Runtime Repair and Enhancement of Mobile App Accessibility

Xiaoyi Zhang

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington

2018

Reading Committee:

James Fogarty, Chair

Jacob O. Wobbrock

Jennifer Mankoff

Program Authorized to Offer Degree:

Computer Science & Engineering

© Copyright 2018

Xiaoyi Zhang

University of Washington

Abstract

Runtime Repair and Enhancement of Mobile App Accessibility

Xiaoyi Zhang

Chair of the Supervisory Committee:

Professor James Fogarty

Computer Science & Engineering

Mobile devices and applications (apps) have become ubiquitous in daily life. Ensuring full access to the wealth of information and services they provide is a matter of social justice. Unfortunately, many capabilities and services offered by apps today remain inaccessible for people with disabilities. Built-in accessibility tools rely on correct app implementation, but app developers often fail to implement accessibility guidelines. In addition, the absence of tactile cues makes mobile touchscreens difficult to navigate for people with visual impairments.

This dissertation describes the research I have done in runtime repair and enhancement of mobile app accessibility. First, I explored a design space of interaction re-mapping, which provided examples of re-mapping existing inaccessible interactions into new accessible interactions. I also implemented interaction proxies, a strategy to modify an interaction at runtime without rooting the phone or accessing app source code. This strategy enables third-party developers and

researchers to repair and enhance mobile app accessibility. Second, I developed a system for robust annotations on mobile app interfaces to make the accessibility repairs reliable and scalable. Third, I built Interactiles, a low-cost, portable, and unpowered system to enhance tactile interaction on touchscreen phones for people with visual impairments.

The thesis of this dissertation is: ***An interaction remapping strategy can enable third-party developers and researchers to robustly repair and enhance the accessibility of mobile applications at runtime, while preserving the platform's security model and accessibility infrastructure***.

Table of Contents

LIST OF FIGURES iv

LIST OF TABLES viii

ACKNOWLEDGEMENTS ix

Chapter 1 INTRODUCTION 1

1.1 Thesis Statement 2

1.2 Thesis Overview 2

Chapter 2 RELATED WORK 4

2.1 Ability-Based Design 4

2.2 Mobile Accessibility Implementation 5

2.3 Runtime Interface Modification 6

2.4 Accessibility Repair and Enhancement 7

2.5 Social Annotation 8

2.6 Screen and UI Element Identifiers 9

2.7 Challenges in Graphical Interface Accessibility 11

2.8 Software-Based Touchscreen Assistive Technologies 12

2.9 Tangible GUI Accessibility 14

2.10 Security of Mobile Runtime Repairs and Enhancements 15

Chapter 3 DESIGN SPACE OF INTERACTION RE-MAPPING 18

3.1 From Zero to One Remapping 19

3.2 From One to Zero Remapping 20

3.3 From One to One Remapping 21

3.4 From One to Many Remapping 22

3.5 From Many to One Remapping 22

3.6 Discussion 23

Chapter 4 INTERACTION PROXIES FOR RUNTIME REPAIR AND ENHANCEMENT OF MOBILE APPLICATION ACCESSIBILITY 25

4.1 Interaction Proxies 26

4.2 Implementation in Android 28

4.2.1 Abstractions for Android Interaction Proxies 28

4.2.2 Coordinating Perception and Manipulation 31

4.3 Demonstration Implementations 33

4.3.1 Adding or Correcting Accessibility Metadata 33

4.3.2 Restoring Missing Interactions 34
4.3.3 Modifying Navigation Order 35
4.3.4 Fully-Proxied Interfaces 35

4.4 Interviews Regarding Interaction Proxies 37
4.4.1 Method 38
4.4.2 Results 39

4.5 Discussion and Conclusion 41

Chapter 5 ROBUST ANNOTATION ON MOBILE APP INTERFACES 43

5.1 Android Background 45

5.2 Approach and System Overview 47
5.2.1 Supporting a Range of Accessibility Repair Scenarios 48

5.3 Implementing Robust Annotation 49
5.3.1 Screen Identifier 49
5.3.2 Element Identifier 50
5.3.3 Screen Equivalence Heuristics 51
5.3.4 Annotation Storage 54

5.4 Data Collection and Annotation Tools 54
5.4.1 Capture Tool 55
5.4.2 Template Screen Tool 55
5.4.3 Annotation Tool 55
5.4.4 Runtime Library 56
5.4.5 Alternative Collection and Annotation Tools 56

5.5 Evaluation of Screen Equivalence Heuristics 56

5.6 Case Studies of Runtime Accessibility Repair 59
5.6.1 Missing or Misleading Labels 60
5.6.2 Navigation Order Issues 61
5.6.3 Inaccessible Customized Widgets 61

5.7 Evaluation of Runtime Repair 62
5.7.1 Participant Feedback on Accessibility Repairs 62
5.7.2 Repairs in Additional Mobile Apps 64

5.8 Current Limitations 66

5.9 Conclusion 67

Chapter 6 INTERACTILES: A SET OF 3D-PRINTED TACTILE INTERFACES TO ENHANCE MOBILE TOUCHSCREEN ACCESSIBILITY 68

6.1 Use Case Scenario 70

6.2 Design and Implementation of Interactiles 71
6.2.1 Design Goals 72
6.2.2 Hardware Components 73
6.2.2 Software Proxies 74
6.2.3 System Improvements Based on Pilot Study Input 78
6.2.3 System Validation in Android Apps 78

6.3 Usability Study 80
6.3.1 Participants 80
6.3.2 Comparative Study Method 80

6.3.3 Tasks....81
6.3.4 Data Collection and Analysis....82
6.3.5 Results....83
6.3.6 Customizability....85

6.4 Discussion 86

6.5 Conclusion....87

Chapter 7 ADDITIONAL RESEARCH THEMES....89

7.1 Mobile Interactions....89

7.2 Large-Scale Mobile App Accessibility Analysis....90

7.3 Mobile Eye Tracking for People with Motor Impairments 92

7.4 Conclusion....93

Chapter 8 CONCLUSION 94

8.1 Contributions 94
8.1.1 Concepts 95
8.1.2 Artifacts....95
8.1.3 Experimental Results 96

8.2 Future Directions 97
8.2.1 Deploying New Accessibility Repairs and Enhancements 97
8.2.2 Scaling Mobile App Interface Annotation by Crowdsourcing 97
8.2.3 Applying Robust Annotation Approaches Outside Accessibility 97
8.2.4 Making More Tasks Tangible....98

8.3 Limitations 99

8.4 Reflections 99

8.5 Conclusion....100

8.6 Final Remarks 102

REFERENCES 103

LIST OF FIGURES

Figure 2.1. (a) A user whose abilities match those presumed by the system. (b) A user whose abilities do not match those presumed by the system. Because the system is inflexible, the user must be adapted to it. (c) An ability-based system is designed to accommodate the user's abilities. It may adapt or be adapted to them. This figure with caption is taken from [105]. 4

Figure 2.2. User interfaces for the synthetic application. The baseline interface is shown in comparison to interfaces generated automatically by SUPPLE based on two participants' preferences. Able-bodied participants like AB03, preferred lists to combo boxes but preferred them to be short; all able-bodied participants also preferred default target sizes to larger ones. As was typical for many participants with motor-impairments, MI09 preferred lists to combo boxes and frequently preferred the lists to reveal a large number of items; MI09 also preferred buttons to either checkboxes or radio buttons and liked larger target sizes. This figure with caption is taken from [30]. 7

Figure 2.3. The Rico dataset contains a set of user interaction traces and the unique UIs discovered during crawling. This figure with caption is taken from [19]. 9

Figure 2.4. Slide Rule uses multi-touch gestures to interact with apps. (1) A one-finger scan is used to browse lists. (2) A second-finger tap is used to select items. (3) A flick gesture is used to flip between pages of items. (4) An L-select gesture is used to browse the hierarchy of items. This figure with caption is taken from [56]. 13

