *star* icon above each layout. Scout pins these layouts to the top of the Layout Ideas panel (Figure 3.1.4). After Eunice has found and saved 3 diverse layouts, she decides to refine them by exporting them out of Scout to edit in her favorite interface design tool (Figure 3.2.3). Scout exports each layout as an SVG with editable shapes and properties. Using Adobe XD, Eunice adjusts the alignment and relative size of the layouts until she feels they are ready for further feedback from her colleagues.

3.2 Architecture & Implementation

Before developing the current version of Scout, I conducted informal interviews with 6 interface designers, including their use and feedback on an early version of the tool. I draw upon their insights to support several key system design choices, including: (1) prioritizing interactive performance, (2) improving design quality through a design quality ranking model and utilizing a layout grid, and (3) preserving a designer’s current set of designs through a feedback resolver which can repair designs that conflict with new designer feedback.

Figure 3.3 illustrates Scout’s architecture. A designer provides a set of interface elements, each as an SVG (Figure 3.3.1). When the designer requests new layouts, Scout sends their interface elements and high-level constraints to the server (Figure 3.3.2), which launches

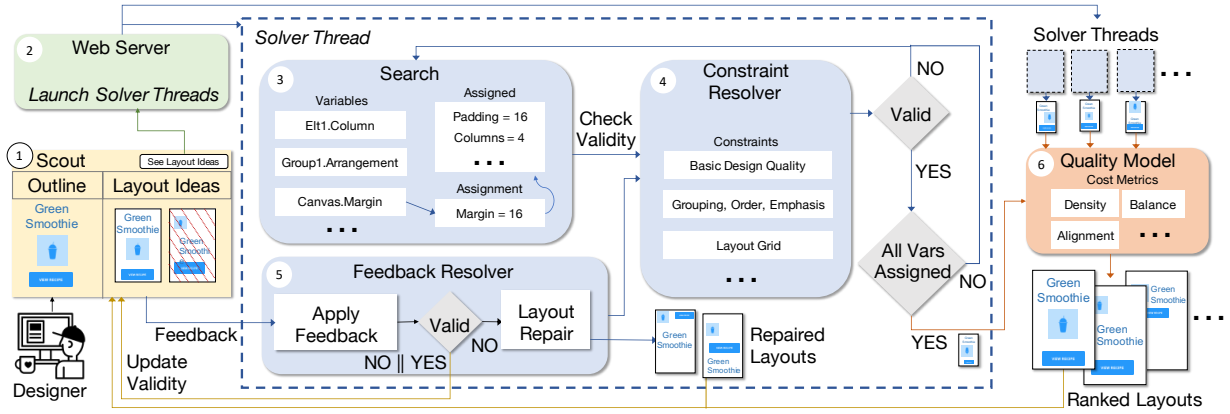


Figure 3.3: Scout System Overview: (1) A designer gives input to Scout via an outline of interface elements and feedback on layout alternatives. (2) A web server generates layouts by launching multiple solver threads. (3) Each solver thread searches over variable assignments. (4) A constraint resolver checks the assignments against constraints. (5) A feedback resolver applies designer feedback and repairs layouts. (6) A quality model ranks resulting layouts.

multiple solver threads to generate layouts with interactive performance (Figure 3.3.3-5). Each thread produces a layout, consisting of an *x position*, *y position*, *width*, and *height* for each element. Scout ranks each layout by a score computed with a quality model (Figure 3.3.6) based on design quality metrics (e.g., alignment, balance). Scout then displays the ranked set of layouts to the designer visually as an SVG *layout canvas*. A designer can give feedback on layouts, and a feedback resolver (Figure 3.3.5) applies the feedback and attempts to repair conflicting layouts. Scout currently supports mobile interface dimensions, but could be extended in the future to support arbitrary device dimensions.

3.2.1 Generating a Layout Alternative

Scout generates layouts through a modified branch and bound search [184], which generates a satisfying set of variable assignments (e.g., *alignment*, *arrangement*) (Figure 3.3.3) with respect to a set of design and high-level constraints on interface elements (Figure 3.3.4). Each variable has a domain of values Scout can assign through its search (e.g., *alignment* is one of top, left, x-center, y-center, bottom, right). Each constraint is a formalized as an equation encoded into the Z3 [57] constraint solver operating on one or more variables (e.g., element *size* and *position*).

Throughout this section, I format `constraint names` in a typewriter font and *variable names* in italics. The next section details Scout's constraints and variables.

Figure 3.3 shows Scout's process to generate a layout. First, Scout's search process (Figure 3.3.3) generates a single variable assignment for an element or group. The constraint resolver (Figure 3.3.4) then uses the Z3 [57] constraint solver to determine whether the assignment is valid. The constraint resolver translates high-level constraints specified by designers into formalized low-level variables and constraints on interface elements and layout behavior, which I detail in a later section. If the assignment is not valid, Scout backtracks in the search and reassigns the variable. If the assignment is valid and other variables remain unassigned, Scout assigns another variable and checks it through the constraint resolver. Finally, when Scout has assigned all variables, it produces a layout canvas which has a position and size for each interface element.

To ensure spatially diverse layouts, Scout randomizes the assignment order of variables and values using a uniform distribution. After Scout produces a layout, it encodes a constraint that prevents that same layout from appearing again. If a solver thread cannot produce a layout, Scout discards that thread. Scout can be configured to launch a variable number of threads based on system capabilities. For the evaluation, I configured Scout to launch 20 solver threads each time the designer requests new layouts. On the machine I used (Ubuntu 18.10 with AMD Ryzen 7 1800x processor, 8 cores x 16 threads, 32 GB memory), Scout typically returned 15 layouts per request in less than 5 seconds. Such resources are common for many designers (e.g., who also work with image and video data), but Scout could also run in a configuration with solver threads shifted to a scalable cloud service.

3.2.2 Ranking Layouts by Quality Metrics

Scout's layout search space is extremely large (i.e., trillions). Some layouts are not well-aligned or visually pleasing, and designers need a way to prioritize higher-quality layouts during their exploration. We created a quality model (Figure 3.3.6) to compute a *layout quality* score for each layout, formalizing some design principles described in interface design literature. Scout

uses these scores to rank higher-scored layouts toward the top of the Layout Ideas panel. We adapted this model from [178], which presents a set of metrics to computationally measure mobile interface complexity (e.g., misalignment, imbalance, density).

Scout's *layout quality* score formalizes the design principles of *alignment*, *balance*, and *simplicity*. For an overview of these principles, see Chapter 2 (Section 2.1). *Simplicity* is a broad principle that covers several aspects of a design including functional elements of behavior, which elements are included in an interface, and *visual simplicity* which can be improved by reducing visual noise and clutter. Scout formalizes visual simplicity by measuring the density of a layout, under the assumption that an interface should be neither too dense (i.e., too little whitespace which can cause the design to seem cluttered) or too sparse (i.e., too much whitespace). One limitation of this is that preferences for interface density differ across cultures. Some cultures prefer interfaces that are more cluttered and some prefer interfaces that are more sparse [175]. This indicates a need to consider designing culturally adaptive cost functions in future work.

Computation of Layout Quality Score

For each group of elements $G = \{e_1, \ldots, e_n\}$ in a layout, Scout computes $quality_g(G)$ which consits of three parts: element size, balance, and alignment. Here I describe each part and include definitions of the design principles that provide motivation and rationale for each part of the layout quality score, $quality_g(G)$.

$$quality_g(G) = s_{size}(G) + s_{balance}(G) + s_{alignment}(G)$$

Minimum Sizing Design Principle: *Small elements tend to be more difficult to interact with. Elements that are touch targets should not be too small. Avoid making font size of text elements too small, which could compromise readability for visually impaired users* [37, 71, 169].

Clear Hierarchy/Emphasis Principle: *Too many excessively large elements in an interface can compete for the users attention, causing a poor visual hierarchy* [126, 213].

The *size* score penalizes groups with excessively large or small elements. It computes the

sum of the normalized width and height of each element (i.e., normalized by the width and height of the canvas), divided by the number of elements.

$$s_{size}(G) = \frac{1}{2|G|}\sum_{e \in G}(\frac{e.w}{W} + \frac{e.h}{H})$$

Balance Design Principle: *The aesthetics, stability, and unity of a design can be improved by placing elements and whitespace in such a way that no area of the design overpowers another. Balance is defined as having an even distribution of visual weight* [126].

The *balance* score rewards groups with evenly-spaced margins between consecutive pairs of elements. It computes the difference between the average horizontal and vertical margins $G.avg\_margin_h$, $G.avg\_margin_v$ and the maximum horizontal and vertical margins $G.max\_margin_h$, $G.max\_margin_v$.

$$s_{balance}(G) = \frac{1}{2}(\frac{G.avg\_margin_h}{G.max\_margin_h} + \frac{G.avg\_margin_v}{G.max\_margin_v})$$

Alignment Design Principle: *The placement of elements such that edges line up along common rows or columns, or their bodies align along a common center. Alignment can create unity and cohesion across a design improving its aesthetic* [126].

The *alignment* score measures the quality of alignment within a group by maximizing the number of alignment edges between elements. For each pair of elements e_1, e_2, *NumAlignment* returns the number of horizontal (i.e., top, y-center, bottom) and vertical (i.e., left, x-center, right) alignments between those elements. *NumPossibleAlignments* returns the maximum number of alignments the two elements could have. For example, if e_1 and e_2 are horizontally arranged and have the same height, they can have a maximum of 3 alignments (i.e., top, y-center, bottom). The score therefore measures the proportion of alignment pairs out of the total number of alignments.

$$s_{align}(G) = \frac{1}{|G|}\sum_{(e_1,e_2) \in G}\frac{NumAlignment(e_1, e_2)}{NumPossibleAlignment(e_1, e_2)}$$

Visual Simplicity/Density Design Principle: *Use whitespace effectively to simplify your inter-*

face and separate unrelated elements. Without whitespace, your design may seem cluttered. [213].

Finally, Scout computes an overall *layout quality* score as a weighted-average of each group quality score $quality_g(G)$, where each group is weighted by its area. The *layout quality* score also includes: (1) a *density score* s_d to measure the ratio of the entire layout area covered by elements, and (2) a *group quality* score treating the top-level set of groups as an additional group (i.e., to measure the quality of layout of those top-level groups on the canvas).

$$quality_l(L) = \frac{\sum_{G \in L} G.area \cdot quality_g(G)}{\sum_{G \in L} G.area} + s_d + quality_g(L)$$

3.2.3 Feedback & Layout Repair

After Scout produces an initial set of layouts, a designer can update the outline (i.e., change the grouping, emphasis, or order of elements) or give feedback on variables (i.e., canvas, group, or element variables), prompting Scout's feedback resolver (Figure 3.3.5) to recheck the validity of each layout. For any "Keep" feedback, Scout encodes a equality constraint into the solver (e.g., *group.arrangement* = "vertical"). Conversely, "Prevent" feedback is encoded as an inequality constraint (e.g.,*group.arrangement* $\neq$ "vertical"). Scout checks validity of each layout with respect to the outline and constraints, then updates their validity in the interface (i.e., with red diagonal stripes over invalid layouts). Scout uses Z3's [57] `unsat core` to obtain the smallest set of constraint clauses that cannot be satisfied. When a designer hovers over an invalid layout, Scout examines these conflicting clauses and highlights the corresponding feedback annotations.

A designer can apply feedback that makes many layouts invalid. To prevent the designer from needing to frequently request new layouts, Scout continuously generates layouts as a designer applies feedback. To minimize change to the current set of layouts, Scout tries to return similar layouts through a layout repair module (Figure 3.3.5). Layout repair iteratively removes a variable assignment from an existing invalid layout until it becomes valid (Figure 3.3.4, Layout Repair). To prevent overwhelming a designer with too many layouts, Scout does not repair or generate new layouts if the number of valid layouts in the Layout Ideas panel is more than 50

(i.e., the number that could reasonably be visible on a 24-inch monitor). However, a designer can still explicitly request new layouts with the "See more layout ideas" button.

3.2.4 Constraint Encodings & Design Variables

Scout generates layouts through an assignment of concrete values to a set of variables, allowing it to explore many combinations of element arrangement, alignment, position, and size. Scout defines *canvas* variables (e.g., layout grid, margin, baseline grid), *group* variables (e.g., arrangement, alignment, padding), and *element* variables for *position* (e.g., x, y), and *size* (e.g., width, height). Each variable has a domain of values, curated from design guidelines [11] and layout design literature [28], together with constraints that define its behavior. Scout uses these constraints, together with a designers high-level constraints, to check the validity of a layout (Figure 3.3.4).

The following sections give high-level descriptions of these constraints and a set of general design quality constraints that Scout enforces. I provide detailed descriptions and formalized low-level equations for each constraint in Appendix A, Section A.1. The constraints in Appendix A demonstrate the form of the constraints that Scout encodes into Z3 [57] in the constraint resolver to validate against the current set of variable assignments (Figure 3.3.4).

Ensuring Basic Design Quality

Scout encodes three `basic design quality` constraints for every layout, an approach also used by Beilik et al. [38] in encoding a set of "Robustness Properties" for Android layouts. For each element, Scout enforces a `stay-in-bounds` constraint that keeps elements inside the bounds of the layout canvas (implemented the same as `inside_screen` in [38]). Scout also encodes a pairwise `non-overlapping` constraint on the bounding boxes of each element. Finally, Scout encodes `minimum` and `maximum sizing` constraints for each element, based on design guidelines (e.g., touch targets should be at least 48x48 pixels [11]).

Placing Elements on the Layout Canvas

Scout uses a layout grid to place elements on a canvas by encoding constraints on an element's bounding box. A *layout grid* is a common method designers use to place elements, which can improve alignment, consistency, and visual organization [212]. It consists of *margins* (i.e., spacing on the outside of the canvas that all elements must be placed inside), *columns* (i.e., vertical containers for placing elements on the canvas), and *gutters* (i.e., spacing between columns where elements must not be placed). To see an example layout grid with these components highlighted, see Figure 2.2. Mobile interfaces typically use a 2 to 4 column layout grid [11], within which elements or groups must begin and end on a column and not in a gutter. Scout defines 4 layout grid variables for a canvas: *margin*, *columns*, *gutter width*, and *column width*. Based on these variables' values, Scout encodes **`layout grid`** constraints that require the left and right edges of elements and groups that are direct children of the canvas to begin and end on the edge of a column.

Baseline grids define the vertical spacing of a design, aid horizontal alignment, and create hierarchy [28]. They consist of horizontal lines at even intervals to which all components should align. Figure 2.2 in Chapter 2 shows an example of a baseline grid in an interface prototyping tool (i.e., Adobe XD). Scout defines a *baseline grid* variable that allows designers to examine different baseline grid options. Based on this, Scout encodes **`baseline grid`** constraints specifying that elements have a *y position* aligned to a baseline grid line and a *height* that is a multiple of the **`baseline grid`** value.

Resizing Elements

To explore different element sizes, Scout defines a *size* variable for each element with a domain of the form `(`*width*, *height*, *sizing_factor*`)`, where *sizing_factor* is used to enforce consistent resizing within groups and repeat groups. Scout pre-computes *width* and *height* domains using two strategies: *maintain aspect ratio* and *increase width*. For both strategies, Scout computes a set of `(`*width*, *height*, *sizing_factor*`)` triples along baseline grid increments, where *width* values

range from a minimum determined by element type to the canvas width. For *maintain aspect ratio* elements (e.g., images, icons), *height* values vary from a minimum for each element to the canvas size. For *increase width*, height values do not vary. Scout encodes the minimum sizes for each element based on its type, which it determines based on usability principles for minimum sizing of elements [13, 14, 169]. Scout encodes each pre-computed set of triples as the domain to a *size* variable. This is a performance optimization because Z3 does not efficiently compute multiplication constraints (i.e., needed for maintaining an aspect ratio).

Grouping and Order

Designers can group elements in the Outline panel to keep them together. Scout varies layout of grouped elements based on three variables: *alignment*, *arrangement*, and *padding*. Scout encodes constraints aligning grouped elements along 6 possible *alignment axes*: left, x-center, right, top, y-center, and bottom. Scout currently aligns all elements within a group to a single axis. Scout defines four *arrangement* domain values for each group: horizontal, vertical, balanced rows, and balanced columns. Each **`arrangement`** constraint encodes rules based on the position and size of grouped elements. Scout defines **`padding`** constraints that work with **`arrangement`** constraints to add spacing between grouped elements while keeping them relatively close to other. Finally, Scout defines **`visual hierarchy`** constraints to keep the within-group *padding* smaller than the group's distance to other groups in the layout to visual separate them.

To allow designers to control the order of elements, Scout allows specifying order as *important* or *unimportant* in a group. For groups with *important* order, Scout encodes an **`ordering`** constraint that combines with **`arrangement`** constraints to keep the elements in the fixed order specified in the outline. For groups with *unimportant* order, Scout encodes a constraint on the height and width of the group bounding box, according to the *arrangement* variable. This allows elements to change position if other constraints are met (e.g., horizontal arrangement). If order is *important* for the top-level canvas, Scout encodes a constraint on each pair of elements such that the bottom edge of an element must be above any element that is later in the ordering. Scout also allows specifying that an element should be *first* or *last* in a group, which enables

specifying a fixed position for elements like a label, header, or footer. Scout encodes constraints requiring these elements to be first or last in the group. For the top-level canvas, Scout encodes pairwise constraints stating the top edge of any *first* element should be above all other elements and the bottom edge of any *last* element should be below all other elements.

Emphasis

To support designers specifying a visual hierarchy, Scout includes emphasis constraints based on design guidelines [213] that state emphasis can be increased or decreased by modifying an element's size in relation to other elements. Scout supports 3 levels of emphasis levels: low, normal, and high. All elements have normal emphasis by default. For elements with low or high emphasis, Scout encodes a `size decrease only` or `size increase only` constraint on the element's `size` variable that allows the element's size to decrease or increase. Scout also specifies an `area` constraint stating that (1) elements with *high* emphasis should have a larger area than elements without high emphasis, and (2) elements with a larger area should appear earlier in the order. Scout encodes similar `low emphasis` constraints, constraining *low* emphasis elements to have a smaller area and constraining smaller elements to appear later in the order.

Alternate Representations and Repeating Patterns

Alternate groups let a designer show alternate elements (i.e., SVGs) in different layouts. For each alternate group, Scout creates a *representation* variable with a domain corresponding to the elements the designer groups. Scout's search (Figure 3.3.3) assigns a value to this variable, which a designer could "Keep" or "Prevent" like other variables.

Repeat groups indicate a layout should be kept consistent across multiple subgroups (e.g., a list, a grid). A repeat group contains a set of subgroups, each with the same number of elements, the same types (e.g., button, text, image), in the same order. Each element in a subgroup has a corresponding element in all other subgroups, determined by their order. Figure 3.1.2 shows a repeat group containing two pairs of icon and label (i.e., minutes and calories labels with

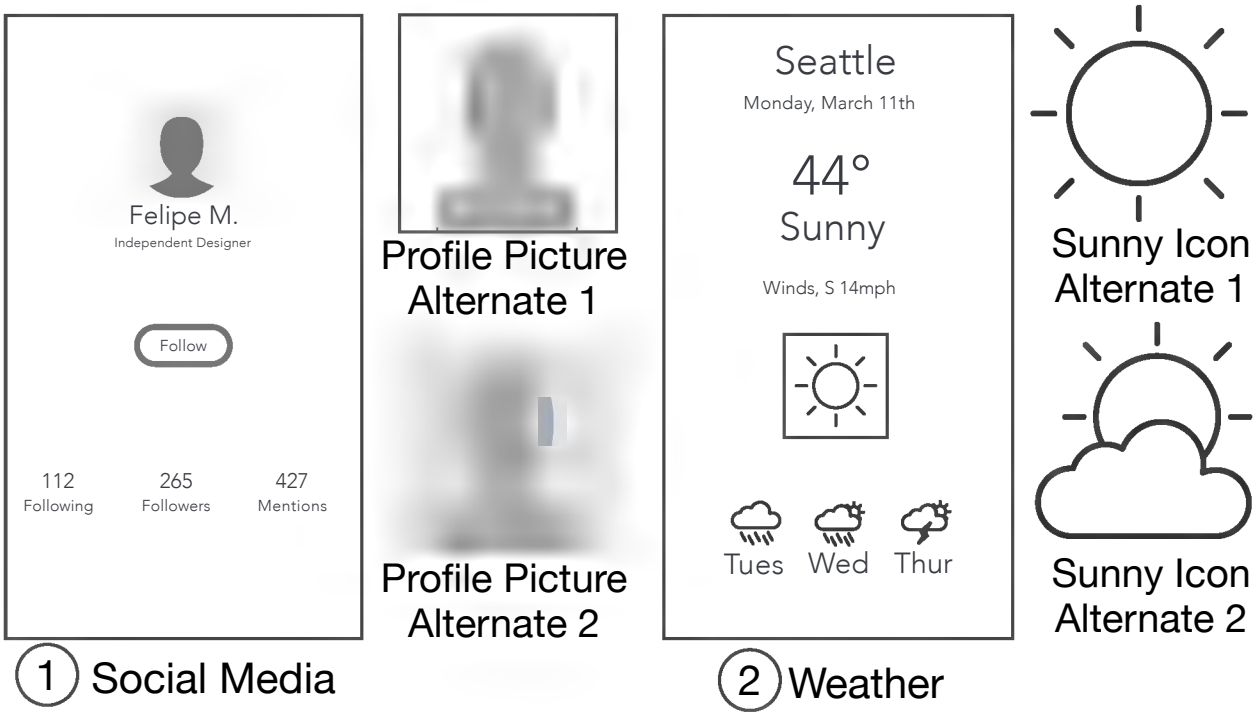


Figure 3.4: The components I provided designers for the *Social Media* and *Weather* scenarios, including alternate images for the profile picture and sunny icon.

corresponding icons that should always be arranged similarly). For each subgroup in a repeat group, Scout encodes a constraint that requires the *arrangement*, *alignment*, *padding*, and *order* variable values of all subgroups to be equal. Scout also requires the *size* variable increase or decrease be the same for corresponding elements in each subgroup.

3.3 User Study

To understand Scout's benefits and limitations, and to examine how different designers might use mixed-initiative layout ideation, I conducted a within-subjects, mixed-methods evaluation centered on three research questions:

- RQ1: Does Scout help designers of varying expertise generate **more diverse** interface layouts than with a baseline tool?
- RQ2: Does Scout help designers of varying expertise generate **higher quality** interface layouts than with a baseline tool?
- RQ3: How does Scout **affect designer processes** of exploring potential interface layouts?

3.3.1 Participants

I recruited 18 interface designers (5M, 13F, ages 18-32), 9 in each of 2 *Experience Level* groups: (1) *Professional Designers*, with >=1 year of professional UI/UX design experience; and (2) *Non-Professional Designers*, who had taken at least one interface design course and/or built a complete interface prototype, but had <1 year of professional experience. *Professional* designers reported a range of professional experience (1 to 3 years of experience: 5; 3 to 5 years: 2; more than 5 years: 2). Five *Non-Professional* designers self-reported no professional UI/UX experience, while four reported less than 1 year of experience.

3.3.2 Procedure

Each participant completed two 30-minute wireframe prototyping tasks, varying *Interface* to use *Scout* and a *Baseline* prototyping tool, Adobe Experience Design (XD). To better examine the use of Scout, rather than participant learning of Scout's interface, participants completed a 20-minute Scout tutorial and warmup task (i.e., exploring layouts for a To Do List). After the tutorial and before proceeding with the task, participants demonstrated how to use Scout's outline grouping and its alternate and repeat group constraints. All participants had experience with Adobe XD and had used similar tools (e.g., Figma, Sketch), so I did not include a corresponding *Baseline* warmup task. I collected screen and audio recordings and notes while participants completed tasks, then interviewed them after each task to reflect on their process using each *Interface* and on differences in using Scout versus their current process. The total amount of time for each session was 1 1/2-2 hours based on the length of the interview portion.

I developed two *Scenario* as a hypothetical setting for participant tasks: redesigning two app screens for a design agency: (1) a *Social Media* profile screen (Figure 3.4.1), and (2) a *Weather* app screen (Figure 3.4.2). I asked the participants to create three diverse alternative layouts for both the *Social Media* and *Weather* app screens. I selected familiar screen types so designers could focus on improving screen layout rather than the content. As in the scenario with Eunice described earlier, the task scenarios described that the agency had conducted a desirability

study [36] and that keywords assigned to the weather and social media app screens were “dull” and “familiar”. The instructions asked the designers to redesign for the keywords “clean” (i.e., "uncluttered and well-aligned") and “compelling” (i.e., "has a clear point of emphasis"), attributes of good layouts from design guidelines [5, 126, 213].

3.3.3 Materials

I provided pre-created wireframe components for the original design, 2 alternate profile pictures for *Social Media*, and two alternate sunny icons for *Weather* (Figure 3.4). The task content encouraged using Scout’s *repeat group* (e.g., for the days of the week and weather icons), but I did not require participants to use any particular constraint. The app screens were similar in complexity (e.g., number of components, groups of elements).

Because Scout is focused on layout, I limited participant use of Adobe XD to spatial features (e.g., position, size, font size) and not non-spatial features (e.g., color, font type). I required participants to use only the provided components without overlapping or rotating them.

I limited each task to 30 minutes. I allowed sketching on scratch paper in both *Interface* conditions. In *Baseline*, participants could use available time to sketch and create their alternatives. For *Scout*, I told participants to use Scout for 20 minutes, save 3 alternative layouts, and spend 10 minutes refining and finalizing the layouts in Adobe XD. I include the task instructions for the *Baseline - Social Media* and *Scout - Weather* conditions in Appendix A, Section A.2.

3.3.4 Analysis

To address learning or other carryover effects, I counterbalanced *Interface* (i.e., *Scout* or *Baseline*) and *Scenario* (i.e., *Social Media* or *Weather*) using a Latin square design. I performed the analysis using mixed effect models, treating *Participant* as a random effect and modeling *Interface*, *Scenario*, and *Experience Level* as fixed effects.

	Group	RG	AG	Order	Emphasis	Feedback
Count (% of 180)	36 (20%)	15 (9%)	16 (9%)	44 (24%)	38 (21%)	31 (17%)
% Designers (n=18)	94%	78%	89%	94%	83%	72%

Table 3.1: Summary counts of the number and proportion of high-level constraints of each type specified by designers following the Scout task, and the percentage of designers who specified each type of high-level constraint.

3.3.5 Results

Overall, the designers explored a large number of layout ideas using Scout during the session, generating an average of 97 layouts (min: 19, max: 280, SD: 81). At the end of the Scout task, each designer had an average of 10 high-level constraints specified (min: 6, max: 17: SD: 3.8). Note that we did not capture modifications to high-level constraints that the designers made during the study. We instead counted the types and amounts of high-level constraints that were currently specified by each designer in Scout's Outline panel at the end of the Scout task. Therefore, designers likely specified a larger amount of high-level constraints in Scout because we observed many of them iteratively modifying the high-level constraints throughout the study after viewing the Scout layouts. Most designers specified high-level constraints of all types supported by Scout (i.e., group, alternate group, repeat group, order, emphasis, feedback). Table 3.1 summarizes the percentage of designers that used each type of high-level constraint and the number of high-level constraints of each type specified at the end of the study. Next, I summarize the results per each research question.

RQ1: Does Scout help designers of varying expertise generate more diverse interface layouts than with a baseline tool?

I wanted to understand Scout's impact on helping designers explore more diverse layouts. Given Scout's focus on spatial arrangement, I defined diversity as *spatial diversity*. Although there are existing computer vision dissimilarity metrics [143], they are not suitable to compare the wireframes from the Scout study (i.e., the fact that wireframes are primarily whitespace causes

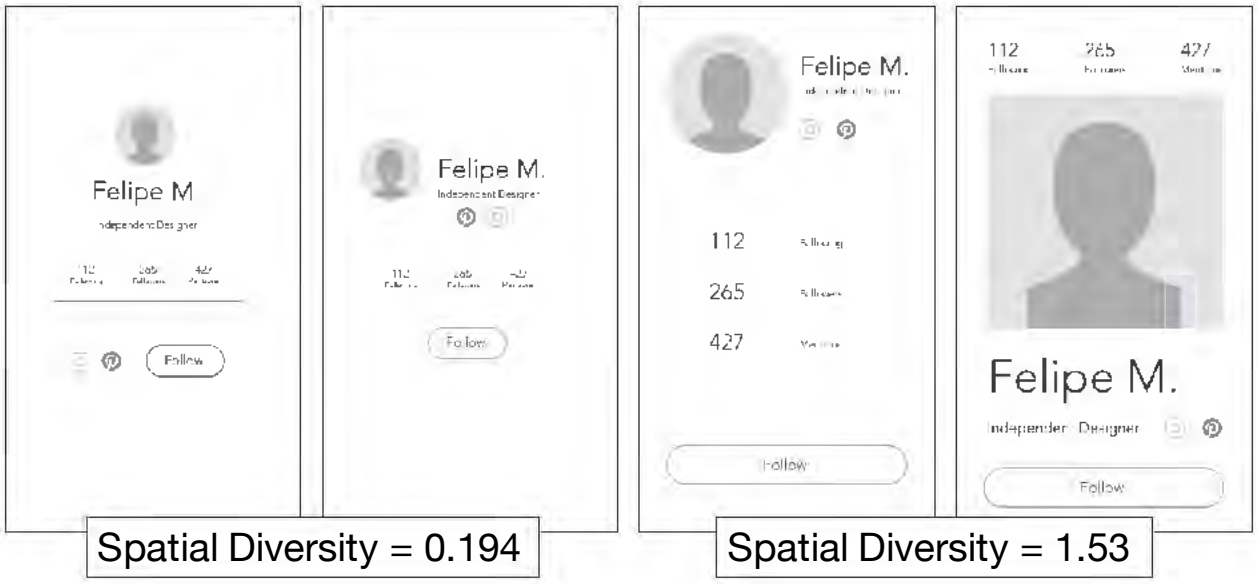


Figure 3.5: To illustrate our spatial diversity score, the least diverse (left) and most diverse (right) pairs of participant-produced *Social Media* layouts.

these approaches to fail). I collaboratively developed a spatial diversity score to estimate the effort needed to adapt one layout to another (i.e, transformation distance [80]). Gajos et al. [73] present a dissimiliarity metric capturing transformation distance by comparing each layout along a set of dimensions (e.g., orientation, representation). With Scout, we adopt a similar approach, defining a *spatial diversity* score for a pair of layouts containing the same elements in terms of 3 metrics: (1) mean *position change* (s_{dist}) computes the mean of the distance that each element moved between the two layouts, (2) mean *size change* (s_{size}) computes the mean of how much the area of each element changed between the two layouts, and (3) mean *relational distance change* (s_{rel}) measures the mean of the position change of an element in relation to all other elements in the layout. We compute these scores as follows.

First, the mean *position change score* s_{dist} computes the mean position change between the centers for each matching element in the two layouts. Given two layouts L and L' with elements $L = \{e_1, \ldots, e_n\}$ and $L' = \{e'_1, \ldots, e'_n\}$, Scout calculates s_{dist} as follows:

$$s_{dist}(L, L') = \frac{1}{n} \cdot \sum_{i=1}^{n} \sqrt{(e.x - e'.x)^2 + (e.y - e'.y)^2}$$

Second, the mean *size change score* s_{size} measures the average change in size for each matching pair of elements in the two layouts. Given two layouts $L = \{e_1, \ldots, e_n\}$ and $L' = \{e'_1, \ldots, e'_n\}$, Scout calculates s_{size} as follows:

$$s_{size}(L, L') = \frac{1}{n} \cdot \sum_{i=1}^{n} |e_i.width \cdot e_i.height - e'_i.width \cdot e'_i.height|$$

Third, we compute a *relational distance score* score to measure how much each element moved in relation to all other elements in the layout. Given two layouts $L = \{e_1, \ldots, e_n\}$ and $L' = \{e'_1, \ldots, e'_n\}$, we calculate the relational distance score s_{rel} as follows:

$$s_{rel}(L, L') = \frac{2}{n(n-1)} \cdot \sum_{1 \leq i < j \leq n} |dist(e_i, e_j) - dist(e'_i, e'_j)|$$

where $dist(e_i, e_j) = \sqrt{(e_i.x - e_j.x)^2 + (e_i.y - e_j.y)^2}$ calculates the distance between centers of the two elements.

Finally, we compute an overall *spatial diversity score* $s_{diversity}$ for a pair of layouts as the weighted sum of the three metrics $s_{dist}, s_{size}, s_{rel}$:

$$s_{diversity} = w_{dist} \cdot s_{dist} + w_{size} \cdot s_{size} + w_{rel} \cdot s_{rel}$$

To ensure each metric is weighted equally, we normalize the metrics into the range of $[0, 1]$. Given the entire set of layouts designers created in our evaluation $\{L_1, \ldots L_n\}$, we compute the weights $w_{dist} = \frac{1}{max\_dist\_change}$, $w_{size} = \frac{1}{max\_size\_change}$, $w_{rel} = \frac{1}{max\_rel\_dist\_change}$. where each weight ($w_{dist}$, w_{size}, w_{rel}) divides a score by the maximum distance change, size change, and relational distance change for any pair of elements in the evaluation set of layouts created by our designer participants, L_i, L_j, respectively. Figure 3.5 shows two pairs of designs that designers created in the Scout study that received the smallest and largest spatial diversity scores.

To examine Scout's impact on spatial diversity within designs created by an individual designer, I conducted a within-designer analysis. I computed the *spatial diversity score*, $s_{diversity}$,

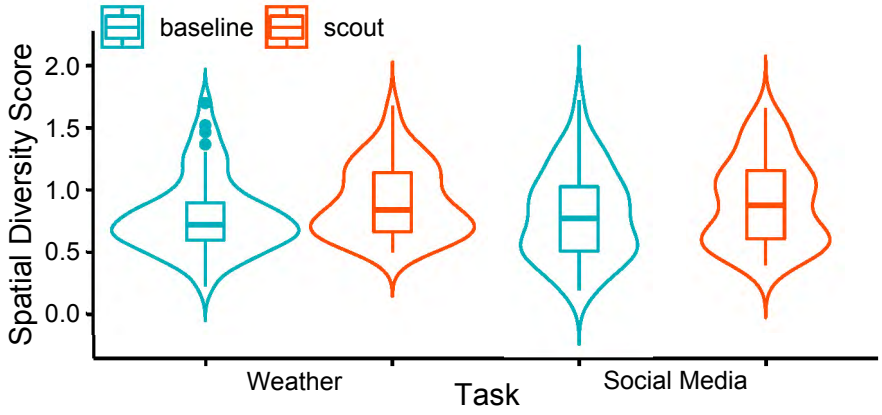


Figure 3.6: Violin plots of the spatial diversity scores for each set of pairs by a designer within an Interface/Scenario combination demonstrating that the Scout designs had higher spatial diversity for both scenarios.

for each pair of designs created by a designer with each *Interface*, excluding the original design (i.e., 3 pairs per designer per *Interface*). Figure 3.6 shows violin plots by *Interface* and *Scenario*. The 54 pairs of *Scout* designs were 12 percent more spatially diverse ($M = 0.880, SD = 0.290$) than the 54 *Baseline* pairs ($M = 0.788, SD = 0.356$). Spatial diversity scores were not normally distributed, so I conducted an aligned-rank-transform analysis [215], which indicated a significant effect of *Interface* on spatial diversity ($F_{1,86} = 5.05$, $p < 0.027, d = 0.435$). The analysis did not find a significant effect of *Experience Level* on spatial diversity score ($F_{1,14} = 0.009$, $p < 0.926$).

To examine whether Scout helped designers create layouts that were more spatially diverse relative to the original design, I computed a spatial diversity score for each layout relative to the original (i.e., 3 pairs per designer per *Interface*). Scout helped designers create layouts that were 15 percent more spatially different ($F_{1,86} = 5.35, p < 0.023, d = 0.45$) than the original design ($Scout : M = 0.926, SD = 0.343$, $Baseline : M = 0.807, SD = 0.315$). Although the effect of *Experience Level* on spatial diversity was not significant ($F_{1,14} = 0.038, p < 0.848$), the analysis showed a significant interaction effect between *Interface* and *Experience Level* ($F_{1,86} = 4.46, p < 0.038$). Using Scout increased spatial diversity by 35% for *Non-Professional* participants ($Baseline : 0.749, Scout : 1.01$), while decreasing spatial diversity for *Professional* participants

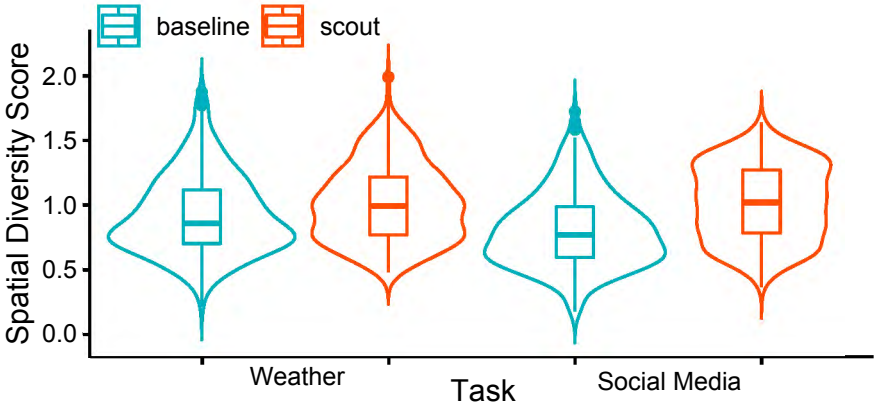


Figure 3.7: Violin plots of the spatial diversity scores across all pairs of designs by all designers within a Interface/Scenario combination showing that the Scout layouts had higher overall spatial diversity than the Baseline layouts for both the weather and social media scenarios.

by 2 percent ($Baseline$: $0.866, Scout$: 0.847). An interaction contrast, corrected with Holm's sequential Bonferroni procedure, indicated this difference when using Scout according to *Experience Level* was significant ($\chi^2(1, n = 27) = 4.46, p < 0.035, d = 0.41$).

Finally, I examined Scout's effect on overall spatial diversity across designers. I computed the entire set of pairwise spatial diversity scores within the *Social Media* and *Weather* scenarios. Figure 3.7 shows Scout increased the overall mean spatial diversity score for the *Social Media* scenario by 26 percent ($Baseline : M = 0.811, SD = 0.290, n = 351$, $Scout : M = 1.02, SD = 0.289, n = 351$) and for the *Weather* scenario by 10 percent ($Baseline : M = 0.926, SD = 0.296, n = 351$, $Scout : M = 1.02, SD = 0.306, n = 351$). Spatial diversity scores were not normally distributed (Shapiro-Wilk $W > 0.974, p < .0001$). Using the Wilcoxon rank sum test, I found a significant difference in means for both the *Social Media* ($W = 50640, p < .0001, r = 0.342$) and *Weather* ($W = 37243, p < .0001, r = 0.154$) scenarios.

RQ2: Does Scout help designers of varying expertise generate higher quality interface layouts than with a baseline tool?

I assessed the quality of participant designs with a panel of 2 independent interface designers, each with at least 3 years of professional UX design experience, who each evaluated each

n = 54	VB	TH	E	A	W	LQ
Scout (M)	2.67	3.07	2.39	2.5	2.82	5.37
Scout (Std)	0.97	0.84	1.09	0.82	0.87	1.0
Baseline (M)	3.09	3.01	2.65	2.83	2.77	5.73
Baseline (Std)	0.96	0.79	1.13	0.88	0.98	1.24

Table 3.2: Summary statistics of the quality scores awarded by the expert evaluators to designers' layouts from the Scout user study including visual balance (VB), typographical hierarchy (TH), emphasis (E), alignment (A), whitespace (W), and overall layout quality (LQ).

design on a layout evaluation rubric. The rubric included 5 items. The first 3 focused a design being "compelling": (1) `visual balance` - *"The layout is easy to scan, and all elements are aligned with respect to axes of symmetry"*, (2) `typographical hierarchy` - *"All elements follow a typographical hierarchy and are easily readable and proportionally sized with respect to each other."*, and (3) `clear point of emphasis` - *"The wireframe has a clear point of entry or a single visually salient feature, that does not overwhelm the design."*. The final 2 focused on a design being "clean": (4) `alignment` - *"All elements in the wireframe are aligned with one or more other elements."*, and (5) `whitespace` - *"Whitespace effectively used to separate unrelated components."* The designers score each rubric item as "Great" (2 points), "Good" (1 point), or "Needs Improvement" (0 points). Appendix A, Section A.3 presents the full rubric, including the concrete criteria for assigning a score for each rubric item. I developed this rubric with the aid of the first designer, who first evaluated a subset of the designs, and we updated the rubric together based on their feedback. Both designers then evaluated the entire set of designs using the updated rubric.

For each design, I computed an overall *layout_quality* score as a weighted sum of the "compelling" (i.e., visual balance (vb), typographical hierarchy (th), and emphasis (e)) and "clean" (i.e., alignment (a) and whitespace (w)) scores, summed across the two designers (d).

$$layout\_quality(L) = \sum_{d \in D} \frac{vb + th + e}{3} + \frac{a + w}{2}$$

Overall, the *Scout* designs had slightly lower *layout_quality* scores (M = 5.37, SD = 1.0, n =

54) than the *Baseline* (M = 5.73, SD = 1.24, n = 54). The layout quality scores were not normally distributed, so I assessed their significance using an aligned-rank-transform analysis [215] which indicated the difference in the mean *layout_quality* score was not significant ($F_{1,87} = 2.35$, $p < 0.13$). Table 3.2 shows the results for individual quality rubric items, suggesting that Scout designs can be improved for visual balance, emphasis, and alignment.

RQ3: How does Scout affect designer processes of exploring potential interface layouts?

I asked designers to reflect on the process they used to explore alternative layouts. After each task, I conducted semi-structured interviews asking designers to describe the strategy they followed to develop diverse, compelling, and clean designs. I then conducted an additional semi-structured interview at the end of the session, asking designers to compare their experiences using each tool to discuss how they might or might not use a tool like Scout in their design process. I include the full set of interview questions in Appendix A, Section A.4. To analyze this data, I collaboratively conducted a qualitative inductive content analysis with another researcher [170] on my notes, with a sensitizing concept of *differences across design processes* when using Scout versus using other tools. Overall, I organized the results into 3 key themes that reflect Scout's impact on *diversity*, *quality*, and *design process*. I also summarize the designers' *envisioned uses of Scout*, and their *suggestions for improvement*.

While I was able to capture audio recordings of most sessions, a malfunction with the recording software led to the loss of audio for about half of the sessions. This meant I could not retrieve direct quotes from those participants' sessions. However, a collaborator on this project compared my notes to audio recordings I did have, and found that the notes were reasonably accurate representations of what participants said, since I had tried to capture participants' statements as accurately as possible. Below, I present the most salient themes from the interviews, supported by direct quotes (in italics) when available, and evidence from my notes if not (in prose format).

Impact on Ideation

Our interviews with designers revealed two main themes on the positive outcomes that Scout had on the designs that designers created in the user study. These themes were that Scout helped designers *come up with new ideas they thought they might not have thought of on their own*, and Scout helped designers *surface new perspectives on conventional design concepts*, and *break out of their personal design style*.

Scout helped designers come up ideas they might not have thought of on their own. Nine designers mentioned that Scout helped them think of new ideas they might not have had on their own. P2 mentioned in the Baseline task they had struggled to come up with three diverse designs:

> *"I feel like I was able to get two really good designs, but the middle one I really don't like. I wasn't able to come up with a third one...I feel like I probably just needed more time..."*

After the Scout task, P2 described Scout helping them create a design that they would not otherwise have come up with:

> *"[T]he way Scout helped with that was, I wasn't even looking to make something like the third layout. I wouldn't have thought to put the image at the bottom of the page, this gigantic one. So I think Scout helped with that."*

Another designer, P16, mentioned that Scout helped them consider combinations of elements that they wouldn't have thought of, leading to a good wireframe.

> *"The first one is pretty similar, but it has a different font size and locations. I wouldn't have thought to use the larger font size with the smaller header. So it's pretty fascinating. I would not have ever made the font so big in comparison to the photo. Its a good wireframe."*

Scout can surface new perspectives on conventional design concepts.
11 designers mentioned that some or all of their Scout layouts were different than a typical weather or social media profile screen. In particular, P10 mentioned that it could be challenging to brainstorm when designing a screen that typically uses conventional layouts, like a weather or social media screen. She mentioned that because there were so many weather apps out

there, it was easy to come up with the normal one, but that it looks kind of boring and normal and like many other weather apps. For the social media profile page that she created without Scout, she had tried to be creative and think of different layouts, but she mentioned that she was *constrained* by her previous experience in terms of a social media profile page.

Some designers reflected on the fact that they had a strong *bias* towards specific styles or patterns of elements. For one designer, Scout helped her in exploring options for low-level layout patterns in the social media profile screen. She was at first adamant that the number should be placed below the "Followers" header and other labels. Scout showed her some other options that had the numbers above the labels, and she expressed that Scout helped her see that as an option. Another designer, P17, reflected that Scout could potentially "*remove the bias*" that comes from getting used to other designers' styles.

Because we asked the designers to create diverse layouts for our study, coming up with new patterns for common layouts was desirable. However, 2 designers noted that breaking design conventions might not always be desirable. For instance, P4 mentioned they preferred to follow typical design conventions, even though they felt constrained by it:

> *"When you look at weather reports on television, right, on the weather channel, there is only so much iteration on the ways they present the information. Because if you try to break the mold, if you try to diversify too much, it only serves to be distracting or confusing for the user...It was a bit of a struggle to diversify the different ideas just because I felt my creativity was constrained by [that]. I prioritized more on usability and familiarity and less on diversity."*

Scout can help designers break out of their personal style.

P9 mentioned that they tend to focus on their own personal style, and thought that Scout could help them break out of that and open their eyes to atypical ways that things could be arranged.

> *"I focus on my personal style. Scout breaks out of that by showing a broad range of ideas. I could generate new ideas and have it show me more. [Scout] shows me more possibility and opens my eyes to how things could be arranged that are atypical. Often times when I design, its based on what I have seen and what I've liked. It [Scout] comes up with its own things."*

P18 explicitly explained that she initially was adamant that she would only want to use the rounded placeholder icon for the social media screen and nothing else. However, because Scout gave her to option to easily look at ideas with other shapes, she was more open to it.

> *"I thought, wow it's square and I don't like it, but because it said diverse and I had the option to easily look at different ideas with different shapes, I was more open to it. If I had done this on my own, I probably would have used the circle and nothing else."*

Impact on Design Diversity & Quality

In the interviews, I asked designers to reflect on which tool workflow helped them come up with more *diverse*, *clean*, and *compelling* designs. Overall, designers viewed the designs they created using Scout to be more diverse, more compelling, but less clean.

Designers viewed Scout layouts as more diverse.

When comparing the sets of designs the designers produced in the *Scout* and *Baseline* conditions, 12 designers thought that the designs they produced with Scout were more diverse, while 4 designers thought they were the same. Four designers thought their *Baseline* designs were more diverse. P2 said her Scout designs were more diverse, stating that some of the elements were portrayed in a way that she didn't think of before. She said she was able to take some aspects of each design and mix them up, and that was helpful. She felt she could build on them.

On the other hand P6 felt that neither her Scout or her Baseline designs were more diverse. She felt that what Scout did was to eliminate some sort of brainpower and just randomize the ways of putting things together, letting her just set the constraints and tell the machine to do it. She felt that between the two workflows, it might just be the process getting to the final result that is different.

When comparing the diversity of his Scout and Baseline designs, P7 reflected on the challenging nature of creating alternatives in the Baseline task, saying he had to imagine the possibilities and contest with his own self-assessment of not being able to meet the goals. His ego got in the way, and he was freed from that in Scout. Looking at the Baseline, he initially had no inspiration

for how to make it different. He self-assessed his own lack of inspiration and thought a better designer would have had more inspiration. This set the tone for his work as he went through creating the alternatives.

Designers viewed Scout layouts as more compelling, interesting, and atypical.

Eight designers felt their Scout designs were more "compelling" than the designs they made in the Baseline task, while 4 designers thought their Baseline designs were more compelling. P13 described that with Scout, it gave her some good ideas she probably wouldn't have thought of herself. She still felt like would need to go to Adobe XD to make it look clean and compelling, but that Scout gave her a foundation to improve upon.

Through these reflections, we found that the designers often interpreted compelling in different ways. To the designers, "compelling" did not necessarily mean having a clear point of entry and clean hierarchy (i.e., as we defined in the task). Five designers interpreted "compelling" as "interesting", or "atypical", like P4:

> P4:*"It does a good job with the compelling thing...The hierarchy is not dull or boring or and to some extent is not even familiar. ... Like this [Scout design], it breaks [design] cliches, that's for sure. It does a good job of not being boring..."*

Designers viewed Scout layouts as less clean.

On the other hand, when comparing their own Baseline and Scout designs, the majority of designers (10 designers) felt they produced "cleaner" designs in the baseline. P17 noted that the Baseline designs were more clean because she had to do a fair amount of cleanup of alignment and other issues on the Scout designs after exporting them to XD. However, she thought that it wasn't particularly difficult to make her Scout designs clean as she could simply import the SVGs into XD, and the cleanup didn't take long. P20 noted that making clean designs within Scout might take more work, by needing to specify more constraints:

> *"You'd have to put a lot of rules on it to get it as clean as you'd want it to be. For example, this [Scout design I made] is not very clean-looking, but I could picture moving it around a bit, and it would be clean...it seemed like more effort to make it clean in Scout than it was in XD."*

Alternately, 5 designers thought their Scout designs were cleaner. P11 said that this was because the Scout designs helped her use whitespace more effectively, and she believed they looked cleaner as a result. She stated that she was not good at intentionally using whitespace, and that the designs she chose from Scout reflected her desire to be a more intentional user of whitespace. She pointed out two designs with different whitespace patterns, and said that she rarely comes up with wild uses of whitespace like Scout did in the early phases of design.

Impact on Design Process

In the interviews, I also had designers reflect on their process for exploring alternatives. I had them reflect on the last interface design they created before the study and discuss the process they followed to create alternative designs. I also had them compare and contrast the approaches they followed to create alternative designs with and without the use of Scout in our user study. These interviews revealed two key themes. The first was that designers *considered the structure of of an interface over the details* more when using Scout. With Scout, they were able to *follow a less linear design process* that contrasted with their reflections on the *Baseline* task and on the process they followed when creating alternatives for the last design they created before the study.

Designers considered the structure of an interface more with Scout.

Scout helped designers in the study consider the high-level structure of a design more than in the Baseline task. When comparing their design processes, more designers mentioned considering emphasis of elements with Scout (13 designers) than without (11 designers). More designers mentioned considering the high-level grouping structure with Scout (14 designers) than without (6 designers). Finally, more designers mentioned considering the order of elements with Scout (7 designers) than without (1 designer).

Of these designers, some explicitly pointed out that Scout helped them have a *more structured process*, *more intention*, and a *better mental model* of the interface before beginning to design. One designer, P17, reflected that Scout gave her a more structured process. She said that in

Scout, she had more intention behind the choices she made. She contrasted that to when working on the Baseline task, she didn't follow a process as much. She described trying out a bunch of things to see what worked.

P10 commented that while she had to spend more initial time setting up the high-level constraints in Scout, she said it was a good way to keep her on the right track. She mentioned spending time early to consider which elements are more important than others, and figuring out the relationships between them. She likened it to "*creating a spec before beginning*". Another designer, P4, likened this to helping develop a better mental model of the interface content before beginning.

> *"Being able to set the emphasis was pretty cool. This helped give me a better mental model of what I should be looking at."*

Designers mentioned a linear design process less with Scout.

When discussing a previous project or their experience with the Baseline task, 12 designers mentioned a linear process of looking at a design and thinking about how to change it into a new design. In contrast, only 2 designers mentioned a linear design process in Scout. When reflecting on their past approach to exploring alternatives, P21 mentioned:

> *"It's something I need to work on. Usually I just end up work on one [idea] and then iterating on that single idea."*

She mentioned that she finds she struggles with not making the design to high fidelity right away. She found it useful to have bunch of different ideas that Scout gave her. P16 also stated that he has used a linear approach to creating alternatives in the past. He mentioned that he makes simple wireframes with groups, and once he has that, he continues to change the layouts and copy/paste elements to create alternatives.

When asked to compare and contrast her process of creating alternatives using Scout and in the *Baseline* task, P2 described that in the *Baseline* task, she sketched out the first layout. For the second and third, she built off the ideas of the first idea, and progressively created different kinds of layouts based on the first one she designed. This was different from the process she

followed in Scout, which consisted of experimenting with setting constraints, and viewing the alternatives Scout generated in response while trying to select diverse alternatives.

P7 hypothesized that Scout could help him break out of a reactionary design approach in the future. When asked how he could see Scout fitting into his design process, P7 mentioned that he would use Scout every day. He saw that a benefit of Scout was that it could *free him from one linear approach*. He mentioned using a "*reaction style approach*" where he follows one path of modifying a design into a new design. They envisioned that in a real-life context, a company could give Scout to individual teams of designers to use as a starting point for brainstorming, enabling them to follow different threads of design alternatives across teams.

Designers Envisioned Uses of Scout

To understand how designers might use Scout in their design process, I had designers reflect on the ways they could envision using Scout in the process of coming up with alternative designs. The themes reflected that the designers would *use Scout during layout ideation*, to *get unstuck* when they were in need of inspiration, and to *quickly visualize and generate alternative ideas*.

Designers would use Scout during layout ideation.

When asked to describe their current ideation strategies, designers mentioned sketching, white-boarding, and looking for examples to come up with new layout ideas. 13 designers mentioned simply placing elements on a design tool's canvas and moving them around to try to generate new ideas. After using Scout, 14 designers said they would use Scout to quickly ideate layouts, or when in need of inspiration. P9 and P18 thought that they would use Scout to help envision layouts physically versus relying on their brain to envision them.

> P18:*"It would be good for seeing physical ideas to think it through and how it fits with what I'm trying to make. Rather than trying to envision something in my head, I'm seeing it here to envision it."*

Nine designers mentioned that they liked how Scout helped them visualize lots of alternate combinations of element layouts. P9 said the whole idea of randomly generating the permuta-

tions of these elements together was useful. This aspect of Scout was an opportunity for P7 to *use an editorial skill set.* By seeing a huge array of alternate arrangements, P7 found it useful to have both both effective an ineffective layouts to choose from.

> *"It stretches your understanding of what's possible. They were wide and broad and messy, and they draw attention to why they don't work... You can look at it more as this is close but we need to change something a little bit to make it better."*

Rather than using Scout for ideating at the beginning of designing, P1 thought she would use it halfway through as a divergent thought process. She thought that she would first make the concept in XD and draw some physical sketches. Once she was clear about which elements she wanted in her design, she would put the elements in Scout to see the alternatives.

When asked to describe how Scout might fit into her design process, P6 mentioned that she typically sketches alternatives, and if she gets stuck, she goes to the internet to look for inspiration in different ways designers use to organize information and preexisting layouts that she can take inspiration from. She stated that Scout could fit into this process by letting her see alternatives that already contained their elements, rather than needing to imagine them based on other examples:

> *"Definitely it helps. Instead of searching on the Internet for alternative layouts or existing things that are out there, [Scout] just makes it easy with what you already have. ... You already see what [the layout] could look like with the information that you have, and not other information."*

Designers would use Scout to get unstuck.

When asked how their approaches to creating alternatives differed with Scout versus the Baseline, 6 designers mentioned that they struggled to think of ideas in the baseline task. P7 felt that compared to the baseline task, thinking of the alternative ideas by himself was *exhausting work*, and that seeing all of the alternatives right there with Scout was exciting.

> *"I did get a big smile when I saw the different alternatives. It was exciting to see all of the ideas right there rather than the exhausting work of trying to come up with these ideas all by*

yourself."

When thinking about the last time she came up with alternatives for a design, P8's process was to *sit and stare at a wall until she thought of something else.* She would try to sketch her ideas on paper, but found that they often didn't look as good as they did in her head. She mentioned it was easy for her to fixate on one idea, and could see herself using Scout *just to get her brain out of her own head.*

P2 mentioned not feeling very successful at creating diverse designs in the Baseline task. She felt she was able to come up with two good designs, but was not able to come up with a third and needed more time to sketch. When asked how she might use Scout in her typical design process, P2 replied:

"Most likely in the initial phase when I'm first designing. I could see Scout replacing sketching for me. I see it as helping start, or if I have a creative block I need to get rid of."

Designers would use Scout to quickly generate alternative ideas.

Five designers mentioned that Scout would enable to them to quickly generate alternate ideas. P18 said that they thought that Scout would be useful *just to generate a bunch of ideas really quickly* and for testing out and visualizing alternate shapes quickly using the alternate groups feature. P4 pointed out that Scout made it quicker to come up with alternatives, compared to the Baseline task:

"It was way quicker for me to come up with these three [Scout designs]. What I struggled with the most on the first [Baseline task] was really brainstorming and ideating, these sort of different variations. Scout made brainstorming a much easier process for me, personally."

P5 described how the Baseline task was more time consuming because of the manual effort required to move elements around to see if they would work.

"It was definitely more time consuming because I wanted to see a bunch of different things upfront, just to see if different concepts would even work...[P5 describes different ways they moved the elements around the screen.] It would have been nice to quickly see that, like, I didn't want every [element] up there [top of screen], I just wanted profile picture, name and title."

Designers Suggestions for Improvement

Finally, my interviews with designers revealed that Scout could be improved in several different ways including by *giving designers more control* over the space of layout alternatives it generates, *scaffolding designers learning of constraints*, and *improving the presentation of feedback options*.

Scout could be improved by giving designers more control

After completing both the *Scout* and *Baseline* tasks, I asked the designers to reflect on what aspects of Scout were useful and not useful in their alternative creation task. Five designers expressed that they would like to have more control over the aspects of variation that Scout explores. P2 typically is very picky about font sizes, so she wanted to be able to set a typography hierarchy or a range of font sizes that Scout would use, as she thought that Scout made too many variations in the font sizes. Controlling font size variations was also a request of P13. P4 suggested allowing initializing Scout's range of alternatives from a company style guide, which typically specific a range of header and label options, and element-specific alternatives.

Four designers mentioned the alternate group constraint as a useful feature of Scout. However, several designers requested more nuanced control. P7 thought that it would be useful to point Scout to a library of alternative icons, and then use the feature to quickly generate alternatives using each icon. P8 and P20 found the alternate group helpful, but would want to be able to toggle the alternate group on a layout to another option so they could quickly visualize the same layout idea with a different alternate.

The desire to specify an emphasis hierarchy was also a common theme among the designers. Eight designers mentioned and used emphasis levels during the study, and six designers mentioned this as a useful aspect of Scout. However, P16 and P18 mentioned they wanted to have more emphasis levels to enable a more fine-grained specification of the visual hierarchy.

Eleven designers mentioned seeing parts of different Scout layouts that they liked, but would change an aspect of the layout or use only part. In our study, the designers were able to refine the Scout designs once they had exported them from Scout, and some designers saved and exported layouts that they liked only part of and then combined the part they liked with the

remaining elements from a layout. P2 enjoyed this aspect of Scout and liked that they were able to pick aspects of different layouts, and *mix and match the different elements of each* that she liked. Several designers mentioned it would help if Scout could make it easier to *mix and match* designs, or explore variations to sub-parts of a design rather than changing the entire layout.

Scout could be improved by scaffolding designers' learning of constraints

Five designers mentioned it was initially confusing to understand how to use each type of group in Scout's high-level constraints outline (i.e., alternate group, repeat group, regular group). P16 noted that he used groups a lot, but struggled to use them initially. He said he initially used a lot of groups because it makes the process really clean and provides a good mental model. With Scout, he found the grouping terminology "tricky" because groups are normally a static thing and do not have other functionality, while in Scout, groups are "smart" with other functionalities and behaviors (e.g., order). Once he learned how each group worked, he was enthusiastic about them. Overall, designers feedback on the challenges of learning how to use the groups suggests that Scout could *better scaffold designers' learning*. Additionally, designers suggested that it would be helpful if group behavior could be changed (i.e., a group to an alternate group) by modifying a property, rather than having to create another type of group.

Scout could present the feedback options better.

Seven designers mentioned confusion over some of Scout's feedback options. P11 mentioned that being able to specify feedback was powerful and helpful, however, she was reluctant to try them sometimes because she felt like she could lose some of the layout ideas and not be able to go back. She suggested that it would be helpful to make applying feedback "feedforward" and provide a preview option to see what would happen before applying the feedback. P21 found all of the feedback options "*a bit overwhelming*", but felt that in any new prototyping tool there are typically a lot of functions as a new user that you really don't learn until you have used the tool for a while. Overall, the designers' feedback on "feedback" suggests that providing tooltips to describe each feedback option, and a preview of feedback feature could be potential ways to help them overcome their initial confusion.

Summary

In summary, my interviews revealed that Scout can aid designers in ideation by helping them think of new ideas, and surface new design concepts outside of common patterns. Overall, while the designers saw their Scout designs as *less clean*, they viewed their Scout designs as *more diverse* and *more compelling*. When examining the processes designers used in our study, the process designers followed using Scout was *less linear.* Rather than iteratively transforming designs into alternatives, Scout can enable them to visualize a number of diverse designs as a starting point. Many of our designers did envision themselves using Scout for ideation, and to quickly visualize many combinations of elements in a layout. They also thought that Scout could help them overcome creative blocks.

Although these findings demonstrate mainly positive outcomes of Scout, designers did struggle with some elements of Scout. Some designers were confused over some of the feedback options, and thought that Scout's high-level grouping constraints were difficult to learn. They gave some useful suggestions for improvement including *better scaffolding of learning* and providing a *preview of feedback behavior*. Designers also wanted to have more control over Scout's generated alternatives by being able to *control the range of variation* (e.g., specify the font sizes to explore), and by being able to *mix and match* different parts of designs to create better combinations.

3.4 Discussion

Scout can enhance designer ideation by helping them rapidly visualize many layouts through mixed-initiative interaction with high-level constraints and feedback on alternatives. An evaluation found Scout can aid designers in creating layout ideas they do not believe they would have otherwise thought of, can help designers avoid developing too early of a focus on a single design, and can help designers consider layouts different from established patterns. Scout designs were also more spatially diverse both within and across designers.

Although not statistically significant, the quality analysis I conducted found Scout designs were awarded slightly lower overall quality scores by expert designers. This suggests opportuni-

ties to improve Scout in terms of balance, emphasis, and alignment. However, participants also had access to the functionality of Adobe XD to refine their designs after Scout ideation (i.e., the same tool used in the baseline). Any difference in quality may therefore be due to a lack of time. Scout required upfront time for specifying elements as well as their grouping, ordering, and emphasis before designers can see layouts, which may have left less time for refinement of designs at the end of the task. Future work could explore integrating capabilities developed in Scout as a feature in an existing design tool (e.g., Adobe XD), such that elements, grouping, ordering, and emphasis could be inferred from an existing layout to generate new alternatives.

Scout points to a new approach to using constraints to support ideation and presents new techniques for providing feedback to systems applying constraint solving. Future systems can explore: (1) formalizations of interface design principles into tools that help designers apply those principles, especially when supporting novice designers, (2) scaling interactive constraint solving to larger interfaces (e.g., webpages), and (3) defining more layout variables and constraints to enable systems like Scout to explore larger and higher-quality spaces of alternatives.

3.5 Contributions

This work was conditionally accepted to CHI 2020 as ***Scout: Rapid Exploration of Interface Layout Alternatives through High-Level Design Constraints*** [202] with co-authors Chenglong Wang, Alannah Oleson, James Fogarty, and Amy Ko. This publication was not yet public at the time of this writing, therefore I do not have the complete reference. In the future, it can be found by searching the above publication title in the ACM DL. A preliminary version of the work was published as a demo at UIST 2018 [200] along with co-authors James Fogarty and Amy Ko. James and Amy helped in developing the initial ideas for Scout, and gave valuable guidance and feedback throughout the project. I developed the Scout interface & system, along with encoding of design principles into formalized constraints. I also conducted the Scout evaluations, and quantitative analyses in the paper. Alannah Oleson and I collaborated in evaluation planning, developing the set of qualitative interview questions for the evaluation,

and in qualitative coding of the Scout interviews. Alannah transcribed quotes and drafted the qualitative evaluation section for the Scout paper. Chenglong Wang and I collaborated to develop the Scout cost model for ranking layout ideas, and the spatial diversity score used in the Scout evaluation, and experimentation with the Scout synthesis algorithms. Once published in the CHI proceedings, you will be able to find a demo video for Scout on the publication page under "Source Materials".

Chapter 4

Interface Design Assistance from Examples

Examples play a critical role in the interface design process. During ideation, designers browse and curate example galleries for inspiration [47,87] and explore alternative solutions [67, 87]. During prototyping, designers frequently use examples as building blocks when developing new designs [120]. Designers use examples throughout the design process, and they are often pivotal to the success of projects [87].

During *prototyping*, designers create prototypes in tools like Sketch [19] or Adobe XD [98], demonstrating the visual design and layout of one or more interface screens in detail. In this process, designers may want to reuse or edit parts of example designs they have collected. For instance, a designer may want to edit the colors of interface elements to explore different palettes, change text labels based on the target application, or reuse part of an example interface in a new design. Performing such operations requires example designs to be represented in an editable way, predominantly as vector graphics where interface elements are specified as objects with properties that define their shape and appearance.

While vector representations enable reuse and editing, example designs are frequently not

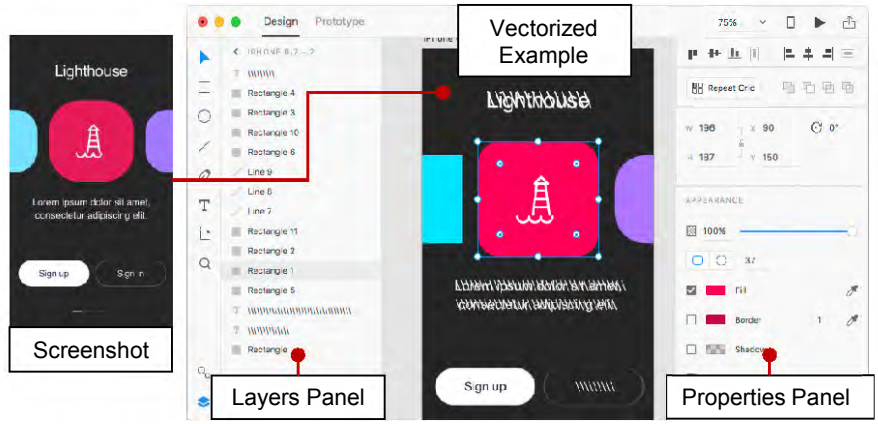


Figure 4.1: Rewire's Full Vector design assistance mode, in the Adobe Experience Design (XD) canvas. Designers activate the mode by right-clicking on a screenshot, that they drag into an artboard in the design document. Designers can then edit the properties and layering of the vectorized output in XD's Properties and Layers panels.

shared in this form. Designers may also find examples in real interfaces, where they cannot access a vector representation, and may want to quickly take a screenshot of the example. Thus designers frequently collect example designs in the form of screenshots (i.e., raster images). Most devices have shortcuts to copy portions of the screen to an image, which makes it easy for designers to capture examples. At the same time, unlike vector graphics, screenshots are flat, unstructured, and hard to edit. As a result, when a designer wants to modify a design from an image, they need to reconstruct all or relevant parts of the content to produce an editable vector representation. This representation must be created by hand by drawing and specifying the properties of shapes through trial-and-error.

Designers can use commercial vectorization tools, like Illustrator's ImageTrace [1], shown in Figure 4.2, to ease this process. However, because these tools aim for visual fidelity, they represent their output with path objects representing visually distinct boundaries in an image. With paths, changing properties like rectangle corner radius requires editing many individual control points to modify the relevant boundaries. Changing font properties of text represented as a path is impossible without creating a text box.

[1] https://www.adobe.com/products/illustrator.html

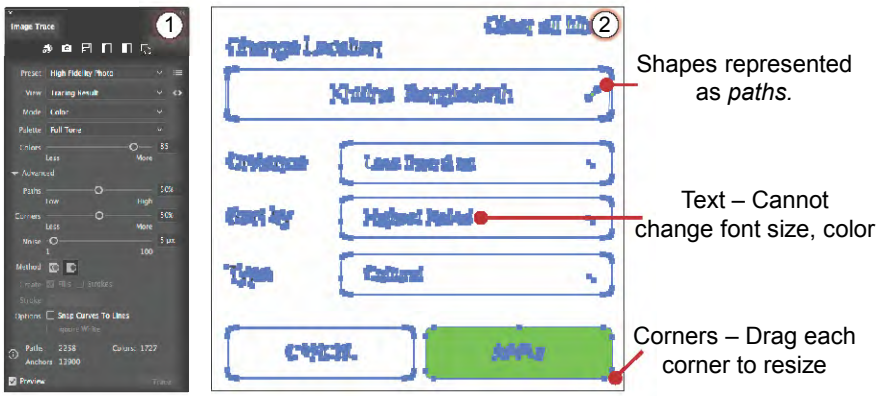


Figure 4.2: Commercial vectorization tools, like Illustrator's ImageTrace, require a designer to specify a complex number of vectorization options (1). They represent their output with path objects (2) representing boundaries in an image. A designer cannot change font size and color because there isn't text box to edit. To adjust a rectangle's corner radius, they need to drag and resize each individual corner.

In this chapter, I present Rewire, a system that helps designers reuse example screenshots. Rewire infers a semantic vector representation from a screenshot where each interface element is an editable object with shape and style properties. With Rewire, I aim to infer higher-level semantic objects (e.g., rectangles, text boxes) with properties of common interface elements (e.g., corner radius, font type), rather than the vector paths as inferred by common features in commercial tools. Rewire applies *semantic analysis* using image analysis techniques to uncover semantic structure and properties from an interface screenshot. Through Rewire, I demonstrate that if we augment interface design prototyping tools with the capability to infer semantic structure from example screenshots, we can make it easier for designers to transform and adapt examples into another design. We can thus enable designers to ideate and prototype from example screenshots more efficiently.

In addition to inferring a vector representation of an example screenshot, Rewire provides three design assistance modes from these inferred vectors. These modes can aid designers through different goals in example adaptation. *Full Vector*, shown in Figure 4.1 and 4.3 (a), creates an artboard with a set of vector shapes inferred from a screenshot. This mode allows designers to quickly reconstruct or edit the example design. *Smart-Snap*, shown in Figure 4.3

(b), provides snapping guides that become active when drawing new shapes. These guides can assist designers in aligning new interface shapes with those in the example design. *Wireframe*, shown in Figure 4.3 (c), displays the inferred vector shapes with a simple black outline. This mode helps designers reuse the layout of the example design while abstracting away visual details. The contributions of this chapter include the following:

1. An automatic method for extracting semantic vector objects from screenshots that combines low-level image processing with UI-specific reverse engineering techniques.
2. Three new design assistance modes that leverage the extracted vector objects to help designers create new designs.
3. Quantitative and qualitative evaluations demonstrating the accuracy of Rewire's pipeline and the benefits of Rewire's design assistance modes in helping designers adapt and transform example designs.

4.1 Formative Interviews

I conducted formative interviews with 10 user interface designers working at both small and large companies to discover more about common use cases for Rewire. All designers frequently used screenshots in their work. In one extreme case, one designer recreated an entire legacy interface design to use as a template, spending days drawing a complex vectorized design document from a screenshot. However, most designers said that they mostly recreate only parts of a design they need to change for making quick mockups. Sometimes they will simply cut, paste, and resize parts of screenshots into their designs for quick prototyping, and then recreate interface shapes when moving to high-fidelity. Designers also mentioned recreating from screenshots when the original design assets were lost and when clients sent them interface screenshots to incorporate into their designs.

4.2 Motivating Example

From the use cases I observed in my formative interviews, I developed three design assistance modes that help designers leverage interface screenshots. I describe and motivate these modes in the context of an example scenario. In the scenario, Maria, a user experience designer,

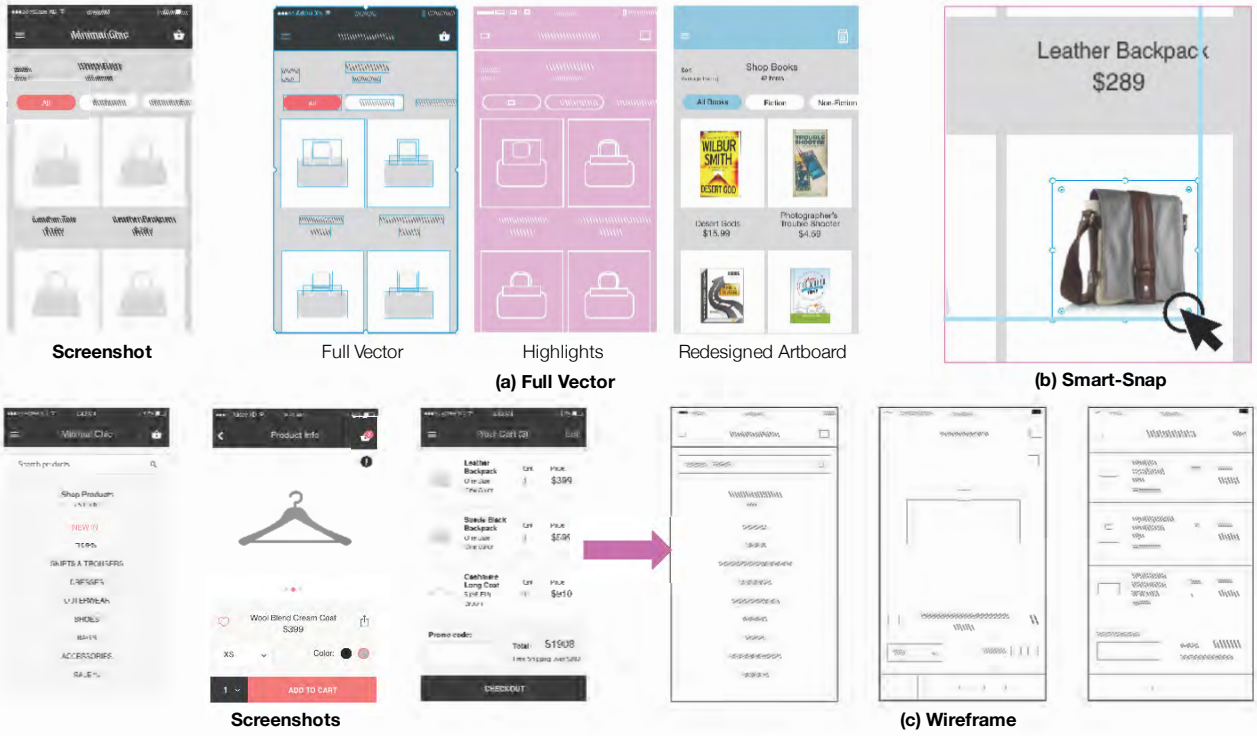


Figure 4.3: Rewire provides three modes of design assistance. Full Vector (a) creates vector objects for shapes in the image. Designers can highlight the vector objects by toggling the pink Highlights layer. Designers can then update and redesign the vectorized artboard, as shown on the right. Smart-Snap (b), displays alignment and spacing guides to help designers align newly drawn shapes to shapes in the screenshot. Wireframe (c), generates abstract wireframes of the screenshot, removing most visual details.

performs several design tasks using an existing vector-based design tool, Adobe Experience Design (XD), that has been augmented with the Rewire design assistance modes.

Maria is creating a mockup for a shopping cart page. Her project manager sends her the screenshot shown in Figure 4.3 and asks for a similar design with realistic bag images in place of the grey bag icons. To accomplish the task, Maria opens the screenshot in XD and activates Rewire's Smart-Snap (Figure 4.3, (b)) mode. As she drags an image of a leather purse onto the canvas, blue snapping guides visualize how the image aligns with the interface shapes in the example design. These guides enable Maria to quickly align and resize four realistic bag images over the original bag icons, without having to carefully manipulate the size and position of each image.

Maria's next assignment is to modify the same shopping cart page to show a set of sample books rather than bags, with book titles and prices below each item. The project manager wants Maria to create several variations of the design with different color schemes. Instead of redrawing, retyping, and matching the properties of the screenshot by hand, Maria activates the Full Vector mode to automatically generate vectorized objects from the screenshot (Figure 4.3 (a), Full Vector). To see the generated objects, Maria enables Rewire's highlighting feature as shown in Figure 4.3 ((a), Highlights). Rewire renders shapes (e.g., rectangles, circles, lines) in pink with white outlines, and text objects with white backslashes to indicate they are editable. Maria edits the text for each book and modifies the header and button colors to create several design variations. In this setting, the Full Vector mode helps Maria to quickly create designs using the elements in the original screenshot.

Finally, Maria's manager sends her several screenshots of inspirational examples and asks her to show the client a range of potential designs based on these images. Since the goal is to present high-level ideas, Maria wants to show abstracted versions of the example designs that leave out unnecessary (and possibly distracting) design details like the specific fonts or icons. Maria drags the screenshots into XD and activates Rewire's Wireframe mode, shown in Figure 4.3 (c), to automatically create wireframe representations. Rewire draws these with a simple black outline with no additional styling. Maria then labels the elements of the wireframe to highlight key parts of the app such as the header and shopping cart items.

4.3 Architecture & Implementation

I assume the input to Rewire to be an image of an interface. Because interfaces consist of an array of geometric shapes and natural images, contain complex hierarchies, and contain a large set of properties that frequently interact, Rewire focuses on detecting and vectorizing four primitive shape types: rectangles, circles, lines, and text. These shapes can be combined or used individually to represent most interface elements. The output of Rewire's processing pipeline is a vectorized artboard containing editable shapes, as shown in Figure 4.4 (Output). These shapes contain styling (e.g. corner radius) and size properties. Rewire populates these

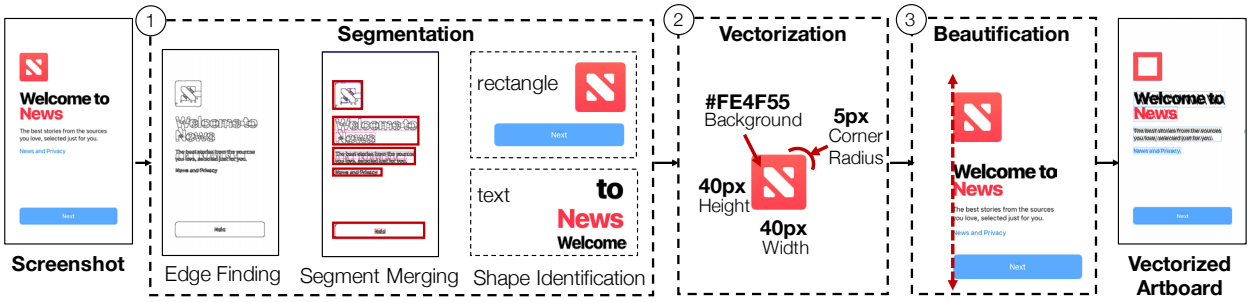


Figure 4.4: System overview of Rewire. The system input is a screenshot. Rewire segments shapes from the image and classifies them by primitive shape type (1), extracts properties of segments to create vector shapes (2), and beautifies (i.e., aligns & normalizes) the resulting layout (3).

shapes into a vector-drawing tool, where designers can edit, resize, or move them.

To support the *Smart-Snap*, *Full Vector* and *Wireframe* design assistance modes, each vectorized shape is presented as demonstrated in Figure 4.3. The screenshot processing pipeline consists of three stages, as illustrated in Figure 4.4:

1. ***Segmentation***. First, Rewire segments the screenshot into regions of distinct geometric elements by leveraging existing low-level computer vision algorithms (Figure 4.4, Stage 1). Rewire also classifies segments into a predefined set of primitives (e.g. rectangles, lines, circles, text).
2. ***Vectorization***. For each segment, Rewire generates a corresponding vector object by estimating the relevant shape (e.g., size, position) and style (e.g., color, border thickness) properties (Figure 4.4, Stage 2).
3. ***Beautification***. Finally, Rewire refines the properties of individual vector objects via a global optimization to improve alignment and consistency across the whole artboard (Figure 4.4, Stage 3).

Each design assistance mode uses the vectorized output in different ways. The *Full Vector* mode presents the fully vectorized artboard to the designer. *Wireframe* mode removes all the style properties and only preserves the shape of each object. Finally, the *Smart-Snap* mode creates snapping guidelines based on the bounding box of each object but does not add the objects to the artboard. I implemented each design assistance mode as an extension to Adobe XD.

4.3.1 Segmentation

The segmentation phase of Rewire's processing pipeline consists of two sub-phases which are *text box* detection, and *shape* detection.

Text Box Detection

Rewire extracts primitive shapes from the input screenshot in two steps. First, Rewire detects all text regions via OCR and then segments the remaining parts of the screenshot. Identifying the text up front improves the quality of the subsequent segmentation because it allows the algorithm to filter out extraneous small segments that often arise within text regions.

For text detection, Rewire uses an OCR library called Tesseract [193] to obtain bounding boxes that correspond to potential text shapes. To optimize Tesseract's performance, Rewire sets the page segmentation mode to *sparse text* which tries to find as much text as possible, in no particular order. With these parameters, Tesseract outputs individual lines of text.

To filter out obvious false positives, Rewire computes two geometric properties, *solidity* and *extent*, for the pixels contained within each text word. Prior work found these to be good at discriminating between text and non-text regions [77]. The system removes any text words with solidity greater than 0.3 and extent > 0.9, which I based on previous techniques [77, 124] and experimentation with the example dataset. Finally, to create a concise and easily editable set of text segments, Rewire merges adjacent text lines that are close to each other and similar in color using the Golden Ratio Φ (1/1.618), a typography ratio that relates font size, line height, and line width in an aesthetically pleasing way. Rewire merges lines if the vertical distance between their bounding boxes is less than the Golden Ratio Φ times the mean line height and the correlation between their color histograms (measured via Pearson's coefficient) is greater than 95%. I set these thresholds empirically and use them for all of Rewire's results and experiments.

Shape Detection

To segment the remaining parts of the screenshot, Rewire decomposes the image into an over-complete set of candidate segments and then iteratively merges and classifies segments to obtain a final set of shapes. Rewire computes candidate segments by constructing an Ultra Metric Contour Map (UCM) [27], which uses low-level image features to partition the image into a set of closed regions. Because Rewire has already detected text regions, it removes UCM region boundaries that overlap with any of the extracted text, and use the remaining segments as its initial set of candidates.

Given the candidate segments, Rewire iteratively merges or removes segments until it has attempted to merge all segments. The first step is to put all candidate segments into a working set S. For each segment $s \in S$, Rewire determines whether it is one of the primitive shapes (i.e. circle, rectangle, line) that Rewire handles. To detect rectangles and lines, Rewire counts the non-segment pixels within the axis-aligned bounding box of s. If there are no non-segment pixels and the height or width of the smaller dimension is less than 5px, Rewire classifies the s as a line. This threshold of 5px was set by a manual exploration of common patterns in hand-created designs. If s is not a line but the fraction of non-segment pixels is less than 90%, Rewire classifies it as a rectangle. Finally, if the s is not a line or a rectangle, Rewire computes a circle Hough transform [33] to check whether it is circular.

If the segment s is classified as one of these shapes and its larger bounding box dimension is bigger than 20px, then Rewire removes it from the working set and add it to the final set of segments. Otherwise, Rewire tries to merge s with its adjacent segments. If s is a line, it tries to merge it with any adjacent co-linear line segments. If s is not a line, it merges it with adjacent segment t if they are a similar size (i.e., neither segment is more than three times larger than the other) or both s and t are small (i.e., have a width or height less than 5px). If s is merged with any segments, Rewire puts the resulting segment back in the working set S. If s is not merged with any segments, Rewire adds it to the final set of segments. Eventually, all segments are removed from the working set.

When the working set is empty, Rewire does one last clean-up pass, and removes any segments that are not primitive shapes and smaller than 25px in area. I discovered through experimentation that these segments frequently correspond to noise produced by the UCM segmentation. For each rectangle, Rewire removes any lines or nested rectangles that are within 2px of the rectangle boundaries, since these extraneous segments typically correspond to styled rectangle borders. Finally, for every primitive shape, Rewire removes neighboring segments that are less than one tenth the size of the shape because they are likely to be edge segments from border effects or shadowing around the detected segments. The final output of the segmentation is a set of segments labeled with a primitive shape type. Rewire leaves segments not classified as a primitive shape unlabeled.