Figure 2.5. A QWERTY keyboard touchplate, cut from acrylic plastic, provides tactile feedback for a large touchscreen user interface. The touchscreen recognizes the guide and moves the virtual keyboard beneath it. This figure with caption is taken from [59]. 14

Figure 2.6. An example of "clickjacking". When user clicks the "next" button on the top-level overlay, the click event passes to the "install" button in the system dialog covered by the overlay. 16

Figure 3.1. This interaction re-mapping adds third-party accessibility ratings directly within the app store. (a) An "Accessibility Ratings" button is added in previously empty space. (b) Selecting the button shows third-party accessibility information for that app. 19

Figure 3.2. iOS Guided Access can disable touch in multiple screen areas. (a) The guidance to circle areas on the screen to disable is shown. (b) The bottom center area is circled and disabled. (c) Users cannot activate any UI element inside the circled area. 20

Figure 3.3. This interaction re-mapping repairs accessibility metadata that an app provides to a platform screen reader. (a) Toggl is a popular time-tracking app. (b) Its interface includes elements with missing or inappropriate metadata, which a screen reader manifests as "Navigate Up" and "One, Unlabeled". (c) Our third-party interaction proxy repairs the interaction using appropriate metadata for each element, so a screen reader correctly manifests these elements as "Menu" and "Start Timer". 21

Figure 3.4. This interaction re-mapping replaces one interaction with a sequence of two interactions. (a) An interface contains small adjacent targets. (b) The targets are replaced with a single larger "Tools" button. (c) Selection displays a menu of the original targets. (d) A selected item is activated. 22

Figure 3.5. This interaction re-mapping replaces many interactions with one interaction. (a) An interface contains scrollable content. (b) The "Scroll to Top" macro button is shown after the content scrolls. (c) Once the macro button is activated, the content scrolls back to the top. . 23

Figure 4.1. Interaction proxies are inserted between an app's original interface and the manifest interface to allow third-party modification of interactions.. 27

Figure 4.2. Direct interaction is generally straightforward to coordinate, with interface layers behaving as expected in their occlusion and in mapping input to the appropriate element. .. 31

Figure 4.3. Indirect interaction can require more coordination. Here a gesture-based navigation requires an event listener to watch for "One, Unlabeled" to get focus and then immediately gives focus to "Start Timer". Perception in the manifest interface is seamless because the screen reader truncates the reading of "One, Unlabeled", but this example illustrates the type of coordination an interaction proxy may need to implement to remain seamless................ 32

Figure 4.4. This interaction proxy corrects a "Menu" label and repairs interaction with the app's dropdown menu, which is otherwise completely inaccessible with a screen reader. 33

Figure 4.5. The Yelp app manifests its five-star rating system as a single element that cannot meaningfully be manipulated using a screen reader, and its navigation order using a screen reader makes it needlessly difficult to access "Search". 34

Figure 4.6. This interaction proxy implements a stencils-based tutorial. At each tutorial step, only the appropriate element is available. All other elements are obscured and disabled. 36

Figure 4.7. Motivated by research in personalized interface layout, this interaction proxy creates a completely new personalized layout for interacting with the underlying app. 37

Figure 5.1. Missing and misleading labels are a common and important accessibility issue that can be addressed by new approaches to robust annotation for accessibility repair. 46

Figure 5.2. We develop and evaluate new methods for robust annotation of mobile app interface elements appropriate for runtime accessibility repair, together with end-to-end tool support for developers implementing accessibility repairs. 47

Figure 5.3. Illustration of annotation interface to correct navigation order. 7 elements are listed in the navigation order determined by Talkback. 60

Figure 6.1. Interactiles allows people with visual impairments to interact with mobile touchscreen phones using physical attachments, including a number pad (left) and a multi-purpose physical scrollbar (right). 69

Figure 6.2. The Interactiles hardware base is a 3D-printed plastic shell that snaps on a phone, and three hardware components (Scrollbar, Control Button, and Number Pad) are attached to the shell. 73

Figure 6.3. Floating windows created for number pad (left), scrollbar (right) and control button (right bottom). The windows can be transparent; we use colors for demonstration.............. 75

Figure 6.4. Average task completion time for each task in the study. P4 did not complete app switching on the control condition, and P5 did not complete the holistic task. 82

Figure 6.5. The average Likert scale rating (strongly disagree = -2, strongly agree = 2) given by participants for the study tasks. Participants were asked how easy, quick, intuitive, and how confident they felt completing each task with the control condition (only TalkBack) and Interactiles. 83

Figure 6.6. Individual task completion time of locate and relocate tasks. 84

Figure 7.1. We examined unlock journaling, in which unlocking a phone also journals an in situ self-report. We presented the first study comparing it to diaries and notification-based reminders. 89

Figure 7.2. The distribution of the proportion of image-based buttons within an app with a missing label. A total of 5,753 apps were tested. A higher proportion indicates an app with more errors. The high number of apps at the extremes along with the uniform, non-zero distribution between the extremes hints at a rich ecosystem of factors influencing whether an app's image-based buttons are labeled. 91

Figure 7.3. The communication partner of a person with motor disabilities can use the GazeSpeak smartphone app to translate eye gestures into words. (a) The interpreter interface is displayed on the smartphone's screen. (b) The speaker saw a sticker depicting 4 groupings of letters affixed to the phone's case. 92

LIST OF TABLES

Table 3.1. Modify perception, manipulation, or both to re-map existing interaction into new interaction. The number of interactions may change before and after a re-mapping. 18

Table 5.1. Improvements in error rates resulting from adding each screen equivalence heuristic.57

Table 5.2. The number of accessibility issues repaired in each of the 26 apps used in our evaluation of runtime repair. 65

Table 6.1. The number of apps in which Interactiles features worked as expected, out of 50 sample apps. One app did not include any text field to test number entry. 78

Table 6.2. Information on study participants, all of whom were VoiceOver users. Proficiency was self-rated as basic, intermediate, or advanced. 80

ACKNOWLEDGEMENTS

My advisor, James Fogarty, deserves special thanks for everything he has done for me. James really understood and supported my priorities and career decisions. He gave very detailed suggestions to improve my scientific writing. When we disagreed, he encouraged me to share my thoughts and challenge him. I also see him as a great example of keeping work-life balance. I cannot imagine what my Ph.D. life would be without him.

I would also like to thank other members of my all-star committee for their guidance, support, and insight. Leah Findlater encouraged me to share ideas in accessibility seminars. Jennifer Mankoff and Jacob O. Wobbrock were both deeply involved in my thesis research and served on my reading committee. Meredith Ringel Morris opened a door to industrial accessibility research for me and mentored me through two internships.

I am thankful for the help from faculty outside my committee. Anat Caspi provided valuable insight into my first thesis work. Richard Ladner gave suggestions at different stages of my research. Jon Froehlich invited me to give talks about my work. Jeffery Bigham discussed future research directions with me; together, we look forward to making them happen.

I enjoyed working with all my internship mentors. Kayur Patel let me explore the intersection of machine learning and human-computer interaction. After the internship, he continued to guide me along my career path. Harish S. Kulkarni trusted me and gave me the freedom to explore eye tracking on mobile devices. I would like to thank all teams where I interned: Ability and Enable at Microsoft Research, Katamari and Colaboratory at Google Research, and iCloud at Apple.

My undergraduate research experience at UCLA was invaluable to me. Ming-Chun Huang, Majid Sarrafzadeh, and Wenyao Xu first guided me through the research process and then encouraged me to lead research projects. I also thank Yutao He for his mentorship in both coursework and life.

Sincere thanks to my peer fogies (labmates) who are always willing to help: Daniel Epstein, Ravi Karkar, Laura R. Pina, Anne Spencer Ross, Jessica Schroeder, Amanda Swearngin, and Jin A Suh. I am also glad that we keep the tradition of Fogies Fun Fun Day — the fun never ends!

Many talented and kind researchers and students have positively influenced my Ph.D. journey. I would like to thank Alex Fiannaca, Anhong Guo, Donny Huang, Hanchuan Li, Jiajun Li, Xi Lin, Tracy Tran, Aditya Vashistha, Kanit Ham Wongsuphasawat, Shaomei Wu, Yang Zhang, and Yu Zhong.

My gratitude goes to friends outside academia: Mingru Bai, Cunpu Bo, Yufei Chen, Yucheng Wang, Zhiyang Wang, Yuxiang Xie, Daheng Yang, and Tianfu Zhang. Thank you for listening to my secrets and keeping them safe.

This work received generous support from government and industry sponsors: The Agency for Healthcare Research and Quality, the National Institute on Disability, Independent Living and Rehabilitation Research, the National Science Foundation, Google Faculty Award, Intel Science and Technology Center for Pervasive Computing, and Nokia Research.

It was a pleasant journey to pursue my Ph.D. at the Paul G. Allen School of Computer Science & Engineering at the University of Washington. In addition to research, I got a chance to practice the art of cooking and plating. I want to thank everyone who saw or will see my posts on WeChat, FB, Instagram, and other social media platforms because I really enjoy sharing my life moments with you. Thank you for coming into my life. I will always cherish the memories.

I love my whole family. My parents, Min Qu and Gaofeng Zhang, deserve my deep thanks for their lifelong encouragement and support. Without their education with love, I could never have become who I am now.

A more visually-appealing version is at: www.XiaoyiZhang.me/dissertation.

To my loved ones.

Chapter 1 INTRODUCTION

Mobile touchscreen devices and their applications (apps) play increasingly important roles in daily life. They are used to access a wide variety of services online and offline (e.g., transit schedules, medical services). Ensuring full access to the wealth of information and services provided by such apps is a matter of social justice [63]. Unfortunately, many apps remain difficult or impossible to access for people with disabilities, an estimated 15% of the world population [77]. Recent research examined the prevalence of accessibility issues in a sample of 100 Android apps, finding that every app included at least one accessibility issue [85]. For example, 95% of the examined apps included touchable elements that were smaller than recommended by Android's accessibility guidelines [33], making them difficult to access for many people (e.g., people with motor impairments). 94% included elements that lacked alternative text, making them difficult to access using a screen reader (e.g., for people with visual impairments). 85% included elements with low text contrast, another barrier for people with vision impairments. Pursuing more complete access requires contributions from the developers of thousands of individual apps, from developers of mobile platforms (e.g., Apple, Google), and from third-party accessibility developers and researchers.

There are many reasons that an app is inaccessible. App developers may fail to implement fundamental accessibility support suggested by platform accessibility guidelines. Developers may lack awareness or knowledge needed to repair accessibility issues. They may delay a fix due to competing development priorities, and other app updates may even introduce new accessibility failures. Alternatively, it might be difficult to obtain updates if the original development of an app was contracted out or if the app was abandoned by its developer.

Mobile platforms also face significant challenges and tradeoffs in prioritizing, designing, and implementing platform-level accessibility improvements. Although the impact of platform support can be large, many promising techniques never achieve platform-level adoption. Platforms also have begun to support correction on some accessibility issues. For example, the Talkback [34] screen reader allows end-users to add custom labels to unlabeled elements. However, in our evaluation of 50 apps, Talkback can apply custom labels to only 13.6% of the elements it visits.

Third-party developers and researchers often examine new accessibility improvements in prototype apps rather than in existing apps or across the entire mobile platform. In most of these cases, they cannot reasonably rebuild a platform's entire accessibility infrastructure to evaluate a new approach. Therefore, the exploration of accessibility techniques is often limited to prototype apps with a small number of user study participants.

1.1 Thesis Statement

In my thesis, I explored a new approach to deploying third-party accessibility repairs and enhancements at runtime that could both: (1) allow promising accessibility repairs and enhancements to reach more people, and (2) improve accessibility research by enabling broader deployment and evaluation of potential techniques.

My thesis statement is: ***An interaction remapping strategy can enable third-party developers and researchers to robustly repair and enhance the accessibility of mobile applications at runtime, while preserving the platform's security model and accessibility infrastructure.***

1.2 Thesis Overview

The following chapters are organized as follows:

Chapter 2 reviews an overview of related work, including the ability-based design approach, prior research in mobile app accessibility, runtime interface modification, annotation methods, tangible GUI accessibility, and security in runtime repair.

Chapter 3 describes the **design space** of interaction re-mapping. It provides examples of re-mapping existing inaccessible interactions into new accessible interactions. The design space supports researchers and developers who design accessibility repairs and enhancements for mobile apps.

Chapter 4 introduces the design and implementation of **interaction proxies**, a strategy that enables the runtime repair and enhancement of mobile app accessibility. This strategy (1) allows individuals or communities to quickly address accessibility failures that an app's original developer is unable or

unwilling to address, and (2) accelerates research by enabling deployment and evaluation of potential techniques and by allowing promising accessibility enhancements to reach more people.

Chapter 5 presents **a mobile app interface annotation system** that makes runtime accessibility repair reliable and scalable. It allows developers to robustly annotate accessibility metadata and develop either (1) targeted repair in one or two screens of an app, or (2) more general-purpose repair of a class of errors across many different apps. This chapter also describes the implementation of repairs to 3 common accessibility issues using interaction proxies and examines the repairs of real-world accessibility issues in 26 apps.

Chapter 6 shows **Interactiles**, an inexpensive, unpowered, general-purpose system that provides tactile feedback on touchscreen smartphones using 3D-printed interfaces. Mobile screen readers transform a 2D spatial interface into a linear audio stream, but information is inevitably lost during the transformation. Tactile feedback has been shown to improve accessibility, but previous approaches have an emphasis on larger touchscreens and have cost and compatibility limitations. Therefore, Interactiles (1) focuses on mobile systems, (2) does not require expensive customization, and (3) maximizes the compatibility across mobile apps by implementing abstractions of interaction proxies.

Chapter 7 highlights additional research themes in my doctoral research. The first theme is mobile interactions. The second theme is mobile eye tracking for people with motor impairments. The third theme is large-scale mobile app accessibility analysis.

Chapter 8 discusses future research opportunities suggested by this dissertation and summarizes thesis contributions.

Chapter 2 RELATED WORK

This chapter covers related work in ten areas. Section 2.1 introduces the ability-based design approach that my dissertation research may benefit. Sections 2.2, 2.3, and 2.4 present the implementation of mobile app accessibility, prior research in runtime app interface modification, and practices in accessibility repair and enhancement. These three research areas inspired my work of interaction proxies (Chapter 4). Sections 2.5 and 2.6 review existing approaches in social annotation and methods in screen and UI (User Interface) element identification. These approaches informed the robust mobile app interface annotation system (Chapter 5) for accessibility repair. Sections 2.7, 2.8, and 2.9 describe challenges in graphical user interface (GUI) accessibility, software assistive technologies, and tangible solutions that enhance GUI accessibility. The limitations of these existing approaches reveal an untapped opportunity for the Interactiles system (Chapter 6) to provide tactile feedback on mobile devices and apps. Section 2.10 discusses the security of runtime repairs and enhancements on the mobile platform**.**

2.1 Ability-Based Design

Accessible technologies aim at making things accessible for people with disabilities. Such framing therefore places more emphasis on dis-ability than ability. Ability-based design aims at shifting the design focus from dis-ability to ability by taking advantage of each user's individual abilities [105]. It also shifts the burden of accommodation from the human to the system. Research guided by ability-based design may personalize interfaces based on environment, devices, tasks, preferences, and abilities [31,32,60].