4.3.2 Vectorization

Rewire generates vector objects from the set of segments. In this phase, text segments become text objects, line segments become lines, circle segments generate circles, and unlabeled and rectangle segments become rectangles. For each segment, the position of the its bounding box determines the position of the corresponding vector object. To ensure shapes are correctly layered, Rewire builds a partial hierarchy of interface shapes based on visual containment. Rewire does not support vectorizing icons or complex vector graphics. Based on formative interviews, there are likely few cases where designers would want to reconstruct an entire logo from a screenshot. Instead, they typically want to abstract it away or replace it with a different icon or logo. Thus, Rewire's aim is to reconstruct whole interface elements, which can be represented mainly with primitive shapes. Rewire computes the vector properties via the following segment-specific vectorization procedures.

Text

For each text segment, Rewire generates a text object and estimates the `baseline`, `font size` and `color`. For text segments containing more than one line of text, Rewire also estimates the

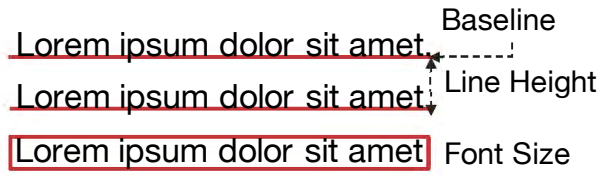


Figure 4.5: Rewire extracts the baseline, line height, and font size of text shapes.

line height property (see Figure 4.5).

To estimate the `baseline`, Rewire converts the region of the original image inside the bounding box of the text area into an edge detected image which contains only white and black pixels on edge boundaries. Rewire analyzes a y-coordinate distribution of the white pixels in the region and sets the `baseline` as the y-coordinate of the largest group with the highest y-coordinate.

For text areas with more than one line, Rewire estimates the `line height` property by computing distances between adjacent baselines in a text area and by computing an average between adjacent baselines.

To estimate the `font size` for text shapes, Rewire uses the bounding box height of the tallest text line in the text area, as shown in Figure 4.5, which directly converts into a fixed pixel value. Typography defines the font size as the distance between the highest ascender line (i.e., the capital letter L in Figure 4.5) and the lowest descender line (i.e., the bottom of the p in Figure 4.5) possible in a line of text. This means that Rewire's font size estimate will be less accurate if the text line does not have ascender and descender lines. To address this, Rewire normalizes font size estimates during the beautification stage (Stage 3 in Figure 4.4), and snaps font size estimates upward toward similarly sized text shapes in the document.

To extract `text color`, Rewire first finds the background color of the text by computing a histogram of all pixel colors found at the boundary of the text box and merging them into groups of indistinguishable colors using the Delta-E metric [187]. The finds the foreground color by clustering pixels in the foreground and computing a weighted average between the number of pixels in a group and the amount of contrast with the background, setting the font

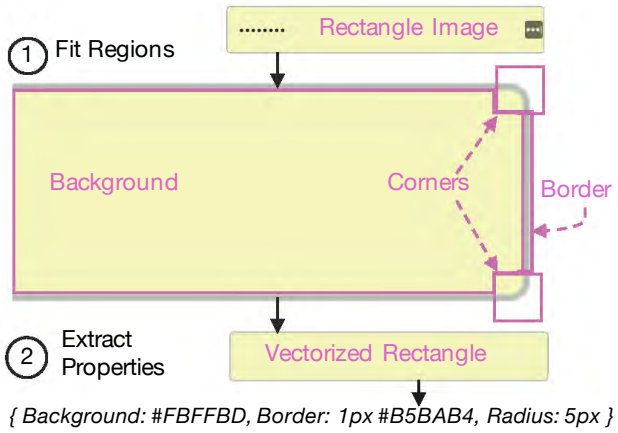


Figure 4.6: The original bitmap, and the Prefab extracted segments. Prefab discovered the background color, border color, border thickness, and corner radius by segmenting the image bitmap into 6 regions.

color to be the color of the group with the highest weighted average.

Rectangles & Lines

For rectangle segments, Rewire generates a rectangle object and estimates the `background color`, `border color`, `border thickness`, and `corner radius`. To enable this, I created a new rectangle model in Prefab [62], a system for reverse engineering the pixels of graphical user interfaces. Prefab recognizes widget shapes in an image by fitting the pixels of an image to nine sub-regions: the interior (background), four borders, and four corners. For Rewire, I created a simplified six-region Prefab model that only includes a single corner region. This makes the computation more efficient but restricts the system to generating rectangles with a single fixed corner radius (which is by far the common case).

Prefab's output is the size of each region, and the colors of the longest repeating patterns discovered in the border and background regions. From this, the model can infer the `corner radius` (i.e., width/height of the corner region) and `border thickness` (i.e., width/height of border regions). For `background color` and `border color`, if Prefab finds a single solid color, Rewire returns this color. If a solid color is not found, Rewire returns the most common color from this region. If the extracted border regions and background regions have the same color,

Rewire collapses them and returns the `background color` and `corner radius` and does not return a `border`. Note that Rewire does not currently extract gradients or patterned fills for rectangles. Figure 4.6 shows an example image and the segments that Prefab infers through this process. Prefab first segments the image into the 6 regions (Figure 4.6.1), and infers the rectangles vectorized properties which in this case are `{background: yellow, border: 1px gray, corner_radius: 5px}`.

Rewire uses a six-region Prefab model for vertical and horizontal lines but allows the size of the side regions and corner regions to be zero. For lines, Rewire sets the `background color` and `line thickness` to the size and color of the background region of the Prefab model.

Circles

For circles, Rewire finds the `radius` based on the dimensions of the segment bounding box. It extracts the `background color` by clustering the pixels into foreground and background regions, and selecting the most common `background color`. Rewire does not currently extract `border color` and `thickness` for circles. However, I believe these could be extracted similarly to rectangles using a parametric Prefab [62] model.

Non-text, non-primitive segment

Rewire generates a rectangle shape that aligns with the segment bounding box and clips out the corresponding screenshot pixels. Although designers cannot easily edit the contents of such clipping regions, they can repurpose them by copying, scaling, and rearranging the clipped pixels. Figure 4.1 shows a lighthouse icon shape that can be rescaled and moved.

4.3.3 Layout Beautification

The first two stages of Rewire's computer vision processing pipeline can introduce small misalignments and inaccuracies in the properties of generated vectors. In particular, inexact edge detection (e.g., due to anti-aliasing or border effects) during segmentation can propagate to the

vectorized objects. Such inaccuracies impact the quality of the design by creating misalignment between objects or inconsistencies across related elements (e.g., text objects with slightly different font sizes). This can violate the design principles *simplicity*, *consistency*, and *alignment*. I provide detailed definitions for these principles in Chapter 2, Section 2.1.

One way to simplify an interface is to *reduce visual clutter* [177]. As Cooper notes in *About Face* [56], if the spacing between consecutive elements is nearly the same size, make it the same. Additionally, if the font size of neighboring fonts is nearly the same size, make it the same [56]. These unnecessary variations are also related to the *consistency* principle, which recommends to make elements have a consistent style and appearance [126]. *Alignments* line up elements along common rows, columns or centers [126]. To ensure that Rewire's vectorized designs do not violate these principles, Rewire adjusts shape sizes, positions, and properties (e.g., font size, line height), and produces a new set of shape sizes, positions, and properties that better follow these design principles.

Rewire repairs *alignment* by correcting small misalignments (e.g., less than 2px) between the boundaries and centers of shapes or baselines for text objects. Rewire ensures *simplicity & consistency* by checking for distribution relationships (i.e., nearly uniform gaps between neighboring aligned elements) and making them uniform. Rewire also ensures *simplicity & consistency* by repairing small differences in font size, line height, and baseline of pairs of text objects. To make these repairs, Rewire formulates the problem as a *constrained optimization problem*, defined in Chapter 2, Section 2.2, with the following soft and hard constraints.

Soft Constraints

Rewire uses soft constraints to discourage small differences in the size, alignment, distribution, and text properties of objects. For every pair of non-text vector objects, Rewire checks whether the widths or heights of the bounding boxes match within a small threshold and if so, adds a constraint penalizing the difference in the relevant dimension. For every pair of non-text objects, Rewire penalizes small misalignments between the boundaries (top, bottom, left, right) and centers (horizontal, vertical) of the bounding boxes. Similarly, for every pair of text vector

objects, Rewire penalizes misalignments along the vertical center and baseline axis, but does not add constraints for the other boundaries because it only makes sense to align text boxes along their baselines. For text/non-text pairs, Rewire adds constraints to align the bottom of the bounding box of the non-text shape to the baseline of the text shape.

In addition, for groups of three or more objects that are approximately aligned along the same axis, Rewire checks for potential distribution relationships between the shapes (i.e., when the gaps between adjacent shapes are nearly uniform) and if so, add constraints that penalize discrepancies. Finally, for every pair of text objects, Rewire penalizes small differences between the baseline, line height, and font size. The assumption is that the original designer manually aligned the shapes in the original design, so small misalignments have likely appeared due to small inaccuracies of the screenshot processing pipeline. Rewire uses a threshold of 2 pixels to determine when to apply the non-text (size, alignment, distribution) constraints and a 1 pixel threshold for the text constraints.

Hard Constraints

To prevent the optimization from introducing new artifacts or transforming objects too drastically, Rewire imposes two types of hard constraints. First, it sets a hard limit on how much any shape or text property can change during the optimization. Through experimentation, I found a threshold of 2 pixels to work well. Second, Rewire constrains every object to stay contained within the bounds of its parent object, if it has one. Unlike the soft constraints, these hard constraints are guaranteed not to conflict.

Optimization

Rewire combines these constraints into a cost function which tries to maximize the number of soft constraints satisfied and minimize the distance the movement of shapes from their original locations. Given the set of soft and hard constraints and the cost function, Rewire uses Z3 [57], described in Chapter 2, Section 2.2, to obtain a solution through the Z3Py library's optimize

	# Elements	Rect.	Circ.	Line	Text	Other
Mean	27.5	7.6	1.8	0.9	10.8	0.6
Median	25.5	6	1	0	9.5	0
Min	7	0	0	0	2	0
Max	55	31	10	6	23	6

Table 4.1: The summary statistics for the total and number of elements of each type per artboard included in Rewire's technical evaluation dataset.

solver, which enables solving using a cost function and weighted soft constraints.

4.4 Technical Evaluation

To evaluate the accuracy of Rewire's screenshot processing pipeline, and to understand the challenges in of using Rewire in the wild, I collected a dataset of interface designs in the form of vector drawings, ran Rewire's pipeline on screenshot images of each drawing, and compared the vectorized output from Rewire to the ground truth vector representation. Here, I describe my process for creating the evaluation dataset and the evaluation metrics I used to measure Rewire's performance.

4.4.1 Dataset

To obtain a representative collection of user interface designs, I collected vectorized design documents from popular online design sharing galleries, including Dribbble [2] and DesignerMill [3]. I restricted my search to Adobe XD design files so that I could view all the vectorized designs with a single tool. In addition, to keep the evaluation dataset self-consistent, I only considered mobile interface designs, which accounted for 39% of the design files. Finally, since Rewire is not yet designed to identify or segment natural images, I filtered out designs with large background images that cover more than 80% of the artboard. I also removed any documents that contained only UI kits, which are large collections of vectorized widgets that typically do

[2] https://dribbble.com/

[3] https://www.designermill.com/

not contain any interface designs. Using these rules, I downloaded (on June 20, 2017) a total of 88 XD files containing 203 mobile design artboards across 6 websites.

While these designs were representative, they did not have the appropriate vector structure to use directly as ground truth. Many designs include vector icons or logos that consist of many grouped geometric objects. Since the primary goal of Rewire's pipeline is to reconstruct whole interface elements rather than their visual parts, it did not make sense to treat individual objects within icon or logo groups as part of the ground truth vector representation. However, the naming and granularity of such groups was not consistent, which made it hard to automatically extract the ideal vector structure from each design. In addition, some designs included objects hidden behind other parts of the drawing. Such objects are likely artifacts of the design process designers left behind by accident.

To resolve these issues, I randomly selected a single design artboard from each of the 31 XD design files in the dataset and manually edited its vector structure. Specifically, I removed any hidden objects and created specially named groups for sets of geometric primitives that form icons or logos. In all cases, icons and logos were easy to identify, and Figure 4.8 shows some examples (e.g. flower, car). After cleanup, I ended up with an evaluation dataset of 31 ground truth vectorized design artboards, with a median of 25 vector objects (see Table 4.1 for statistics).

4.4.2 Evaluation Method

For the evaluation, I compare Rewire's Full Vector output to the ground truth images. I evaluate the accuracy of Smart-Snap or Wireframe modes in combination with Full Vector mode because they use the same object boundaries as the Full Vector output. I compute *precision*, *recall*, and *f-score* for two different evaluation metrics: *type detection* and *property accuracy*.

For *type detection*, I first find corresponding shapes between the Rewire output and ground truth. To measure precision, I consider each Rewire object, compute the standard intersection-over-union (IoU) score for all ground truth objects, and I select the one with the highest IoU as the match. If the types of the two matched objects are the same, I count a hit. Otherwise, I

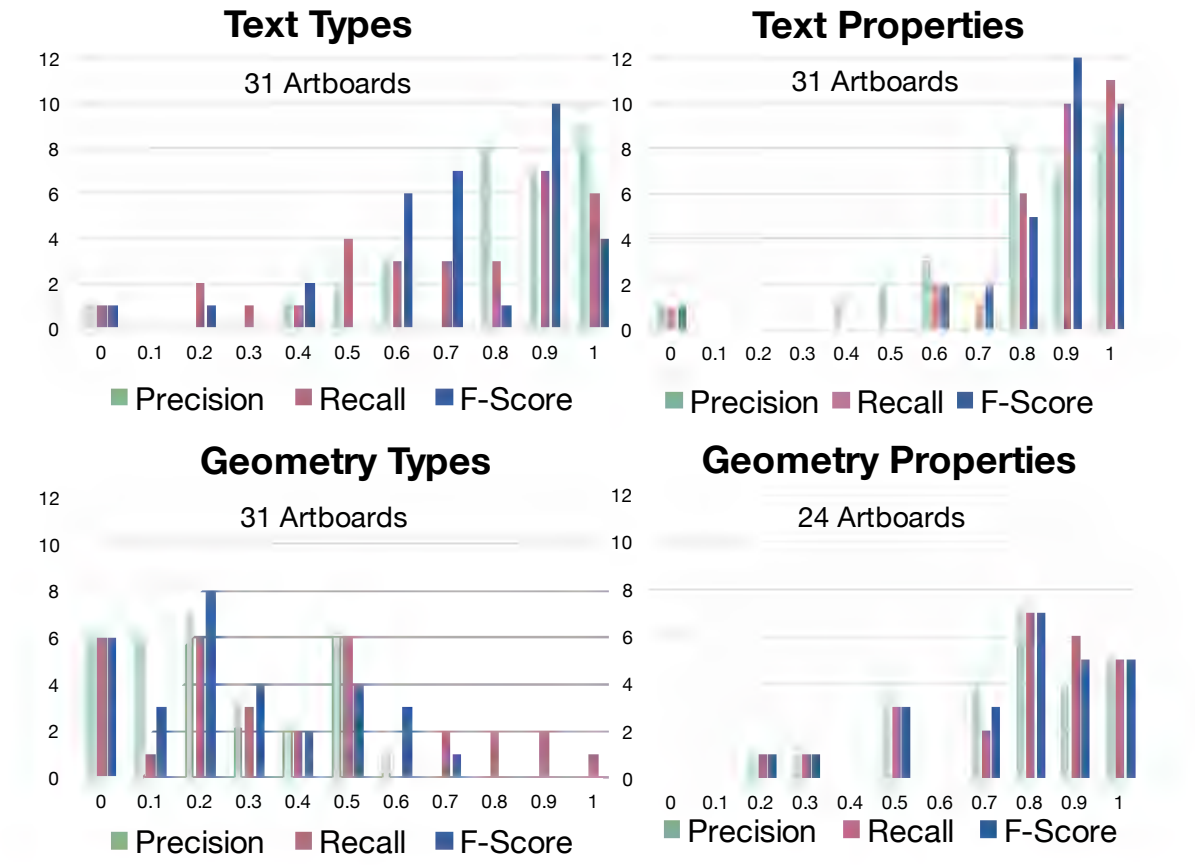


Figure 4.7: Histograms of Rewire's f-score, precision, and accuracy on the dataset of real artboards collected from popular design sharing galleries. The height of each bar represents the amount of artboards at that accuracy level.

count a miss. To measure recall, I perform the inverse procedure starting with each ground truth object. This metric describes the accuracy of Rewire's segmentation and shape identification.

To measure the *property accuracy*, I consider each matching pair of Rewire and ground truth objects that have the same type. I check if the objects overlap enough (90% for rectangles and circles, and any amount for lines and text), and for lines I check if they are colinear. For each pair that meets these requirements, test for matching property values using a 2 pixel threshold for pixel-based properties and the Delta-E metric [187] for color similarity with a threshold of 1. I measure property accuracy as the number of properties within this threshold across the artboard, and I report these results for Text and Geometry (i.e., rectangles, circles, lines).

4.4.3 Results

Figure 4.7 shows the distribution of precision, recall, and f-scores for Text and Geometry for the *type detection* and *property accuracy* metrics. The Text Types histogram shows Rewire is able to identify most text shapes, and for both Geometry and Text objects, Rewire is able to successfully extract most properties. For Text Properties, 27 out of 31 artboards have f-scores over 70%, and for Geometry Properties 17 out of 24 artboards have f-scores over 70% accuracy. Note that the Geometry Properties histogram does not include the 7 artboards where Rewire did not match any of the Geometry Types well enough to extract their properties.

In general, type detection is harder than property extraction because it requires the initial segmentation to be correct. Moreover, identifying the type for Geometry objects is challenging for three key reasons. *Natural images* result in many extra segments, as shown by the low precision scores in the Geometry Types histogram. If I remove the 12 artboards with natural images from the evaluation dataset, 60% of the remaining artboards have Geometry Type f-scores above 50%. Additionally, *small objects* are sometimes mistaken for noise and filtered out by Rewire's segmentation. For example, 11 artboards in the evaluation dataset have standard mobile header bars with small elements. Finally, many designs use *alternate representations* for interface elements. For example, designers sometimes use closed paths to draw rectangles and circles, while Rewire treats all geometric shapes as primitives. Layering relationships can also be ambiguous. For the designs in Figure 4.3, the grey background layer extends beneath the black header rectangle. Rewire extracts two adjacent rectangles. In many cases, Rewire's output may be equivalent in terms of utility and editability.

Extracting more accurate Geometry objects is the biggest opportunity for improvement in Rewire's processing pipeline. As I discuss later, there are many directions for future work to address the current limitations. Yet, as I demonstrate in my user study of Rewire, extracting even a subset of the interface shapes in a screenshot can have practical benefits for design tasks.

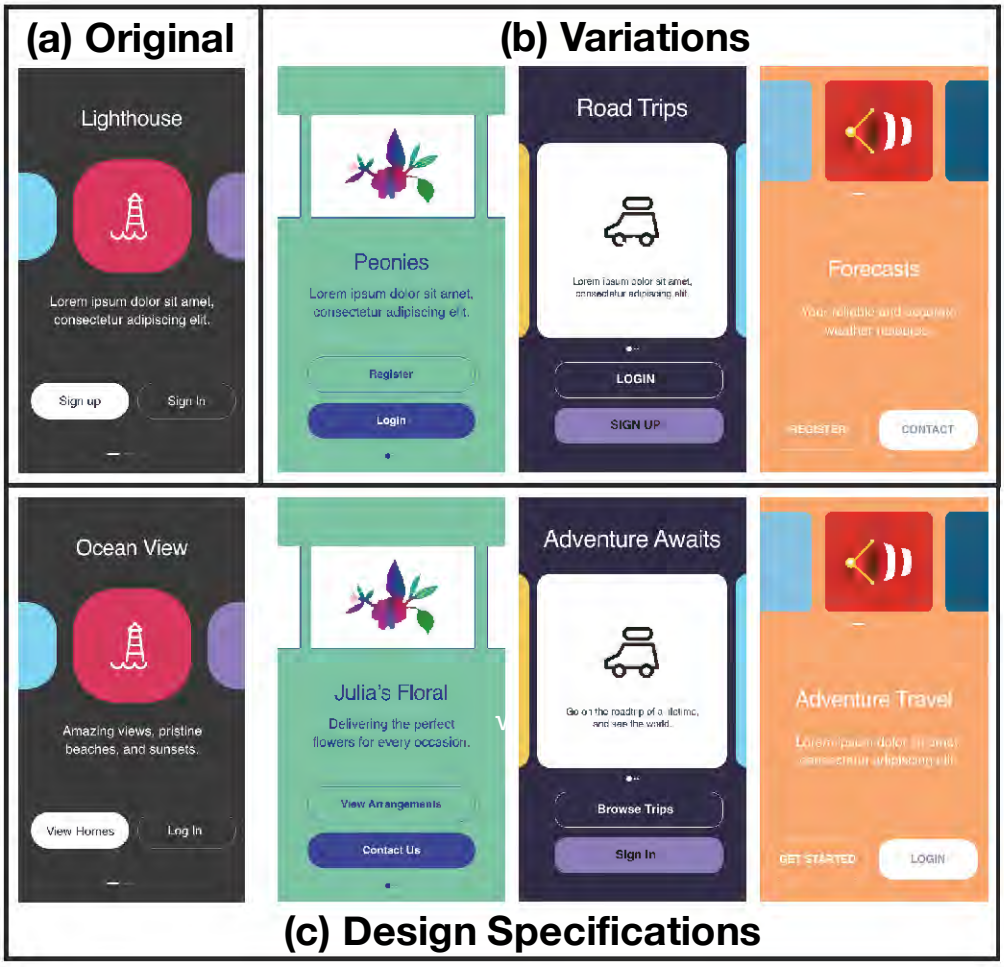


Figure 4.8: The original screenshot (a), variations (b), and design specifications that designers recreated for the Rewire user study (c).

4.5 User Study

To understand more about the benefits and limitations of Rewire's design assistance modes, I conducted a user study with 16 professional interface designers. I investigated the following research questions:

RQ1: Do Rewire's modes of design assistance improve the *accuracy* and *efficiency* of designers when creating a vector graphics design from an example screenshot?

RQ2: What aspects of each design assistance mode do designers like and dislike?

4.5.1 Participants

For the study, I recruited 16 former or current user interface designers (6M, 9F) between the ages of 21 and 50, all of whom had at least one year of professional experience. All participants completed four design tasks over the course of 1 hour using a version of Adobe Experience Design (XD) augmented with Rewire's design assistance modes, shown in Figure 4.1, running on a MacBook Pro (OSX Sierra). While nine participants had little or no previous exposure to XD, they used very similar vector design tools like Bohemian Sketch and Axure in their daily work. The other six designers had at least one year of experience using XD.

4.5.2 Procedure

To evaluate RQ1, I used a within-subjects design with four conditions. Each condition included one task in which I asked participants to produce a vector representation of all of the interface elements of an interface (see Figure 4.8). Rather than asking them to produce an exact vector replica of the screenshot, I made versions of the designs with different text (see Figure 4.8b) and gave them the target designs on paper. I also gave them printed task instructions for reference. To reduce the overall study time, I told the designers to use the default font for all text, even if it didn't match the screenshot. However, I did ask them to match other text properties (e.g., color, size) to preserve the overall appearance and layout of the design. I did not want the designers to vectorize icons, such as the lighthouse shown in Figure 4.8 (a), as it could take too much time. Thus, I instructed designers to use screenshots for all icons. Within XD, I provided the input screenshot in one artboard and asked designers to create the new design in an adjacent artboard. To facilitate tracing, I also added the screenshot to the working artboard as a background layer at 30% opacity. I asked participants to complete the task as quickly and accurately as possible and gave them a time limit of seven minutes.

In the *Baseline* condition, the designer used standard XD tools. In the *Smart-Snap* and *Full Vector* conditions, the designers used the corresponding design assistance modes provided by Rewire. Finally, I also measured the performance of an "idealized" version of Rewire by creating

an *Ideal Vector* condition that provided a perfect vector representation of the screenshot. In this last condition, designers only needed to edit the text. Figure 4.1 shows the XD experience for the Full Vector condition. Designers were able to edit the properties and layering of the auto-generated vectors. The Smart-Snap and Screenshot Only modes started with no shapes in the layers panel, while the Ideal Vector contained the fully vectorized set of shapes.

I did not include the *Wireframe* mode as a test condition because, as noted earlier, designers described this mode as being most useful in communicating with clients. In addition, since designers often create wireframes at various levels of fidelity based on the situation, the appropriate target output for a wireframing task is not as well defined.

Before performing any tasks, I asked designers to recreate a small screenshot as a warm-up to familiarize themselves with the task instructions and baseline XD features. After the warm-up, participants performed the tasks under each condition. I fixed the task order and counterbalanced the order of conditions using a Latin square. Before each of the Rewire conditions (*Smart-Snap* and *Full Vector*), I described the relevant features of the design assistance modes and let the designers experiment with them on a small screenshot.

To evaluate RQ2, designers completed an open-ended survey after each task about what they liked and disliked about each condition. Additionally, they ranked all conditions in terms of preference and described contexts in which they would find the different modes most useful.

4.5.3 Materials

Since the Rewire study design counterbalanced the four conditions across the four tasks, it was important for the tasks to be equivalent in difficulty. I based the four tasks on an example interface from the evaluation dataset of online designs. I chose a design with a small number of elements to make the tasks manageable (Figure 4.8a), and with an average f-score of Rewire performance of 47% which was around the median of results for segmentation from the technical evaluation dataset.

Using this example as a reference, I then created three variations that had the same number and distribution of object types and rendered the corresponding screenshots (Figure 4.8b).

Finally, I verified that Rewire produced similar quality output for all four input images (e.g., the number of incorrect properties or misclassified objects is comparable).

4.5.4 Analysis

To measure *accuracy*, I exported each of the designer's artboards to a screenshot and computed the pixel difference using the Delta-E metric [187]. I averaged this metric across all pixels for each screenshot, and I assigned each designer an error score. I selected this metric because I instructed designers to match the original designs as close visually as possible. I did not expect that they should or could recreate the exact shapes and structure of the original artboards. Thus, I manually verified that every participant created shapes as instructed.

To analyze task durations, I first ran a repeated measures ANOVA to check for significant differences between the conditions. Then, I ran pairwise t-tests (i.e., paired two sample for means) across the six pairs of conditions. I then used the Holm-Bonferroni post-hoc method [89] to analyze the significance of the paired conditions and did not reject any null hypothesis where the p-value was greater than this metric. I report the adjusted p-values in the results. I calculated the effect sizes using Cohen's *d*.

4.5.5 Results

I report the results and analyses by each research question.

*RQ1: Do Rewire's modes of design assistance improve the **accuracy** and **efficiency** of designers when creating a vector graphics design from an example screenshot?*

Figure 4.9 shows a box-plot of the designers' times to completion. For the *Full Vector* condition, I found that designers were able to complete the tasks 52 seconds on average faster than the Smart-Snap condition ($t(11) = 3.26$, $p' < 0.008$, $d=0.91$), and 65 seconds faster than the Screenshot Only condition ($t(11) = 4.32$, $p' < 0.002$, $d=1.07$). For *Smart-Snap*, designers completed the tasks 13 seconds faster than the Screenshot Only condition ($t(11) = 2.20$, $p' <$

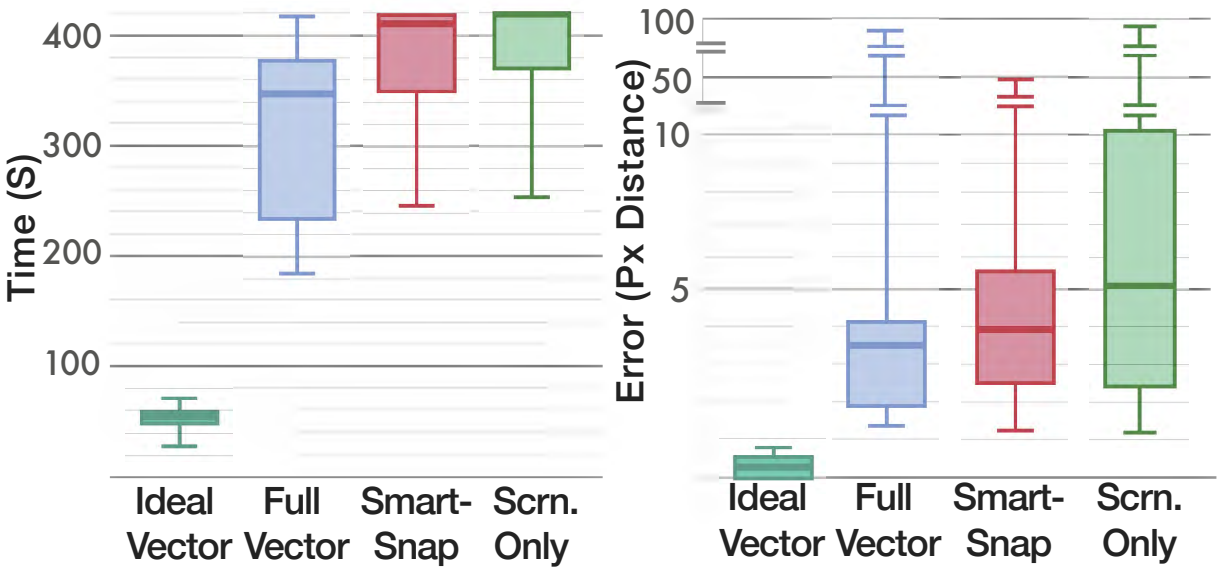


Figure 4.9: The left shows a box-plot of the the designers' task completion times for Rewire (Smart-Snap and Full Vector) and baseline (Ideal Vector and Screenshot Only) conditions. The right shows amount of error in the designers' output, as measured by the average of pixel color distance.

0.025, d=0.36). However, since they still had to recreate all of the shapes, the time savings were not as significant as the Full Vector condition. Additionally, since I stopped designers after 7 minutes, Smart-Snap and Screenshot Only conditions had a significant ceiling effect. For the Full Vector condition, most participants completed the tasks within the allotted time. Thus, the differences may have been more dramatic had there been no ceiling effect. The *Ideal Vector* was 5.5 minutes faster than Smart-Snap ($t(11) = 23.12$, $p' < 2.81E\text{-}10$, $d=7.01$), 4.67 minutes faster than Full Vector ($t(11) = 13.20$, $p' < 8.70E\text{-}08$, $d=1.07$), and 5.7 minutes faster than Screenshot Only ($t(11) = 28.13$, $p' < 4.02E\text{-}11$, $d=8.22$). This demonstrates that we can find significant time improvements by being able to produce a perfectly vectorized output, providing a strong motivation to improve the accuracy of Rewire's Full Vector mode.

Figure 4.9 shows box-plots of the designers' error score, measured by the average pixel distance. This shows that Ideal Vector has the lowest inter-quartile range, followed by Full Vector. I found no significant differences between any of the pairs of conditions demonstrating that Rewire's design assistance modes helped designers complete the tasks faster with no trade-offs in accuracy.

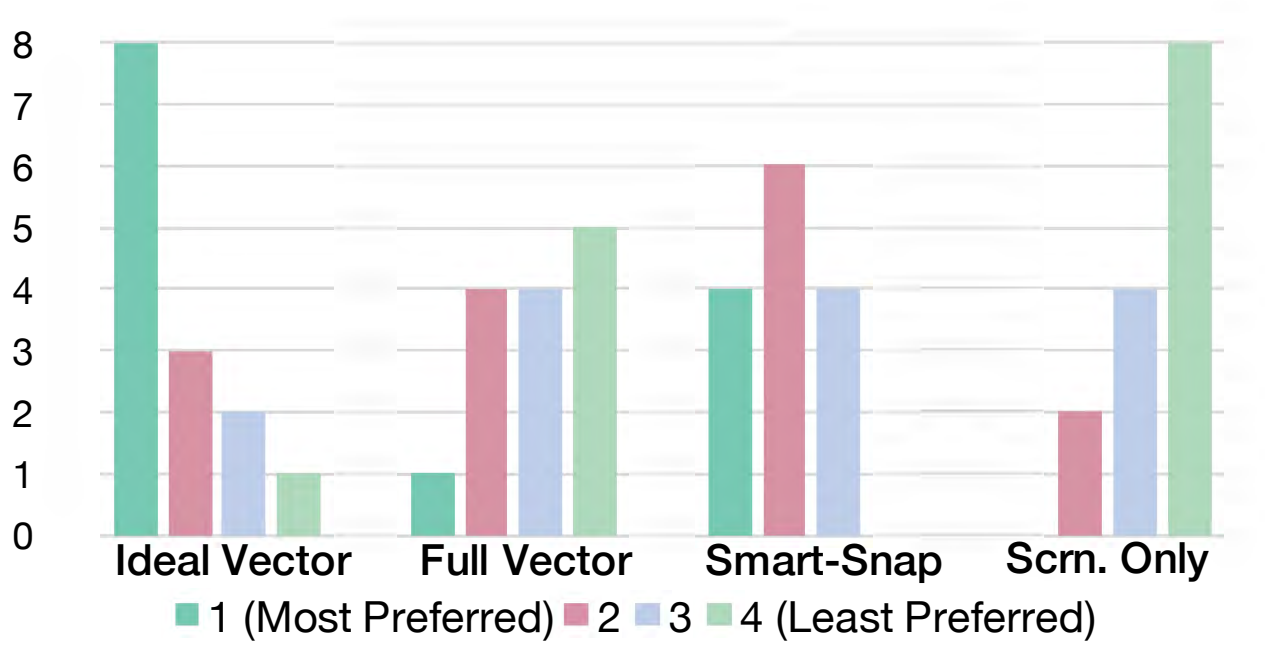


Figure 4.10: The designers' overall rankings of design assistance mode from most to least preferred.

RQ2: What aspects of each design assistance mode do designers like and dislike?

Figure 4.10 shows the designers' rankings of the design assistance modes from 1 (Most Preferred) to 4 (Least Preferred), showing that 10 designers prefer the Ideal Vector mode the most, followed by Smart-Snap. Smart-Snap was most consistently ranked second, followed by Rewire's Full Vector mode. The factors that most affected designers' rankings were *perceived effort and time*, which were mentioned by 10 designers. However, the Full Vector mode required more fixes by designers so they perceived it as less accurate and more work than the Smart-Snap mode.

Designers' favorite part of having the ideal vector template was that it was more accurate, and required less effort. Designers liked not having to redraw shapes or manually align them (9 designers). P10 mentioned "*It was way easier! Now I can spend my time working on actual design.*" However, designers did not always think that they would want it in every scenario. P13 said "*It would be easier if I wanted to copy the exact screenshot. I usually change up colours, shapes, etc., so this wouldn't be helpful in that case.*"

The second most preferred mode was Rewire's *Smart-Snap*. The designers' favorite part *Smart-Snap* was that it made it easier and quicker to achieve a more accurate alignment (8

designers). P8 said "*I was able to get an idea of where exactly each element is properly placed with close to pixel perfect alignment.*" P5 said "*The snapping guidelines are helpful and make for most accurate tracing of shapes - much better than doing them by hand.*" Five designers also mentioned Smart-Snap's help in drawing and matching the correct size for rectangles and other shapes. P11 said "*I really like the snapping guidelines because it takes the guesswork out of shape sizes and where items are on the page.*"

Smart-Snap seemed especially helpful for drawing rectangles and placing icon shapes; however, it was less helpful for text. Snapping guidelines did not always appear in the correct place to help align new text boxes to the baselines of image text, so designers still had to align them manually. The snapping condition also did not help with matching any text or shape properties, so designers still found that part of the task to be challenging. Additionally, four designers mentioned that adding support for snapping to help obtain corner radii for rectangles would be useful. Currently, Smart-Snap only displays snapping guidelines for vertical and horizontal axes.

Rewire's *Full Vector* mode was most frequently ranked third. Designers mostly would not want to use it if the Ideal Vector mode was available, since there were shapes and text that required fixes in the vectorized output. However, designers did like Full Vector mode's auto-generated shapes and text (9 designers), and they felt that Full Vector mode required less effort overall than the Screenshot-Only mode (4 designers). P5 mentioned "*It was a nice balance of providing elements and still allowing the user to make decisions.*"

The most common dislike (6 designers) for *Full Vector* was having to manually fix issues in the auto-generated output. Rewire's Full Vector mode was not perfect. It may have required more cognitive load to detect and find the issues. P10 said "*It requires more brain computing to determine how much more needs to be done. I would prefer to have it draw only the objects it is most confident about.*" P1 said "*The fact that objects were not created accurately requires me to go through the auto-generated objects to make sure they are up to spec.*" Despite some designers' dislike of the fixes required in this mode, three designers mentioned it was easier to make these fixes than recreating from scratch.

Text shapes confused some designers because Rewire generates masked vector shapes instead of editable text boxes when it cannot detect text. We instructed them of this behavior in the warm-up, but several designers found this behavior confusing, and they tried to edit the text before realizing Rewire had not generated the text box. Also, four designers mentioned it was difficult to distinguish Rewire's vector shapes from newly drawn shapes, despite the Rewire Highlights panel. Rewire's vector shapes are currently indistinguishable from hand-drawn shapes in the XD canvas. Thus it is difficult for designers to check them for accuracy. One designer suggested adding an indicator to distinguish them in the layers panel.

As Figure 4.10 shows, the *Screenshot-Only* mode was most commonly ranked last. None of the designers preferred it most. The designers most common dislike was the *lack of precision and accuracy* (7 designers). Designers mentioned needing to use more strategies (e.g, zooming, eyeballing, using guides) to ensure accurate alignment of shapes to the image, and they disliked the precision in their output. Seven designers felt the task was more difficult, more work, and more tedious than other modes. However, designers liked having more control and freedom with this mode. Five designers mentioned having more trust in its accuracy since it is their current method.

After ranking the modes, we had the designers describe scenarios in which they might find each mode most useful. For Smart-Snap, the designers thought it would be most useful for tracing and getting exact alignment (P12, P13) and when creating new shapes similar in size or layout to the screenshot but with different designs (P5, P13). They also thought it would provide more control than the other modes. Designers thought the Ideal Vector and Full Vector modes would be most useful for situations where the original assets were lost (P9, P13), having to match an interface or existing design language (P2, P3, P8), or in making quick mockups based an existing interface. However, accuracy was also important (P7, P11, P14). Designers felt that they would need to build trust in the auto-vectorized mode before integrating it into their design process. Screenshot-Only was only mentioned as useful when creating quick mockups if the designers did not care about accuracy or wanted a more loose recreation (P2, P8).

4.6 Discussion & Conclusion

In this chapter, I presented Rewire, a system that automatically infers a semantic vector-based representation of interface shapes from a pixel-based input screenshot. Rewire provides new forms of design assistance to ease the adaptation of example screenshots directly in designs. If designers can save time in their prototyping from recreating interface elements, they would potentially have more time to consider alternative designs, which would lead them to better final products [65]. I believe that systems like Rewire can enable researchers to explore new forms of intelligent design assistance enabling new possibilities in user interface design.

Rewire presents new tools for creating user interface designs based on example images. I see this work as an initial exploration of intelligent design assistance, and I see many opportunities to develop more sophisticated tools in the future. One area that could be explored is additional forms of design assistance. While evaluating Rewire, I discovered that interface design documents frequently contain complex hierarchies and shapes with ambiguous representations. Rewire could infer multiple hierarchies and shape types and allow the designer to select from these candidate representations. Also, because achieving a perfect vectorization is difficult and designers may have different preferences for the vectorized output that should be produced, Rewire could be a user-in-the-loop system where designers can repair the vectorized output while training Rewire to improve its accuracy.

Additionally, deep learning could be used to improve the accuracy of Rewire's vectorization pipeline. The pix2code system [35] presents a convolutional and recurrent neural network to infer interface code from screenshot images. The network represents the desired structure via a constrained domain-specific language (DSL) that encodes simple geometric relationships between a fixed set of UI components in addition to a small set of style properties, like color. Yet, this DSL is not designed to handle the much broader range of component types, appearances and arrangements that arise in many example screenshots. While adapting this network to output the type of structured representation that Rewire requires is an interesting direction for future work, the pix2code approach in its current form is not directly applicable to our problem. Training an end-to-end network would likely require either a lot of training data (e.g., collected

from online galleries and cleaned) or using data augmentation. Another option is to fine tune a pre-trained segmentation network (e.g., [29]) to handle interface screenshots.

Another area for future work would be to explore better detection of natural images from interface shapes. To do this, researchers could potentially train a network to distinguish these segments. Also, small interface shapes elements frequently get filtered by Rewire's segmentation algorithms, thus an area of exploration could be to improve these low-level techniques. Researchers could also explore an extension of Prefab's models [64] to discover more properties of shapes like shadows and gradients.

Finally, it is possible that tools like Rewire may unintentionally facilitate unsanctioned copying. My formative work suggests that the tasks Rewire supports (e.g., creating derived designs, recreating vector designs when original assets are lost) are common practice in the design community and not viewed as "stealing". However, researchers and practitioners should consider the ethical implications of tools like Rewire when adopting them into their practices.

4.7 Contributions

This work was published at CHI 2018 as ***Rewire: Interface Design Assistance from Examples*** [198] with co-authors from Adobe Research Mira Dontcheva, Wilmot Li, Morgan Dixon, and Joel Brandt. Amy Ko is also a co-author on this paper. I developed the initial ideas with Joel Brandt, Mira Dontcheva, and Morgan Dixon. All co-authors provided valuable feedback and ideas towards this project throughout the implementation, evaluation, and paper writing. I built the Rewire system and add-ins into the Adobe XD codebase through collaboration with the XD team at Adobe. I conducted the exploratory evaluation before beginning the implementation of Rewire and the full evaluations with 16 UX designers after completing the project. I completed the majority of the work during two internships at Adobe Research. You can watch the demo video of Rewire at http://doi.acm.org/10.1145/3173574.3174078 under "Source Materials".

Chapter 5

Modeling Mobile Interface Tappability

Designers conduct *usability testing* in the latter stages of the design process to discover problems in their interfaces. Designers can conduct these studies at various fidelity levels, from paper prototypes to full interface implementations. A recent trend in usability testing is to conduct large crowdsourced studies where designers deploy their designs to online services (e.g., Mechanical Turk, 5 Second Test[1]) and have crowd workers annotate them with interaction data or provide brief impressions. One such study is a *tappability study* or a *visual affordance test*, that can help designers understand whether their users will perceive the right set of tappable elements in a mobile app design. Designers can conduct such a study in a lab where they have users label tappable and not tappable elements on paper, or their users can label them digitally through a crowdsourcing website. Such a study can help designers understand and prevent tappability issues, which can lead to user frustration and error.

Tapping is arguably the most important gesture on mobile interfaces. Yet, it can still be difficult for people to distinguish tappable and not-tappable elements in a mobile interface. In

[1]https://fivesecondtest.com/

traditional desktop GUIs, the style of clickable elements (e.g., buttons) are often conventionally defined. However, with the diverse styles of mobile interfaces, tappability has become a crucial usability issue. Poor tappability can lead to a lack of discoverability [162] and false affordances [75] that can lead to user frustration, uncertainty, and errors [3, 7].

Designers can use properties like color or depth to signify "clickability" [3] or "tappability" in mobile interfaces. In Chapter 2, Section 2.1, I review common *signifiers* [162] which can indicate to a user if an interface element is tappable. To design for tappability, designers can apply existing design clickability guidelines [3]. These are important and can cover typical cases, however, it is not always clear when to apply them. Frequently, mobile app developers are not equipped with such knowledge. Additionally, designers frequently introduce new design patterns and interface elements in modern platforms for mobile apps. Designing these to include appropriate tappability signifiers is challenging. Furthermore, mobile interfaces cannot utilize some clickability signifiers available in desktop interfaces (e.g., hover states).

When designers are introducing new design elements and patterns, conducting a tappability study can be highly useful to ensure their users have the correct perception of tappability. However, it is time-consuming and expensive to conduct such studies. In addition, the findings from these studies are often limited to a specific app, interface design, or design library. In this work, I seek to understand how we can help designers understand tappability signifiers at a large scale across a diverse array of interface designs. For example, I found that despite the existence of tappability guidelines, there is a *significant amount of tappability misperception in real mobile interfaces*. Another goal of this work is to build tools to help designers diagnose tappability issues in new apps automatically without needing to conduct a usability study. This can help designers avoid the time and expense for conducting such studies, and it can enable them to diagnose tappability issues early in the design process and understand the effects of small design changes on tappability perception.

In this work, I present an approach for modeling interface tappability at scale. In addition to acquiring a deeper understanding about tappability, I develop tools that can automatically identify tappability issues in a mobile app interface (see Figure 5.1). I trained a deep learn-

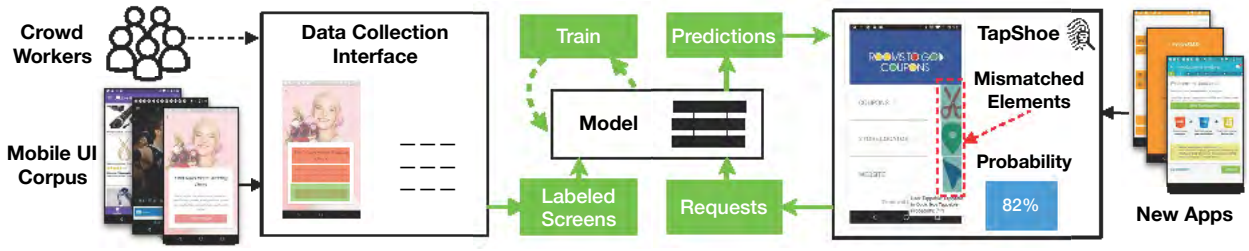


Figure 5.1: Our deep model learns from a large-scale dataset of mobile tappability collected via crowdsourcing. It predicts tappability of interface elements and identifies mismatches between designer intention and user perception, and is served in the TapShoe tool that can help designers and developers to uncover potential usability issues about their mobile interfaces.

ing model based on a large dataset of labeled tappability of mobile interfaces collected via crowdsourcing. The dataset includes more than 20,000 examples from more than 3,000 mobile screens. The tappability model achieved reasonable accuracy with mean precision 90.2% and recall 87.0% on identifying tappable elements as perceived by humans. To showcase a potential use of the model, I built TapShoe, a web interface that diagnoses mismatches between the human perception of the tappability of an interface element and its actual state in the interface code. I conducted informal interviews with 7 professional interface designers who were positive about the TapShoe interface, and could envision intriguing uses of the tappability model in realistic design situations. The contributions of this chapter include the following:

1. An approach for understanding interface tappability at scale using crowdsourcing and computational signifier analysis, and a set of findings about mobile tappability;
2. A deep neural network model that learns human perceived tappability of interface elements from a range of interface features, including the spatial, semantic and visual aspects of an interface element and its screen, and an in-depth analysis about the model behavior;
3. An interactive system that uses the model to examine a mobile interface by automatically scanning the tappability of each element on the interface, and identifies mismatches with their intended tappable behavior.

5.1 Understanding Tappability at Scale

A common type of usability testing is a *tappability study* or a visual affordance test [12]. In these studies, designers have crowd workers or lab participants label interfaces for which elements

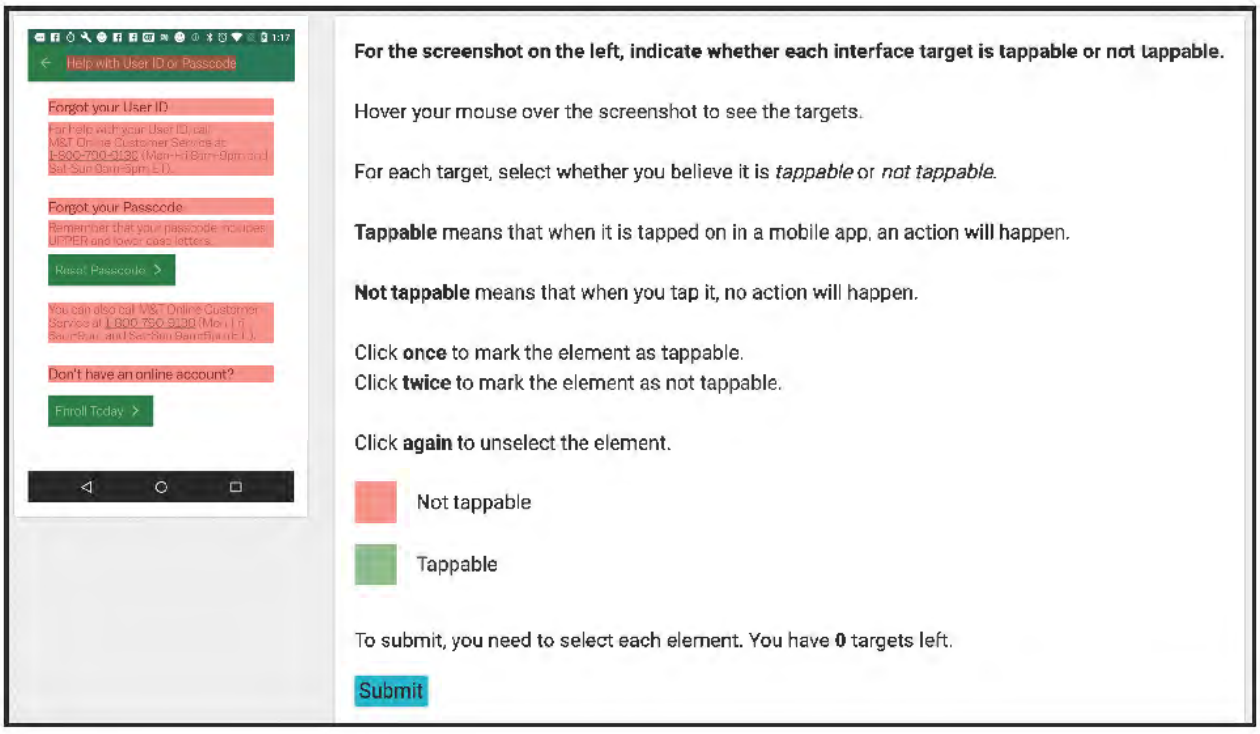


Figure 5.2: The interface that workers used to label the tappability of UI elements via crowdsourcing. It displays a mobile interface screen with interactive hotspots that can be clicked to label an element as either tappable or not tappable.

they think are tappable and not tappable digitally or on paper. Based on this data, designers can construct heatmaps to visualize where users would tap in the app being tested. These studies can help designers discover which elements have missing or false tappability signifiers. However, in general, there is a lack of a dataset and deep understanding about interface tappability across diverse mobile apps. Having such a dataset and knowledge is required for researchers to develop automated techniques to help designers diagnose tappability issues in their interfaces.

5.1.1 Crowdsourcing Data Collection

I designed a crowdsourcing task to simulate a tappability study across a large corpus of Android mobile apps [59], using the interface shown in Figure 5.2. The left side of the interface displayed a mobile app screenshot. The right side of the task interface displayed instructions for the task and an explanation about what was meant by tappable and not tappable. For tappable elements,

it was "*When you tap this in a mobile interface, an action will happen*", and for not tappable, the explanation was "*When you tap on it, no action will happen*".

To collect the tappability dataset, I selected a set of 3,470 unique, randomly chosen screens from the Rico dataset [59], and I had crowd workers label elements randomly sampled from these screens as either `tappable` or `not tappable`. I built a web interface that selected the elements for the workers to label in the following manner. Each UI screen in the Rico dataset has an Android view hierarchy—JSON tree structure of all of the interface elements on the screen, similar to a DOM tree for a web interface. Each element in the hierarchy has a `clickable` property that marks whether an element will respond to a tapping event. For each screen, the labeling interface selected up to five unique `clickable` and non-`clickable` elements. When selecting `clickable` elements, starting from a leaf element, the web interface selects the top-most `clickable` element in the hierarchy for labeling. When a clickable element contains a sub-tree of elements, these elements are typically presented as a single interface element to the user, which is more appropriate for the worker to label as a whole. When the web interface selects a clickable container (e.g., ViewGroup), it does not select any of its child elements thus preventing any duplicate counting or labeling. The web interface does not select elements in the status bar or navigation bar as they are standard across most screens in the dataset.

To perform a labeling task, a crowd worker hovers their mouse over the interface screenshot, and the labeling interface displays grey hotspots over the interface elements pre-selected based on the above process. Workers click on each hotspot to toggle the label as either tappable or not tappable, which are colored in green and red, respectively. The labeling interface asked each worker to label around six elements for each screen. Depending on the screen complexity, the amount of elements could vary. The labeling interface randomized the elements as well as the order to be labeled across each worker.

5.1.2 Results

I collected 20,174 unique interface elements from 3,470 app screens using the tappability labeling interface (Figure 5.2). These elements were labeled by 743 unique workers in two

	Positive Class	#Elements	Precision	Recall
R1	`clickable=True`	6,101	79.81%	89.07%
	`clickable=False`	3,631	78.56%	61.75%
R2	`clickable=True`	6,560	79.55%	90.02%
	`clickable=False`	3,882	78.30%	60.90%
All	`clickable=True`	12,661	79.67%	89.99%
	`clickable=False`	7,513	78.43%	61.31%

Table 5.1: The number of elements labeled by the crowd workers in two rounds, along the precision and recall of human workers in perceiving the actual clickable state of an element as specified in the view hierarchy metadata.

rounds where each round involved different sets of workers (see Table 5.1). Each worker could complete up to 8 tasks. On average, each worker completed 4.67 tasks. Of these elements, 12,661 of them are indeed tappable (i.e., the view hierarchy attribute `clickable=true`), and 7,513 of them are not.

How well can human users perceive the actual clickable state of an element as specified by developers or designers?

To answer this question, I treat the `clickable` value of an element in the view hierarchy as the actual value and human labels as the predicted value for a precision and recall analysis. In this dataset of real mobile app screens, there were still many false signifiers for tappability potentially causing workers to misidentify tappable and not-tappable elements (see Table 5.1). The workers labeled non-clickable elements as tappable 39% of time. While the workers were significantly more precise in labeling clickable elements, workers still marked clickable elements as not tappable 10% of the time. The results were quite consistent across two rounds of data collection involving different workers and interface screens. These results further confirmed that tappability is an important usability issue worth investigation.

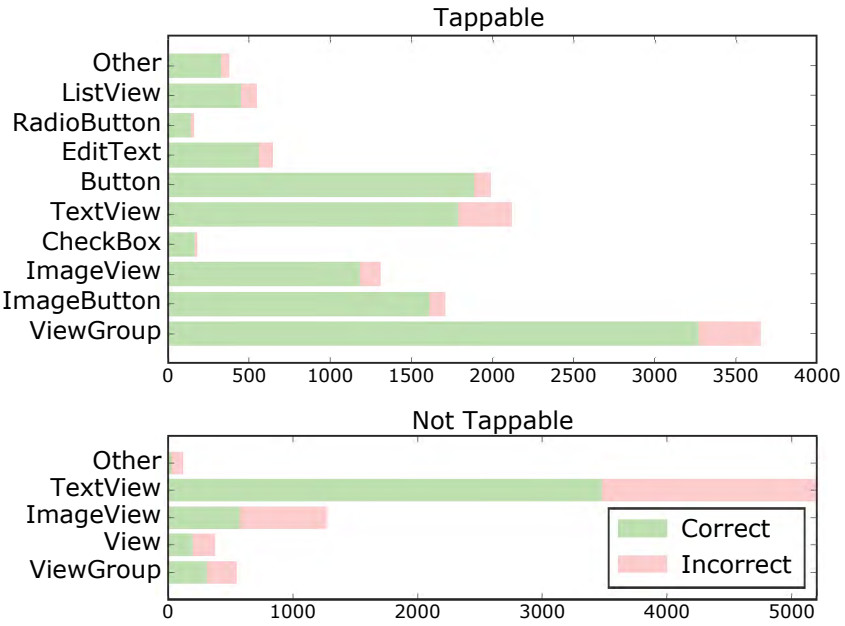


Figure 5.3: The number of tappable and not-tappable elements in several type categories with the bars colored by the relative amounts of correct and incorrect labels.

5.1.3 Signifier Analysis

To understand how users perceive tappability, I analyzed the potential signifiers affecting tappability in real mobile apps. These findings can help us understand human perception of tappability and help develop features to build machine learning models to predict tappability. I investigated several visual and non-visual features based on previous understandings of common visual signifiers [2, 3, 11] and through exploration of the characteristics of the dataset.

Element Type

Several element types have conventions for visual appearance, thus users would consistently perceive them as tappable [160] (e.g., buttons). I examined how accurately workers label each interface element type from a subset of Android class types in the Rico dataset [59]. Figure 5.3 shows the distribution of tappable and not-tappable elements by type labeled by human workers. Common tappable interface elements like Button and Checkbox appeared more frequently in the set of tappable elements. For each element type, I computed the accuracy by comparing

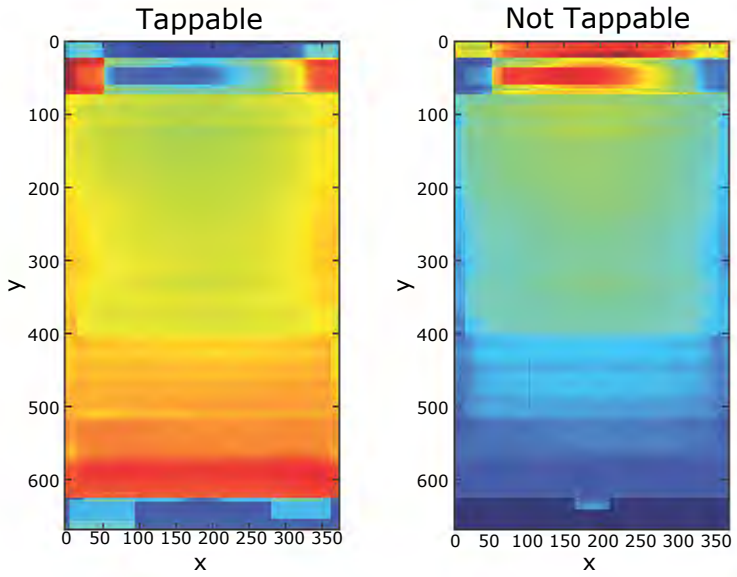


Figure 5.4: Heatmaps displaying the accuracy of tappable and not tappable elements by location where warmer colors represent areas of higher accuracy. Workers labeled not-tappable elements more accurately towards the upper center of the interface and tappable elements towards the bottom center of the interface.

the worker labels to the view hierarchy `clickable` values. For tappable elements, the workers achieved high accuracy for most types. For not-tappable elements, the two most common types, TextView and ImageView, had low accuracy of only 67% and 45%, respectively. These interface types allow more flexibility in design than standard element types (e.g., RadioButton). Unconventional styles may make an element more prone to ambiguity in tappability.

Location

I hypothesized that an element's location on the screen may have influenced the accuracy of workers in labeling its tappability. Figure 5.4 displays a heatmap of the accuracy of the workers' labels by location. I created the heatmap by computing the accuracy per pixel, using the `clickable` attribute, across the 20,174 labeled elements I collected using the bounding box of each element. Warm colors represent higher accuracy values. For tappable elements, workers were more accurate towards the bottom of the screen than the center top area. Placing a not-tappable element in these areas might confuse people. For tappable elements, there

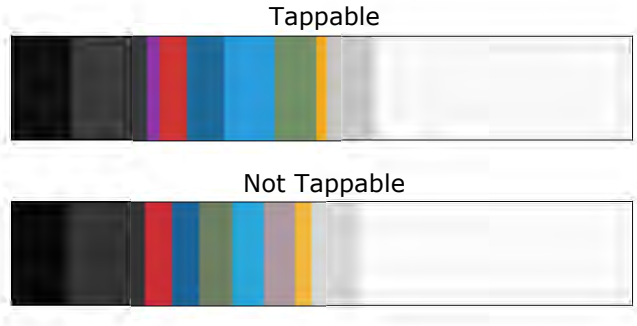


Figure 5.5: The aggregated RGB pixel colors of tappable and not-tappable elements clustered into the 10 most prominent colors using K-Means clustering.

are two spots at the top region of high accuracy. I speculate that this is because these spots are where apps tend to place their Back and Forward buttons. For not-tappable elements, the workers were less accurate towards the screen bottom and highly accurate in the app header bar area with a corresponding area of low accuracy for tappable elements. This area is not tappable in many apps, so people may not realize any element placed there is tappable.

Size

There was only a small difference in average size between labeled tappable and not-tappable elements. However, tappable elements labeled as not tappable were 1.9 times larger than tappable elements labeled as tappable indicating that elements with large sizes were more often seen as not tappable. Examining specific element types can reveal possible insights into why the workers may have labeled larger elements as not tappable. TextView elements tend to display labels but can also be tappable elements. From design recommendations, tappable elements should be labeled with short, actionable phrases [208]. The text labels of not-tappable TextView elements have an average and median size of 1.48 and 1.55 times larger respectively than those of tappable TextView elements. This gives us a hint that TextView elements may be following these recommendations. For ImageView elements, the average and median size for not-tappable elements were 2.39 and 3.58 times larger than for tappable elements. People may believe larger ImageView elements, typically displaying images, to be less likely tappable than smaller ImageView elements.

Color

Based on design recommendations [3], color can also be used to signify tappability. Figure 5.5 displays the top 10 dominant colors in each class of labeled tappable and not-tappable elements, which are computed using K-Means clustering. The dominant colors for each class do not necessarily denote the same set. The brighter colors such as blue and red have more presence (i.e., wider bars) in the pixel clusters for tappable elements than those for not-tappable ones. In contrast, not-tappable elements have more grey and white colors. I computed these clusters across the image pixels for 12 thousand tappable and 7 thousand not-tappable elements and scaled them by the proportion of elements in each set. These differences indicate that color is likely a useful distinguishing factor.

Words

As not-tappable textual elements are often used to convey information, the number of words in these elements tend to be large. The mean number of words per element, based on the log-transformed word count in each element, was 1.84 times greater for not-tappable elements (Mean: 2.62, Median: 2) than tappable ones (Mean: 1.42, Median: 1). Additionally, the semantic content of an element's label may be a distinguishing factor based on design recommendations [208]. I hypothesized that tappable elements would contain keywords indicating tappability (e.g., "Login"). To test this, I examined the top five keywords of tappable and not-tappable elements using TF-IDF analysis, with the set of words in all the tappable and not-tappable elements as two individual documents. The top 2 keywords extracted for tappable elements were "submit" and "close", which are common signifiers of actions. However, the remaining keywords for tappable elements (i.e., "brown", "grace" and "beauty"), and the top five keywords for not-tappable elements (i.e., "wall", "accordance", "recently", "computer" and "trying") do not appear to be actionable signifiers.

5.2 Model Architecture

Because it is expensive and time consuming to conduct user studies, it is desirable to develop automated techniques to examine the tappability of mobile interfaces. Although we can use the signifiers previously discussed as heuristics for this purpose, it would be difficult to manually combine them appropriately. It is also challenging to capture factors that are not obvious or hard to articulate. As such, I employed a deep learning approach to address the problem. Overall, our model is a feedforward neural network with a deep architecture (multiple hidden layers). It takes a concatenation of a range of features about the element and its screen and outputs a probability of how likely a human user would perceive an interface element as tappable.

5.2.1 Feature Encoding

Our model takes as input several features collected from the view hierarchy metadata and the screenshot pixel data of an interface. For each element under examination, our features include 1) semantics and functionality of the element, 2) the visual appearance of the element and the screen, and 3) the spatial context of the element on the screen.

Semantic Features

The length and the semantics of an element's text content are both potential tappability signifiers. For each element, the model scans the text using OCR. To represent the semantics of the text, the model uses a word embedding that is a standard way of mapping word tokens into a continuous dense vector that can be fed into a deep learning model. The model encodes each word token in an element as a 50-dimensional vector representation that is pre-learned from a Wikipedia corpus [171]. When an element contains multiple words, the model treats them as a bag of words and apply max pooling to their embedding vectors to acquire a single 50-dimensional vector as the semantic representation of the element. The model also encodes the number of word tokens each element contains as a scalar value normalized by an exponential function.

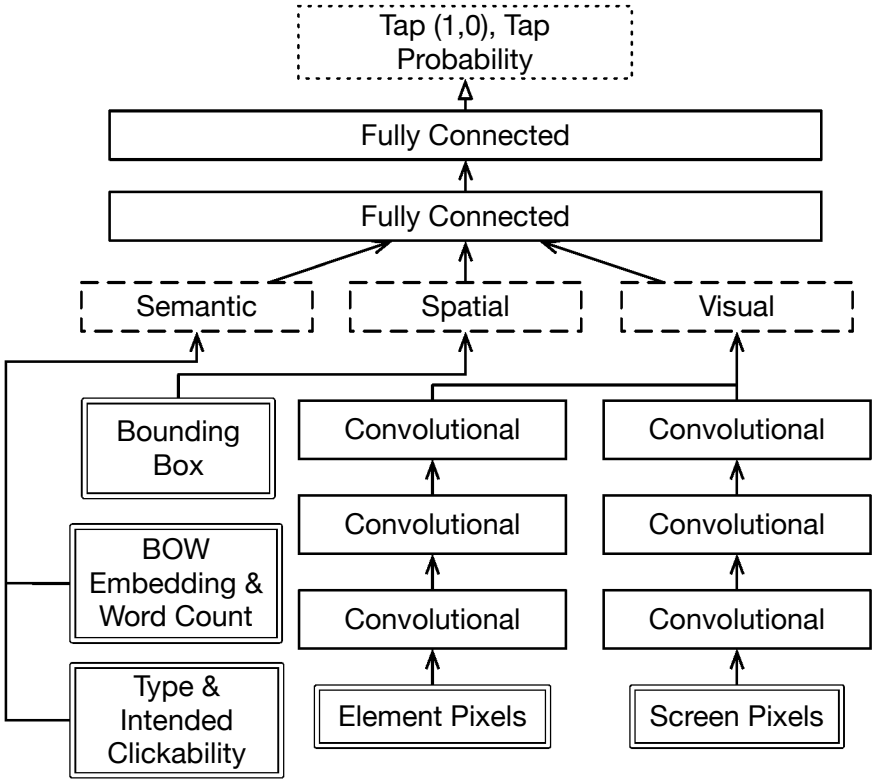


Figure 5.6: A deep neural network model for predicting tappability, leveraging semantic, spatial and visual features. The model produces a prediction and continuous probability of an interface element being perceived as tappable.

Type Features

There are many standard element types that users have learned over time (e.g., buttons and checkboxes) [160]. However, new element types are frequently introduced (e.g., floating action button). Our model includes an element type feature as an indicator of the element's semantics. This feature allows the model to potentially account for these learned conventions as a users' background plays an important role in their decision. To encode the Type feature, the model includes a set of the 22 most common interface element types (e.g. TextView, Button) represented as a 22-dimensional categorical feature. For training, the model collapses the vector into a 6-dimensional embedding vector for training, which provides better performance over sparse input. Each type comes with a built-in or specified clickable attribute that is encoded as either 0 or 1.

Visual Features

As previously discussed, visual design signifiers such as color distribution can help distinguish an element's tappability. It is difficult to articulate the visual perception that might come into play and realize it as an executable rule. As a result, the model feeds an element's raw pixel values and the screen to which the element belongs to the network, through convolutional layers—a popular method for image processing. The model resizes the pixels of each element and formats them as a 3D matrix in the shape of 32x32x3 where the height and width are 32, and 3 is the number of RGB channels. Contextual factors on the screen may affect the human's perception of tappability. To capture the context, the model resizes and formats the entire screen as another visual feature. This manifests as a 3D matrix in the shape of 300x168x3 and preserves the original aspect ratio. As I will discuss later, a screen contains useful information for predicting an element's tappability even though such information is not easy to articulate.

Spatial Features

As location and size can be signifiers of tappability, we include them as features. The model captures the element's bounding box as four scalar values: x, y, *width*, and *height*. The model scales each of these values to the range of 0 and 1 by normalizing them using the screen width and height.

5.2.2 Model Architecture & Learning

Figure 5.6 illustrates our model architecture. To process the element and screenshot pixels, our network has three convolutional layers with ReLU [151] activation. Each convolutional layer applies a series of 8 3x3 filters to the image to help the model progressively create a feature map. Each convolutional layer is followed by a 2x2 max pooling layer to reduce the dimensionality of the image data for processing. Finally, the output of the image layers is concatenated with the rest of the features into a series of two fully connected 100-dimensional dense layers using ReLU [151] as the activation function. The output layer produces a binary

classification of an element's tappability using a sigmoid activation function to transform the output into probabilities from zero to one. The probability indicates how likely the user would perceive the element as tappable. I trained the model by minimizing the sigmoid cross-entropy loss between the predicted values and the binary human labels on tappability of each element in the training data. For loss minimization, the model uses the Ada adaptive gradient descent optimizer with a learning rate of 0.01 and a batch size of 64. To avoid model overfitting, the model applies a dropout ratio of 40% to each fully connected layer to regularize the learning. I built our model using Tensorflow [23] in Python and trained it on a Tesla V100 GPU.

5.2.3 Model Performance Results

I evaluated our model using 10-fold cross validation with the crowdsourced dataset. In each fold, I used 90% of the data for training and 10% for validation. I trained our model for 100,000 iterations. Similar to an information retrieval task, I examine how well our model can correctly retrieve elements that users would perceive as tappable. I select an optimal threshold based on Precision-Recall AUC. Our model achieved a mean precision and recall, across the 10 folds of the experiment, of 90.2% (SD: 0.3%) and 87.0% (SD: 1.6%). To understand what these numbers imply, I analyzed how well the `clickable` attribute in the view hierarchy predicts user tappability perception: precision 89.9% (SD: 0.6%) and recall 79.6% (SD: 0.8%). While our model has a minor improvement on precision, it outperforms the `clickable` attribute on recall considerably by over 7%.

Although identifying not-tappable elements is less important in real scenarios, to better understand the model, I report the performance with not-tappable elements as the target class. Our model achieved a mean precision 70% (SD: 2%) and recall 78% (SD: 3%), which improves precision by 9%, with a similar recall, over the `clickable` attribute (precision 61%, SD: 1% and recall 78%, SD: 2%). One potential reason that not-tappable elements have a relatively low accuracy is that they tend to be more diverse, leading to more variance in the data.

In addition, the original dataset had an uneven number of tappable and not-tappable elements (14,301 versus 5,871), likely causing our model to achieve higher precision and

	Predicted Tappable	Predicted Not Tappable
Actually Tappable	1195	260
Actually Not Tappable	235	1170

Table 5.2: A confusion matrix for the balanced dataset, averaged across the 10 cross-validation experiments.

recall for tappable elements than not-tappable ones. Therefore I created a balanced dataset by upsampling the minority class (i.e., not-tappable). On the balanced dataset, our model achieved a mean precision and recall of 82% and 84% for identifying tappable elements, and a mean precision and recall of 81% and 86% for not-tappable elements. Table 5.2 shows the confusion matrix for the balanced dataset. Compared to using view hierarchy `clickable` attribute alone, which achieved mean precision 79% and recall 80% for predicting tappable elements, and 79% and 78% for not-tappable ones, our model is consistently more accurate across all the metrics. These performance improvements show that our model can effectively help developers or designers identify tappability misperceptions in their mobile interfaces.

5.3 Human Consistency & Model Behaviors

We speculated that the model did not achieve even higher accuracy because human perception of tappability can be inherently inconsistent as people have their own experience in using and learning different sets of mobile apps. This can make it challenging for the model to achieve perfect accuracy. To examine this hypothesis, I collected another dataset via crowdsourcing using the same interface as shown in Figure 5.2. I selected 334 screens from the Rico dataset, which I did not select in previous rounds of data collection. I recruited 290 workers to perform the same task of marking each selected element as either tappable or not tappable. However, each element was labeled by 5 different workers to enable examining how much these workers agree on the tappability of an element. In total, there were 2,000 unique interface elements and each was labeled 5 times. In total, 1,163 elements (58%) were entirely consistent among all 5 workers which include both tappable and not-tappable elements. I report two metrics to analyze the consistency of the data statistically. The first is in terms of an agreement score [214]

that I compute using the following formula:

$$A = \frac{\sum_{e \in E} \sum_{r \in R} \left(\frac{|R_i|}{|R_e|} \right)^2}{|E|} \times 100\% \tag{5.1}$$

Here, e is an element in the set of all interface elements E that were rated by the workers, R_e is the set of ratings for an interface element e, and R_i is the set of ratings in a single category (0: not tappable, 1: tappable). I also report the consistency of the data using Fleiss' Kappa [72], a standard inter-rater reliability measure for the agreement between a fixed number of raters assigning categorical ratings to items. This measure is useful because it computes the degree of agreement over what would be expected by chance. As there are only two categories, the agreement by chance is high. The overall agreement score across all the elements using Equation 5.1 is *0.8343*. The number of raters is 5 for each element on a screen, and across 334 screens, resulting in an overall Fleiss' Kappa value of 0.520 *(SD=0.597, 95% CI [0.575,0.618], P=0)*. This corresponds to a "Moderate" level agreement according to [116]. What these results demonstrate is that while there is a significant amount of consistency in the data, there still exists a certain level of disagreement on what elements are tappable versus not tappable. Particularly, consistency varies across element *Type* categories. For example, *View* and *ImageView* elements were labeled far less consistently (*0.52, 0.63*) than commonplace tappable element types such as *Button* (94%), *Toolbar* (100%), and *CheckBox* (95%). View and ImageView elements have more flexibility in design, which may lead to more disagreement.

To understand how the model predicts elements with ambiguous tappability, I test the trained model on this new dataset. The model matches the uncertainty in human perception of tappability surprisingly well (see Figure 5.7). When workers are consistent on an element's tappability (two ends on the X axis), the model tends to give a more definite answer—a probability close to 1 for tappable and close to 0 for not tappable. When workers are less consistent on an element (towards the middle of the X axis), the model predicts a probability closer to 0.5.

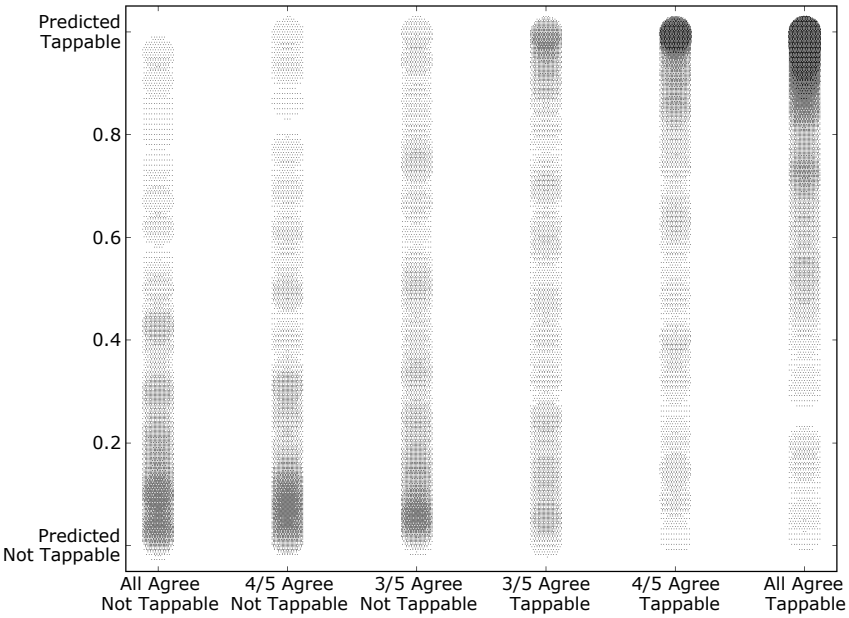


Figure 5.7: The scatterplot of the tappability probability output by the model (Y axis) versus the consistency in the human worker labels (X axis) for each element in the consistency dataset.

5.3.1 Usefulness of Individual Features

One motivation to use deep learning is to alleviate the need for extensive feature engineering. Recall that we feed the entire screenshot of an interface to the model to capture contextual factors affecting the user's decision that can not be easily articulated. Without the screenshot pixels as input, there is a noticeable drop in precision and recall for tappable of 3% and 1%, and for not-tappable, an 8% drop in precision but no change in recall. This indicates that there is useful contextual information in the screenshot affecting the users' decisions on tappability. I also examined removing the *Type* feature from the model, and found a slight drop in precision about 1% but no change in recall for identifying tappable elements. The performance change is similar for the not-tappable case with 1.8% drop in precision and no drop in recall. We speculate that removing the Type feature only caused a minor impact likely because the model has captured some of element type information through its pixels.

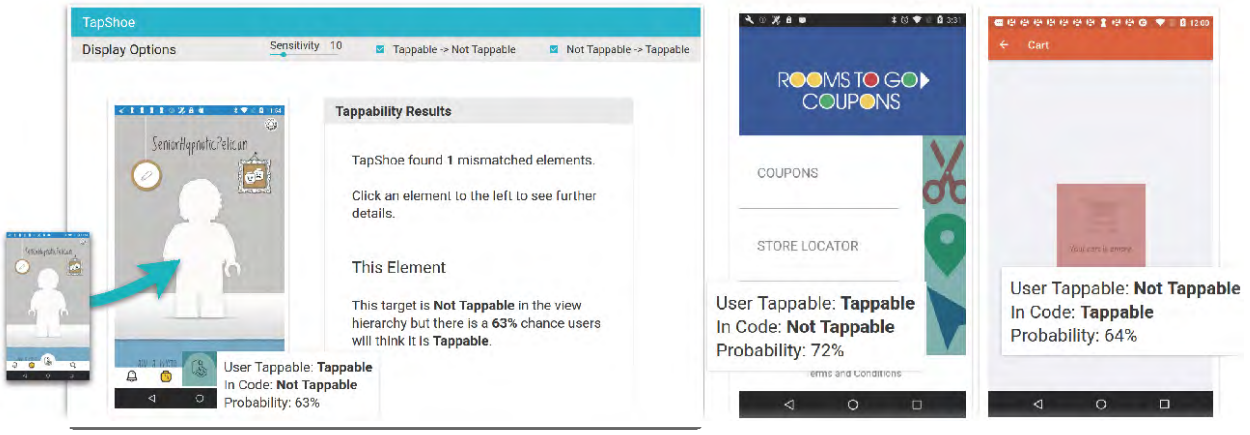


Figure 5.8: The TapShoe interface. An app designer drag and drops a UI screen on the left. TapShoe highlights interface elements whose predicted tappability is different from its actual tappable state as specified in its view hierarchy.

5.4 Interface

I created a web interface for our tappability model called TapShoe (see Figure 5.8). The interface is a proof-of-concept tool to help app designers and developers examine their UI's tappability. I describe the TapShoe interface from the perspective of an app designer, Zoey, who is designing an app for deal shopping, shown in the right hand side of Figure 5.8. Zoey has redesigned some icons to be more colorful on the home page links for "Coupons", "Store Locator", and "Shopping". Zoey wants to understand how the changes she has made would affect the users' perception of which elements in her app are tappable. First, Zoey uploads a screenshot image along its view hierarchy for her app by dragging and dropping them into the left hand side of the TapShoe interface. Once Zoey drops her screenshot and view hierarchy, TapShoe analyzes her interface elements and returns a tappable or not-tappable prediction for each element. The TapShoe interface highlights the interface elements with a tappable state, as specified by Zoey in the view hierarchy, that does not match up with user perception as predicted by the model.

Zoey sees that the TapShoe interface highlighted the three colorful icons she redesigned. These icons were not tappable in her app but TapShoe predicted that the users would perceive them as tappable. She examines the probability scores for each element by clicking on the green hotspots on the screenshot to see informational tooltips. She adjusts the sensitivity slider to

change the threshold for the model's prediction. Now, she sees that the "Coupons" and "Store Locator" icon are not highlighted and that the arrow icon has the highest probability of being perceived as tappable. She decides to make all three colorful icon elements interactive and extend the tappable area next to "Coupons", "Store Locator", and "Website". These fixes prevent her users from the frustration of tapping on these elements with no response.

I implemented the TapShoe interface as a web application (JavaScript) with a Python web server. The web client accepts an image and a JSON view hierarchy to locate interface elements. The web server queries a trained model, hosted via a Docker container with the Tensorflow model serving API, to retrieve the predictions for each element.

5.5 Informal Designer Evaluation

To understand how the TapShoe interface and tappability model would be useful in a real design context, I conducted informal design walkthroughs with 7 professional interface designers at a large technology company. The designers worked on design teams for three different products. I demonstrated TapShoe to them and collected informal feedback on the idea of getting predictions from the tappability model and on the TapShoe interface for helping app designers identify tappability mismatches. I also asked them to envision new ways they could use the tappability prediction model beyond the functionality of the TapShoe interface. The designers responded positively to the use of the tappability model and TapShoe interface, and gave several directions to improve the tool. Particularly, the following themes have emerged.

5.5.1 Visualizing Probabilities

The designers saw high potential in being able to get a tappability probability score for their interface elements. Currently, the TapShoe interface displays only probabilities for elements with a mismatch based on the threshold set by the sensitivity slider. However, several of the designers mentioned that they would want to *see the scores for all the elements*. This could give them a quick glance at the tappability of their designs as a whole. Presenting this information in a *heatmap that adjusts the colors based on the tappability scores* could help them compare the

relative level of tappability of each element. This would allow them to deeply examine and compare interface elements for which tappability signifiers are having an impact.

The designers also mentioned that sometimes, they do not necessarily aim for tappability to be completely binary. Tappability could be aimed to be higher or lower along a continuous scale depending on an element's importance. In an interface with a primary action and a secondary action, they would be more concerned that people perceive the primary action as tappable than the secondary action.

5.5.2 Exploring Variations

The designers also pointed out the potential of the tappability model for helping them systematically explore variations. TapShoe's interface only allows a designer to upload a single screen. However, the designers envisioned an interface to allow them to upload and compare multiple versions of their designs to systematically change signifiers and observe how they impact the model's prediction. This could help them discover new design principles to make interface elements look more or less tappable. It could also help them compare more granular changes at an element level, such as different versions of a button design. As context within a design can also affect an element's tappability, they would want to move elements around and change contextual design attributes to have a more thorough understanding of how context affects tappability. Currently, the only way for them to have this information is to conduct a large tappability study, which limits them to trying out a small number of design changes at a time. Having the tappability model output could greatly expand their current capabilities for exploring design changes that may affect tappability.

5.5.3 Model Extension and Accuracy

Several designers wondered whether the model could extend to other platforms. For example, their design for desktop or web interfaces could benefit from this type of model. Additionally, they have collected data that the tappability model could already use for training. I believe

the model we created could help them in this case as it would be simple to extend to other platforms or to use existing tappability data for training.

I also asked the designers about how they feel about the accuracy of our tappability model. The designers already thought that the model could be useful in its current state even for helping them understand the relative tappability of different elements. Providing a confidence interval for the prediction could aid in giving them more trust in the prediction.

5.6 Discussion & Conclusion

The tappability model achieves good accuracy at predicting tappable and not-tappable interface elements and designers gave positive feedback on the TapShoe model and tool. Here I discuss the limitations and directions for future work.

One limitation is that the TapShoe interface, as a proof-of-concept, demonstrates only one of many potential uses for the tappability model. Future work could explore building a more complete design analytics tool based on designers' suggestions and conduct further studies of the tool by following its use in a real design project.

Particularly, the TapShoe interface could be updated to take early stage prototypes created in interface prototyping tools (e.g., Sketch, Adobe XD), rather than interface screens that have a view hierarchy. Additionally, rather than having the TapShoe interface outside of a designer's main prototyping tool, it could be built as an extension or feature inside their tool. Prior work [180] suggests that this is an important feature in integrating computational models of usability or aesthetics into designers workflows.

Currently, I trained our tappability model on Android interfaces and therefore the results may not generalize well to other platforms. However, the model relies on general features available in many interface platforms (e.g., element bounding boxes and types). If a similar design dataset could be collected for other platforms (e.g., iOS, Sketch design documents), it would be feasible to collect a tappability dataset for these platforms to train the model, and the cost for crowdsourcing labeling is relatively small. In fact, researchers could potentially apply a similar approach to new interface styles that involve drastically different design concepts (e.g.,

emerging UI styles in AR/VR).

From the consistency evaluation, I learned that people's perception of tappability is not always consistent. Future work could explore ways to improve the model's performance with inconsistent data. These methods could extend the tappability annotation task beyond a simple binary rating of tappable versus not-tappable to a rating that incorporates uncertainty (e.g., adding a "Not sure" option or a scale of confidence in labels).

The tappability model that we developed is a first step towards modeling tappability. There may potentially be other features that could add predictive power to the model. As we begin to understand more of the features that people use to determine which elements are tappable and not tappable, we can incorporate these new features into a deep learning model as long as they are manifested in the data. For example, the tappability model presented in this chapter used the Type feature as a way to account for learned conventions (i.e., the behavior that users have learned over time). As users are not making a tappable decision solely based on the visual properties of the current screen, future work could explore more features that can capture user background.