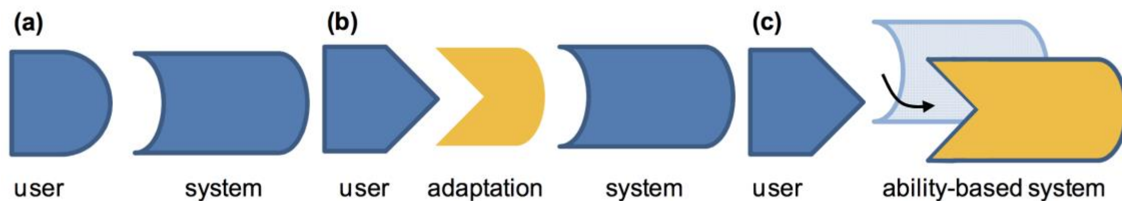


Figure 2.1. (a) A user whose abilities match those presumed by the system. (b) A user whose abilities do not match those presumed by the system. Because the system is inflexible, the user must be adapted to it. (c) An ability-based system is designed to accommodate the user's abilities. It may adapt or be adapted to them. This figure with caption is taken from [105].

However, current mobile apps seldom adapt to user's abilities as shown in Figure 2.1. While my dissertation is not strictly guided by ability-based design, it explores ways to help developers and researchers achieve their goals using ability-based design. For example, Chapter 4 introduces an interaction re-mapping strategy that allows third-party developers and researchers to practice and evaluate ability-based design at the system level. This strategy can potentially bring ability-based design to more apps.

2.2 Mobile Accessibility Implementation

Designing and implementing support for mobile accessibility requires diverse contributions from researchers, mobile platforms, and app developers. One example can be seen in the design, adoption, and implementation of support for screen readers. Screen readers make apps more accessible to blind and low-vision people by transforming graphical interfaces into auditory interfaces [73,74]. Motivated by the research challenge of supporting blind people in using mobile touchscreens, Slide Rule introduced techniques for re-mapping gestures to support navigating and exploring the screen separately from activating targets in an interface [56]. These techniques were adopted by built-in screen readers for mobile platforms (e.g., Android's TalkBack [34] and iOS's VoiceOver [4]), and now improve the accessibility of apps on those platforms [64]. Prior research investigated the architecture for transforming graphical interfaces into other interfaces [26]. Current mobile platforms support such transformations with native accessibility APIs which expose screen content (e.g., view hierarchy) to assistive tools such as screen readers.

However, the accessibility of mobile apps critically depends on the developers of individual apps [33]. One key requirement is that app developers should provide metadata to support platform accessibility enhancements. Developers fail to provide metadata for a variety of reasons (e.g., being unaware they should, lacking knowledge of how to do so, or failing to prioritize accessibility). This problem has been most studied on the web (e.g., with missing image alternative text [11]). Despite education and improved testing [1,15,80], such accessibility failures remain common on the web [44]. The same underlying challenge occurs in mobile platforms when app developers fail to follow platform accessibility guidelines. Recent research examined the prevalence of accessibility issues in

a sample of the top 100 Android free apps, finding that every app included at least one accessibility issue [85]. It is best for apps to be designed and developed to be broadly accessible [105]. However, when apps fall short of this goal, my research on third-party interaction proxies in Chapter 4 provides a complementary strategy to repair accessibility issues.

2.3 Runtime Interface Modification

Techniques for modifying the web benefit from the ability to directly modify a page prior to its rendering by a browser (e.g., modifying the HTML, CSS, or DOM). In contrast, non-web architectures generally lack an ability to access or modify internal representation, requiring different approaches to runtime modification. Early work replaced a toolkit drawing object and intercepted commands [27,76]. More recently, Scotty examined runtime toolkit overloading, developing abstractions for implementing modifications in code injected directly into the interface runtime [25]. Complementary work explored input-output re-direction in the window manager [99], using information from the desktop accessibility API to make changes in an interface façade [93]. To overcome limitations of the desktop accessibility API and incomplete implementations of that API, recent work examined the interpretation of interface pixels [50]. Runtime modification can be implemented using only pixel-level analysis [22–24,109] or in combination with information from an accessibility API [16]. Such approaches can allow authoring of more specialized or accessible manifestations of underlying application functionality [115]. Our work builds on insights from the desktop, but the mobile app context does not allow implementation of prior approaches. We therefore developed a new approach for runtime mobile interface modifications (i.e., using floating windows while coordinating perception and manipulation in the manifest interface) in Chapter 4.

2.4 Accessibility Repair and Enhancement

This dissertation is informed by accessibility repair and enhancement research on different platforms. Prior accessibility repairs often focus on the web, where the representation rendered by a browser is available and can be directly modified. Some accessibility repairs can be automated, such as attempting to automatically identify sources of alternative text for web images [11] or automatically increasing font size to improve readability [10]. Other approaches emphasize social annotation, wherein people contribute accessibility repairs that benefit additional people who encounter the same accessibility failure [61]. The next section provides further discussion about social annotation. In Chapter 4, we use the term interaction "proxies" to be analogous to prior research using web proxies to improve web accessibility.

In addition to work on visual accessibility, many people with motor impairments choose accessibility techniques that modify how they manipulate an interface. As shown in Figure 2.2, SUPPLE modifies interface elements according to personal motor abilities [30,105]. Other systems leave interface layout unchanged but apply techniques to ease pointing. Examples include making targets "sticky" [51,106], dynamically modifying cursor behavior according to a person's movement profile [104], and breaking a pointing interaction into multiple interactions that disambiguate the

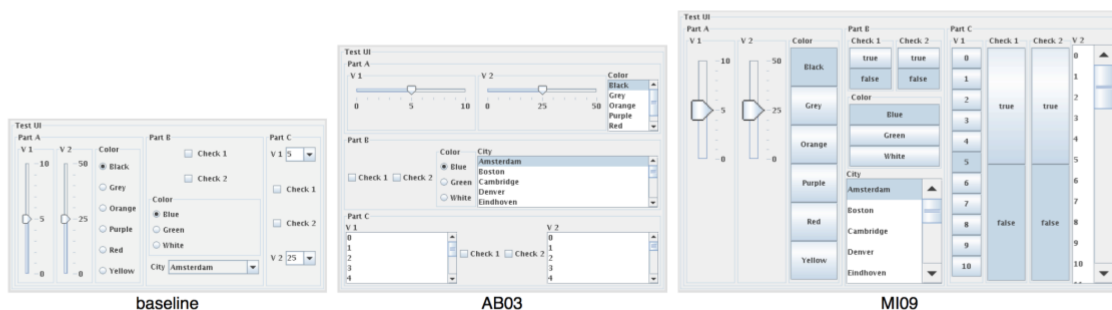


Figure 2.2. User interfaces for the synthetic application. The baseline interface is shown in comparison to interfaces generated automatically by SUPPLE based on two participants' preferences. Able-bodied participants like AB03, preferred lists to combo boxes but preferred them to be short; all able-bodied participants also preferred default target sizes to larger ones. As was typical for many participants with motor-impairments, MI09 preferred lists to combo boxes and frequently preferred the lists to reveal a large number of items; MI09 also preferred buttons to either checkboxes or radio buttons and liked larger target sizes. This figure with caption is taken from [30].

intended target [29,55,116]. Other research has explored alternative pointing for physically large devices [58].

Less research has explored runtime interface modification for mobile platforms, in part due to their security architectures and greater concern for performance (e.g., a challenge for pixel-based techniques developed in the desktop context). Notable examples of runtime accessibility enhancements have included macro support [81] and pointing enhancement [116]. The SWAT framework examines system-level instrumentation of content and events to support developer creation of accessibility services [82]. However, it requires rooting a device, which creates a significant security risk and presents a technical expertise barrier. On the contrary, our interaction re-mapping strategy in Chapter 4 works within the Android accessibility and security model. In addition, this dissertation explores a larger design space of re-mappings for accessibility repairs and enhancements in Chapter 3.

2.5 Social Annotation

The web's representation has long encouraged content enhancement through annotation, from annotation capabilities in Mosaic [101] to W3C efforts in interoperable annotations [103]. Extensive research in social accessibility has applied social annotation to web accessibility (e.g., [12,48,87,88,97]). Systems have explored various techniques, with a core that: (1) a person observes an accessibility failure, (2) that person or another person annotates the interface with metadata used by a tool that can repair the failure, and (3) annotation data is shared so that future users of the interface benefit from the repair. Examples of social annotation research for web accessibility include providing image alternative text and other metadata [88,97,98], repairing navigation order [87], sharing scripts for site-specific repairs [12] and designing infrastructure for crowdsourcing contributions [48].

Social annotation on the web has generally been implemented via the combination of a URL (i.e., indicating the context to apply an annotation) and an XPath (i.e., indicating the element to annotate within the context of a URL). For desktop environments, pixel-based UI interpretation [24]

allows robust annotation on interface elements with metadata needed to enable runtime enhancements. The implementation of annotation on the web and desktop cannot be directly applied in mobile apps because: (1) the various screens of an app lack robust identifiers (i.e., the equivalent of a web URL), and (2) pixel-level interpretation on mobile app is limited by the potential lack of access to screenshots and concerns about performance challenges in runtime analysis. Chapter 5 describes how robust annotation of mobile UI can enhance app accessibility. In the next section of related work, we discuss methods to implement annotation on mobile platform.

2.6 Screen and UI Element Identifiers

For web social annotation, a URL is used to locate a webpage and an XPath is used to locate the element to annotate within that webpage. Similarly, robust annotation of mobile apps requires both: (1) methods for identifying a screen within an app, and (2) methods for identifying specific elements within that screen.

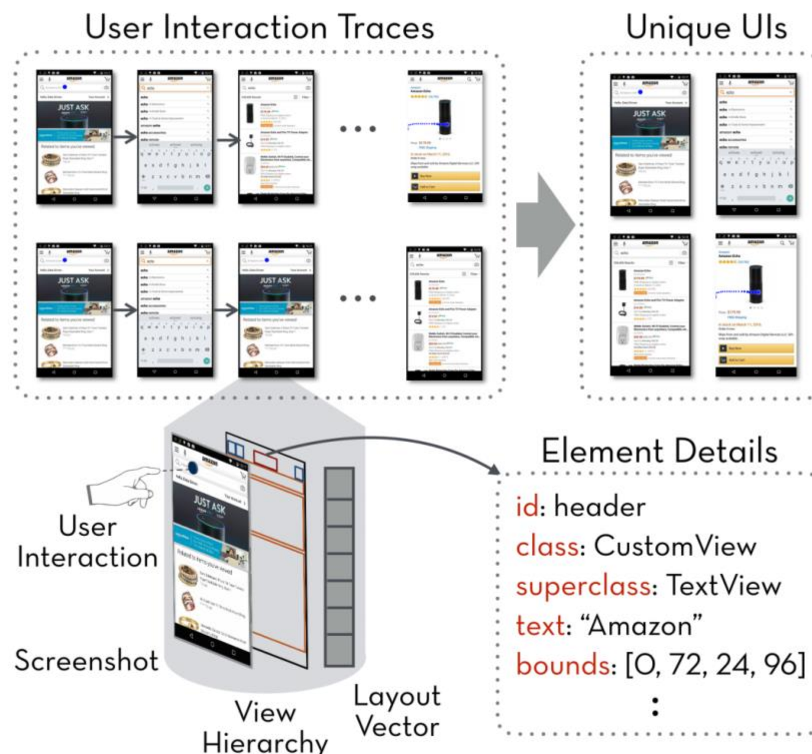


Figure 2.3. The Rico dataset contains a set of user interaction traces and the unique UIs discovered during crawling. This figure with caption is taken from [19].

Prior research exploring screen identifiers has not been motivated by runtime interface modification and is generally inappropriate for that purpose. For example, Flow is an Android developer toolkit that allows naming interface states, navigating between them by name, and remembering the history of states [90]. However, such a toolkit must be integrated by an app's developer and cannot be used to reason about screens as part of an external repair.

The Rico project developed a large dataset of mobile app designs that were captured during crawls of mobile apps, as shown in Figure 2.3 [19]. Rico encounters a screen identification problem in determining whether an interaction during a crawl results in an app entering a new interface state or a state that was previously captured. Rico defines a context-agnostic similarity heuristic that compares two screens based on: (1) the number of pixels that differ in the two screen images, and (2) the number of differences when comparing the values of *ViewIDResourceName* for elements of the two screens. Two screens were considered the same if both were below manually tuned thresholds, requiring the same value for 99.8% of pixels and all but 1 *ViewIDResourceName* value. These thresholds resulted in an estimated 9% error rate (6% error incorrectly determining two screens were the same and 3% error incorrectly determining two screens were different). Rico's use of pixel comparison is appropriate for crawling, but problematic for runtime modification (e.g., it requires additional permission to access screen images, and it can present performance challenges).

Other mobile app crawls similarly attempt to minimize the re-visitation of known screens by testing for similarity. For example, DECAF and PUMA define a generic feature vector that encodes the structure of a screen's interface hierarchy and then use a cosine-similarity metric to determine screen equivalence according to a threshold [45,67]. An evaluation in DECAF with a .92 threshold estimated a 20% error rate (including 8% error incorrectly determining two screens were the same and 12% error incorrectly determining two screens were different). The threshold can be varied to obtain different tradeoffs between thoroughness and speed of a crawl, but a developer cannot otherwise correct either class of error.

In contrast to both methods above, the screen identification methods I develop in Chapter 5: (1) use more information in the structure of the interface hierarchy to reduce overall error, and (2) allow the developer of an accessibility repair to explicitly correct any errors in screen identification to ensure robust annotation for runtime interface modification.

Automated testing tools address a need to be in a known state by executing pre-defined interaction sequences that bring an app to known screens (e.g., [3,35,89]). Because developers of a test know what screen will be active in each step of that test, they can reference elements of that screen. For example, UiSelector is an element identifier used in Android tools [36] to specify elements by properties such as *ContentDescription*, *ClassName*, State information, Text value, and location in an interface hierarchy. Within the context defined by a screen identifier, our element identifiers in Chapter 5 use a similar approach to leverage developer familiarity with them.

2.7 Challenges in Graphical Interface Accessibility

Built-in screen readers (e.g., TalkBack [34] and VoiceOver [4]) on mobile touchscreen devices have become widely adopted by people with visual impairments. However, visual information is lost during the text-to-speech conversion (e.g., spatial layout or view hierarchies). Qualitative studies have investigated difficulties faced by people with visual impairments when using screen readers: Baldwin et al. conducted a long-term field study of novices learning general computer skills on desktops [8], McGookin et al. investigated touchscreen usage across many types of devices such as phones and media players [70], and Kane et al. specifically focused on mobile touchscreens [56]. These studies consistently found problems with locating items within graphical interfaces that were designed to be seen rather than heard. Furthermore, once objects are located, it is still a challenge to understand their location in the context of the app, remember where they are, and relocate them. This location and relocation process is time-consuming, with users needing to make multiple passes through content and listen to increasingly longer amounts of audio. Even then, they may not realize that their desired target is not reachable in the current context [8]. As Kane et al. and McGookin et al. demonstrated, this basic location problem exists not only on desktops but also on

mobile touchscreens. We would expect this problem to be compounded on phones because their targets are smaller and denser.