Lastly, identifying the reasons behind tappable or not-tappable perception could potentially enable interfaces to offer recommendations for a fix. This would also require communicating these reasons with the designer in a human-understandable fashion. There are two approaches to pursue this. One is to analyze how the model relies on each feature, although understanding the behavior of a deep learning model is challenging and it is an active area in the deep learning field. The other approach is to train the model to recognize the human reasoning behind their selection. Progress in this direction will allow a tool to provide a more complete and useful output to the designers.

5.7 Contributions

This work was published at CHI 2019 as ***Modeling Mobile Interface Tappability Using Crowdsourcing and Deep Learning*** [201] with co-author Yang Li at Google Research. I conducted initial interviews with interface designers at Google, and came up with the initial idea to model and

predict tappability. I developed the model architecture collaboratively with Yang Li, and I implemented the crowdsourcing interfaces, deep learning model code, and the TapShoe interface. I developed and conducted the design walkthroughs and interviews with interface designers. You can watch the demo video for this project at http://doi.acm.org/10.1145/3290605.3300305 under "Source Materials".

Chapter 6

Prototyping Input Retargeting for Web Interfaces

Throughout the design process, designers need to consider the needs of people with highly diverse abilities and expertise. Factors like age [90], gender [190], culture [175], and physical abilities [70] can all impact peoples' interactions with interfaces. It is important for designers and developers to consider these needs when creating interfaces. For web interfaces, this is especially important as people rely on them for critical information.

People often interact with web interfaces in ways that designers may not anticipate. For example, many blind and low vision people use a screen reader to interact with a web page [40]. However, a web page needs to be instrumented with ARIA attributes [21] in order for a screen reader to be able to interpret its content. Despite the existence of guidelines for applying these attributes [105], 60% of screen reader users stated that web content became less accessible or did not change in 2018. These guidelines are also far from comprehensive. One study [173] found that only 50% of the problems blind web users encountered were covered by the success criteria in these guidelines. On a study of government and high-traffic websites, most did not even implement the guidelines that are helpful [81].

Alternatively, people with motor impairments may be able to see a web page, but their limited ability to use a mouse may leave interactive parts of the web almost impossible to use [70]. These people can benefit from customized interfaces created for their unique physical capabilities [74]. However, such techniques are rarely deployed in web interfaces.

A persons culture, gender, and age can also impact their interactions with websites. What interfaces people perceive as usable often depends on their cultural background [175]. By making culturally adaptive interfaces, Reinecke et al. [175] saw that people's task performance increased by 22%. A person's gender can influence the appeal and trustworthiness of a website [190], and it can also impact how usable an interface is to them [46]. A person's age can also affect perception, hearing, cognitive, and motor abilities, requiring new interface designs [90]. Despite the existence of factors impacting interaction like gender, age, and culture, most web applications continue to be one-size-fits-all.

Given the challenges of designing for people with diverse abilities, researchers and designers have explored a variety of application-agnostic tools for adapting existing interfaces [51, 62, 68, 221, 228, 231]. Such tools can enable designers, researchers, and developers to *prototype* new interactions on top of existing interfaces. For example, Prefab [62] enables prototyping novel interaction techniques on desktop interfaces, such as a target-aware implementation of the Bubble Cursor [79]. For mobile phones, Interaction Proxies enable prototyping accessibility repairs for mobile interfaces [229]. Gesture Avatar enables prototyping new gesture-based interactions for arbitrary phone apps [118]. On the web, people can use scripting languages [41, 131] to interact with and automate tasks and repair accessibility issues [39, 95], and can implement web plugins to modify webpage behavior [15, 18].

Although prior work demonstrates the power of systems that enable prototyping new interactions on top of existing interfaces, these techniques have fundamental limitations. First, they often rely on modifying an application through *understanding and interacting with an interface's visible features* (i.e., interface elements). Interfaces can contain many behaviors that cannot be discovered from visible features alone, including keyboard events, touch gestures, and custom commands. On the web, this is particularly true, where websites can use a vast array

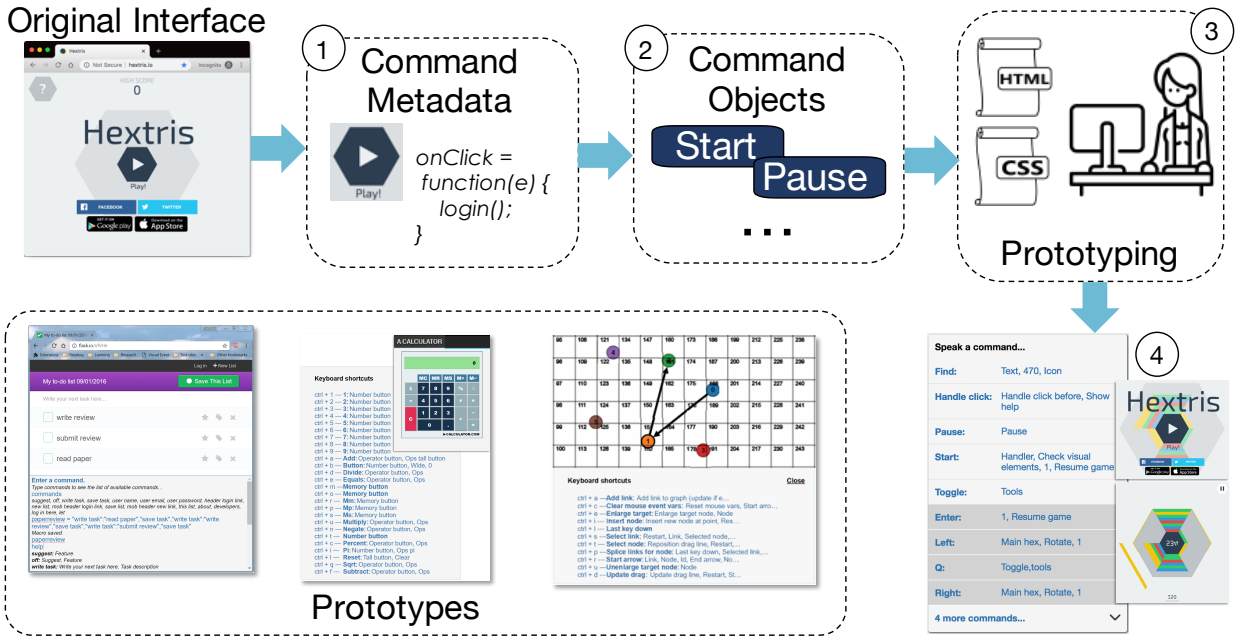


Figure 6.1: Genie uses program analysis techniques to reverse engineer a web application's interactive commands (1-2). Designers can create interaction prototypes (3) to prototype new interactions with input modalities for existing web applications. With Genie, I created several application-agnostic prototypes that automatically retarget input to add speech, keyboard, and command-line input capabilities to arbitrary web applications.

of different user interface toolkits and custom controls. Second, web interface customization toolkits and scripting languages typically *modify a specific website or subset of websites*, by *modifying a specific aspect of behavior* or adding a new functionality.

In this chapter, I present Genie, a system and approach to automatically derive an abstract model of a web application's interactive commands and enable designers to prototype with these commands to transform interfaces to support new modalities (e.g., speech, keyboard, command line input). Genie, shown in Figure 6.1, uses program analysis techniques from software engineering to reverse engineer application commands and aspects of *non-visible behavior* from source code. Genie then enables designers to create interaction prototypes to retarget the inputs of *arbitrary* web applications to new modalities (e.g., voice interface for a mouse-based website).

I explore Genie's approach on the web, where reverse engineering is simplified due to the

Figure 6.2: The Hextris web game (hextris.io) shown with a list of speech commands created by Genie. Speaking the bolded text label for each command triggers the corresponding actions.

open access to interface structure in the DOM and the source code that handles inputs. Figure 6.2 shows an example web game that originally supported only mouse and keyboard input. Genie recovers the application's underlying commands and provides an application-agnostic speech interface to access the game's commands. Speaking the command label "Left" (a command label derived by Genie's analysis) rotates the hexagon left, performing the same functionality as clicking the left mouse button or typing the left arrow key. This *input retargeting* approach allows a web application designed to support one input modality to be mapped onto any other input modality, ensuring that users can access any website according to their diverse needs. The contributions of this chapter are:

- An application-agnostic abstract model of interactive commands and their properties.
- A method for reverse engineering these commands from an existing web interface into this abstract representation.
- An API that exposes a website's commands, which can be used to prototype interfaces that support other input modalities.

Available: Dependent on the *enabled* and *visible* state.
Enabled: Command is *enabled* if DOM element disabled attribute is not set, and current conditions in its event handler will produce at least one side effect. `True` if at least one *data dependency* with at least one *side effect* is currently satisfied.
Visible: The command element is visible on the screen.
Data Dependency: A condition in an event listener that can be evaluated to have the value of either *true* or *false* and is associated with at least one side effect.
Side Effect: A statement in an event listener that has an effect on the system when executed.
Device Dependencies: A list of commands that must be performed before (*pre*) or after (*post*) a command based on the implicit device requirements.
Required Input: A list of required inputs to the command. Possibilities are *mouse location*, *mouse button*, *key code*, and *value*. Knowing each required input value allows Genie to generate them automatically, or request them from users, when required.
Perform: A method to trigger the command, supply the *required input* values, and trigger commands corresponding to the *pre* and *post device dependencies*.
Label: Genie provides command labels, extracted from event listener source code, and additional command metadata to describe commands in a new interface.

Figure 6.3: Genie abstract data model properties, metadata, and behaviors.

- Several application-agnostic interface prototypes that automatically retarget input to add speech, keyboard, and command-line input capabilities to arbitrary web applications.

6.1 Architecture & Implementation

Genie models web application user interfaces as a set of available *commands*. Each command has a set of *properties* (shown in Figure 6.3), which represent command availability, dependencies, and other metadata. Genie periodically refreshes commands and their properties as the state of the interface changes in response to user interaction or other application background activity. Genie also provides generic support for executing each command, allowing the implementation of alternative interfaces that support other input modalities.

Genie is implemented as a Google Chrome extension that accesses the web page DOM, event registrations, and source code. Genie consists of five algorithms to (1) *detect* commands, (2) *filter* commands to those that are directly executable, (3) *analyze* properties of commands to update their status in an interface, (4) *describe* commands to obtain appropriate labels for an interface, and (5) *invoke* commands by recreating their inputs and event sequences. The

following sections describe each algorithm and how they discover and update the properties and behaviors of the Genie data model.

6.1.1 Command Detection

The Genie system interposes application event registration to detect commands. Genie assumes graphical user interfaces consist of graphical elements that make up the interface, events that are triggered by input devices performing various actions on elements (e.g., `click`, `keydown`), and *event listeners* that receive and respond to events. This event subscription model is the dominant way of receiving and responding to input in modern user interface frameworks. Web interfaces also consist of a set of default *interactive elements* such as links (i.e., `<a>`) or input fields. Genie maps each *interactive element* and registered *event listener* to a unique command.

Genie detects commands by intercepting each event listener registration as it occurs. The default DOM APIs, as described by the W3C (www.w3.org), do not provide a method of accessing all currently subscribed event listeners in the DOM. Genie therefore monitors each registered event listener by overriding the default DOM API for `addEventListener`. This override notifies Genie that a new listener has been registered, and Genie calls the original `addEventListener` method to register the listener with the browser. Event listeners can also be registered using a secondary library (e.g., jQuery, D3). These libraries wrap the `addEventListener` method, so intercepting `addEventListener` calls is sufficient to capture event listeners registered using these libraries. The override is achieved by injecting a script into the original page before DOM initialization so that all registered event listeners are intercepted. The script then intercepts all event listeners registered after DOM initialization to keep a currently updated list of events, each of which corresponds to a new *command*.

The default DOM APIs provide a second method of registering event listeners through global attributes. These listeners are registered in two ways. One is through an inline attribute on the DOM element in the format `onclick="listenerName()"` where the attribute name can

be `onclick` or any of the supported event types [1]. Applications can also register global event listeners using the format `element.onclick=listenerName`. Genie locates and detects all registered global event listeners at document initialization and monitors any updates to them using the MutationObserver API [2], which notifies Genie of any updates to element attributes in the DOM. Genie also collects and monitors *interactive elements* (e.g., `<input>`, `<a>`) using the MutationObserver API.

6.1.2 Command Filtering

There are hundreds of events that can be triggered in a browser. A small subset of these events correspond to interactive actions that a user can trigger by clicking, touching, or typing a key. Genie distinguishes between events that can be triggered by a user action and those that happen as side effects of user actions or are triggered by the system.

Genie categorizes events into three groups: *direct*, *indirect*, and *system*. *Direct* events are those that can be triggered by human actions (e.g., a mouse click - `mousedown`, a key stroke - `keydown`). *Indirect* events happen as a side-effect of triggering *direct* events (e.g., an element gains focus - `focusin`, a field loses focus - `blur`). *System* events are events that are triggered without direct user intervention, but occur during interactive use of the system (e.g., a resource failed to load - `error`, a resource has finished loading - `load`). Genie collects each event listener registered to each *direct* event as a command and filters out all event listeners registered to *system* events. Genie monitors *indirect* event listener registrations but does not expose them as commands, instead using them in the *invoke* module which I will describe later. The output of the command filtering process is: (1) a set of registered direct and indirect event listener and element combinations, and (2) a set of default interactive elements.

---

[1] https://html.spec.whatwg.org/multipage/dom.html#global-attributes

[2] https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver

6.1.3 Command Property Analysis

User interfaces can change at any time in response to user interactions, events, and system status. To give an accurate picture of what a person can do at any given point in time, and to prevent needless interactions with disabled commands, Genie analyzes and monitors the visibility and enabled state of each command, keeping the *visible* and *enabled* states, defined in Table 6.3, current with the original user interface.

To discover the value of these properties, for each intercepted event listener registration, Genie captures the event type, event listener source code, and the DOM element the listener is associated with. For the `visible` property, Genie queries the DOM element for its visibility through a set of properties that can hide any element, such as setting the `display` attribute to `none` or other standard methods of hiding elements. Elements can also be off-screen or opaque, and Genie uses existing DOM APIs to inspect these forms of visibility.

A command can be *disabled* (`enabled=false`) in two ways. First, a DOM element can set the `disabled` attribute. However, this attribute is not required, so it is possible that an element will appear interactive even if current conditions in its event listener prevent it from having effect. The second way a command can be *disabled* is if the conditions in its event listener that result in *side effects* are not currently satisfied. I define each condition as a *data dependency* and each method call, state update, event, or other response in an event listener as a *side effect*. Each event listener has one or more *data dependencies*, and each *data dependency* can have one or more *side effects*. If none of the *data dependencies* of a listener are currently satisfied, a command will have no effect, and Genie will mark the command's `enabled` state as `false`.

Genie discovers *data dependencies* by analyzing the event listener source code, shown in Figure 6.4. First, Genie parses the event listener to build an abstract syntax tree (AST). Genie traverses it to build an expression for each data dependency. Genie computes data dependency expressions by tagging each variable node in the AST with the node where it was either previously defined or declared. Then, Genie locates each conditional (e.g., `if`, `while`). Within each conditional expression (i.e., the expression that must be true to reach the inside of the

Input: AST for handleStartGame()

```
function handleStartGame(e) {
  var startBtn = $("#startBtn");
  if(e.button == 0
    && !startBtn.attr("disabled")) {
    startBtn.attr("disabled",false);
    // Start the game
    // If there are lives remaining
    if(livesRemaining > 0) {
      startGame();
    }
  }
}
```

Output: data dependencies (DD) & side effects (SE)

DD: `!$('startBtn'.attr('disabled') && livesRemaining > 0`
SE: `startBtn.attr('disabled', false), startGame()`
DD: `!$('startBtn').attr('disabled') && !(livesRemaining > 0)`
SE: `startBtn.attr('disabled', false)`
DD: `!$('startBtn').attr('disabled')`
SE: None

Figure 6.4: Genie parses the event listener source code to extract data dependency (DD) and side effect (SE) expressions, used to evaluate a command's availability (`enabled`). Genie extracts imperative statements and command metadata to describe and label commands.

conditional block), Genie finds each variable identifier and replaces it first with where it was last assigned, and secondly, where it was last declared if it does not find other assignments besides the variable's declaration. This process continues recursively until Genie replaces all identifiers in a conditional expression with their corresponding assignments or declarations.

Figure 6.4 shows an event listener that handles clicking on a game's start button. There are three possible *data dependencies* corresponding to the three potential paths through the code, labeled by "DD" under output. These expressions depend on the current state of the Start button and the number of lives remaining. The first expression is constructed by combining the first `if` conditional with the second nested `if` conditional. The `startBtn` reference in the first conditional is resolved to the value `$("#startBtn")` based on where it was last assigned in the listener, which occurs on the previous line.

Each data dependency is associated with one or more *side effects*. Identifying side effects is important for interpreting and displaying to a person what effects the command will have when it is executed. In Figure 6.4, `startGame()` is a side effect. Each line of code that would be executed if a data dependency expression is satisfied is a potential side effect. This method of side effect detection is potentially accurate if each method call, state update, or response

Input: AST for `rotateHexagonLeft`

```
// Rotate the hexagon left
function rotateHexagonLeft() {
  if(MainHex && gameState !== 0) {
    // Rotate hexagon left
    MainHex.rotate(1);
    // Update the position counter
    MainHex.hexagonPosition--;
  }
}
```

Output: Rotate the hexagon left,
rotate hexagon left, rotate,
update the position counter, hexagon position

Figure 6.5: Inputs and outputs of Genie's command description algorithm for the `rotateHexagonLeft` event listener.

is causing an update to the state of the system, affecting future interactions. However, this may not always be the case. To improve this method, Genie could potentially identify output affecting statements (e.g., as in [108]). It could then link each method call or state update to its last assigned location outside of the event listener, linking it to its origin in the website's source code. These could be searched for output affecting statements, or Ajax calls that cause data changes. However, this method would mostly be an approximation.

A command is disabled (`enabled=false`) if none of its data dependencies are currently satisfied, if none of its data dependencies have side effects, and if it has no side effects outside of conditional expressions. For example, Figure 6.2 shows a set of four disabled commands: *Enter*, *Left*, *Q*, and *Right*. This event listener for the *Left* command, shown in Figure 6.5, consists of only a single data dependency (`MainHex && gameState !== 0`), and two side effects (`MainHex.rotate(1)`, `MainHex.hexagonPosition`——). Before the game starts, the value of `gameState` is `0`. After a person starts the game, its value is `1`, resulting in this expression evaluating to `true`, and allowing the side effects to occur when the event listener is called.

6.1.4 Command Monitoring

After reverse engineering each command's data dependencies and side effects, Genie evaluates them to determine each command's availability. This process runs in an *update service* which calls the JavaScript method `eval` to evaluate each data dependency in a global scope. The results of `eval` are sent to each Genie interface through the update service. For example, the speech interface shown in Figure 6.2 updates the status in the interface by giving the disabled commands a grey color. Genie's analysis currently only resolves data dependencies in the local event listener scope (excluding expressions defined outside this scope such as `MainHex`). This is because it is not possible to access the closure of registered event listeners to recreate the scoping of these variables. Future work could explore how to evaluate these expressions more accurately within the constraints of JavaScript scoping.

Genie's update service runs every second, evaluating the enabled and visible states of each command as a person interacts with the system, providing them with timely feedback about command availability.

6.1.5 Describing Commands

Alternative interfaces that display available commands need some way to describe the commands so that people know what effect each command will have before they invoke it. For example, Genie derives and displays command labels in the speech interface shown in Figure 6.2, giving each command a name and basic description of its effects.

To derive these labels, Genie identifies labels from command metadata by searching for natural language phrases starting with an imperative verb followed by a noun. For example, the phrases in Figure 6.2 include "Resume game" or "Show help". If a command label cannot be found in the "*verb noun*" format, Genie searches each metadata string for verbs and nouns to use as labels, if they can be found.

Genie collects command metadata from two sources: *element* and *listener* metadata. *Element* metadata comes directly from the attributes of a command's DOM element. *Global* attributes

are those common to all types of DOM elements, which include `title`, `id`, and `class`. These three attributes frequently and conventionally contain semantic metadata to describe the object or concept that a DOM element represents. `<input>` elements have an additional set of commonly useful attributes: `placeholder`, `alt`, and `value`. Another subset of elements, `<input>`, `<button>`, `<fieldset>`, `<textarea>`, and `<select>` have a name attribute. `<a>` elements have an `href` attribute that also frequently contains descriptive metadata.

Listener metadata comes from command event listener source code. Genie analyzes four sources of listener metadata: (1) comments on the event listener, (2) the event listener name, (3) comments on expression calls and assignments (i.e., side effects), and (4) the expression calls and assignments. To parse these sources, the algorithm splits each assignment, function call, and function name into separate identifiers (e.g., each token in Figure 6.5, such as `MainHex` and `hexagonPosition`), parsing each identifier and attribute value separately and identifying them as either a phrase or an individual word.

Genie identifies phrases from both element and listener metadata by splitting on common identifier conventions (e.g., camel casing, underscores, dashes). A part-of-speech tagger tags each sentence, phrase, or individual word, while the system discards non-English strings using an open source spell checker library [3]. The algorithm then searches each string for the first verb, followed by the first noun. If both can be found, the system generates a *command label*.

Genie uses command labels to uniquely identify and trigger commands. The system prioritizes imperative phrases and verbs to use as these labels, but if none can be found will fall back to remaining metadata. Developers can use the remaining metadata to provide a more detailed description of the command to be shown in an interface, such as the one shown in Figure 6.6. This remaining metadata is labeled as *description metadata*. Thus, the two outputs of this process are a command label and description metadata. In Figure 6.6, the multiplication command has a command label *Multiply* and description metadata of *Operator button* and *Ops*.

[3] https://github.com/cfinke/Typo.js

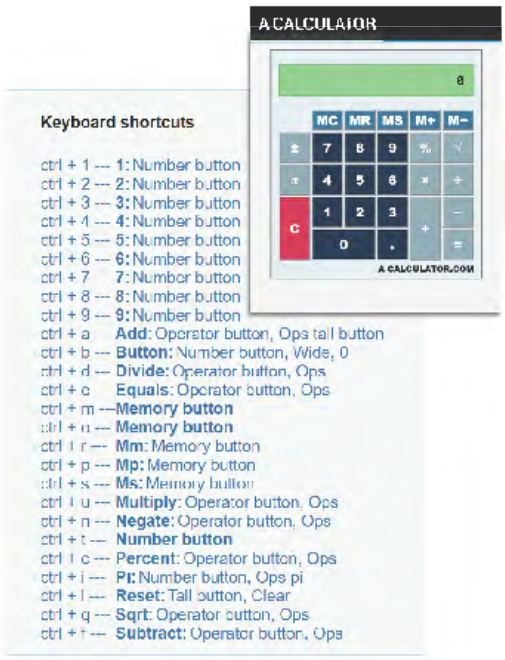


Figure 6.6: A calculator interface with incomplete keyboard support (a-calculator.com), enhanced to provide a keyboard shortcut for each command, as enabled by Genie's analyses.

6.1.6 Invoking Commands

Performing commands through an alternate interface requires some way of triggering the original functionality through the new interface. The JavaScript DOM API provides a `dispatchEvent` method for creating and triggering custom events. Genie uses this API to allow Genie interfaces to dispatch events to the original interface, creating a new `Event` object with the necessary inputs, and triggering them through the `dispatchEvent` method defined in the `EventTarget` DOM API.

However, Genie cannot simply trigger only the event corresponding to the command. Web interfaces expect one or more sequences of events to be triggered by a motor action (e.g., clicking the left mouse button, typing a key). For example, a `mouseup` event and a `mousedown` event must be triggered before the `click` event is triggered, as an application's semantics may depend on these events occurring. I define these event ordering requirements as *device dependency* events (see definition in Figure 6.3). Each command has a list of *pre* device dependencies and *post* device dependencies describing events that need to be triggered before and after the event,

Input: mouseMoveHandler AST

```
function mouseMoveHandler(e) {
   var relativeX = e.clientX;
   relativeX = relativeX - canvas.offsetLeft;
   if(relativeX > 0
      && relativeX < canvas.width){
        paddleX = relativeX - paddleWidth/2;
   }
}
```

Output: True - Command dependent on mouse position

Figure 6.7: This event listener references the `clientX` property of the event object. This is stored in the variable `relativeX` which is referenced in the conditional statement, which guards a side effect. Genie detects these dependencies and determines that the command is dependent upon mouse location

in the correct order. Device dependencies consist of both *direct* events, such as `mouseup` and `mousedown`, and *indirect* events, such as `blur` and `focus`. When a Genie interface requests to perform a command, Genie executes pre and post device dependencies before and after a command in the correct order if there are commands corresponding to those events.

Many events also require additional input that comes from device specific input, like mouse location or key code. Genie discovers these dependencies through a command's event listener. To do this, Genie traverses the AST of the event listener to locate mouse location and keyboard dependencies. Event listeners typically reference device location through properties on the event object (e.g., `evt.clientX`, `evt.clientY`, `evt.x`, and `evt.y`). Genie's algorithm searches the AST for references to these properties, typically in assignment or conditional expressions. For example, Figure 6.7 shows that `mouseMoveHandler()` references the `clientX` property of the event object and stores it in the `relativeX` variable. The conditional test expression then references the variable `relativeX`. Genie transitively detects any dependencies that it can statically determine will effect the control flow through the event listener.

Genie detects keyboard dependencies similarly (Figure 6.8), looking for references in the AST to `code`, `key`, or `keyCode` properties on the event object. If the key code of the event

Input: keyDownHandler AST

```
function keyDownHandler(e) {
   if(e.keyCode == 13) {
    submitOnEnter();
   }
}
```

Output: `KeyCode = 13`, `submitOnEnter()`

Figure 6.8: This event listener references the `keyCode` property of the event object and compares it to the value. Genie returns the value `13` and the corresponding side effect.

is assigned to a variable that is referenced on a conditional expression, or if the key code is referenced directly, Genie collects the corresponding value that the `keyCode` is compared to (e.g., `if(evt.keyCode == 13)`). If the `keyCode` value is not hard-coded, Genie transitively determines the value, if possible.

As Table 6.3 shows, each command has a *required input* property. Genie adds each key code value (e.g., `13`) it discovers to the *required input* list. If Genie cannot find a possible value for a key code reference, in cases where the key code value is assigned to a global variable or variable declared outside the function, and not referenced later in a conditional, it does not add a value to *required input* because it cannot determine that value statically. Genie detects mouse button dependencies by traversing references to the `button` and `buttons` properties of the event object, and collects the corresponding values in a similar manner to `keyCode`.

Each key code or mouse button value in the required input property is mapped to one or more side effects. When a command has multiple required input values, Genie splits the command into multiple *pseudo-commands*, where each required input value is a command, has a command label corresponding to its input value or side effect metadata, and has a set of side effects that will occur if the command is given that specific input. Figure 6.2 is an example where the commands "*Left*" and "*Right*" originate from the same event listener. In the Genie system, a pseudo-command is represented in the same way as a regular command.

Each Genie command also has a *perform* method that triggers the command, supplying the *required input* and *pre* and *post device dependencies*. For a pseudo-command, Genie supplies the

associated *required input* value.

6.1.7 Genie API

To support designers and developers in prototyping applications with Genie models, I built an API that exposes any web page's current set of commands, properties, and behaviors. An application prototyper can build a Genie interface which subscribes to an abstract list of commands that Genie keeps up to date. Genie notifies each interface when the state of a property (e.g., `visible`, `enabled`) has changed, or when a command is added or removed.

To prototype a Genie interface, the prototyper needs to create a visual representation to display for each command (i.e., HTML structure and CSS), and a small amount of JavaScript code that defines how to update an interface when a command's enabled or visible status changes. The developer would also need to implement generic command triggers for each command, which may require the use of an API to receive input (e.g., WebSpeech API [4]).

Genie will then invoke these commands automatically when a user interacts with the Genie prototype. Genie also provides each interface a set of automatically inferred labels for a command, including a command trigger label unique to each command (e.g., the bolded labels shown in Figure 6.2) and an additional set of labeling metadata. The framework is meant to be simple, only requiring the developer of the Genie interface to implement the behaviors required for receiving command inputs and visualizing a command in alternate ways if desired.

6.2 Interface Prototypes & Use Cases

The benefit of having an abstract model of application commands is that designers and developers can easily use Genie to enable web interface prototypes to support a range of input types, without having to design built-in support. I validated Genie by building several diverse applications showcasing these prototypes of new interactions for existing web pages. I motivate them through their potential for making the web more powerful and accessible. I demonstrate each

[4] https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API/

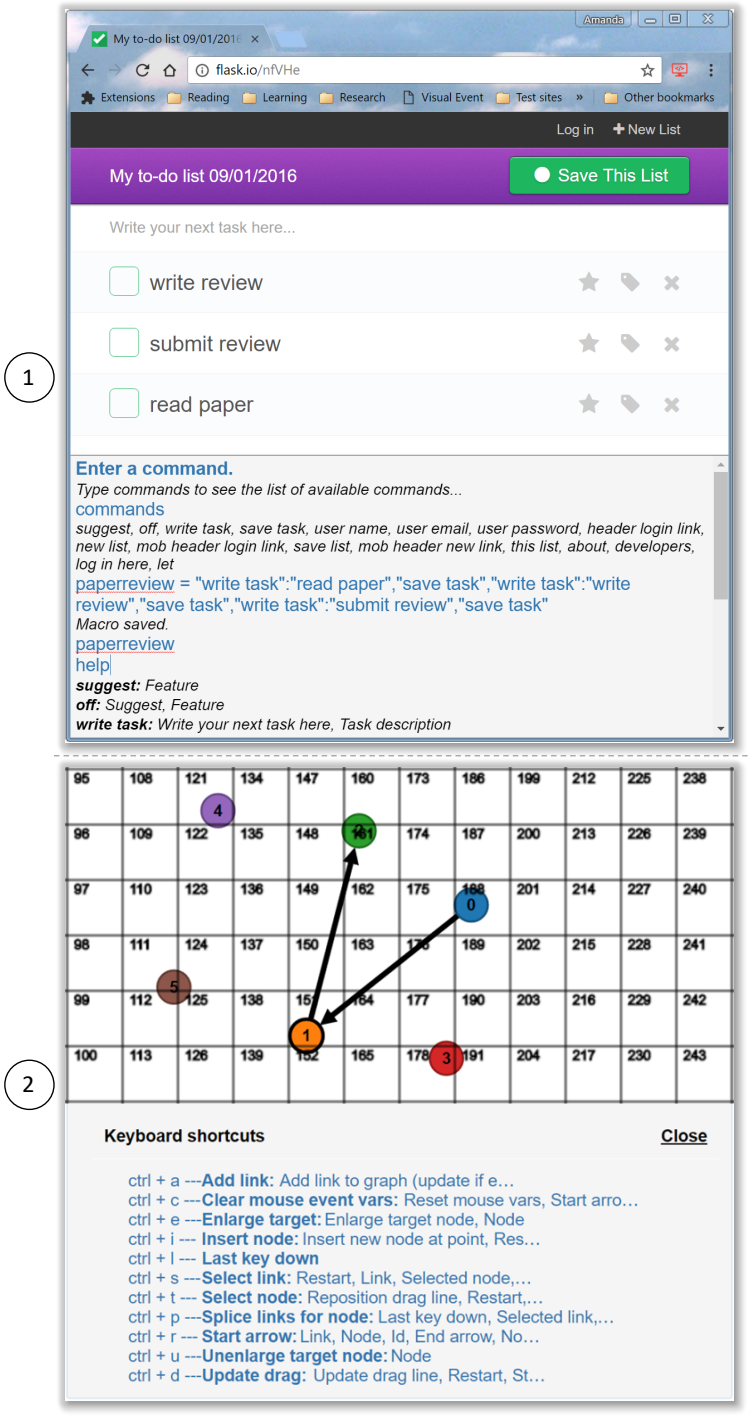


Figure 6.9: (1) A Genie-enabled command line terminal that allows command automation and macro creation, and (2) a graph builder augmented with Genie's input grid for capturing mouse coordinates via keyboard

of these applications on a single website, but the applications themselves are generic and are meant to be able to be applied on any website.

6.2.1 Automatic Speech Input

Many people have severe motor impairments that make using a mouse or physical keyboard almost impossible [70]. However, many people with motor impairments can use speech interfaces. Using the Genie API, I built the speech interface shown in Figure 6.2. The Genie interface displays a list of currently available commands on the page. People can trigger any of the commands by simply speaking the label shown in bold. Commands shown in dark grey are currently disabled and cannot be triggered. Genie monitors and updates the states of these commands as the user interacts with the web page. I implemented this interface using the Genie API to define the interface structure and styles and to integrate the Web Speech API to process speech input. I map each speech input keyword to its corresponding Genie command, if there is one. I discard spoken keywords that do not correspond to a Genie command. This only required about 150 lines of JavaScript code.

The web site shown in Figure 6.2 demonstrates the speech interface active on a game called Hextris, which consists of a rotating hexagon. The objective of the game is to prevent blocks from leaving the outside of the gray hexagon. The two main commands to play the game are speaking "Left" and "Right" to rotate the hexagon in either direction. Genie allows anyone to play this game via speech, in addition to using the built-in mouse and keyboard commands.

6.2.2 Automatically Generated Keyboard Shortcuts

Most websites only support mouse input, or selectively implement support for a small set of keyboard accelerators. For example, the calculator shown in Figure 6.6 is a basic calculator with simple number, operator (e.g., +, *), and memory functions (e.g., MR, MC). It has keyboard shortcuts for numbers, but not for operators or memory functions, rendering the calculator useless without a mouse. Such applications are less accessible to people who do not use a

mouse, such as people with severe motor impairments or people who use screen readers.

I used Genie to build the interface prototype shown in Figure 6.6. This interface displays a list of commands and a corresponding automatically generated keyboard shortcut for each command. Implementing this interface required defining shortcut triggers using the Keypress API[5] and defining a function to generate a shortcut for each command. I defined a simple method to define shortcuts using the first letter of the command label and the modifier `ctrl`. If a shortcut is already used, the interface assigns the second letter as the shortcut, and so on. Each keyboard shortcut has a unique command label and a corresponding metadata description. This required about 130 lines of code.

Figure 6.6 shows the calculator interface with the Genie interface active. This interface can be used on any website to activate automatically generated shortcuts, so the calculator interface is used here as just an example. The interface displays a shortcut that can be used to trigger each number, operator, and memory command. Number commands contain the metadata "number button", operators have the metadata "operator button", and memory buttons have the metadata "memory button", along with other collected metadata, shown by each command description. For this interface, the labels happen to originate from the referenced listener name in the `onclick` attribute which for each of the buttons has the value `onclick="operatorButton('+')"`.

6.2.3 A Command Line Interface for Web Automation

Most web applications are not scriptable. Many web forms require painstaking input needing a skilled programmer to automate. Web automation tools such as CoScripter [131] and Chickenfoot [41] provided powerful solutions to this problem. Genie can easily recreate such functionality, providing a command line scripting console for arbitrary web applications.

Figure 6.9.1 shows the example command-line prototype active for a simple to-do list application (*flask.io*). This application allows people to create to-do lists, save to-do lists to a profile, and share tasks with other people. Typing "commands" into the terminal presents the

[5]http://dmauro.github.io/Keypress/

user with a list of the available commands discovered by Genie. Commands for the application shown include "write task" which corresponds to typing a value into the field labeled "Write your next task here", and "save task" which saves the task to the interface. Typing "help" into the terminal displays the list of command triggers along with more detailed descriptions from the collected command metadata. The command-line interface prototype supports macro creation allowing for automation of multiple commands and inputs. The user creates a macro using the following format.

```
<macroName>="<commandName1>":"<commandInput1 (optional)>",
            "<commandName2>"...
```

The **paperReview** macro in Figure 6.9.1 creates and saves three tasks to the list: *read paper*, *write review*, and *submit review*. These are three hypothetical tasks that a reviewer might create to remember to complete all steps of submitting a paper review. The macro could potentially be persisted across sessions so that adding subsequent paper reviews would require simply typing the command **paperReview**.

6.2.4 Keyboard-Based Mouse Input

Many people cannot use a mouse to interact with the web, preventing them from using applications that rely on fine grained pointer input such as drawing or diagramming tools [70,97]. With Genie, designers and developers can easily prototype new interactions for supplying pointer input using the keyboard.

Figure 6.9.2 shows an automatic keyboard-based mouse input application I built currently active on a graph drawing application. Typing **ctrl-i** triggers the command "Insert node" as labeled in the figure. Triggering the "Insert node" command displays a grid covering the surface of the canvas. Typing in the numbers corresponding to the desired cell generates a mouse location used as input to Genie when the command is triggered. Implementing this Genie prototype using the framework simply required implementing the method of inputting

coordinates, integrating the Canvas API to draw the grid, and registering `keydown` listeners to process location input. In all, this interface required only about 75 lines of JavaScript.

This method of input, while not having the precision of clicking a mouse on the canvas, does provide a method of entering this input. This prototype could easily be extended with more accurate methods of input, such as an onscreen cursor that could be moved with keyboard or voice commands, it could also work with other input devices. For example, people could speak the cell numbers to input the location or use the command line interface that I describe in the next section. The prototyper of each Genie interface would need to create this mapping, but as I have shown, such mappings require very little implementation work.

6.3 Limitations

Many of Genie's limitations are due to limited availability of metadata in an application's source code. For instance, a key limitation of Genie's command labels and the descriptions Genie discovers is that many websites use "minification" to improve performance and obfuscate code, limiting the information Genie can extract and present in alternate interfaces. Even if a site is not minified, many websites do not include descriptors in the comments or source code of their event listeners or on their command elements. Even if the information exposed by Genie is not detailed or high quality, the ability of Genie to at least expose what actions are possible may still be useful. Better metadata might also be attached through social annotation techniques [103], applied to the original interface or to Genie's representation.

Other limitations are due to imprecision in the program analyses that I used in the Genie prototype. For example, Genie only analyzes the functions registered as event listeners, and not the functions they transitively call. Genie could have done a full program analysis, tracking the full extent of downstream side effects following a function call, potentially discovering a more precise set of command states (e.g., enabled, disabled), side effects, and descriptions. Not having these precise states might mean that a command label is not descriptive enough, or that a command that is currently disabled is shown as enabled. However, triggering the command through Genie will have the same behavior as triggering the command through

the regular interface, and relevant feedback of the disabled command will still be presented. Discovering more precise command states is a matter of applying more advanced program analysis techniques from prior work in software engineering, but it was outside the scope of the Genie prototype.

Another limitation I discovered with larger websites was that Genie discovered large numbers of commands that made it difficult to discern which command triggered which functionality. Future work should explore reverse engineering more metadata to help organize and group relevant commands together so that they are more discoverable.

6.4 Discussion

This chapter has presented Genie, a framework to reverse engineer the interactive commands from a website, retargeting their inputs to alternate input modalities. Genie enables alternate access to a broad range of websites that were not designed for diverse abilities. By implementing a set of alternate interfaces using the Genie framework, I have shown that this approach has the potential to enable prototyping more efficient and customizable interfaces that can enhance the ability to interact with existing websites.

During the process of creating our Genie interface prototypes, I generated several additional ideas for Genie interfaces. Many of these involved ways that *input retargeting* could add support for other modalities, including touch input, brain-computer input, or any other types of future input devices. However, I also generated ideas for several other interface enhancements that go beyond input. For example, Genie might be used to automatically create a help interface that display tooltips that describe each command and its side effects. In investigating this, I found that the metadata I could collect from the real applications used in this chapter was not detailed enough to support this type of description, but would still give some indication as to the behavior of the command and could be useful in many cases. More advanced analysis of command side effects might allow systems to generate tutorials that suggest a specific sequence of commands to complete a particular task, thus providing more advanced and automated help.

Another promising use for Genie might be in adding enhanced ARIA metadata to existing

websites, which is a standard for making interactive websites screen readable. For example, the attribute `aria-disabled` indicates that an element is perceivable but not interactive. Hidden elements in a page should be marked with the attribute `aria-hidden` which indicates that the interactive element is not visible or perceivable. As the Genie data model already exposes and notifies a Genie interface when these two properties are updated for any command, it would be simple to implement a Genie interface that keeps these two properties up to date for any web interface. In fact, utilizing a combination of static and dynamic analysis has the potential of being able to monitor the state of many attributes, such as `aria-invalid`, `aria-expanded`, and others. Future work could explore extending Genie to monitor additional properties.

There are a diverse set of usability issues with existing web interfaces that could also be enhanced or repaired with the metadata Genie collects. It could be possible to build a Genie interface that would automatically detect and repair usability issues in an interface. For example, previous work [108] analyzed a set of 115 web applications, and found 37% did not provide feedback to users after completing an action. Augmenting Genie to detect when a command does not provide feedback and generating customized feedback messages could be a promising application. Genie additionally analyzes the disabled state of commands, so using this information to provide reasoning to users about why a command is currently disabled, or what commands can be performed to enable it, could also be useful applications of Genie's analysis.

Future work could scale up Genie's evaluation to demonstrate the effectiveness of these techniques on a larger set of websites selected from the Alexa.com rankings. This could allow researchers to fully evaluate the benefits and limitations of Genie across a more representative set of websites, and it could also help them discover areas of improvement for future work.

6.5 Contributions

This work was published at CHI 2017 as ***Genie: Input Retargeting for the Web through Command Reverse Engineering*** [199] with co-authors James Fogarty and Amy Ko. I developed the initial idea with Amy and received feedback from both James and Amy throughout the

project to refine the ideas about the Genie framework and implementation. I implemented the code for the Genie prototype, API, and example interfaces. You can watch a demo video of Genie at http://doi.acm.org/10.1145/3025453.3025506 under "Source Materials".

Chapter 7

Future Work

This dissertation surfaces numerous opportunities for future work. Furthering the work in theses areas can enhance designers' capabilities to prototype with existing webpage interfaces, support better mixed-initiative interaction to explore alternatives, build models to predict attributes of interface usability, and extend integrated design prototyping tools.

7.0.1 Inference and Prototyping with Existing Webpage Commands

With Genie (Chapter 6), I built a few different prototypes to demonstrate how Genie could add support for new modalities (e.g., keyboard, voice). However, future work could examine the creation of prototypes for other modalities (e.g., touch input, brain-computer interface). Genie's prototypes also currently require writing JavaScript code and HTML to specify support for new input modalities, which is likely to be difficult for a less technical or novice designers. Future work should explore how to enable these prototypes to be more easily specified by a larger group of designers.

One of the most promising areas that Genie could be used is for the repair of ARIA attributes

required to make a webpage accessible (e.g., aria-disabled). Howver, such repair would likely require more accurate inference than Genie currently supports due to webpage minification and lack of detailed metadata that Genie can mine from source code. We could explore more accurate inference through social annotation techniques like [95] or machine learning techniques that can automatically infer labels for interface commands based on their visual presentation, source code, or the resulting change to the interface by performing the command. Due to the open access to webpage DOM and source code, collecting large amounts of structured data for labeling may be more possible than other platforms (i.e., mobile apps). Additionally, collecting such data to support design tasks has been fruitful for prior work [113, 114].

7.0.2 Mixed-Initiative Exploration of Alternatives

My work on Scout (Chapter 3) reveals a rich set of future directions in mixed-initiative tools for exploring alternatives. I see four main opportunities for future work revealed through my user studies with Scout: *inference of high-level constraints from design documents*, *giving designers more control*, *providing faceted exploration of alternatives*, and *expanding the language of high-level constraints*. I describe the motivations for each of these and potential directions for research in each area in the following sections.

Inference of High-Level Constraints from Design Documents

Scout is currently limited to helping designers explore alternatives in mobile interface dimensions. The nature of mobile interfaces means that the number of elements and potential high-level constraints that a designer needs to define would be manageable. However, for more complex interfaces (e.g., webpages, GUIs) defining high-level constraints across the entire interface could become unmanageable. Additionally, designers may have already created a prototype for their interface and want to redesign it by exploring alternative layouts. Currently, to use Scout, they would have to import each interface element individually and create high-level constraints on these elements in Scout. A helpful focus for future work could be

to automatically infer high-level constraints from existing wireframes and design documents created in prototyping tools. A designer could drag their entire design into Scout's outline, and Scout could automatically extract their components and infer high-level constraints from them.

Such an inference would be drastically easier if the design document already contained groups and structure. This is the case for many design prototypes, as designers are used to using groups to structure their designs. Scout could infer this grouping structure by directly matching groups in the prototype to groups in the design hierarchy. Scout could infer ordering constraints by analyzing grouped content for implicit ordering relationships through a machine learning classifier or heuristics. A designer would likely need to define which elements should be emphasized, however, this would be a much more manageable task than creating all of Scout's high-level constraints from scratch.

Giving Designers More Control

Through my Scout user studies (Chapter 3, Section 3.3), I found that designers wanted more control over the inputs to Scout (e.g., font and element sizes, alternate group elements). Scout could support this in two ways. First, design firms and companies typically have internal design language systems that provide rules, guidelines, interface elements, and styles to create consistency across design organizations in a company. Scout could support importing the range of variations for a property (e.g., font sizes) from a company's style guide. Supporting this may require designing an interaction to specify the style guide or structured format for importing it to Scout, as these style guides are unlikely to have a consistent format across companies. Scout could also support this by providing an interface to set the range of input parameters (e.g., font sizes, element sizes). However, not all parameters are numerical (e.g., a set of alternate group elements) thus a variety of interaction techniques for specifying parameters may be needed.

Another insight revealed from interviews with designers in Scout was that Scout's high-level constraints could provide a structured format for designers to share high-level interface rules across designers and teams. In Scout, designers need to specify interface structure and semantics (e.g., grouping, order, emphasis) to explore alternatives. Future work could explore the use

of this structure as an output format for sharing across designers. This could enable them to explore a larger range of alternatives, as multiple designers could use the same high-level constraints to explore separate threads of alternatives. It could also act as a specification for an interface design, which could help future designers who are completing a redesign task to understand the original designers intent.

During my user studies on Scout, I found that Scout was helpful to the designers for visualizing many combinations of interface elements. Frequently, designers would see a layout idea where they didn't necessarily like the entire arrangement, but liked a smaller part of the layout or the combination of two or more elements. Designers requested the ability to *mix and match* various parts of Scout layouts to create their design. Future work could explore giving designers the ability to explore alternatives to low-level layout patterns within a design and more easily extract and combine element subgroups of elements to create an entire layout.

Another type of control that Scout could support is direct manipulation of interface elements in Scout's layout ideas. Currently, designers cannot directly modify layouts within Scout. We could use this interaction paradigm to specify high-level relational constraints. However, Scout was initially designed such that a designer might not know how they want the elements to be laid out. This enabled us to evaluate the value of high-level constraints independently of the features in a direct manipulation canvas. However, if Scout were a feature in a design prototyping tool, Scout could enable designers to explore alternatives to an existing layout.

Faceted Exploration of Alternatives

Currently, Scout supports exploring alternatives to interface layouts, varying size and position of interface elements. However, during early interviews regarding Scout, designers requested that Scout explore variations on other aspects of design (e.g., color schemes, font types). An initial version of Scout supported some variation of non-layout properties (e.g., background color); however, some designers mentioned they would want to consider the layout and visual properties during separate stages of the design process. Future work could explore providing faceted exploration of alternatives. Designers could first explore a variety of wireframe layouts.

They could then explore those layouts under alternate color schemes, followed by exploration of alternate font types. Such an interaction would likely require an interface where design properties that can be varied (e.g., layout, color, fonts) would be broken down into facets so that a designer can explore one dimension at a time, or switch back and forth between design dimensions. Designing this interaction could take insights from similar efforts on faceted design search of examples [52, 113].

Expanding the Language of High-Level Constraints

Scout provides an initial set of high-level constraints. However, more complex interfaces may require a more sophisticated set of high-level constraints. Designers may want to use relational constraints (e.g., element A should be to the left of element B), or constraints that define allowed behavior (e.g., rotation). Design principles like whitespace and balance could be adapted to constraints where a designer could specify a constraint that "asymmetrical balance" is satisfied. Future work could explore taking inspiration from design principles to expand this language.

7.0.3 Modeling Human Perception of Usability

My work on TapShoe (Chapter 5) revealed that we can build machine learning models that predict human perception of tappability for interface designs. Designers can use these models to help understand a key aspect of the usability of their interfaces without the need to collect any data. Beyond tappability, researchers have explored modeling importance (i.e., emphasis) [48] of graphic design elements, and similar models could likely be built for interface designs. Researchers have also explored modeling human perception of brand personality (e.g., excitement, sophistication) [217] of mobile applications. One area of work could be to study whether we can model tappability for other platforms (i.e., webpages). Another area of future work could be to explore other attributes of usability that can be modeled and predicted in a similar way. For example, research could study how to systematize and model usability questions such as "Can a person predict what will happen a when interacting with an interface?" (i.e., predictability),

or "Is a person going to be able to understand this error message and know what to do next?" (e.g., "Help users recognize, diagnose, and recover from errors" [159]).

7.0.4 Building a Dataset of Design Documents

One limitation of Rewire (Chapter 4) is that while its accuracy at inferring element properties was reasonable, it was not able to infer element hierarchies as accurately. Interface design documents can be challenging to infer because they frequently contain complex hierarchies and shapes with ambiguous representations. For example, a button can be represented by a vector path or a rectangle. Rewire's accuracy to infer this hierarchy and to infer ambiguous elements might be improved by applying machine learning methods trained on a large dataset of designs.

Data-driven design [59, 61, 113, 132] is a recent focus of work in interface design research. Some recent work has explored techniques to enable novel design search [52, 94] and building models to predict human design perception [217]. Such techniques could potentially be used to improve Rewire's accuracy. However, throughout my work, I have noticed that design documents created in tools like Sketch and Adobe XD do not reflect the same structure as the datasets that data-driven design systems have recently relied on. Android app designs have far more complex hierarchies and different types of low-level interface elements than design prototypes.

Additionally, design prototypes for other platforms use other design languages (e.g., iOS) and element types that do not exist in Android apps. Therefore, a separate dataset of design prototype documents may be needed to create better data-driven design interactions, like those of Rewire, in interface prototyping tools. Creating such a dataset is a huge challenge for future work, given the diversity in interface prototyping tools, their output formats, and the diversity of design formats shared in online design galleries. However, creating such a dataset could be immensely helpful in example adaptation tasks, design search, and modeling human perception of interface designs.

7.0.5 Advancing Design Prototyping Tools

The work I reviewed in data-driven design mainly focuses on high-level insights while little has been integrated into design prototyping tools (e.g., Sketch, Adobe XD). Although high level insights can be useful to designers, they do not have much impact on the daily work of designers when they are using their prototyping tools. Throughout my work, I have found that there is a huge opportunity to adapt research insights from data-driven design, machine learning, and semantic analysis into interactive prototyping tools to solve real-world interface design challenges. I explored this in Rewire by building tools for example adaptation directly into Adobe XD [198].

With TapShoe [201], I built an interface for designers to explore tappability, however, this interface was outside of a prototyping environment. Design walkthroughs of this tool with designers revealed that an integrated prototyping tool would be far more useful. This would enable designers to interactively explore properties of usability on a fine grained level, where they can make a single design change (e.g., change a color, move an element around) to explore the model's output. Having the capability to explore these changes on a fine-grained level could potentially help designers gain new insights on tappability and create tappability guidelines.

For exploring alternatives, tools like Scout could be integrated into interface prototyping tools. The creation of high-level constraints could map well to this interaction paradigm as designers are already used to specifying interface prototypes in a hierarchical design outline. Techniques for exploring alternatives to data visualizations [93] could be applied to enable designers to explore low-level variations to sub-components of their designs for more fine-grained exploration.

Chapter 8

Conclusion

Interface design is crucial to the success of software. Therefore, it is important that we give interface designers the tools to make them more efficient and creative, and to better understand their users. Currently, designers still face many challenges in using their design prototyping tools for ideation, prototyping, and testing. Alternatives can be difficult to ideate and require manual work to transform, limiting the number of alternatives a designer can explore. Example designs are frequently too rigid to easily adapt and modify during prototyping. Example designs can also exist as software prototypes, which can be difficult to customize or prototype new interactions for. Many forms of large-scale usability testing are too expensive and time consuming to conduct throughout the design process. Such forms of usability testing rely on collecting data for each design change, and the results of these tests are separated from a designer's main tools.

A common thread across these challenges is that interface designers frequently need to transform interface designs to create prototypes, analyze usability, or ideate. Current design prototyping tools do not let a designer easily make this transformation. To aid this transition, we can augment interface design tools with *semantic analysis* capabilities to understand, analyze,

transform, and augment an interface design. Enhancing design tools with such capabilities can aid the design process by making it more efficient, more creative, and less rigid. Automatically inferring high-level interface semantics from incomplete inputs (e.g., pixels, code), and allowing designers to specify high-level semantics to enable a system to rapidly generate alternatives can ease this transition. We can enable this by applying technological advances from program synthesis, machine learning, and data-driven design. In this dissertation, I demonstrated the following thesis statement:

Augmenting interface design tools with high-level semantic knowledge gained through semantic and data-driven analyses can help designers more easily analyze, transform, and augment a design. This can enable them to ideate and prototype more efficiently, and more thoroughly analyze the usability of their interface designs.

In support of this statement, I developed Scout (Chapter 3), a system that helps designers rapidly visualize layout alternatives. In Scout, designers specify high-level constraints, and Scout translates them into low-level spatial constraints to enable them to *rapidly ideate alternate layouts*. Designers can iteratively refine and explore alternatives through mixed-initiative interaction with high-level constraints and design feedback. To enable designers to more easily adapt example designs, I developed Rewire (Chapter 4) which automatically infers a vector representation from an example interface screenshot where each interface component is a separate object with editable shape and style properties. Rewire can help designers *prototype more efficiently* by avoiding the manual recreation of design shapes and properties from example screenshots. To help designers in analyzing the usability of their interfaces, I created TapShoe (Chapter 5), an approach to gather usability data on mobile interface tappability at scale, and a deep learning model that designers can use to explore the tappability of their interfaces without the need to collect any data. This can help designers avoid the time and cost of conducting this type of usability study. If we can integrate such models into tools that designers can use iteratively throughout the design process, it can also help designers *more thoroughly analyze the usability of their interface designs*. Finally, I presented Genie (Chapter 6), a system that reverse engineers an abstract model of web interface commands to enable prototyping interactions

with the webpage under new input modalities. Genie provides a prototyping framework to let designers more easily transform and augment a design by *quickly prototyping* existing interfaces under new input modalities (e.g., speech, keyboard, command line input) and presentations.

Through these systems, I demonstrated that we can use semantic analysis and data-driven design to accelerate an interface designers ideation, prototyping, and testing process. Future work can build upon the work in this dissertation to advance designers' capabilities within their prototyping tools. This dissertation focuses mainly on ideating and prototyping within the design of a single interface screen. Future work can explore how to enable a designer to explicate interface semantics that can enable a system generate interface workflows based on the semantics of individual screens and the semantics of a task the designer wants to support.

If future systems present many design alternatives based on these semantics to a designer, they will also need to give designers the capabilities to prototype with and compare between multiple design alternatives. A system could generate these alternatives based on a designers' semantics or on the needs of users with diverse mental and physical capabilities. A designer could also quickly adapt them from example designs to include their own interface components.

Once a designer has been shown or has created multiple design alternatives, they will need to evaluate them and select the best alternative to support a given task. The work I present in Chapter 5 can help designers evaluate one aspect of these alternatives; however, there are numerous more aspects of usability, visual design, and human performance that can likely be modeled. It will be important to enable designers to inspect the output of these models to help them understand the trade-offs between multiple alternatives to make better decisions. The work in this dissertation can provide a foundation for the development of a mixed-initiative design workflow throughout ideation, prototyping, and testing, where a designer can offload tedious design work to a system, and a designer and a system can collaborate to produce better design solutions.

Bibliography

[1] Andy Rutledge :: Gestalt Principles - 3: Proximity, Uniform Connectedness, and Good Continuation. https://andyrutledge.com/gestalt-principles-3.html, 2009.

[2] Affordances and Design. https://www.interaction-design.org/literature/article/affordances-and-design, 2015.

[3] Beyond Blue Links: Making Clickable Elements Recognizable. https://www.nngroup.com/articles/clickable-elements/, 2015.

[4] 7 Elements of Design and How to Use Them Properly. https://blog.thepapermillstore.com/7-elements-of-design-and-how-to-use-them-properly/, 2016.

[5] Emphasis: Setting up the focal point of your design. https://www.interaction-design.org/literature/article/emphasis-setting-up-the-focal-point-of-your-design, 2016.

[6] Data Spotlight: Fierce Demand for UI/UX Designers. https://www.economicmodeling.com/2017/01/12/data-spotlight-fierce-demand-uiux-designers/, 2017.

[7] Flat UI Elements Attract Less Attention and Cause Uncertainty. https://www.nngroup.com/articles/flat-ui-less-attention-cause-uncertainty/, 2017.

[8] Build a Responsive UI with ConstraintLayout. https://developer.android.com/training/constraint-layout/, 2018.

[9] Building Adaptive User Interfaces. https://developer.apple.com/design/adaptivity/, 2018.

[10] Design for Kids Based on Their Stage of Physical Development. https://www.nngroup.com/articles/children-ux-physical-development/, 2018.

[11] Material Design Guidelines. https://material.io/design/, 2018.

[12] Visual Affordance Testing. http://practicaluxmethods.com/product/visual-affordance-testing/, 2018.

[13] Accessibility. https://material.io/design/usability/accessibility.html#assistive-technology, 2019.

[14] Adaptivity and Layout. https://developer.apple.com/design/human-interface-guidelines/ios/visual-design/adaptivity-and-layout/, 2019.

[15] Add-ons for Firefox (en-US). https://addons.mozilla.org/en-US/firefox/, 2019.

[16] Appsee Mobile App Analytics. https://www.appsee.com, 2019.

[17] Five Second Test. https://fivesecondtest.com/, 2019.

[18] Getting Started Tutorial. https://developer.chrome.com/extensions/getstarted, 2019.

[19] Sketch: The digital design toolkit. https://www.sketch.com/, 2019.

[20] Typography. https://developer.apple.com/design/human-interface-guidelines/ios/visual-design/typography/, 2019.

[21] WAI-ARIA Overview. https://www.w3.org/WAI/standards-guidelines/aria/, 2019.

[22] WebAIM: Screen Reader User Survey #8 Results. https://webaim.org/projects/screenreadersurvey8/, 2019.

[23] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, and others. Tensorflow: A System for Large-Scale Machine Learning. In *OSDI*, volume 16, pages 265–283, 2016.

[24] Julio Abascal, Amaia Aizpurua, Idoia Cearreta, Borja Gamecho, Nestor Garay-Vitoria, and Raúl Miñón. Automatically Generating Tailored Accessible User Interfaces for Ubiquitous Services. In *The Proceedings of the 13th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '11, pages 187–194, New York, NY, USA, 2011. ACM. doi: 10.1145/2049536.2049570

[25] Khalid Alharbi and Tom Yeh. Collect, Decompile, Extract, Stats, and Diff. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '15, pages 515–524, New York, New York, USA, 2015. ACM. doi: 10.1145/2785830.2785892

[26] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N. Bennett, Kori Inkpen, Jaime Teevan, Ruth Kikin-Gil, and Eric Horvitz. Guidelines for Human-AI Interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '19, pages 3:1–3:13, New York, NY, USA, 2019. ACM. doi: 10.1145/3290605.3300233

[27] Pablo Arbelaez. Boundary Extraction in Natural Images Using Ultrametric Contour Maps. In *Conference on Computer Vision and Pattern Recognition Workshop*, CVPRW '06, pages 182–182. IEEE, June 2006. doi: 10.1109/CVPRW.2006.48

[28] Nick Babich. Building Better UI Designs With Layout Grids. https://www.smashingmagazine.com/2017/12/building-better-ui-designs-layout-grids/, 2017.

[29] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, December 2017. doi: 10.1109/TPAMI.2016.2644615

[30] Greg J. Badros, Alan Borning, Kim Marriott, and Peter Stuckey. Constraint Cascading Style Sheets for the Web. In *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology*, UIST '99, pages 73–82, New York, NY, USA, 1999. ACM. doi: 10.1145/320719.322588

[31] Greg J. Badros, Alan Borning, and Peter J. Stuckey. The Cassowary Linear Arithmetic Constraint Solving Algorithm. *ACM Transactions on Computer-Human Interaction*, 8(4):267–306, December 2001. doi: 10.1145/504704.504705

[32] Helen Y. Balinsky, Anthony J. Wiley, and Matthew C. Roberts. Aesthetic Measure of Alignment and Regularity. In *Proceedings of the 9th ACM Symposium on Document Engineering*, DocEng '09, pages 56–65, New York, NY, USA, 2009. ACM. doi: 10.1145/1600193.1600207

[33] Dana H Ballard. Generalizing the Hough Transform to Detect Arbitrary Shapes. *Pattern Recognition*, 13(2):111–122, 1981.

[34] Nikola Banovic, Tovi Grossman, Justin Matejka, and George Fitzmaurice. Waken: Reverse Engineering Usage Information and Interface Structure from Software Videos. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 83–92, New York, NY, USA, 2012. ACM. doi: 10.1145/2380116.2380129

[35] Tony Beltramelli. Pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '18, pages 3:1–3:6, New York, NY, USA, 2018. ACM. doi: 10.1145/3220134.3220135

[36] Joey Benedek and Trish Miner. Measuring Desirability: New Methods for Evaluating Desirability in a Usability Lab Setting. *Proceedings of Usability Professionals Association*, 2003(8-12):57, 2002.

[37] Michael Bernard, Chia Hui Liao, and Melissa Mills. The Effects of Font Type and Size on the Legibility and Reading Time of Online Text by Older Adults. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems Extended Abstracts*, CHI EA '01, pages 175–176, New York, NY, USA, 2001. ACM. doi: 10.1145/634067.634173

[38] Pavol Bielik, Marc Fischer, and Martin Vechev. Robust Relational Layout Synthesis from Examples for Android. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):156:1–156:29, October 2018. doi: 10.1145/3276526

[39] Jeffrey P. Bigham and Richard E. Ladner. Accessmonkey: A Collaborative Scripting Framework for Web Users and Developers. In *Proceedings of the 2007 International Cross-disciplinary Conference on Web Accessibility*, W4A '07, page 25, New York, New York, USA, 2007. ACM. doi: 10.1145/1243441.1243452

[40] Jeffrey P. Bigham, Tessa Lau, and Jeffrey Nichols. Trailblazer: Enabling Blind Users to Blaze Trails through the Web. In *Proceedings of the 13th International Conference on Intelligent User Interfaces*, IUI '09, page 177, New York, New York, USA, 2008. ACM. doi: 10.1145/1502650.1502677

[41] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. Automation and Customization of Rendered Web Pages. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology*, UIST '05, pages 163–172, New York, New York, USA, 2005. ACM. doi: 10.1145/1095034.1095062

[42] Alan Borning, Richard Kuang-Hsu Lin, and Kim Marriott. Constraint-based Document Layout for the Web. *Multimedia Systems*, 8(3):177–189, October 2000. doi: 10.1007/s005300000043

[43] Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving Linear Arithmetic Constraints for User Interface Applications. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*, UIST '97, pages 87–96, Banff, Alberta, Canada, 1997. ACM. doi: 10.1145/263407.263518

[44] Yevgen Borodin, Jeffrey P. Bigham, Rohit Raman, and I. V. Ramakrishnan. What's New?: Making Web Page Updates Accessible. In *Proceedings of the 10th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '08, pages 145–152, New York, NY, USA, 2008. ACM. doi: 10.1145/1414471.1414499

[45] Andy Brown and Simon Harper. Dynamic Injection of WAI-ARIA into Web Content. In *Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility*, W4A '13, pages 14:1–14:4, New York, NY, USA, 2013. ACM. doi: 10.1145/2461121.2461141

[46] Margaret Burnett, Anicia Peters, Charles Hill, and Noha Elarief. Finding Gender-Inclusiveness Software Issues with GenderMag: A Field Investigation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '16, pages 2586–2598, New York, NY, USA, 2016. ACM. doi: 10.1145/2858036.2858274

[47] William Buxton. *Sketching User Experiences: Getting the Design Right and the Right Design*. Elsevier/Morgan Kaufmann, 2007.

[48] Zoya Bylinskii, Nam Wook Kim, Peter O'Donovan, Sami Alsheikh, Spandan Madan, Hanspeter Pfister, Fredo Durand, Bryan Russell, and Aaron Hertzmann. Learning Visual Importance for Graphic Designs and Data Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 57–69, New York, NY, USA, 2017. ACM. doi: 10.1145/3126594.3126653

[49] Luca Cardelli. Building User Interfaces by Direct Manipulation. In *Proceedings of the 1st Annual ACM SIGGRAPH Symposium on User Interface Software*, UIST '88, pages 152–166, New York, NY, USA, 1988. ACM. doi: 10.1145/62402.62428

[50] Kerry Shih-Ping Chang and Brad A. Myers. WebCrystal: Understanding and Reusing Examples in Web Authoring. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 3205–3214, New York, NY, USA, 2012. ACM. doi: 10.1145/2207676.2208740

[51] Tsung-Hsiang Chang, Tom Yeh, and Rob Miller. Associating the Visual Representation of User Interfaces With Their Internal Structures and Metadata. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, page 245, New York, New York, USA, 2011. ACM. doi: 10.1145/2047196.2047228

[52] Chunyang Chen, Sidong Feng, Zhenchang Xing, Linda Liu, Shengdong Zhao, and Jinshui Wang. Gallery D.C.: Design Search and Knowledge Discovery Through Auto-created GUI Component Gallery. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):180:1–180:22, November 2019. doi: 10.1145/3359282

[53] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. From UI Design Image to GUI Skeleton: A Neural Machine Translator to Bootstrap Mobile GUI Implementation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 665–676, New York, NY, USA, 2018. ACM. doi: 10.1145/3180155.3180240

[54] Sen Chen, Lingling Fan, Ting Su, Lei Ma, Yang Liu, and Lihua Xu. Automated Cross-Platform GUI Code Generation for Mobile Apps. In *2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile)*, pages 13–16, February 2019. doi: 10.1109/AI4Mobile.2019.8672718

[55] Larry L. Constantine and Lucy AD Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Pearson Education, 1999.

[56] Alan Cooper, Robert Reimann, David Cronin, and Christopher Noessel. *About Face: The Essentials of Interaction Design*. John Wiley & Sons, August 2014.

[57] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[58] Marco de Sá, Luís Carriço, Luís Duarte, and Tiago Reis. A Mixed-fidelity Prototyping Tool for Mobile Devices. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '08, pages 225–232, New York, NY, USA, 2008. ACM. doi: 10.1145/1385569.1385606

[59] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 845–854, New York, NY, USA, 2017. ACM. doi: 10.1145/3126594.3126651

[60] Biplab Deka, Zifeng Huang, Chad Franzen, Jeffrey Nichols, Yang Li, and Ranjitha Kumar. ZIPT: Zero-Integration Performance Testing of Mobile App Designs. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 727–736, New York, NY, USA, 2017. ACM. doi: 10.1145/3126594.3126647

[61] Biplab Deka, Zifeng Huang, and Ranjitha Kumar. ERICA: Interaction Mining Mobile Apps. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 767–776, New York, NY, USA, 2016. ACM. doi: 10.1145/2984511.2984581

[62] Morgan Dixon and James Fogarty. Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1525–1534, New York, NY, USA, 2010. ACM. doi: 10.1145/1753326.1753554

[63] Morgan Dixon, James Fogarty, and Jacob Wobbrock. A General-purpose Target-aware Pointing Enhancement Using Pixel-level Analysis of Graphical Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 3167–3176, New York, NY, USA, 2012. ACM. doi: 10.1145/2207676.2208734

[64] Morgan Dixon, Daniel Leventhal, and James Fogarty. Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 969–978, New York, NY, USA, 2011. ACM. doi: 10.1145/1978942.1979086

[65] Steven P. Dow, Julie Fortuna, Dan Schwartz, Beth Altringer, Daniel L. Schwartz, and Scott R. Klemmer. Prototyping Dynamics: Sharing Multiple Designs Improves Exploration, Group Rapport, and Results. In *Design Thinking Research*, pages 47–70, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi: 10.1007/978-3-642-31991-4_4

[66] Steven P. Dow, Alana Glassco, Jonathan Kass, Melissa Schwarz, Daniel L. Schwartz, and Scott R. Klemmer. Parallel Prototyping Leads to Better Design Results, More Divergence, and Increased Self-efficacy. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 17(4):18:1–18:24, December 2010. doi: 10.1145/1879831.1879836

[67] Steven P. Dow, Alana Glassco, Jonathan Kass, Melissa Schwarz, Daniel L. Schwartz, and Scott R. Klemmer. Parallel Prototyping Leads to Better Design Results, More Divergence, and Increased Self-efficacy. In *Design Thinking Research*, pages 127–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi: 10.1007/978-3-642-21643-5_8

[68] James R. Eagan, Michel Beaudouin-Lafon, and Wendy E. Mackay. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 225–234, New York, NY, USA, 2011. ACM. doi: 10.1145/2047196.2047226

[69] W. Keith Edwards, Scott E. Hudson, Joshua Marinacci, Roy Rodenstein, Thomas Rodriguez, and Ian Smith. Systematic Output Modification in a 2d User Interface Toolkit. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*, UIST '97, pages 151–158, New York, New York, USA, 1997. ACM. doi: 10.1145/263407.263537

[70] Leah Findlater, Alex Jansen, Kristen Shinohara, Morgan Dixon, Peter Kamb, Joshua Rakita, and Jacob O. Wobbrock. Enhanced Area Cursors: Reducing Fine Pointing

Demands for People with Motor Impairments. In *Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology*, UIST '10, pages 153–162, New York, New York, USA, 2010. ACM. doi: 10.1145/1866029.1866055

[71] Paul M. Fitts. The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement. *Journal of Experimental Psychology*, 47(6):381–391, 1954. doi: 10.1037/h0055392

[72] Joseph L Fleiss. Measuring Nominal Scale Agreement Among Many Raters. *Psychological Bulletin*, 76(5):378, 1971.

[73] Krzysztof Gajos, Anthony Wu, and Daniel S Weld. Cross-Device Consistency in Automatically Generated User Interfaces. In *Proceedings of the 2nd Workshop on Multi-User and Ubiquitous User Interfaces*, pages 7–8, 2005.

[74] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. Automatically Generating User Interfaces Adapted to Users' Motor and Vision Capabilities. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, page 231, New York, New York, USA, 2007. ACM. doi: 10.1145/1294211.1294253

[75] William W. Gaver. Technology Affordances. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 79–84, New York, NY, USA, 1991. ACM. doi: 10.1145/108844.108856

[76] James J Gibson. *The Ecological Approach to Visual Perception: Classic Edition*. Psychology Press, 2014.

[77] Álvaro González, Luis M. Bergasa, J. Javier Yebes, and Sebastián Bronte. Text Location in Complex Images. In *Proceedings of the International Conference on Pattern Recognition*, ICPR '12, pages 617–620. IEEE, November 2012.

[78] Michael D. Greenberg, Matthew W. Easterday, and Elizabeth M. Gerber. Critiki: A Scaffolded Approach to Gathering Design Feedback from Paid Crowdworkers. In *Proceedings of the 2015 ACM SIGCHI Conference on Creativity and Cognition*, C&C '15, pages 235–244, New York, NY, USA, 2015. ACM. doi: 10.1145/2757226.2757249

[79] Tovi Grossman and Ravin Balakrishnan. The Bubble Cursor: Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '05, pages 281–290, New York, NY, USA, 2005. ACM. doi: 10.1145/1054972.1055012

[80] Ulrike Hahn, Nick Chater, and Lucy B Richardson. Similarity as Transformation. *Cognition*, 87(1):1–32, 2003.

[81] Vicki L. Hanson and John T. Richards. Progress on Website Accessibility? *ACM Transactions on the Web*, 7(1):2:1–2:30, March 2013. doi: 10.1145/2435215.2435217

[82] Steven J. Harrington and Paul Roetling. Aesthetic Measures for Automated Document Layout. In *ACM Symposium on Document Engineering*, DocEng '04, pages 109–111, 2004.

[83] Björn Hartmann, Sean Follmer, Antonio Ricciardi, Timothy Cardenas, and Scott R. Klemmer. D.Note: Revising User Interfaces Through Change Tracking, Annotations, and Alternatives. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 493–502, New York, NY, USA, 2010. ACM. doi: 10.1145/1753326.1753400

[84] Björn Hartmann, Leslie Wu, Kevin Collins, and Scott R. Klemmer. Programming by a Sample: Rapidly Creating Web Applications with D.Mix. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, pages 241–250, New York, NY, USA, 2007. ACM. doi: 10.1145/1294211.1294254

[85] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*, UIST '08, page 91, New York, New York, USA, 2008. ACM. doi: 10.1145/1449715.1449732

[86] Osamu Hashimoto and Brad A. Myers. Graphical Styles for Building Interfaces by Demonstration. In *Proceedings of the 5th Annual ACM Symposium on User Interface Software and Technology*, UIST '92, pages 117–124, New York, NY, USA, 1992. ACM. doi: 10.1145/142621.142635

[87] Scarlett R. Herring, Chia-Chen Chang, Jesse Krantzler, and Brian P. Bailey. Getting Inspired!: Understanding How and Why Examples Are Used in Creative Design Practice. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 87–96, New York, NY, USA, 2009. ACM. doi: 10.1145/1518701.1518717

[88] Jane Hoffswell, Alan Borning, and Jeffrey Heer. SetCoLa: High-Level Constraints for Graph Layout. *Computer Graphics Forum*, 37(3):537–548, June 2018. doi: 10.1111/cgf.13440

[89] Sture Holm. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979.

[90] Andreas Holzinger, Gig Searle, and Alexander Nischelwitzer. On Some Aspects of Improving Mobile Applications for the Elderly. In *Universal Access in Human Computer Interaction. Coping with Diversity*, pages 923–932, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[91] Eric Horvitz. Principles of Mixed-initiative User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '99, pages 159–166, New York, NY, USA, 1999. ACM. doi: 10.1145/302979.303030

[92] Hiroshi Hosobe. A Scalable Linear Constraint Solver for User Interface Construction. In *Principles and Practice of Constraint Programming*, CP '00, pages 218–233, Berlin, Heidelberg, 2000. Springer. doi: 10.1007/3-540-45349-0_17

[93] Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. Programming by Manipulation for Layout. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 231–241, New York, NY, USA, 2014. ACM. doi: 10.1145/2642918.2647378

[94] Forrest Huang, John F. Canny, and Jeffrey Nichols. Swire: Sketch-based User Interface Retrieval. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, pages 104:1–104:10, New York, NY, USA, 2019. ACM. doi: 10.1145/3290605.3300334

[95] Yun Huang, Brian Dobreski, Bijay Bhaskar Deo, Jiahang Xin, Natã Miccael Barbosa, Yang Wang, and Jeffrey P. Bigham. CAN: Composable Accessibility Infrastructure via Data-driven Crowdsourcing. In *Proceedings of the 12th Web for All Conference*, W4A '15, pages 2:1–2:10, New York, NY, USA, 2015. ACM. doi: 10.1145/2745555.2746651

[96] Scott E. Hudson and Shamim P. Mohamed. Interactive Specification of Flexible User Interface Displays. *ACM Transactions on Information Systems*, 8(3):269–288, July 1990. doi: 10.1145/98188.98201

[97] Faustina Hwang, Simeon Keates, Patrick Langdon, and John Clarkson. Mouse Movements of Motion-impaired Users: A Submovement Analysis. In *Proceedings of the 6th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '04, pages 102–109, New York, NY, USA, 2004. ACM. doi: 10.1145/1028630.1028649

[98] Adobe Inc. Download free Adobe XD | UX/UI design and collaboration tool. https://www.adobe.com/products/xd.html, October 2019.

[99] David G. Jansson and Steven M. Smith. Design Fixation. *Design Studies*, 12(1):3–11, January 1991. doi: 10.1016/0142-694X(91)90003-F

[100] Yue Jiang, Ruofei Du, Christof Lutteroth, and Wolfgang Stuerzlinger. ORC Layout: Adaptive GUI Layout with OR-Constraints. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '19, pages 413:1–413:12, New York, NY, USA, 2019. ACM. doi: 10.1145/3290605.3300643

[101] Eunice Jun, Maureen Daum, Jared Roesch, Sarah Chasins, Emery Berger, Rene Just, and Katharina Reinecke. Tea: A High-level Language and Runtime System for Automating Statistical Analysis. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, pages 591–603, New Orleans, LA, USA, 2019. ACM. doi: 10.1145/3332165.3347940

[102] Solange Karsenty, Chris Weikart, and James A. Landay. Inferring Graphical Constraints with Rockit. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '93, page 531, New York, New York, USA, 1993. ACM. doi: 10.1145/169059.169528

[103] Shinya Kawanaka, Yevgen Borodin, Jeffrey P. Bigham, Darren Lunn, Hironobu Takagi, and Chieko Asakawa. Accessibility Commons: A Metadata Infrastructure for Web Accessibility. In *Proceedings of the 10th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '08, pages 153–160, New York, NY, USA, 2008. ACM. doi: 10.1145/1414471.1414500

[104] Rubaiat Habib Kazi, Tovi Grossman, Hyunmin Cheong, Ali Hashemi, and George Fitzmaurice. DreamSketch: Early Stage 3d Design Explorations with Sketching and Generative Design. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 401–414, New York, NY, USA, 2017. ACM. doi: 10.1145/3126594.3126662

[105] Andrew Kirpatrick, Campbell Alastair O Connor, Joshue, and Michael Cooper. Web Content Accessibility Guidelines. https://www.w3.org/TR/WCAG20/, 2018.

[106] Aniket Kittur, Ed H. Chi, and Bongwon Suh. Crowdsourcing User Studies with Mechanical Turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 453–456, New York, NY, USA, 2008. ACM. doi: 10.1145/1357054.1357127

[107] Scott R. Klemmer, Mark W. Newman, Ryan Farrell, Mark Bilezikjian, and James A. Landay. The Designers' Outpost: A Tangible Interface for Collaborative Web Site Design. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, UIST '01, pages 1–10, New York, NY, USA, 2001. ACM. doi: 10.1145/502348.502350

[108] Amy J. Ko and Xing Zhang. Feedlack Detects Missing Feedback in Web Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2177–2186, New York, NY, USA, 2011. ACM. doi: 10.1145/1978942.1979260

[109] Wolfgang Kohler. Gestalt Psychology. *Psychological Research*, 31(1):XVIII–XXX, 1967. doi: 10.1007/BF00422382

[110] Steven Komarov, Katharina Reinecke, and Krzysztof Z. Gajos. Crowdsourcing Performance Evaluations of User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 207–216, New York, NY, USA, 2013. ACM. doi: 10.1145/2470654.2470684

[111] Yuki Koyama, Daisuke Sakamoto, and Takeo Igarashi. Crowd-powered Parameter Analysis for Visual Design Exploration. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 65–74, New York, NY, USA, 2014. ACM. doi: 10.1145/2642918.2647386

[112] Chinmay Kulkarni, Steven P. Dow, and Scott R Klemmer. Early and Repeated Exposure to Examples Improves Creative Work. In *Design Thinking Research*, pages 49–62. Springer International Publishing, 2014. doi: 10.1007/978-3-319-01303-9_4

[113] Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R. Klemmer, and Jerry O. Talton. Webzeitgeist: Design Mining the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 3083–3092, New York, NY, USA, 2013. ACM. doi: 10.1145/2470654.2466420

[114] Ranjitha Kumar, Jerry O. Talton, Salman Ahmad, and Scott R. Klemmer. Bricolage: Example-based Retargeting for Web Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2197–2206, New York, NY, USA, 2011. ACM. doi: 10.1145/1978942.1979262

[115] James A. Landay. SILK: Sketching Interfaces like Krazy. In *Conference Companion on Human Factors in Computing Systems*, CHI '96, pages 398–399, Vancouver, British Columbia, Canada, 1996. ACM. doi: 10.1145/257089.257396

[116] J Richard Landis and Gary G Koch. An Application of Hierarchical Kappa-type Statistics in the Assessment of Majority Agreement Among Multiple Observers. *Biometrics*, pages 363–374, 1977.

[117] Walter S. Lasecki, Juho Kim, Nick Rafter, Onkur Sen, Jeffrey P. Bigham, and Michael S. Bernstein. Apparition: Crowdsourced User Interfaces That Come to Life As You Sketch Them. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 1925–1934, New York, NY, USA, 2015. ACM. doi: 10.1145/2702123.2702565

[118] Hao L and Yang Li. Gesture Avatar: A Technique for Operating Mobile User Interfaces Using Gestures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 207–216, New York, New York, USA, 2011. ACM. doi: 10.1145/1978942.1978972

[119] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 521(7553):436, 2015.

[120] Brian Lee, Savil Srivastava, Ranjitha Kumar, Ronen Brafman, and Scott R. Klemmer. Designing with Interactive Example Galleries. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Syfstems*, CHI '10, pages 2257–2266, New York, NY, USA, 2010. ACM. doi: 10.1145/1753326.1753667

[121] Sang Won Lee, Yujin Zhang, Isabelle Wong, Yiwei Yang, Stephanie D. O'Keefe, and Walter S. Lasecki. SketchExpress: Remixing Animations for More Effective Crowd-Powered Prototyping of Interactive Interfaces. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 817–828. ACM, 2017. doi: 10.1145/3126594.3126595

[122] Jianan Li, Jimei Yang, Aaron Hertzmann, Jianming Zhang, and Tingfa Xu. LayoutGAN: Generating Graphic Layouts with Wireframe Discriminators. *arXiv:1901.06767 [cs]*, January 2019. arXiv: 1901.06767.

[123] Yang Li, Samy Bengio, and Gilles Bailly. Predicting Human Performance in Vertical Menu Selection Using Deep Learning. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '18, pages 29:1–29:7. ACM, 2018. doi: 10.1145/3173574.3173603

[124] Yao Li and Huchuan Lu. Scene Text Detection via Stroke Width. In *Proceedings of the International Conference on Pattern Recognition*, ICPR '12, pages 681–684. IEEE, November 2012.

[125] Horst Lichter, Matthias Schneider-Hufschmidt, and Heinz Zullighoven. Prototyping in Industrial Software Projects - Bridging the Gap Between Theory and Practice. *IEEE Transactions on Software Engineering*, 20(11):825–832, 1994.

[126] William Lidwell, Kritina Holden, and Jill Butler. *Universal Principles of Design, Revised and Updated: 125 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach Through Design*. Rockport Publishers, 2010.

[127] Youn-Kyung Lim, Erik Stolterman, and Josh Tenenberg. The Anatomy of Prototypes: Prototypes As Filters, Prototypes As Manifestations of Design Ideas. *ACM Transactions on Computer-Human Interaction*, 15(2):7:1–7:27, July 2008. doi: 10.1145/1375761.1375762

[128] James Lin and James A. Landay. Damask: A Tool for Early-Stage Design and Prototyping of Cross-Device User Interfaces. In *CHI 2003 Workshop on HCI Patterns: Concepts and Tools*, 2003.