Because of these issues, users often carry multiple devices with overlapping functionality that each offer better accessibility for specific tasks [56,70]. A few examples include the media players with tactile buttons [70], or the popular Victor Reader Stream [49], a handheld media device specifically for people with visual impairments. Although iPods and smartphones support features similar to these devices, their flat screens make them more complex to use. However, carrying multiple devices can be difficult to manage [56] and costly [70] (e.g., the Victor Reader Stream costs $369). For a population that is more likely to live in poverty than those with sight [2], this represents a substantial barrier to equal information access.

These basic problems of locating, understanding, interacting with, and relocating objects via a touchscreen lead to the secondary problems of needing to manage multiple devices and the high cost of additional devices. Prior work to address these problems has taken different approaches to improving accessibility, ranging from software-focused (e.g., [56,58]), hardware-focused (e.g., [28]), to hybrid (e.g., [8,59]). Our Interactiles system in Chapter 6 takes a hybrid software-hardware solution to enhance mobile accessibility.

2.8 Software-Based Touchscreen Assistive Technologies

Software-based approaches to improving accessibility have introduced new interaction techniques designed to ease cognitive load and shorten task completion time. For example, Access Overlays [58] improves task completion times and spatial understanding through software overlays designed to help with target acquisition. However, although Access Overlays showed accessibility improvements on large touchscreens, such an approach may not work on mobile devices due to their significantly smaller screen size.

For mobile phones, Slide Rule re-maps gestures to support screen navigation separately from activating targets, as shown in Figure 2.4 [56], similar to the mechanism now supported by TalkBack and VoiceOver.

Text entry is an important activity on mobile devices. NavTap, a navigational text entry method that divided the alphabet into rows to reduce cognitive load, improved typing speed over a 4-month study [42]. BrailleTouch allowed users to enter Braille characters using three fingers on each hand and offered a significant speed advantage over standard touchscreen braille keyboards [84]. For the task of number entry, DigiTaps encoded gestures that differed from the ten numeric digits [6]. These gestures required a varying number of fingers to tap, swipe, or both. Although the DigiTaps system showed improvement over VoiceOver in number entry speed, it required users to remember the 10 gesture patterns and could also benefit from a tangible component.

In addition to text entry, mobile phones are also used for a much wider variety of activities, such as checking bus schedules, getting weather information, and updating personal calendars. These all require some form of text entry as well as app-specific actions [57]. Indeed, the plethora of apps on mobile phones [91] makes the creation of general interaction patterns challenging. Our Interactiles system in Chapter 6 supports common tasks by implementing 5 frequently used functions. The strategy in Chapter 4 maximizes its compatibility with existing mobile apps within security model.

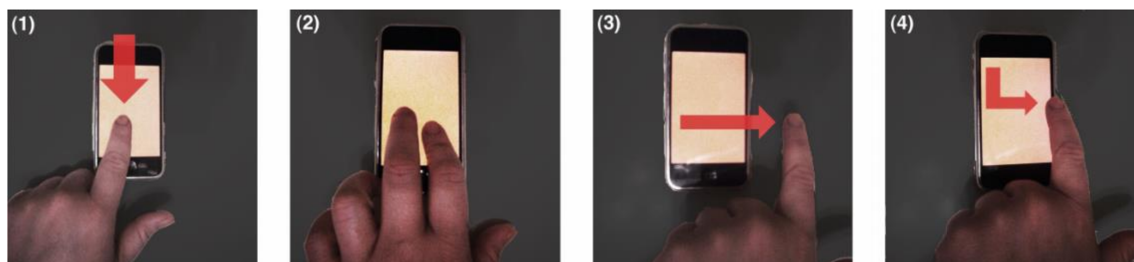


Figure 2.4. Slide Rule uses multi-touch gestures to interact with apps. (1) A one-finger scan is used to browse lists. (2) A second-finger tap is used to select items. (3) A flick gesture is used to flip between pages of items. (4) An L-select gesture is used to browse the hierarchy of items. This figure with caption is taken from [56].

2.9 Tangible GUI Accessibility

Although software solutions can improve accessibility, physical modifications also offer a promising complement to software. There also have been efforts to add hardware for enhancing touchscreen accessibility. One area of research has been in creating physical interaction aids for focused applications such as maps and graphics. For example, tactile maps overlaid on a touchscreen were fabricated using a conductive filament that could transfer touch [100]. In TacTILE, a toolchain was presented for creating arbitrary tactile graphics with both raised areas and cutout regions to present information [47]. Both of these approaches received positive feedback by making it easier to explore spatial information, but the tangible hardware was limited by being only useful for a single screen on a single application. In addition, the hardware was not attached to the phone and thus required the user to carry it separately.

For larger touchscreens, Touchplates introduced a set of guides to provide tactile feedback as shown in Figure 2.5 [59]. The guides are versatile and inexpensive because (1) they are compatible with existing software, (2) they can be simply made by cutting holes in inexpensive materials, such

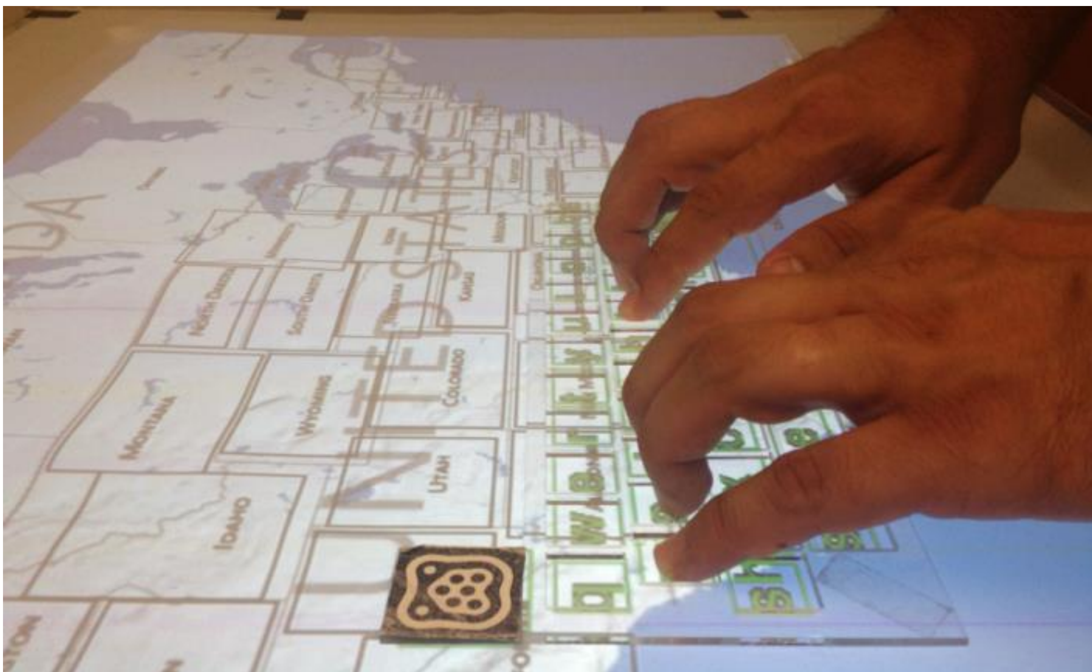


Figure 2.5. A QWERTY keyboard touchplate, cut from acrylic plastic, provides tactile feedback for a large touchscreen user interface. The touchscreen recognizes the guide and moves the virtual keyboard beneath it. This figure with caption is taken from [59].

as cardboard and acrylic sheets, and (3) they contain battery-less visual tags that allow the software event listeners to locate each touchplate on the screen. However, some interaction innovations of Touchplates may not work on smaller mobile devices: (1) removing, attaching, and storing guides while on-the-go is more inconvenient and (2) interacting with the smaller cutout on a mobile screen is more difficult.