[129] James Lin and James A. Landay. Employing Patterns and Layers for Early-Stage Design and Prototyping of Cross-Device User Interfaces. In *Proceeding of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, page 1313, New York, New York, USA, 2008. ACM. doi: 10.1145/1357054.1357260

[130] 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*, CHI '00, pages 510–517, New York, NY, USA, 2000. ACM. doi: 10.1145/332040.332486

[131] Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, Eser Kandogan, and Eser Kandogan. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 943–946, New York, NY, USA, 2007. ACM. doi: 10.1145/1240624.1240767

[132] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. Learning Design Semantics for Mobile Apps. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, pages 569–579, New York, NY, USA, 2018. ACM. doi: 10.1145/3242587.3242650

[133] Aran Lunzer and Kasper Hornbæk. Subjunctive Interfaces: Extending Applications to Support Parallel Setup, Viewing and Control of Alternative Scenarios. *ACM Transactions on Computer-Human Interaction*, 14(4):1–44, January 2008. doi: 10.1145/1314683.1314685

[134] Kurt Luther, Jari-Lee Tolentino, Wei Wu, Amy Pavel, Brian P. Bailey, Maneesh Agrawala, Bjrn Hartmann, and Steven P. Dow. Structuring, Aggregating, and Evaluating Crowdsourced Design Critique. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, pages 473–485, New York, NY, USA, 2015. ACM. doi: 10.1145/2675133.2675283

[135] Christof Lutteroth, Robert Strandh, and Gerald Weber. Domain Specific High-Level Constraints for User Interface Layout. *Constraints*, 13(3):307–342, September 2008. doi: 10.1007/s10601-008-9043-2

[136] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. ReAjax: A Reverse Engineering tool for Ajax Web Applications. *IET Software*, 6(1):33, 2012. doi: 10.1049/iet-sen.2010.0152

[137] Joe Marks, Paul Beardsley, Brad Andalman, William Freeman, Sarah Gibson, Jessica Hodgins, Thomas Kang, Brian Mirtich, Hanspeter Pfister, Wheeler Ruml, et al. Design Galleries: A General Approach to Setting Parameters for Computer Graphics and Animation. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 389–400, New York, New York, USA, 1997. ACM. doi: 10.1145/258734.258887

[138] Dimitri Masson, Alexandre Demeure, and Gaelle Calvary. Magellan, an Evolutionary System to Foster User Interface Design Creativity. In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '10, pages 87–92, New York, NY, USA, 2010. ACM. doi: 10.1145/1822018.1822032

[139] Justin Matejka, Michael Glueck, Erin Bradner, Ali Hashemi, Tovi Grossman, and George Fitzmaurice. Dream Lens: Exploration and Visualization of Large-Scale Generative

Design Datasets. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 369:1–369:12, New York, NY, USA, 2018. ACM. doi: 10.1145/3173574.3173943

[140] Nolwenn Maudet, Germąn Leiva, Michel Beaudouin-Lafon, and Wendy Mackay. Design Breakdowns: Designer-Developer Gaps in Representing and Interpreting Interactive Systems. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, CSCW '17, pages 630–641, New York, NY, USA, 2017. ACM. doi: 10.1145/2998181.2998190

[141] Nicholas Micallef, Erwin Adi, and Gaurav Misra. Investigating Login Features in Smartphone Apps. In *Proceedings of the 2018 ACM International Joint Conference and 2018 International Symposium on Pervasive and Ubiquitous Computing and Wearable Computers*, UbiComp '18, pages 842–851, New York, NY, USA, 2018. ACM. doi: 10.1145/3267305.3274172

[142] Aliaksei Miniukovich and Antonella De Angeli. Computation of Interface Aesthetics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '15, pages 1163–1172, Seoul, Republic of Korea, 2015. ACM. doi: 10.1145/2702123.2702575

[143] Aliaksei Miniukovich and Antonella De Angeli. Visual Diversity and User Interface Quality. In *Proceedings of the 2015 British HCI Conference*, pages 101–109. ACM, 2015.

[144] Bill Moggridge. *Designing Interactions*. The MIT Press, 2006.

[145] Rolf Molich and Jakob Nielsen. Improving a Human - Computer Dialogue. *Communications of the ACM*, 33(3):11, 1990.

[146] Kevin Patrick Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *IEEE Transactions on Software Engineering*, 2018.

[147] Dominik Moritz, Chenglong Wang, Greg L Nelson, Halden Lin, Adam M Smith, Bill Howe, and Jeffrey Heer. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):438–448, 2019.

[148] Brad Myers. Challenges of HCI Design and Implementation. *Interactions*, 1(1):73–83, January 1994. doi: 10.1145/174800.174808

[149] Brad Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Amy Ko. How Designers Design and Program Interactive Behaviors. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 177–184, September 2008. doi: 10.1109/VLHCC.2008.4639081

[150] Brad A. Myers and William Buxton. Creating Highly-interactive and Graphical User Interfaces by Demonstration. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 249–258, New York, NY, USA, 1986. ACM. doi: 10.1145/15922.15914

[151] Vinod Nair and Geoffrey E Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on Machine Learning*, ICML '10, pages 807–814, 2010.

[152] Michael Nebeling, Maximilian Speicher, and Moira C Norrie. CrowdStudy: General Toolkit for Crowdsourced Evaluation of Web Interfaces. In *Proceedings of the SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 255–264. ACM, 2013. doi: 10.1145/2494603.2480303

[153] David Chek Ling Ngo, Lian Seng Teo, and John G. Byrne. Modelling Interface Aesthetics. *Information Sciences*, 152:25–46, June 2003. doi: 10.1016/S0020-0255(02)00404-8

[154] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. GUITAR: An Innovative Tool for Automated Testing of GUI-driven Software. *Automated Software Engineering*, 21(1):65–105, March 2014. doi: 10.1007/s10515-013-0128-9

[155] Tuan Anh Nguyen and Christoph Csallner. Reverse Engineering Mobile Application User Interfaces with REMAUI (T). In *IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pages 248–259. IEEE, November 2015. doi: 10.1109/ASE.2015.32

[156] Jeffrey Nichols, Brad A. Myers, and Kevin Litwack. Improving Automatic Interface Generation with Smart Templates. In *Proceedings of the 9th International Conference on Intelligent User Interfaces*, IUI '04, page 286, New York, New York, USA, 2004. ACM. doi: 10.1145/964442.964507

[157] Jakob Nielsen. *Usability Engineering*. Elsevier, 1994.

[158] Jakob Nielsen and Robert L Mack. *Usability Inspection Methods*, volume 1. Wiley New York, 1994.

[159] Jakob Nielsen and Rolf Molich. Heuristic Evaluation of User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 249–256, New York, New York, USA, 1990. ACM. doi: 10.1145/97243.97281

[160] Don Norman. *The Design of Everyday Things: Revised and Expanded Edition*. Constellation, 2013.

[161] Donald A. Norman. Affordance, Conventions, and Design. *Interactions*, 6(3):38–43, May 1999. doi: 10.1145/301153.301168

[162] Donald A. Norman. The Way I See It: Signifiers, Not Affordances. *Interactions*, 15(6):18–19, November 2008. doi: 10.1145/1409040.1409044

[163] Peter O'Donovan, Aseem Agarwala, and Aaron Hertzmann. Learning Layouts for Single-Page Graphic Designs. *IEEE Transactions on Visualization and Computer Graphics*, 20(8):1200–1213, August 2014. doi: 10.1109/TVCG.2014.48

[164] Peter O'Donovan, Aseem Agarwala, and Aaron Hertzmann. DesignScape: Design with Interactive Layout Suggestions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '15, pages 1221–1224, New York, New York, USA, 2015. ACM. doi: 10.1145/2702123.2702149

[165] Dan R. Olsen, Scott E. Hudson, Thom Verratti, Jeremy M. Heiner, and Matt Phelps. Implementing Interface Attachments based on Surface Representations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '99, pages 191–198, New York, New York, USA, 1999. ACM. doi: 10.1145/302979.303038

[166] Dan R. Olsen, Trent Taufer, and Jerry Alan Fails. ScreenCrayons: Annotating Anything. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, UIST '04, page 165, New York, New York, USA, 2004. ACM. doi: 10.1145/1029632.1029663

[167] Antti Oulasvirta, Aliaksei Miniukovich, Gregorio Palmas, Tino Weinkauf, Samuli De Pascale, Janin Koch, Thomas Langerak, Jussi Jokinen, Kashyap Todi, Markku Laine, Manoj Kristhombuge, and Yuxi Zhu. Aalto Interface Metrics (AIM): A Service and Codebase for Computational GUI Evaluation. In *The 31st Annual ACM Symposium on User Interface Software and Technology Adjunct Proceedings*, UIST '18 Adjunct, pages 16–19, Berlin, Germany, 2018. ACM. doi: 10.1145/3266037.3266087

[168] Pavel Panchekha, Adam T. Geller, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. Verifying That Web Pages Have Accessible Layout. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '18, pages 1–14, New York, NY, USA, 2018. ACM. doi: 10.1145/3192366.3192407

[169] Pekka Parhi, Amy K. Karlson, and Benjamin B. Bederson. Target Size Study for One-handed Thumb Use on Small Touchscreen Devices. In *Proceedings of the 8th Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '06, pages 203–210, New York, NY, USA, 2006. ACM. doi: 10.1145/1152215.1152260

[170] Michael Quinn Patton. *Qualitative Research & Evaluation Methods: Integrating Theory and Practice*. SAGE Publications, October 2014.

[171] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, EMLNP '14, pages 1532–1543, 2014.

[172] Ken Pfeuffer and Yang Li. Analysis and Modeling of Grid Performance on Touchscreen Mobile Devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '18, pages 288:1–288:12, New York, NY, USA, 2018. ACM. doi: 10.1145/3173574.3173862

[173] Christopher Power, André Freire, Helen Petrie, and David Swallow. Guidelines Are Only Half of the Story: Accessibility Problems Encountered by Blind Users on the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 433–442, New York, NY, USA, 2012. ACM. doi: 10.1145/2207676.2207736

[174] Raffaela Rein. The Trillion Dollar UX Problem: A Comprehensive Guide to the ROI of UX, March 2017.

[175] Katharina Reinecke and Abraham Bernstein. Improving Performance, Perceived Usability, and Aesthetics with Culturally Adaptive User Interfaces. *ACM Transactions on Computer-Human Interaction*, 18(2):1–29, June 2011. doi: 10.1145/1970378.1970382

[176] Katharina Reinecke, Tom Yeh, Luke Miratrix, Rahmatri Mardiko, Yuechen Zhao, Jenny Liu, and Krzysztof Z. Gajos. Predicting Users' First Impressions of Website Aesthetics with a Quantification of Perceived Visual Complexity and Colorfulness. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 2049–2058, New York, NY, USA, 2013. ACM. doi: 10.1145/2470654.2481281

[177] Garr Reynolds. *Presentation Zen: Simple Ideas on Presentation Design and Delivery*. New Riders, 2011.

[178] Andreas Riegler and Clemens Holzmann. Measuring Visual User Interface Complexity of Mobile Applications With Metrics. *Interacting with Computers*, 30(3):207–223, 2018. doi: 10.1093/iwc/iwy008

[179] Daniel Ritchie, Ankita Arvind Kejriwal, and Scott R. Klemmer. d.tour: Style-Based Exploration of Design Example Galleries. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, page 165, New York, New York, USA, 2011. ACM. doi: 10.1145/2047196.2047216

[180] Ruth Rosenholtz, Amal Dorai, and Rosalind Freeman. Do Predictions of Visual Perception Aid Design? *ACM Transactions on Applied Perception*, 8(2):12:1–12:20, February 2011. doi: 10.1145/1870076.1870080

[181] Ruth Rosenholtz and Zhenlan Jin. A Computational Form of the Statistical Saliency Model for Visual Search. *Journal of Vision*, 5(8):777–777, September 2005. doi: 10.1167/5.8.777

[182] Ruth Rosenholtz, Yuanzhen Li, and Lisa Nakano. Measuring Visual Clutter. *Journal of Vision*, 7(2):17–17, January 2007. doi: 10.1167/7.2.17

[183] Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. Examining Image-Based Button Labeling for Accessibility in Android Apps Through Large-Scale Analysis. In *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '18, pages 119–130, New York, NY, USA, 2018. ACM. doi: 10.1145/3234695.3236364

[184] Stuart J Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002.

[185] Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjörg Schmauder. Insights into Layout Patterns of Mobile User Interfaces by an Automatic Analysis of Android Apps. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, page 275, New York, New York, USA, 2013. ACM. doi: 10.1145/2494603.2480308

[186] Hanna Schneider, Katharina Frison, Julie Wagner, and Andras Butz. CrowdUX: A Case for Using Widespread and Lightweight Tools in the Quest for UX. In *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*, DIS '16, pages 415–426, New York, NY, USA, 2016. ACM. doi: 10.1145/2901790.2901814

[187] Gaurav Sharma, Wencheng Wu, and Edul N. Dalal. The CIEDE2000 Color-Difference Formula: Implementation Notes, Supplementary Test Data, and Mathematical Observations. *Color Research and Application*, 30(1):21–30, 2005. doi: 10.1002/col.20070

[188] Ben Shneiderman, Catherine Plaisant, Maxine Cohen, Steven Jacobs, Niklas Elmqvist, and Nicholas Diakopoulos. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson, 6th edition, 2016.

[189] Carlos E. Silva and José C. Campos. Combining Static and Dynamic Analysis for the Reverse Engineering of Web Applications. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 107–112, New York, New York, USA, June 2013. ACM. doi: 10.1145/2494603.2480324

[190] Steven John Simon and Steven John. The Impact of Culture and Gender on Web Sites. *ACM SIGMIS Database: The Database for Advances in Information Systems*, 32(1):18–37, December 2001. doi: 10.1145/506740.506744

[191] Gurminder Singh, Chun Hong Kok, and Teng Ye Ngan. Druid: A System for Demonstrational Rapid User Interface Development. In *Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*, UIST '90, pages 167–177, New York, NY, USA, 1990. ACM. doi: 10.1145/97924.97943

[192] Nishant Sinha and Rezwana Karim. Responsive Designs in a Snap. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '15, pages 544–554, New York, NY, USA, 2015. ACM. doi: 10.1145/2786805.2786808

[193] Ray Smith. An Overview of the Tesseract OCR Engine. In *International Conference on Document Analysis and Recognition*, volume 2 of *ICDAR '07*, pages 629–633. IEEE, September 2007. doi: 10.1109/ICDAR.2007.4376991

[194] Debbie Stone, Caroline Jarrett, Mark Woodroffe, and Shailey Minocha. *User Interface Design and Evaluation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[195] Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Phillips, and Nicolas Roussel. User Interface Façades: Towards Fully Adaptable User Interfaces. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, UIST '06, pages 309–318, New York, NY, USA, 2006. ACM. doi: 10.1145/1166253.1166301

[196] Piyawadee Sukaviriya, James D. Foley, and Todd Griffith. A Second Generation User Interface Design Environment: The Model and the Runtime Architecture. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '93, pages 375–382, New York, New York, USA, 1993. ACM. doi: 10.1145/169059.169299

[197] Ivan E. Sutherland. SketchPad: A Man-machine Graphical Communication System. In *Proceedings of the SHARE Design Automation Workshop*, DAC '64, pages 6.329–6.346, New York, NY, USA, 1964. ACM. doi: 10.1145/800265.810742

[198] Amanda Swearngin, Mira Dontcheva, Wilmot Li, Joel Brandt, Morgan Dixon, and Amy J. Ko. Rewire: Interface Design Assistance from Examples. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '18, pages 504:1–504:12, New York, NY, USA, 2018. ACM. doi: 10.1145/3173574.3174078

[199] Amanda Swearngin, Amy J. Ko, and James Fogarty. Genie: Input Retargeting on the Web Through Command Reverse Engineering. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '17, pages 4703–4714, New York, NY, USA, 2017. ACM. doi: 10.1145/3025453.3025506

[200] Amanda Swearngin, Amy J. Ko, and James Fogarty. Scout: Mixed-Initiative Exploration of Design Variations Through High-Level Design Constraints. In *The 31st Annual ACM Symposium on User Interface Software and Technology Adjunct Proceedings*, UIST '18 Adjunct, pages 134–136, New York, NY, USA, 2018. ACM. doi: 10.1145/3266037.3271626

[201] Amanda Swearngin and Yang Li. Modeling Mobile Interface Tappability Using Crowdsourcing and Deep Learning. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '19, pages 75:1–75:11, New York, NY, USA, 2019. ACM. doi: 10.1145/3290605.3300305

[202] Amanda Swearngin, Chenglong Wang, Alannah Oleson, Amy J. Ko, and James Fogarty. Scout: Rapid Exploration of Interface Layout Alternatives through High-Level Design Constraints. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '20. ACM, 2020. *To Appear*.

[203] Pedro Szekely, Ping Luo, and Robert Neches. Beyond Interface Builders: Model-Based Interface Tools. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '93, pages 383–390, New York, New York, USA, 1993. ACM. doi: 10.1145/169059.169305

[204] Desney S. Tan, Brian Meyers, and Mary Czerwinski. WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Extended Abstacts*, CHI '04, page 1525, New York, New York, USA, 2004. ACM. doi: 10.1145/985921.986106

[205] Michael Terry and Elizabeth D. Mynatt. Side views: Persistent, On-Demand Previews for Open-Ended Tasks. In *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology*, UIST '02, page 71, New York, New York, USA, 2002. ACM. doi: 10.1145/571985.571996

[206] Michael Terry, Elizabeth D. Mynatt, Kumiyo Nakakoji, and Yasuhiro Yamamoto. Variation in Element and Action: Supporting Simultaneous Development of Alternative Solutions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 711–718, New York, NY, USA, 2004. ACM. doi: 10.1145/985692.985782

[207] Yuan Tian, Meiyappan Nagappan, David Lo, and Ahmed E. Hassan. What are the Characteristics of High-Rated Apps? A Case Study on Free Android Applications. In *2015 IEEE International Conference on Software Maintenance and Evolution*, ICSME '15, pages 301–310. IEEE, September 2015. doi: 10.1109/ICSM.2015.7332476

[208] Jenifer Tidwell. *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly Media, Inc., 2010.

[209] Kashyap Todi, Daryl Weir, and Antti Oulasvirta. Sketchplore: Sketch and Explore with a Layout Optimiser. In *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*, DIS '16, pages 543–555, New York, New York, USA, 2016. ACM. doi: 10.1145/2901790.2901817

[210] Maryam Tohidi, William Buxton, Ronald Baecker, and Abigail Sellen. Getting the Right Design and the Design Right. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '06*, page 1243, New York, New York, USA, 2006. ACM. doi: 10.1145/1124772.1124960

[211] Priyan Vaithilingam and Philip J Guo. Bespoke: Interactively Synthesizing Custom GUIs from Command-Line Applications By Demonstration. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, pages 563–576. ACM, 2019.

[212] Khoi Vinh. *Ordering Disorder: Grid Principles for Web Design*. Pearson Education, 2010.

[213] Alex W White. *The Elements of Graphic Design: Space, Unity, Page Architecture, and Type*. Skyhorse Publishing, Inc., 2011.

[214] Jacob O Wobbrock, Htet Htet Aung, Brandon Rothrock, and Brad A Myers. Maximizing the Guessability of Symbolic Input. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems Extended Abstracts*, pages 1869–1872. ACM, 2005. doi: 10.1145/1056808.1057043

[215] Jacob O Wobbrock, Leah Findlater, Darren Gergle, and James J Higgins. The Aligned-Rank Transform for Non-Parametric Factorial Analyses Using Only ANOVA Procedures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 143–146. ACM, 2011.

[216] Ou Wu, Weiming Hu, and Lei Shi. Measuring the Visual Complexities of Web Pages. *ACM Transactions on the Web*, 7(1):1:1–1:34, March 2013. doi: 10.1145/2435215.2435216

[217] Ziming Wu, Taewook Kim, Quan Li, and Xiaojuan Ma. Understanding and Modeling User-Perceived Brand Personality from Mobile Application UIs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '19, pages 213:1–213:12, New York, NY, USA, 2019. ACM. doi: 10.1145/3290605.3300443

[218] Anbang Xu, Shih-Wen Huang, and Brian Bailey. Voyant: Generating Structured Feedback on Visual Designs Using a Crowd of Non-Experts. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, pages 1433–1444. ACM, 2014. doi: 10.1145/2531602.2531604

[219] Pengfei Xu, Hongbo Fu, Takeo Igarashi, and Chiew-Lan Tai. Global Beautification of Layouts With Interactive Ambiguity Resolution. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 243–252, New York, New York, USA, 2014. ACM. doi: 10.1145/2642918.2647398

[220] Yeonsoo Yang and Scott R Klemmer. Aesthetics Matter: Leveraging Design Heuristics to Synthesize Visually Satisfying Handheld Interfaces. In *Proceedings of the SIGCHI Conference Human Factors in Computing Systems Extended Abstracts*, CHI EA '09, pages 4183–4188. ACM, 2009.

[221] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the 22nd Annual ACM Symposium on User*

Interface Software and Technology, UIST '09, pages 183–192, New York, NY, USA, 2009. ACM. doi: 10.1145/1622176.1622213

[222] Loutfouz Zaman, Wolfgang Stuerzlinger, Christian Neugebauer, Rob Woodbury, Maher Elkhaldi, Naghmi Shireen, and Michael Terry. GEM-NI: A System for Creating and Managing Alternatives In Generative Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '15, pages 1201–1210, New York, New York, USA, 2015. ACM. doi: 10.1145/2702123.2702398

[223] Brad Vander Zanden and Brad A. Myers. Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 27–34, New York, New York, USA, 1990. ACM. doi: 10.1145/97243.97248

[224] Brad Vander Zanden and Brad A. Myers. The Lapidary Graphical Interface Design Tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 465–466, New York, New York, USA, 1991. ACM. doi: 10.1145/108844.109005

[225] Clemens Zeidler, Christof Lutteroth, Wolfgang Sturzlinger, and Gerald Weber. The Auckland Layout Editor: An Improved GUI Layout Specification Process. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 343–352, New York, New York, USA, 2013. ACM. doi: 10.1145/2501988.2502007

[226] Clemens Zeidler, Gerald Weber, Wolfgang Stuerzlinger, and Christof Lutteroth. Automatic Generation of User Interface Layouts for Alternative Screen Orientations. In *IFIP Conference on Human-Computer Interaction*, pages 13–35. Springer, 2017.

[227] Mathieu Zen and Jean Vanderdonckt. Towards an Evaluation of Graphical User Interfaces Based on Metrics. In *2014 IEEE Eighth International Conference on Research Challenges in Information Science*, RCIS '14, pages 1–12. IEEE, 2014.

[228] Luke S. Zettlemoyer and Robert St. Amant. A Visual Medium for Programmatic Control of Interactive Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '99, pages 199–206, New York, New York, USA, 1999. ACM. doi: 10.1145/302979.303039

[229] Xiaoyi Zhang, Anne Spencer Ross, Anat Caspi, James Fogarty, and Jacob O. Wobbrock. Interaction Proxies for Runtime Repair and Enhancement of Mobile Application Accessibility. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 6024–6037, New York, New York, USA, 2017. ACM. doi: 10.1145/3025453.3025846

[230] Xiaoyi Zhang, Anne Spencer Ross, and James Fogarty. Robust Annotation of Mobile Application Interfaces in Methods for Accessibility Repair and Enhancement. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, pages 609–621, New York, NY, USA, 2018. ACM. doi: 10.1145/3242587.3242616

[231] Xiaoyi Zhang, Tracy Tran, Yuqian Sun, Ian Culhane, Shobhit Jain, James Fogarty, and Jennifer Mankoff. Interactiles: 3d Printed Tactile Interfaces to Enhance Mobile Touchscreen Accessibility. In *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '18, pages 131–142, New York, New York, USA, 2018. ACM. doi: 10.1145/3234695.3236349

[232] Xianjun Sam Zheng, Ishani Chakraborty, James Jeng-Weei Lin, and Robert Rauschenberger. Correlating Low-level Image Statistics with Users - Rapid Aesthetic and Affective Judgments of Web Pages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1–10, New York, NY, USA, 2009. ACM. doi: 10.1145/1518701.1518703

APPENDIX A

Mixed Initiative Exploration of Design Alternatives

This appendix extends Chapter 3, and includes the full set of formalized constraint specifications that Scout encodes into Z3 to generate layouts (Section A.1). It also includes the Scout evaluation materials including the user study interview script (Section A.4), task instructions (Section A.2), and quality evaluation rubric (Section A.3).

A.1 Formalized Constraint Specifications

Scout creates layouts by generating an assignment of concrete values to a set of variables, which lets it explore many combinations of element arrangement, alignment, position, and size. Scout defines *canvas* variables (e.g., layout grid, margin, baseline grid), *group* variables (e.g., arrangement, alignment, padding), and *element* variables for *position* (e.g., x, y), and *size* (e.g., width, height).

Once Scout produces a valid variable assignment, it outputs a position (e.g., x, y) and size (e.g., *width*, *height*) for each interface element which Scout then translates into the layout canvas that the designer sees (Chapter 3, Section 3.1). Each of Scout's variables has a domain

of values curated from design guidelines [11, 28] along with constraints defining its behavior. Scout uses these constraints, together with a designers high-level constraints, to check the validity of a layout (See Chapter 3, Section 3.2 for an overview of this process).

The following sections detail the layout variables that Scout defines, and the formalized constraint specifications for each of these variables that define their behavior along with a set of basic design quality constraints that Scout enforces for every layout it generates. Within each constraint definition, I will refer to variables for an interface element or the layout canvas using object-oriented notation (i.e., *canvas.margin*, *element.width*). The variable names will be displayed in *italics* within the text descriptions of each constraint.

A.1.1 Position & Size

To explore different element sizes, Scout defines a *size* variable for each element on the canvas *e* with a domain of the form (*width*, *height*, *sizing_factor*), where *sizing_factor* is used to enforce consistent resizing within groups and repeat groups. Scout pre-computes *width* and *height* domains for each element using two strategies: *maintain aspect ratio* and *increase width*.

For both strategies, Scout computes a set of (*width*, *height*, *sizing_factor*) triples along baseline grid increments (i.e., 4), where *width* values range from a minimum determined by element type to the width of the canvas (i.e., *c*.width). For *maintain aspect ratio* elements (e.g., images, icons), *height* values vary from a minimum for each element to the canvas height (i.e., *c*.height). For *increase width only* elements (e.g., buttons, fields), height values do not vary. Scout encodes each pre-computed set of triples as the domain to a *size* variable for each element.

An example of what the final domain might look like for an interface element using *maintain aspect ratio* (e.g., icon) is `[(20,20,1), (24,24,2), ...]`. For an element using *increase width only* (e.g., button), the domain could be `[(120,40,1), (124,40,2), ...]`.

Scout does not encode any sizing specific constraints, however, a majority of the remaining constraints (e.g., Grouping and Arrangement, Emphasis, Ordering) operate on the element *height* and *width* variables. While each element has only one *size* variable, with resulting values, many of the following constraints operate on only part of the (*width*, *height*, *sizing_factor*) triple.

In the constraints in this appendix, I will refer to each part of the triple as an individual variable for an element e (e.g., e.width, e.height, e.sizing_factor).

Scout also defines x and y position variables for each element. Scout does not initialize a domain for the x and y variables, and instead produces an output value for the variables x and y for each element based on the values of other variables and constraints defining their behavior.

A.1.2 Basic Design Quality

Scout encodes 3 basic design quality constraints for each layout to keep inside the bounds of layout canvas and their parent group (Constraint A.9), prevent them from overlapping (Constraint A.2), and enforce minimum sizing constraints (Constraint A.3) based on usability guidelines that recommend minimum sizes for touch targets and text labels.

Scout's **`stay-in-bounds`** constraint (Constraint A.9) enforces that for the set of all elements on the layout canvas E, the bounding box of each element e (i.e., $[e.x, e.y, e.x + e.width, e.y + e.height]$) remains inside the layout canvas c.

$$\phi_{stay\_in\_bounds}(E, c) \stackrel{\text{def}}{=} \bigwedge_{e \in E} (e.\text{x} \geq 0) \wedge (e.\text{x} + e.\text{width} \leq c.\text{width}) \tag{A.1}$$
$$\wedge (e.\text{y} \geq 0) \wedge (e.\text{y} + e.\text{height} \leq c.\text{height})$$

Scout's **`non-overlapping`** constraint (Constraint A.2) prevents overlap for all elements on the layout canvas by enforcing $\phi_{prevent-overlap}$ constraints on the set of groups on the layout canvas G. Scout considers the top-level layout canvas as an additional group in this set. Here, $g.E$ denotes the set of all elements E inside of a group on the layout canvas, and g.padding denotes the within-group padding (i.e., the variable defining the spacing between elements in the group).

$$\phi_{non\_overlapping}(G) \stackrel{\text{def}}{=} \bigwedge_{g \in G} \phi_{prevent\_overlap}(g.\text{E}, g.\text{padding}) \tag{A.2}$$

The constraint $\phi_{prevent\_overlap}$ (Constraint A.1.2) adds constraints for each pair of elements e_i, e_j in a given set of elements E. This constraint requires that the edges of element e_i falls to the top, bottom, left, or right of e_j. The distance between the pair of elements must be equal to the given padding value p. Below, the value of i cannot be equal to j.

$$\phi_{prevent\_overlap}(\mathrm{E}, \mathrm{p}) \stackrel{\text{def}}{=} \bigwedge_{1 \le i < j \le |E|} (e_i.\mathrm{x} + e_i.\mathrm{width} + \mathrm{p} \le e_j.\mathrm{x}) \vee (e_j.\mathrm{x} + e_j.\mathrm{width} + \mathrm{p} \le e_i.\mathrm{x})$$
$$\vee (e_i.\mathrm{y} + e_i.\mathrm{height} + \mathrm{p} \le e_j.\mathrm{y}) \vee (e_j.\mathrm{y} + e_j.\mathrm{height} + \mathrm{p} \le e_i.\mathrm{y})$$

Finally, Scout encodes **`minimum-sizing`** constraints (Constraint A.3) that enforce a minimum height and width for each element e in the set of all elements on the canvas E. This constraint encodes different minimum sizes for *text* and *touch* element types. Each element has a predefined **`type`** property that I statically define within the interface element SVG. Ideally, this would be specified by the designer in their design tool or automatically inferred in future versions.

$$\phi_{minimum\_size}(E) \stackrel{\text{def}}{=} \bigwedge_{e \in E} \begin{cases} e.\mathrm{width} \ge 44 \wedge e.\mathrm{height} \ge 44, & \text{if } e.\mathrm{type} == \text{"touch"} \\ e.\mathrm{height} \ge 16, & \text{if } e.\mathrm{type} == \text{"text"} \end{cases} \tag{A.3}$$

A.1.3 Layout Grid

Scout enforces a set of **`layout grid`** constraints on elements to place them on the layout canvas using a layout grid. Chapter 2 (Section 2.1) describes the behavior of a layout grid in detail. For a brief overview, a layout grid consists of *margins* (i.e., spacing on the outside of the canvas that all elements must be placed inside), *columns* (i.e., vertical containers for placing elements on the canvas), and *gutters* (i.e., spacing between columns where elements must not be placed).

Layout Grid Variables & Domains

In the following constraints the layout canvas c has four variables: *margin*, *columns*, *gutter_width*, and *column_width*. Scout assigns *margin* the domain of `[4-60]` along 4px increments (i.e., 4,8 ...). For *columns*, Scout initializes the domain to `[2,3,4,6,12]` based on design guidelines [11, 28, 212]. While these guidelines can vary in how many columns they recommend using, Material Design guidelines recommend using a 4 column layout grid for mobile app screens [11]. For *gutter_width*, Scout assigns the domain to `[4,8,16]` based on the same design recommendations [11]. Finally, Scout computes the possible *column_width* domain values from the domain values of *margin*, M, *columns*, C, and *gutter_width*, GW, using the following formula:

$$c.\text{column\_width.domain} = \forall_{m \in M} \forall_{c \in C} \forall_{g \in GW} c.\text{width} - (2 * \text{m}) - ((\text{c} - 1) * \text{g}))/\text{c}. \tag{A.4}$$

For each element e and group g directly located on the layout canvas, and not contained inside of a group, Scout assigns a *left_column* and *right_column* variable that define the left and right column on the layout canvas that an element will begin and end in. The domain of these variables ranges from 1 to the maximum number of columns on the canvas (i.e., `[1-6]`).

Layout Grid Constraints

Scout encodes a $\phi_{column\_soundness}$ (Constraint A.5) constraint for each element e_c in the set of elements and groups that are direct children of the canvas E_c, that the *left_column* is less than or equal to the *right_column* variable, and that both variables are less than or equal to the canvas *columns* variable.

$$\phi_{column\_soundness}(c, E_c) \stackrel{\text{def}}{=} \bigwedge_{e_c \in E_c} e_c.\text{left\_column} \leq c.\text{columns} \tag{A.5}$$

$$\wedge e_c.\text{right\_column} \leq c.\text{columns} \wedge e_c.\text{left\_column} \leq e_c.\text{right\_column}$$

Scout also encodes a $\phi_{begins\_in\_column}$ (Constraint A.6) constraint for each element e_c in the set of direct children of the canvas E_c that the x position of the element e_c.x is located on the edge of a column. The constraint computes the edge of a column from the values of the *margin*, *gutter_width*. and *column_width* variables of the canvas and the *left_column* variable of the element.

$$\phi_{begins\_in\_column}(c, E_c) \stackrel{\text{def}}{=} \bigwedge_{e_c \in E_c} e_c.\text{x} = (c.\text{margin} + (c.\text{column\_width} \cdot (e_c.\text{left\_column} - 1)) \quad \text{(A.6)}$$
$$+(c.\text{gutter\_width} \cdot (e_c.\text{left\_column} - 1)))$$

Scout encodes an $\phi_{ends\_in\_column}$ constraint (Constraint A.7) for each element e_c in the set of child elements of the canvas E_c. This constraint requires that the right side of the element e_c.x + e_c.width ends on the right edge of a column. Each element e_c has a *right_column* variable. This constraint states that the elements right edge (i.e., e_c.x + e_c.width) should be equal to the total sum of the canvas *margin*, and width of all columns and gutters to the left of the element's right edge.

$$\phi_{ends\_in\_column}(c, E_c) \stackrel{\text{def}}{=} \bigwedge_{e_c \in E_c} (e_c.\text{x} + e_c.\text{width}) = (c.\text{margin} \quad \text{(A.7)}$$
$$+(c.\text{col\_width} \cdot e_c.\text{right\_column}) + (c.\text{gutter\_width} \cdot (e_c.\text{right\_column} - 1)))$$

A.1.4 Baseline Grid

Baseline grids define the vertical spacing of a design, aid horizontal alignment, and create hierarchy [28]. They consist of horizontal lines at even intervals to which all components should align (further described in Chapter 2, Section 2.1). Scout defines a set of **`baseline grid`** constraints to ensure it places all elements on baseline grid lines. For the layout canvas c, Scout defines a *baseline_grid* variable that allows designers to examine different baseline grid options. Scout initializes the domain of this variable to `[4,8,16]`, based on design guidelines for baseline grid selection [11].

Baseline Grid Constraints

The $\phi_{baseline\_grid}$ constraint (Constraint A.8) assures that the y position of each element e_c of the set of all direct children of the canvas E_c is placed on a baseline grid line. That is, the y position of the element e_c should be a multiple of the current value of the canvas *baseline_grid* variable. For each text element e_c, the constraint specifies that the *baseline* variable, rather than the y position variable, should be a multiple of the canvas *baseline_grid* value. This is because text should always be vertically aligned by its baseline, and not the top or bottom of the text element which can change depending on the text element's font type.

$$\phi_{baseline\_grid}(E_c, c) \stackrel{\text{def}}{=} \bigwedge_{e_c \in E_c} \begin{cases} ((e_c.\text{baseline} \mod c.\text{baseline\_grid}) = 0) & \text{if } e_c.\text{type} == \text{"text"} \\ ((e_c.\text{y} \mod c.\text{baseline\_grid}) = 0) & \text{otherwise} \end{cases} \tag{A.8}$$

A.1.5 Grouping & Arrangement Constraints

Scout uses **`arrangement`** constraints to place elements within a group. Arrangement constraints work in combination with **`alignment`**, **`order`**, and **`visual hierarchy`** constraints, and Scout conditionally applies them based on the current value of the arrangement variable. Each group g on the layout canvas c has an *arrangement* variable with a domain of `[horizontal, vertical, rows, columns]`, and an *alignment* variable with a domain of `[left, top, x-center, y-center, right, bottom]`. Each group g has a *padding* variable with a domain of `[4, ..., 100]` which Scout computes along baseline grid increments.

Arrangement

The first constraint that Scout applies for **`arrangement`** constraints is $\phi_{stay\_in\_bounds}$ (Constraint A.9). For every group on the canvas g and for the set of all elements in a group E, this constraint requires that all edges of an element (i.e., top, bottom, left, right) fall within

the parent group.

$$\phi_{stay\_in\_bounds}(E, g) \stackrel{\text{def}}{=} \bigwedge_{e \in E} (e.\text{x} \geq g.\text{x}) \wedge (e.\text{x} + e.\text{width} \leq g.\text{x} + g.\text{width}) \tag{A.9}$$
$$\wedge (e.\text{y} \geq g.\text{y}) \wedge (e.\text{y} + e.\text{height} \leq g.\text{y} + g.\text{height})$$

Scout encodes arrangement behavior by setting a maximum height and width on the size of a group that ensures all elements fit within the group bounding box. Scout supports four types of arrangements including `horizontal`, `vertical`, `rows`, and `columns`. Scout applies conditional arrangement constraints based on the current value of the arrangement variable.

To encode the behavior of each arrangement value, Scout uses $\phi_{set\_size\_main\_axis}$ (Constraint A.10) and $\phi_{set\_size\_cross\_axis}$ (Constraint A.15) which require a group to fit within a bounding box based on the size of the elements the group contains. When the *order* of a group is *important*, Scout encodes $\phi_{arrange\_container}$ constraints that enforce a specific order of elements in a group (Constraint A.16). Otherwise, the arrangement of a group is enforced by the bounding box, and $\phi_{non\_overlapping}$ constraints (Constraint A.2) that enforce a specific amount of *padding* between elements based on the value of the *padding* variable. Scout also aligns elements within a group using $\phi_{align\_container}$ constraints (Constraint A.28).

Scout applies $\phi_{set\_size\_main\_axis}$ (Constraint A.10) to the set of all groups on the canvas G. When *arrangement* is `horizontal`, the constraint requires that the *width* of the group be equal to the value returned by `sum_widths(g.E, g.padding)` (Method A.11) which returns the widths of all elements and inner padding between elements. When *arrangement* is `vertical`, the constraint requires that the *height* of the group be equal to the value returned by `sum_heights(g.E, g.padding)` (Method A.12) which returns the height of all elements and inner padding between elements. In both conditions, $g.E$ refers to the set of all elements in a group and g.padding refers to the group *padding* variable.

When *arrangement* is `rows` or `columns`, the constraint relies on an algorithm that first splits the elements in the group into rows and column groups (Algorithm 1) based on the number of elements in the group to ensure a balanced number of elements across rows and columns ($\sqrt{|E|}$).

Algorithm 1 Algorithm for splitting a group of elements into row and column groups.

```
function BALANCED_ROW_SPLIT(E, g)
    g.RC ← [ ]                          ▷ Collection that will hold the set of all row and column groups
    n_groups ← √E.len
    per_group ← E.len/n_groups          ▷ Splits the elements into equal length groups
                                        ▷ with the remainder in the last row or column group
    curr_group ← [ ]
    if g.order ≠ "important" then       ▷ Before dividing, randomize the order of elements
        if E[0].order = "first" and E[E.len − 1].order = "last" then
            inner ← randomize(E[1 : E.len − 1])     ▷ Shuffle the middle elements in the list
            inner ← inner.insertFront(E[0])
            E ← inner.insert(E[E.len − 1])
        else if E[0].order = "first" then
            last ← randomize(E[1 : E.len])          ▷ Shuffle non-first elements in the list
            E ← last.insertFront(E[0])
        else if E[E.len − 1].order = "last" then
            first ← randomize(E[0 : E.len − 1])     ▷ Shuffle non-last elements in the list
            E ← first.insert(E[E.len − 1])
        end if
    end if

    for i ← 1 to E.len do               ▷ Divide the elements into subgroups
        if curr_group.len < per_group then
            curr_group.insert(E[i])
        else if curr_group.len = per_group then     ▷ Begin inserting into the next group
            g.RC.insert(curr_group)
            curr_group ← [ ]
            curr_group.insert(E[i])
        end if
    end for

    if current_group.len > 0 then       ▷ Add the last group to the collection RC
        g.RC.insert(curr_group)
    end if
end function
```

Then, Scout arranges elements within each row or column using the same set of constraints it applies for horizontal and vertical arrangements, respectively. Algorithm 1 considers the current *order* value that the designer has set for the group and for elements in the group, and if *order* is not *important*, Scout will randomize the order of elements before placing them into row and column groups. The output of this procedure is $g.RC$ which represents the set of row and column groups that Scout has split the elements of the group into. Scout then applies the following arrangement constraints as specified to the set of row and column groups.

Scout currently supports a static number of columns or rows for a group, using Algorithm 1, determined by the number of elements in the group. However, it could be possible to support dynamic numbers of columns, and future work could explore creating more flexible encodings to support a more diverse set of row and column arrangements.

In the following constraints, g.RC refers to the set of all column or row groups that the elements in the group have been divided into. $\phi_{set\_size\_main\_axis}$ applies the helper methods **`sum_row_heights`**(g.RC, g.padding) and **`sum_row_widths`**(g.RC, g.padding) which restrict the *height* and *width* of the group to the total height and width of all rows and columns, when *arrangement* is **`rows`** or **`columns`**, respectively.

$$\phi_{set\_size\_main\_axis}(G) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|G|} \begin{cases} g.\text{width} = \text{sum\_widths}(g.E, g.\text{padding}) & \text{if g.arrangement} = \text{"horizontal"} \\ g.\text{height} = \text{sum\_heights}(g.E, g.\text{padding}) & \text{if g.arrangement} = \text{"vertical"} \\ \text{g.height} = \text{sum\_row\_heights}(\text{g.RC, g.padding}) & \text{if g.arrangement} = \text{"rows"} \\ \text{g.width} = \text{sum\_column\_widths}(\text{g.RC, g.padding}) & \text{if g.arrangement} = \text{"columns"} \end{cases} \tag{A.10}$$

Scout uses the following helper methods conditionally when group *arrangement* is

`horizontal` and `vertical` to return an expression of the sum of widths (Method A.11) and heights (Method A.12) of all elements in a given set of elements E. The expression for each will also add an amount of padding p to the sum for every element except for the last element in the set (i.e., $i = |E|$). This ensures that there is some padding between elements in the group.

$$\text{sum\_widths}(E,p) \stackrel{\text{def}}{=} \sum_{i=1}^{|E|} \begin{cases} e_i.\text{width} & \text{if } i = |E| \\ e_i.\text{width} + p & \text{otherwise} \end{cases} \tag{A.11}$$

$$\text{sum\_heights}(E,p) \stackrel{\text{def}}{=} \sum_{i=1}^{|E|} \begin{cases} e_i.\text{height} & \text{if } i = |E| \\ e_i.\text{height} + p & \text{otherwise} \end{cases} \tag{A.12}$$

Scout uses the following helper methods conditionally when group *arrangement* is `rows` or `columns` to set the height and width of a group. Method A.13 computes an expression summing for each row rc in the set of all row groups RC the maximum height element in the row group plus an amount of *padding* for each row excluding the last row. Method A.14 works similarly. For each column rc in the set of all column groups RC, it computes an expression that sums the maximum height element in each column group plus an amount of *padding* for each column excluding the last column.

$$\text{sum\_row\_heights}(RC,p) \stackrel{\text{def}}{=} \sum_{i=1}^{|RC|} \begin{cases} \text{max\_height}(rc) & \text{if } i = |RC| \\ \text{max\_height}(rc) + p & \text{otherwise} \end{cases} \tag{A.13}$$

$$\text{sum\_column\_widths}(RC,p) \stackrel{\text{def}}{=} \sum_{i=1}^{|RC|} \begin{cases} \text{max\_width}(rc) & \text{if } i = |RC| \\ \text{max\_width}(rc) + p & \text{otherwise} \end{cases} \tag{A.14}$$

The following constraint, $\phi_{set\_size\_cross\_axis}$ (Constraint A.15), works in combination with

$\phi_{set\_size\_main\_axis}$ (Constraint A.10) to enforce an arrangement of elements within a group of elements where elements do not necessarily have a specified order. Scout applies the $\phi_{set\_size\_cross\_axis}$ to the set of all groups on the canvas G. If the group *arrangement* is `horizontal`, the constraint requires that the *height* of the group be equal to the height of the maximum height element in the group (i.e., $\texttt{max\_height}(g.E)$). If the group *arrangement* is `vertical`, the constraint requires that the *width* of the group be equal to the width of the maximum width element in the group (i.e., $\texttt{max\_width}(g.E)$). In the following constraints, $g.E$ refers to the set of all elements in a group g. When *arrangement* is `rows`, the constraint requires that the group *width* be equal to the width of the maximum width row. When *arrangement* is `columns`, the constraint requires that the group *height* be equal to the height of the maximum height column.

$$\phi_{set\_size\_cross\_axis}(G) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|G|} \begin{cases} g.\text{height} = \text{max\_height}(g.E) & \text{if } g.\text{arrangement} = \text{"horizontal"} \\ g.\text{width} = \text{max\_width}(g.E) & \text{if } g.\text{arrangement} = \text{"vertical"} \\ g.\text{width} = \text{max\_width}(g.RC) & \text{if } g.\text{arrangement} = \text{"rows"} \\ g.\text{height} = \text{max\_height}(g.RC) & \text{if } g.\text{arrangement} = \text{"columns"} \end{cases} \tag{A.15}$$

Order

Scout applies the arrangement sizing constraints (A.10, A.15) for every group on the canvas. With only the above constraints specified, their relative position in the order within the group can change across alternatives. However, it is also possible for designers to set the *order* of a group to be *important*. In this case, Scout applies an additional set of pairwise arrangement constraints on consecutive element pairs within a group to ensure they appear in the order that the designer has specified. For each group g in the set of all groups on the canvas G, Scout applies $\phi_{arrange\_container}$ (Constraint A.16) which states if the value of the group *arrangement* variable is `horizontal` or `vertical`, Scout should apply the $\phi_{arrange\_horizontal}$ (Constraint A.19) or $\phi_{arrange\_vertical}$ (Constraint A.20) constraint, respectively.

If the group *arrangement* is `rows`, Scout applies $\phi_{arrange\_rows}$ which applies for each row group rc in the set of row groups RC that Scout has divided the group into, $\phi_{arrange\_horizontal}$, which arranges the elements within a row group horizontally. Similarly, if the group *arrangement* is `columns`, Scout applies $\phi_{arrange\_columns}$ which applies for each column group rc in the set of column groups RC that Scout has divided the group into, $\phi_{arrange\_vertical}$, which arranges the elements within a column group vertically.

$$\phi_{arrange\_container}(G) \stackrel{\text{def}}{=} \bigwedge_{g \in G} \begin{cases} \phi_{arrange\_horizontal}(g.\text{E}, g.\text{padding}) & \text{if g.arrangement} = \text{"horizontal"} \\ \phi_{arrange\_vertical}(g.\text{E}, g.\text{padding}) & \text{if g.arrangement} = \text{"vertical"} \\ \phi_{arrange\_rows}(g.\text{RC}, g.\text{padding}) & \text{if g.arrangement} = \text{"rows"} \\ \phi_{arrange\_columns}(g.\text{RC}, g.\text{padding}) & \text{if g.arrangement} = \text{"columns"} \end{cases} \tag{A.16}$$

$$\phi_{arrange\_rows}(RC, p) \stackrel{\text{def}}{=} \bigwedge_{rc \in RC} \phi_{arrange\_horizontal}(rc.\text{E}, p) \tag{A.17}$$

$$\phi_{arrange\_columns}(RC, p) \stackrel{\text{def}}{=} \bigwedge_{rc \in RC} \phi_{arrange\_vertical}(rc.\text{E}, p) \tag{A.18}$$

The constraint $\phi_{arrange\_horizontal}$ (Constraint A.19) encodes a conjunction of pairwise constraints for every consecutive pair of elements e_i, e_{i+1} in a set of elements E that state that e_{i+1} should fall to the right of e_i with an additional amount of horizontal padding p between the

elements. The padding comes from the *padding* variable of the parent group or canvas.

$$\phi_{arrange\_horizontal}(E,p) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|E|-1} (e_{i+1}.\text{x} = e_i.\text{x}) + (e_i.\text{width} + p) \tag{A.19}$$

The constraint $\phi_{arrange\_vertical}$ (Constraint A.20) encodes a conjunction of pairwise constraints for every pair of consecutive elements e_i, e_{i+1} in a set of elements E stating that e_{i+1} should fall below e_i with an additional amount of vertical padding p between the elements. The padding comes from the *padding* variable of the parent group or canvas.

$$\phi_{arrange\_vertical}(E,p) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|E|-1} (e_{i+1}.\text{y} = e_i.\text{y}) + (e_i.\text{height} + p) \tag{A.20}$$

Order First or Last

In Scout, a designer can set an element to be *first* or *last* in a group or on the canvas. Scout will only allow a designer to set the element as first or last if it appears in that position in the Outline panel hierarchy. The following constraints encode this behavior. For every group on the canvas g and for all elements within a group E_g, $\phi_{order\_first\_or\_last}$ applies the constraint ϕ_{order_first} if the first element E_{g_0} has *order* set to *first*, and applies the constraint ϕ_{order_last} if the last element $E_{g_{|E|-1}}$ has the *order* set to *last*.

$$\phi_{order\_first\_or\_last}(E_g, g) \stackrel{\text{def}}{=} \begin{cases} \phi_{order\_first}(E_{g_0}, g) & \text{if } E_{g_0}.\text{order} = \text{"first"} \\ \phi_{order\_last}(E_{g_{|E|-1}}, g) & \text{if } E_{g_{|E|-1}}.\text{order} = \text{"last"} \end{cases} \tag{A.21}$$

Scout applies the ϕ_{order_first} constraint (Constraint A.22) based on the value of the *arrangement* variable. If the group *arrangement* is `horizontal` or `rows`, the element should be placed at the beginning of the group (i.e., $g.x$) and if the group *arrangement* is `vertical` or `columns`, the element should be placed at the top of the group (i.e., $g.y$). Scout ensures that if group *arrangement* is `rows` or `columns`, it places an element with *order* of *first* into the first row or column group in $g.RC$ and places an element with *order* of *last* into the last row or column

group in $g.RC$, using Algorithm 1.

$$\phi_{order\_first}(e,g) \stackrel{\text{def}}{=} \begin{cases} e.\text{x} = g.\text{x} & \text{if g.arrangement} = \text{"horizontal"} \\ & \vee g.\text{arrangement} = \text{"rows"} \\ e.\text{y} = g.\text{y} & \text{if g.arrangement} = \text{"vertical"} \\ & \vee g.\text{arrangement} = \text{"columns"} \end{cases} \tag{A.22}$$

Similarly, Scout applies the ϕ_{order_last} constraint based on the value of the *arrangement* variable. If the group *arrangement* is `horizontal`, the element should be placed at the right end of the group (i.e., $g.x + g.\text{width}$) and if the group *arrangement* is `vertical`, the element should be placed at the bottom of the group (i.e., $g.y + g.\text{height}$).

$$\phi_{order\_last}(e,g) \stackrel{\text{def}}{=} \begin{cases} (e.\text{x} + e.\text{width}) = (g.\text{x} + g.\text{width}) & \text{if } g.\text{arrangement} = \text{"horizontal"} \\ (e.\text{y} + e.\text{height}) = (g.\text{y} + g.\text{height}) & \text{if } g.\text{arrangement} = \text{"vertical"} \end{cases} \tag{A.23}$$

Scout enforces the ordering constraints for elements inside of a group g in combination with the `arrangement` constraints specified above. However, the top level canvas c does not have an *arrangement* variable and uses the `layout_grid` constraints to place elements. The following constraint ϕ_{order_canvas} (Constraint A.24) enforces the order of elements as specified by the designer in Scout's Outline panel when the designer has set the *canvas order* property to `important`. For each element in the set of direct child elements of the canvas E_c, the bottom edge, $e_{c_i}.y + e_{c_i}.\text{height}$, should be above the bottom edge of the element that comes after it in the hierarchy, $e_{c_{i+1}} + e_{c_{i+1}}.\text{height}$ (Constraint A.25), or the bottom edge should be equal to the element that comes after it in the hierarchy and also be placed to the left of the element that

comes after it in the hierarchy (Constraint A.26).

$$\phi_{order\_canvas}(E_c, c) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|E_c|-1} \Big\{ \phi_{above}(e_{c_i}, e_{c_{i+1}}) \vee \phi_{left}(e_{c_i}, e_{c_{i+1}}) \quad \text{if } c.\text{order} = \text{"important"} \tag{A.24}$$

$$\phi_{above}(e_1, e_2) \stackrel{\text{def}}{=} (e_1.\text{y} + e_1.\text{height}) < (e_2.\text{y} + e_2.\text{height}) \tag{A.25}$$

$$\phi_{left}(e_1, e_2) \stackrel{\text{def}}{=} ((e_1.\text{y} + e_1.\text{height}) = (e_2.\text{y} + e_2.\text{height})) \wedge ((e_1.\text{x} + e_1.\text{width}) \leq (e_2.\text{x} + e_2.\text{width})) \tag{A.26}$$

To support the designer in setting *order* `first` and `last` for elements directly on the canvas and not in a group, Scout applies $\phi_{order\_first\_canvas}$ (Constraint A.21) which creates pairwise constraints for each pair of elements e_i, e_j in the set of all elements directly on the canvas E_c and not contained within a group. These constraints specify that if an element e_i has *order* `first`, and e_j does not, e_i should be placed above (Constraint A.25) or to the left (Constraint A.26) of e_j. Similarly, if an element e_i has *order* `last`, and e_j does not, e_j should be placed above or to the left of e_i.

$$\phi_{order\_first\_canvas}(E_c, c) \stackrel{\text{def}}{=} \bigwedge_{\substack{i=1 \\ i \neq j}}^{|E_c|} \bigwedge_{j=1}^{|E_c|} \begin{cases} \phi_{above}(e_i, e_j) \vee \phi_{left}(e_i, e_j) & \text{if } e_i.\text{order} = \text{"first"} \wedge \\ & e_j.\text{order} \neq \text{"first"} \\ \phi_{above}(e_j, e_i) \vee \phi_{left}(e_j, e_i) & \text{if } e_i.\text{order} = \text{"last"} \wedge \\ & e_j.\text{order} \neq \text{"last"} \end{cases} \tag{A.27}$$

Alignment

Scout encodes $\phi_{align\_container}$ constraints (Constraint A.28) based on the current value of a group's *arrangement* variable. For the set of all groups on the canvas G, $\phi_{align\_container}$ specifies that if the *arrangement* of the group is `vertical`, the $\phi_{align\_vertical}$ constraint should be applied, and if the *arrangement* of the group is `horizontal`, the $\phi_{align\_horizontal}$ constraint should be applied. When *arrangement* is `rows`, ϕ_{align_rows} should be applied to the set of all row groups $g.RC$ in the group g. When *arrangement* is `columns`, $\phi_{align\_columns}$ should be applied to the set of all column groups $g.RC$ in the group g.

$$\phi_{align\_container}(G) \stackrel{\text{def}}{=} \bigwedge_{g \in G} \begin{cases} \phi_{align\_vertical}(g.\text{E}, g) & \text{if } g.\text{arrangement} = \text{"vertical"} \\ \phi_{align\_horizontal}(g.\text{E}, g) & \text{if } g.\text{arrangement} = \text{"horizontal"} \\ \phi_{align\_rows}(g.\text{RC}, g) & \text{if } g.\text{arrangement} = \text{"rows"} \\ \phi_{align\_columns}(g.\text{RC}, g) & \text{if } g.\text{arrangement} = \text{"columns"} \end{cases} \tag{A.28}$$

The constraint $\phi_{align\_horizontal}$ (Constraint A.29) specifies the horizontal alignment behavior of a group. For the set of all elements E in a group g, Scout encodes constraints based on the value of the group *alignment* variable. The *alignment* variable has three horizontal values: `top`, `y-center`, and `bottom`. Based on these values, the constraint aligns elements relative to the top, y-center, and bottom of the group. For *text* elements, $\phi_{align\_horizontal}$ aligns the *baseline* of the text to the bottom of the group, rather than aligning the bottom of the text line which could

include descenders.

$$\phi_{align\_horizontal}(E,g) \stackrel{\text{def}}{=} \bigwedge_{e \in E} \begin{cases} e.\text{y} = g.\text{y} & \text{if } g.\text{alignment} = \text{"top"} \\ (e.\text{y} + e.\text{height}/2) = (g.\text{y} + g.\text{height}/2) & \text{if } g.\text{alignment} = \text{"y-center"} \\ e.\text{baseline} = (g.\text{y} + g.\text{height}) & \text{if } g.\text{alignment} = \text{"bottom"} \\ & \wedge e.\text{type} = \text{"text"} \\ (e.\text{y} + e.\text{height}) = (g.\text{y} + g.\text{height}) & \text{if } g.\text{alignment} = \text{"bottom"} \\ & \wedge e.\text{type} \neq \text{"text"} \end{cases} \tag{A.29}$$

The constraint $\phi_{align\_vertical}$ (Constraint A.30) specifies the vertical alignment behavior of a group. For the set of all elements E in a group g, Scout encodes constraints based on the value of the group *alignment* variable. The *alignment* variable has three vertical values: **`left`**, **`x-center`**, and **`right`**. Based on these values, the constraint aligns elements relative to the left, x-center, and right of the group.

$$\phi_{align\_vertical}(E,g) \stackrel{\text{def}}{=} \bigwedge_{e \in E} \begin{cases} e.\text{x} = g.\text{x} & \text{if } g.\text{alignment} = \text{"left"} \\ (e.\text{x} + e.\text{width}/2) = (g.\text{x} + g.\text{width}/2) & \text{if } g.\text{alignment} = \text{"x-center"} \\ (e.\text{x} + e.\text{width}) = (g.\text{x} + g.\text{width}) & \text{if } g.\text{alignment} = \text{"right"} \end{cases} \tag{A.30}$$

Scout applies the constraint ϕ_{align_rows} (Constraint A.31) when the group *arrangement* is **`rows`**. ϕ_{align_rows} applies for every consecutive pair of row groups rc_i, constraints that left align the pair of first elements of each row group $rc_i.e_1, rc_{i+1}.e_1$ and place the second element in the pair $rc_{i+1}.e_1$ below the first element of the pair $rc_i.e_1$ to begin the second and following rows in the group. Scout also applies the constraint $\phi_{align\_horizontal}$ to align elements within a

horizontally arranged row group according to the group *alignment* value.

$$\phi_{align\_rows}(RC, g) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|RC|-1} \begin{cases} \phi_{align\_left\_and\_below}(rc_i.e_1, rc_{i+1}.e_1, g.\text{padding}) & \\ \wedge \phi_{align\_horizontal}(rc_i.\text{E}, g) & \text{if } i < |RC| \\ \\ \phi_{align\_horizontal}(rc_i.\text{E}, g) & \text{otherwise} \end{cases} \tag{A.31}$$

Scout applies the constraint $\phi_{align\_columns}$ (Constraint A.32) when the group *arrangement* is `columns`. $\phi_{align\_columns}$ applies for every consecutive pair of column groups rc_i, constraints that top align the pair of first elements of each column group $rc_i.e_1, rc_{i+1}.e_1$ and place the second element in the pair $rc_{i+1}.e_1$ to the right of the first element of the pair $rc_i.e_1$ to begin the second and following columns in the group. Scout also applies the constraint $\phi_{align\_vertical}$ to align elements within a vertically arranged column group according to the group *alignment* value.

$$\phi_{align\_columns}(RC, g) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|RC|} \begin{cases} \phi_{align\_right\_and\_top}(rc_i.e_1, rc_{i+1}.e_1, g.\text{padding}) & \\ \wedge\ \phi_{align\_vertical}(rc_i.\text{E}, g) & \text{if } i < |RC| \\ \\ \phi_{align\_vertical}(rc_i.\text{E}, g) & \text{otherwise} \end{cases} \tag{A.32}$$

Scout uses the following constraint $\phi_{align\_left\_and\_below}$ (Constraint A.33) for a pair of elements e_1, e_2 to align the second element to the left and below the first element, along with a variable amount of padding p between the elements provided by the parent group or canvas.

$$\phi_{align\_left\_and\_below}(e_1, e_2, p) \stackrel{\text{def}}{=} (e_1.x = e_2.x) \wedge ((e_1.y + e_1.\text{height} + p) = e_2.y) \tag{A.33}$$

Scout uses the following constraint $\phi_{align\_right\_and\_top}$ (Constraint A.34) for a pair of elements e_1, e_2 to align the second element to the right and top of the first element, along with a variable amount of padding p between the elements provided by the parent group or canvas.

$$\phi_{align\_right\_and\_top}(e_1, e_2, p) \stackrel{\text{def}}{=} (e_1.y = e_2.y) \wedge ((e_1.x + e_1.\text{width} + p) = e_2.x) \tag{A.34}$$

Visual Hierarchy

To help maintain visual hierarchy within a group, Scout encodes $\phi_{group\_hierarchy}$ constraints (Constraint A.35) that require the *padding* variable of the group to be less than the *height* or *width* of the minimum height or width element. For vertical arrangements (i.e., `vertical`, `columns`), Scout requires that the *padding* variable be less than or equal to the element in the group with the smallest height. For horizontal arrangements (i.e., `horizontal`, `rows`), Scout requires that the *padding* variable be less than or equal to the element in the group with the smallest width.

$$\phi_{group\_hierarchy}(G) \stackrel{\text{def}}{=} \bigwedge_{g \in G} \begin{cases} g.\text{padding} \leq \text{min\_height(g.E)} & \text{if g.arrangement} = \text{"vertical"} \\ & \vee g.\text{arrangement} = \text{"columns"} \\ g.\text{padding} \leq \text{min\_width(g.E)} & \text{if g.arrangement} = \text{"horizontal"} \\ & \vee g.\text{arrangement} = \text{"rows"} \end{cases} \tag{A.35}$$

Scout also encodes pairwise $\phi_{canvas\_hierarchy}$ constraints (Constraint A.36) that require a minimum amount of spacing between elements on the canvas. These constraints are only applied if one or more of the elements in a pair is a group, and are meant to ensure that elements outside a group do not appear to be a part of neighboring groups. This constraint encodes pairwise constraints for the set of all elements on the canvas E_c, and not within a group. For each pair e_{c_i}, e_{c_j}, if they are both groups, the minimum padding between them should be 2 times the maximum padding in either of the groups. If e_{c_i} is a group and e_{c_j} is not, the minimum padding between the paired elements should be 2 times the padding of e_{c_i}. Conversely, if e_{c_j} is a group, and e_{c_i} is not, the minimum padding between the paired elements should be 2 times the padding of e_{c_j}. I selected the value of 2 experimentally with different mobile hierarchies of

elements. In the following constraint, the value of i cannot be equal to the value of j.

$$\phi_{canvas\_hierarchy}(E_c, c) \stackrel{\text{def}}{=} \bigwedge_{1 \le i < j \le |E_c|} \begin{cases} \phi_{min\_padding}(e_{c_i}, e_{c_j}, & \text{if } (\mathrm{e}_{c_i}.\text{is\_group} \wedge \\ \text{max\_padding}(\mathrm{e}_{c_i}, e_{c_j}) \cdot 2) & e_{c_j}.\text{is\_group}) \\ \phi_{min\_padding}(e_{c_i}, e_{c_j}, e_{c_i}.\text{padding} \cdot 2) & \text{if } e_{c_i}.\text{is\_group} \\ \phi_{min\_padding}(e_{c_i}, e_{c_j}, e_{c_j}.\text{padding} \cdot 2) & \text{if } e_{c_j}.\text{is\_group} \end{cases} \tag{A.36}$$

The $\phi_{canvas\_hierarchy}$ constraint applies the $\phi_{min\_padding}$ constraint to a given pair of elements e_1, e_2 and with a padding argument p, which corresponds to the minimum distance that must be maintained between the two elements in any direction.

$$\phi_{min\_padding}(e_1, e_2, \mathrm{p}) \stackrel{\text{def}}{=} ((e_1.\mathrm{x} + e_1.\text{width} + \mathrm{p}) \le e_2.\mathrm{x}) \vee \tag{A.37}$$
$$((e_2.\mathrm{x} + e_2.\text{width} + \mathrm{p}) \le e_1.\mathrm{x}) \vee ((e_1.\mathrm{y} + e_1.\text{height} + \mathrm{p}) \le e_2.\mathrm{y}) \vee ((e_2.\mathrm{y} + e_2.\text{height} + \mathrm{p}) \le e_1.\mathrm{y})$$

A.1.6 Emphasis Constraints

Scout encodes emphasis constraints when a designer has specified through the Outline panel that an element should have **`high`** or **`low`** emphasis. The first constraint that Scout encodes for these elements is $\phi_{size\_increase\_or\_decrease\_only}$ (Constraint A.38) which specifies for each element on the canvas e, if emphasis e.emph is **`high`** that its area should only increase from its original area (i.e., the size of the element the designer imports into the Outline panel). If emphasis e.emph is **`low`** for the element, its area should only decrease from its original area.

$$\phi_{size\_increase\_or\_decrease\_only}(E) \stackrel{\text{def}}{=} \bigwedge_{e \in E} \begin{cases} (e.\text{area} > e.\text{orig\_area}) & \text{if } e.\text{emph} = \text{"high"} \\ (e.\text{area} < e.\text{orig\_area}) & \text{if } e.\text{emph} = \text{"low"} \end{cases} \tag{A.38}$$

Scout also encodes constraints to adjust the relative size of emphasized and non-emphasized elements in relation to each other to give them more or less emphasis. Scout encodes a

$\phi_{size\_larger\_or\_smaller}$ constraint (Constraint A.39) that specifies for each pair of elements on the canvas e_i, e_j that if the emphasis level of e_i is `high` and the emphasis level of e_j is not, that e_i should have a larger *width* or *height*. Conversely, if the emphasis level of e_i is `low` and the emphasis level of e_j is not, e_i should have a smaller *width* or *height*.

$$\phi_{size\_larger\_or\_smaller}(E) \stackrel{\text{def}}{=} \bigwedge_{\substack{1\le i,j\le|E|\\ i\neq j}} \begin{cases} (\text{e}_i.\text{width} > e_j.\text{width} \vee & \text{if } (\text{e}_i.\text{emph} = \text{"high"} \wedge \\ e_i.\text{height} > e_j.\text{height}) & \text{e}_j.\text{emph} \neq \text{"high"}) \\ (\text{e}_i.\text{width} < e_j.\text{width} \vee & \text{if } (\text{e}_i.\text{emph} = \text{"low"} \wedge \\ e_i.\text{height} < e_j.\text{height}) & \text{e}_j.\text{emph} \neq \text{"low"}) \end{cases} \tag{A.39}$$

Finally, Scout encodes $\phi_{order\_before\_or\_after}$ constraints (Constraint A.40) to adjust the relative position of emphasized elements to place earlier or later in the order than non-emphasized elements. The constraint specifies for each pair of elements on the canvas e_i, e_j that if the emphasis level of e_i is `high` and the emphasis level of e_j is not, and if the area of e_i is less than the area of e_j, the element should be placed earlier in the order to give it more emphasis than the other element.

$$\phi_{order\_before\_or\_after}(E) \stackrel{\text{def}}{=} \bigwedge_{\substack{1\le i,j\le|E|\\ i\neq j}} \begin{cases} e_i.\text{y} < e_j.\text{y} & \text{if } (\text{e}_i.\text{emph} = \text{"high"} \wedge e_j.\text{emph} \neq \text{"high"} \\ & \wedge\, e_i.\text{area} \le e_j.\text{area}) \\ e_i.\text{y} > e_j.\text{y} & \text{if } (\text{e}_i.\text{emph} = \text{"low"} \wedge e_j.\text{emph} \neq \text{"low"} \\ & \wedge\, e_i.\text{area} \ge e_j.\text{area}) \end{cases} \tag{A.40}$$

A.1.7 Alternate Group Constraints

For alternate groups, Scout encodes $\phi_{alternate\_group}$ constraints (Constraint A.41) which apply a set of disjunctions requiring that the representation variable, g.rep, have a value corresponding to the ID of an element that the designer has grouped inside of the alternate group. The representation variable determines which SVG element Scout displays within a layout canvas. The domain of g.rep is thus a set of unique IDs corresponding to the child elements of the alternate group.

$$\phi_{alternate\_group}(G) \stackrel{\text{def}}{=} \bigwedge_{g \in G} \left(\bigvee_{a \in g.A} (g.\text{rep} = a) \right) \tag{A.41}$$

A.1.8 Repeat Group Constraints

For repeat groups, Scout encodes three kinds of constraints. First, Scout encodes a $\phi_{repeat\_group\_vars\_same}$ constraint (Constraint A.43) for each repeat group in the set of all repeat groups, G_r. This constraint applies a $\phi_{match\_subgroup\_vars}$ constraint (Constraint A.43) on each consecutive pair of subgroups, g_i and g_{i+1}. The constraint $\phi_{match\_subgroup\_vars}$ encodes a set of conjunctions on a pair of groups g_1, g_2 requiring the *arrangement*, *alignment*, and *padding* variables of g_1 and g_2 to be the same.

$$\phi_{repeat\_group\_vars\_same}(G_r) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|G|-1} \phi_{match\_subgroup\_vars}(g_i, g_{i+1}) \tag{A.42}$$

$$\begin{aligned} \phi_{match\_subgroup\_vars}(g_1, g_2) \stackrel{\text{def}}{=} & (g_1.\text{arrangement} = g_2.\text{arrangement}) \\ & \wedge (g_1.\text{alignment} = g_2.\text{alignment}) \wedge (g_1.\text{padding} = g_2.\text{padding}) \end{aligned} \tag{A.43}$$

The second type of constraint that Scout encodes for repeat groups is $\phi_{size\_change\_same}$ (Constraint A.44), which requires that the size increase or decrease of *corresponding pairs* of elements (i.e., two elements that have the same position in the group order within different subgroups) be

the same. For each repeat group in the set of all repeat groups G_r, Scout encodes a $\phi_{size\_change\_same}$ constraint which then applies $\phi_{match\_element\_sizes}$ for each consecutive pair of subgroups in the repeat group.

$$\phi_{size\_change\_same}(G_r) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|G|-1} \phi_{match\_element\_sizes}(g_i, g_{i+1}) \tag{A.44}$$

The constraint $\phi_{match\_element\_sizes}$ (Constraint A.45) encodes a conjunction of equality constraints for each corresponding pair of elements e_{1_i}, e_{2_i} in each group of elements E_1, E_2 requiring the *size_factor* variable to be equal. The *size_factor* variable determines the relative amount of size increase or decrease an element will have from its original size. Each element in a *corresponding pair* of elements has the same position in a subgroup within the repeat group.

$$\phi_{match\_element\_sizes}(E_1, E_2) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|E_1|} (e_{1_i}.\text{size\_factor} = e_{2_i}.\text{size\_factor}) \tag{A.45}$$

Finally, Scout encodes a ϕ_{order_same} constraint (Constraint A.46) that requires the order of elements within subgroups of repeat groups to be the same. The constraint ϕ_{order_same} encodes for each repeat group G a constraint, $\phi_{match\_element\_order}$, for each consecutive pair of subgroups g_i, g_{i+1}.

$$\phi_{order\_same}(G) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|G|-1} \phi_{match\_element\_order}(g_i, g_{i+1}) \tag{A.46}$$

The constraint $\phi_{match\_element\_order}$ (Constraint A.47) encodes for each corresponding pair of elements e_{1_i}, e_{2_i} within two groups of elements E_1, E_2 that if e_{1_i} is left or above the next element in the order $e_{1_{i+1}}$ then e_{2_i} should also be left or above the next element in the order $e_{2_{i+1}}$. Conversely, if e_{1_i} is right or below $e_{1_{i+1}}$ then e_{2_i} should be right or below $e_{2_{i+1}}$.

$$\phi_{match\_element\_order}(E_1, E_2) \stackrel{\text{def}}{=} \bigwedge_{i=1}^{|E_1|-1} \begin{cases} \phi_{left\_or\_above}(e_{2_i}, e_{2_{i+1}}) & \text{if } \phi_{left\_or\_above}(e_{1_i}, e_{1_{i+1}}) \\ \phi_{right\_or\_below}(e_{2_i}, e_{2_{i+1}}) & \text{otherwise} \end{cases} \tag{A.47}$$

The constraint $\phi_{left\_or\_above}$ (Constraint A.48) encodes a disjunction stating for a pair of elements e_1, e_2 that e_1 should fall to the left of or below e_2.

$$\phi_{left\_or\_above}(e_1, e_2) \stackrel{\text{def}}{=} ((e_1.\text{x} + e_1.\text{width}) < e_2.\text{x}) \vee ((e_1.\text{y} + e_1.\text{height}) < e_2.\text{y}) \tag{A.48}$$

The constraint $\phi_{right\_or\_below}$ (Constraint A.49) similarly encodes a disjunction that states that for a pair of elements e_1, e_2 that e_1 should fall to the right of or below e_2.

$$\phi_{right\_or\_below}(e_1, e_2) \stackrel{\text{def}}{=} (e_1.\text{x} > (e_2.\text{x} + e_2.\text{width})) \vee (e_1.\text{y} > (e_2.\text{y} + e_2.\text{height})) \tag{A.49}$$

A.2 Sample Task Instructions

For the Scout user study, presented in Chapter 3, I conducted a within-subjects evaluation where I had designers complete two alternative exploration & creation tasks. Each designer completed a *Baseline* and *Scout* task, across two *Scenarios* including a `Social Media` app screen and a `Weather` app screen. I gave the designers the following instructions for the tasks, along with a set of pre-designed components and a Adobe XD design document. I include the instructions for the *Baseline* - `Social Media` task and the *Scout* - `Weather` task on the following page. The instructions for the *Baseline* - `Weather` task and the *Scout* - `Social Media` task were identical other than using an alternate set of components, thus I did not include them here.

A.2.1 Baseline Task Instructions

Redesigning a Social Media App screen (30 minutes)

You are working for a design agency and you are in charge of the redesign of a social media app, shown below. The UX research team has just conducted a *desireability* study. In this kind of study, users assign emotional adjectives to a design to measure its desireability like fun, fresh, and dull. The top two keywords found for this wireframe were:

dull - plain, boring.
familiar - looks like other profile screens.

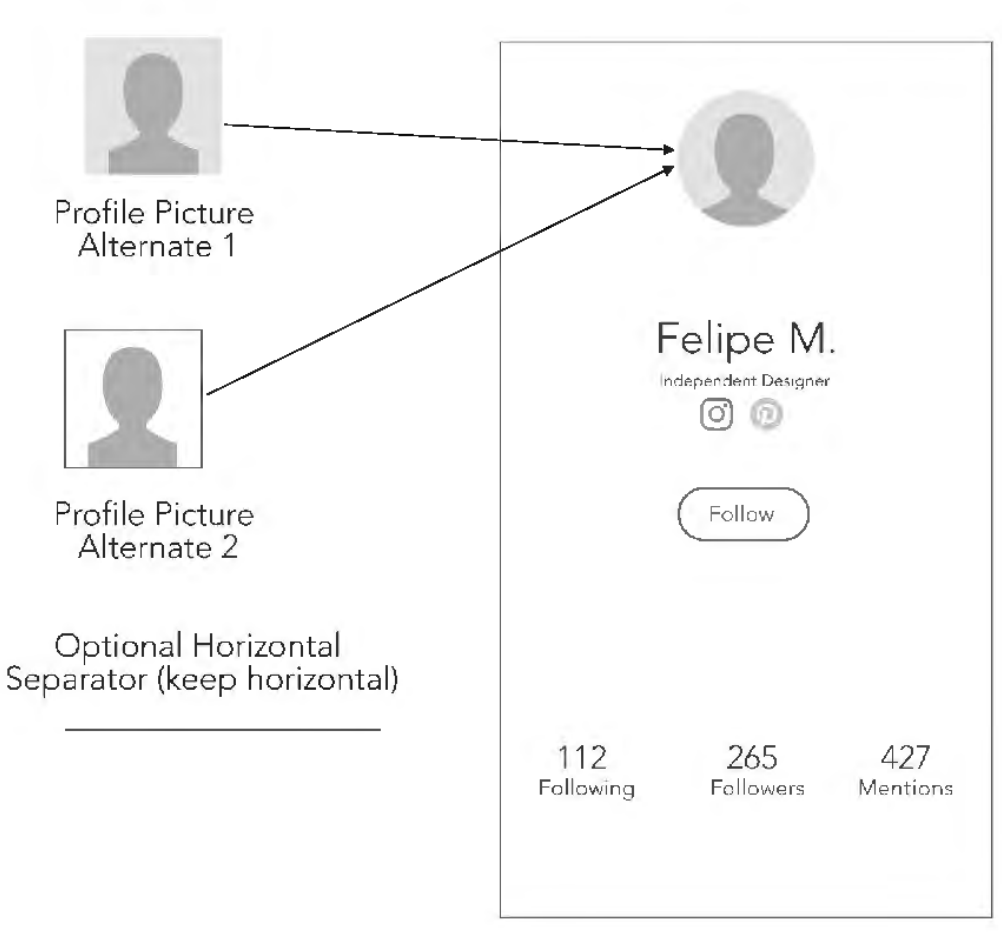
Create 3 alternative wireframe ideas for this design to change these reactions to the original design. You would like the users reactions to be these keywords, which we have defined as:

clean - well aligned, and good use of white space to clearly separate unrelated elements.
compelling - has a clear point of entry (visual salient feature, or an emphasized component.)

For this task, you will create 3 clean and compelling alternative prototypes for the social media profile screen. You should make them distinct and diverse.

You will complete this task with a sheet of paper, and use Adobe XD to make prototypes, with the original components for the social media profile. You will also have a few alternate options to use for the profile picture icon, where the arrows are pointing below.

There are also a few modification rules to follow for this task, so please read the modification rules carefully. Open the Adobe XD document from the dock to begin.



A.2.2 Scout Task Instructions

Redesigning a Weather App screen (30 minutes)

You are working for a design agency and you are in charge of the redesign of a weather app screen, shown below. The UX research team has just conducted a *desireability* study. In this kind of study, users assign emotional adjectives to a design to measure its desireability like fun, fresh, and dull. The top two keywords found for this wireframe were:

dull - plain, boring.
familiar - looks like other weather app screens.

Create 3 alternative wireframe ideas for this design to change these reactions to the original design. You would like the users reactions to be these keywords, which we have defined as:

clean - well aligned, and good use of white space to clearly separate unrelated elements.
compelling - has a clear point of entry (visual salient feature, or an emphasized component).

For this task, you will use Scout to find 3 clean and compelling alternatives to the weather app screen. You should make them distinct and diverse.

This task has two parts: *idea exploration*, and *refinements*.

Part 1: Idea Exploration (20-25 minutes)
Use Scout to find and save 3 distinct alternative layouts that you like. We have imported the components into Scout, including two different sunny icons to use to use shown below (Hint: you can use an alternate group to try out layouts using a single alternate option at a time).

Part 2: Refinements (5-10 minutes)
Export your saved ideas by clicking "Export Saved Ideas" in the Scout toolbar. Then, open XD from the dock and drag the SVGs into Adobe XD. Make any refinements youâĂŹd like to the Scout ideas. We will remind you when there is 5-10 minutes left to begin your refinements.

Goal	Subgoal	Great (2)	Good (1)	Needs Imp. (0)
Compelling	Clear point of emphasis	The wireframe has a clear point of entry or a single visually salient feature, that does not overwhelm the design.	The wireframe has a clear point of entry or single visually salient feature, but it could be more salient without overwhelming the design.	Unable to tell if there is a clear point of entry or visually prominent feature, or there are multiple things that are competing for saliency.
	Visual Balance	Layout is easy to scan, symmetrical and all elements are aligned with respect to axes of symmetry.	Layout is not easy to scan, 1-2 elements are not distributed with respect to axes of symmetry.	Layout is not easy to scan, 3 or more elements are not distributed with respect to axes of symmetry.
	Typographical Hierarchy	All elements follow a typographical hierarchy and are easily readable and proportionally sized with respect to each other.	1-2 elements do not follow a typographical hierarchy and are not easily readable and proportionally sized with respect to each other.	3 or more elements do not follow a typographical hierarchy and are not easily readable and proportionally sized with respect to each other.
Clean	Alignment	All elements in the wireframe are aligned with one or more other elements.	1-2 elements are misaligned to other elements.	1-2 elements are misaligned to other elements.
	Whitespace	Whitespace effectively used to separate unrelated components.	Whitespace sometimes effectively used to separate unrelated components.	Whitespace not effectively used to separate unrelated components.

Table A.1: Quality evaluation rubric that the independent designer panel used to assess the quality of *Scout* and *Baseline* designs created by the designers in the Scout user study.

A.3 Quality Evaluation Rubric

During the Scout user study, each designer created a total of 6 designs. I had a panel of 2 independent interface designers assess each design with a rubric, shown in Table A.1. The designers used the rubric to evaluate designs for 3 **compelling** metrics: *clear point of emphasis*, *visual balance*, and *typographical hierarchy* as detailed in Table A.1. The designers also used the rubric to evaluate each design for two **clean** metrics: *alignment* and *whitespace*. Each designer assigned scores to each design of `0`, `1`, or `2` for each rubric item based on the criteria detailed in the corresponding columns. Overall, the layout quality score, detailed in Chapter 3 (Section 3.3), for each design was a weighted sum of the scores for each rubric item, summed across the two designers.

A.4 Qualitative Interview Questions

As part of the user study for Scout, which I present in Chapter 3, I conducted qualitative interviews to understand more about the impact of Scout on the process of designers in exploring alternatives to their designs early in the design process, and on the diversity and quality of the designers' final design alternatives. I include the full set of interview questions below. Chapter 3 presents the full results of the qualitative analysis. For the questions below, the section of *Scout Post-Task Questions* and the section of *Baseline Post-Task Questions* were not necessarily presented in this order. I asked half of the designers the *Baseline* questions first, and half of the designers the *Scout questions* first.

A.4.1 Scout Post-Task Questions

The following are the set of interview questions that I asked designers immediately after completing the Scout task with Scout (20 minutes) and Adobe XD for refinements (10 minutes). With these questions, I wanted to understand the strategy designers followed in coming up with the alternatives, and the role of Scout in that. Additionally, I wanted to understand the aspects of Scout that were useful and not useful to their process of exploring alternatives.

1. You've just used Scout to explore ideas, and Adobe XD to refine alternatives for a `Social Media|Weather App` interface.
   (a) What strategy did you use to come up with the alternative designs?
   (b) How did Scout play a role in that strategy?
   (c) What aspects of Scout were particularly useful in your process, if any?
   (d) What aspects of Scout were particularly not useful in your process, if any?
2. We asked you to make three diverse design alternatives that were both clean and compelling.
   (a) How successful do you feel you were at creating diverse designs with Scout+Adobe XD together?
   (b) How successful do you feel you were at creating clean and compelling designs with Scout+Adobe XD together?
3. Is there anything else you want to share with me about Scout or this task before we move on?

A.4.2 Baseline Post-Task Questions

The following are the set of interview questions that I asked designers immediately after completing the Baseline task with Adobe XD only (30 minutes). With these questions, I wanted to understand more about the strategy designers followed in coming up with the alternatives. Additionally, I wanted to understand if there were any aspects of XD that were useful or not useful to them in exploring alternatives.

1. You've just used Adobe XD to create alternatives for a `<Social Media/Weather App>` interface.
   (a) What strategy did you use to come up with the alternative designs?
   (b) How did Adobe XD play a role in that strategy?
   (c) What aspects of Adobe XD were particularly useful in your process, if any?
   (d) What aspects of Adobe XD were particularly not useful in your process, if any?

2. We asked you to make three diverse design alternatives that were both clean and compelling.
   (a) How successful do you feel you were at creating diverse designs with Adobe XD?
   (b) How successful do you feel you were at creating clean and compelling designs with Adobe XD?
3. Is there anything else you want to share with me about Adobe XD or this task before we move on?

A.4.3 Post Study Interview Questions

I asked the designers to answer the following questions after completing both the Scout and Baseline tasks. This enabled them to reflect on their past design processes for coming up with and creating alternatives, and how they could imagine using a tool like Scout in that process. I also had them compare and contrast their approach in creating alternatives in both tasks. Finally, I had them reflect on the impact of the tools in both conditions on the quality (i.e., clean and compelling) and diversity of their alternative designs.

1. Think about the last interface you created (before this study). Did you try to come up with alternative designs at any point in that process?
   (a) If so, what did you do? And how well did it work?
   (b) If not, why not?
2. How could you see yourself using a tool like Scout in that process?
3. Today, you used two different workflows to make alternative designs: Adobe XD alone, and then Scout for idea exploration and Adobe XD for refining the Scout ideas.
   (a) How did your approach to creating alternatives using Scout and Adobe XD together differ from your approach using Adobe XD alone?

I asked the designers the following questions to have them reflect on which tool workflow resulted in more diversity in their final designs.

1. Which tool workflow do you feel helped you to come up with more diverse design alternatives?
2. How did **<workflow>** help you to do that better than **<other workflow>**?

I asked the designers the following questions to have them reflect on which tool workflow resulted in more *clean* and *compelling* designs.

1. Which tool workflow you feel helped you to come up with cleaner and more compelling designs?
2. How did <workflow> help you to do that better than <other workflow>?

Finally, I concluded the interview by having the designers give any last thoughts on Scout, and on the differences between Scout and Adobe XD. Typically, this is where they provided any open-ended feedback.

1. Are there any last thoughts on Scout, or the differences between Scout and Adobe XD, that you'd like to share before we wrap up?