Tangible interfaces have also been used to enhance screen reader access in desktop computers without a touchscreen. The Tangible Desktop showed significant improvements in task completion times on a personal computer by replicating traditional desktop metaphors in the physical world [8]. However, its setup of a potentiometer-powered scrollbar and physical icons would be difficult to carry around. Hybrid-Brailler combined physical and gestural interactions to provide fast and accurate Braille input [102]. Its physical interface is on the back of the phone where it does not block gestural interaction on the screen. Therefore, the physical interface communicates with phones via Bluetooth, which requires power and increases the system cost and complexity.

There appears to be an untapped opportunity to use tangible means to improve accessibility in mobile touchscreen computing. Prior work has demonstrated the potential of new software-enabled interaction techniques which make it easier to locate objects, but they do not fully use tangibility as a memory aid and tool to ease cognitive load. It has also been shown that inexpensive, easily fabricated tactile pieces have great promise, but do not extend well to a mobile setup because they have many pieces that are difficult to manage on-the-go or lack broad deployability on a mobile phone due to their narrow applications.

2.10 Security of Mobile Runtime Repairs and Enhancements

Compare to web and desktop platforms, the mobile platform has a stronger security model. System-wide accessibility enhancements on mobile platforms have sometimes been implemented by rooting a device [82]. Rooted devices obtain administrative access in order to examine system-level instrumentation of content and events. However, rooting introduces significant security risks because malware may also obtain administrative access to circumvent the security restrictions put

in place by the operating system. In addition, rooted devices will not receive over-the-air system updates and will therefore fall behind on security patches. Finally, the rooting process presents a technical expertise barrier for people with impairments, and the process may introduce malware from untrusted rooting tools.

Public Android accessibility APIs provide capabilities to develop accessibility enhancements. However, abusing these powerful APIs creates security risks. One significant security threat is "Clickjacking" (Click + Hijack). Originating on the web, clickjacking is a malicious UI redressing attack that applies transparent or opaque overlay(s) to trick a victim into clicking a UI element when the victim intends to click another UI element on the top-level overlay, as shown in Figure 2.6 [78]. For Android, clickjacking is more than a theoretical threat. Symantec found a ransomware Android.Lockdroid.E utilized clickjacking to gain Admin permission as an accessibility service using Android public APIs [95]. 95.4% Android devices were exposed to clickjacking when Symantec first

Figure 2.6. An example of "clickjacking". When user clicks the "next" button on the top-level overlay, the click event passes to the "install" button in the system dialog covered by the overlay.

presented this attack in March 2016. Google patched this security threat after API 22. However, 31.9% of Android devices remain exposed to the threat as of December 2018 [37].

Protecting against malicious accessibility services is an important topic for additional security research [54]. Prior research has explored composing interfaces from mutually distrustful elements [83], and the development of effective abstractions is important for improving the security of accessibility enhancements. Because an accessibility service can be used to implement a keylogger, ransomware attack, or phishing exploit, Google took efforts to protect users from malicious accessibility services by: (1) displaying a warning dialog when enabling accessibility services, (2) setting restrictions on public APIs (e.g., disabling overlay creation on top of system dialog buttons), (3) removing apps from the Play Store if they use accessibility APIs without providing accessibility benefits [107].

In Chapter 4, we will describe implementation detail and required public Android APIs for our approach, which works within the existing mobile security model.

Chapter 3 DESIGN SPACE OF INTERACTION RE-MAPPING

Effective interaction requires bridging both the gulf of evaluation (i.e., perceiving and understanding the state of an app) and the gulf of execution (i.e., manipulating the state of an app) [52]. To bridge these gulfs, our strategy modifies: (1) perception in an interaction, (2) manipulation in an interaction, or (3) both. We consider such modifications in terms of re-mapping existing interaction into new interaction.

The thesis of this dissertation is about ***an interaction remapping strategy that enables third-party repair and enhancement of mobile app accessibility***. To support developers and researchers in designing accessibility repair and enhancement, we present a design space of interaction re-mappings. This chapter considers five identified patterns of re-mapping and discusses potential accessibility repairs and/or enhancements in that region of the design space. Although many potential enhancements fit cleanly into the design space shown in Table 3.1, some enhancements are better considered to be a combination of several interaction re-mappings, each of which re-maps an interaction according to this design space.

Table 3.1. Modify perception, manipulation, or both to re-map existing interaction into new interaction. The number of interactions may change before and after a re-mapping.

Re-Mapping	**Purpose**	**Example**
Zero to One	Add a new interaction	Accessibility rating
One to Zero	Remove an interaction	Stencils-based tutorial
One to One	Replace an existing interaction with another	Label repair
One to Many	Replace an interaction with multiple interactions	2-stage click
Many to One	Replace multiple interactions with one interaction	Macro

3.1 From Zero to One Remapping

A re-mapping from zero to one adds a new interaction where there was none. By definition, this adds new information or functionality to a manifest interface [18], which a person uses to perceive and manipulate that app (e.g., the interface exposed by a screen reader like Android's TalkBack). We contrast this re-mapping with later examples that correct or replace an existing interaction. Re-mappings might integrate new information obtained from an outside service or restore elements of an original interface that are otherwise inaccessible in a person's chosen manifest interface.

Figure 3.1 shows an example of adding information from an outside service. In interviews presented in Chapter 4, blind and low-vision people report they often do not have ready access to information about an app's accessibility prior to attempting to use the app. If a third-party service rated app accessibility, it would be preferable to make ratings available directly at the time that information is needed (i.e., right before downloading the app). This proof-of-concept shows how an interaction re-mapping could add a button for getting accessibility ratings within the app download screen in Google Play Store. Google's Accessibility Scanner app similarly inserts a floating button that allows invoking an accessibility checker for the current screen [38], but it targets developer inspection of an app rather than end-users seeking information about the accessibility of potential apps.

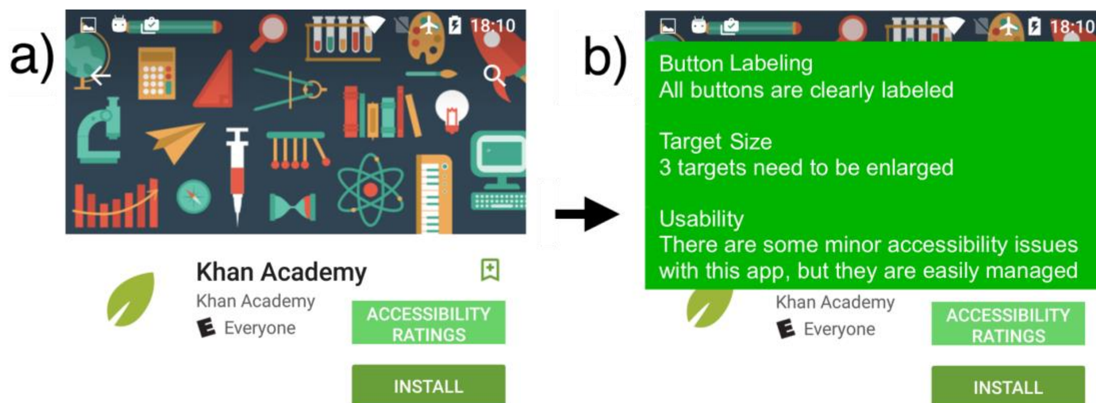


Figure 3.1. This interaction re-mapping adds third-party accessibility ratings directly within the app store. (a) An "Accessibility Ratings" button is added in previously empty space. (b) Selecting the button shows third-party accessibility information for that app.

Chapter 4 will further discuss Figure 4.4, an example of zero-to-one re-mappings that restore functionality missing in an interface manifested by a screen reader. Specifically, the Wells Fargo app fails to expose several of its interface elements, leaving them inaccessible to a person manifesting the interface with a screen reader. An interaction re-mapping repairs this, restoring access to each of the menu items from the original interface. Figure 4.5 presents the same type of problem occurring with the stars on Yelp's rating page. Inaccessible when using a screen reader, an interaction re-mapping restores these to the manifest interface.

3.2 From One to Zero Remapping

A re-mapping from one to zero removes an existing interaction. Examples include: (1) modifying a manifest interface to remove information or functionality from the original interface, and (2) modifications to correct or improve undesirable interactions introduced by a person's chosen manifest interface. For example, stencils-based tutorials remove access to much of an interface by limiting interaction to the current step in the tutorial [46,62]. In Figure 3.2, iOS's Guided Access feature similarly allows disabling a phone's motion detection or disabling touch in a region of an interface [5]. Additional examples can include removing interactions that provide no value or are problematic in a person's chosen manifest interface (e.g., advertising, interstitial screens, spurious animations that generate distracting notifications with a screen reader).

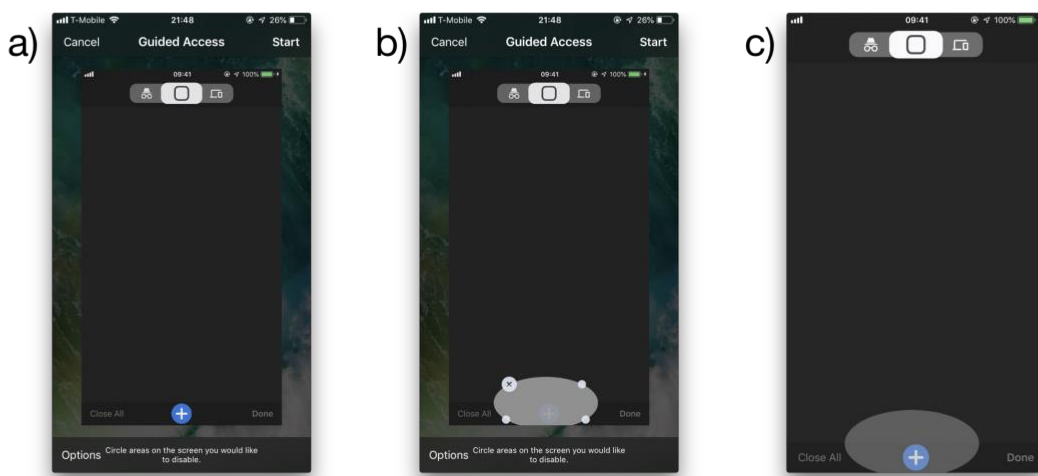


Figure 3.2. iOS Guided Access can disable touch in multiple screen areas. (a) The guidance to circle areas on the screen to disable is shown. (b) The bottom center area is circled and disabled. (c) Users cannot activate any UI element inside the circled area.

3.3 From One to One Remapping

A re-mapping from one to one replaces an existing interaction with another. Figure 3.3 shows an example of elements with missing or incorrect accessibility metadata, which are therefore inappropriately manifested by a screen reader (e.g., presented as "One, Unlabeled"). Manipulation thus works as desired, but elements are difficult to correctly perceive. An interaction re-mapping can correct this (e.g., presenting the button as "Start Timer"), proxying any manipulation to the underlying original interface. Although such a correction could also be considered a combination of a one-to-zero re-mapping (i.e., removing the incorrect data) and a zero-to-one re-mapping (i.e., adding the correct data), it is more straightforward to consider such direct replacement of an existing interaction as a one-to-one re-mapping.

Alternatively, an interaction re-mapping can modify manipulation while leaving information in individual elements unchanged. For example, our blind and low-vision interview participants report that needed functionality can sometimes be difficult to reach when manifested by a screen reader. Because swiping gestures traverse elements serially, a screen reader has to traverse all elements in a list to access the target at the end of the list (e.g., Yelp's search box is readily available in the original interface, but the screen reader reaches it at the end of a list in Figure 4.5).

Existing platform support for re-mapping interaction tends to be strongest for one-to-one re-mapping (e.g., both Android and iOS support interactively labeling elements that are missing screen reader metadata). But our interaction re-mapping strategy further allows third-party developers and researchers to develop, deploy, and evaluate new approaches (e.g., neither Android nor iOS

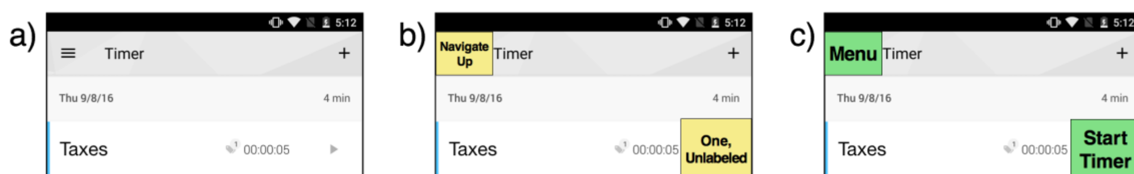


Figure 3.3. This interaction re-mapping repairs accessibility metadata that an app provides to a platform screen reader. (a) Toggl is a popular time-tracking app. (b) Its interface includes elements with missing or inappropriate metadata, which a screen reader manifests as "Navigate Up" and "One, Unlabeled". (c) Our third-party interaction proxy repairs the interaction using appropriate metadata for each element, so a screen reader correctly manifests these elements as "Menu" and "Start Timer".

Figure 3.4. This interaction re-mapping replaces one interaction with a sequence of two interactions. (a) An interface contains small adjacent targets. (b) The targets are replaced with a single larger "Tools" button. (c) Selection displays a menu of the original targets. (d) A selected item is activated.

provides integrated support for social annotation approaches to crowdsourcing accessibility text, as has been proposed and examined in web-based systems [88,97,98]).

3.4 From One to Many Remapping

A re-mapping from one to many replaces a single interaction with multiple interactions. This is often needed because the original interaction makes ability assumptions that are inaccessible to some people, such as fine-grained motor assumptions [105]. Unpacking such an interaction into a sequence of interactions can make it more accessible to more people. Doing so generally requires designing for both perception and manipulation because a person must navigate the new sequence of interactions. For example, prior work explores two-stage selection of small targets [29,55,116]. Figure 3.4 shows an interaction re-mapping that replaces a set of small adjacent targets with a single larger target that invokes a menu of choices.

3.5 From Many to One Remapping

A re-mapping from many to one replaces multiple interactions with a single interaction. For example, prior work has explored this principle in accessibility macros [81]. Figure 3.5 provides an example of a "scroll to top" macro. Such macros can provide a new method for manipulating an app through a sequence of interactions that might otherwise be difficult or laborious (e.g., interfaces that rely upon timing constraints or active modes that are difficult for a person with a

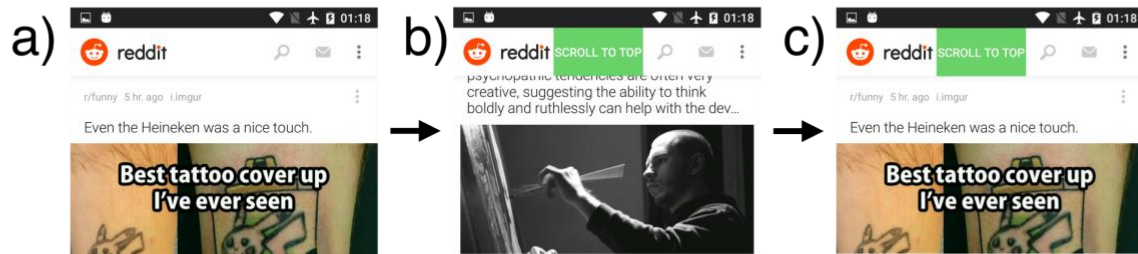


Figure 3.5. This interaction re-mapping replaces many interactions with one interaction. (a) An interface contains scrollable content. (b) The "Scroll to Top" macro button is shown after the content scrolls. (c) Once the macro button is activated, the content scrolls back to the top.

motor impairment to perform). A macro might also be invoked using a different modality (e.g., CoFaçade's support for configuring devices so older adults can access functionality using on-screen buttons, physical buttons, or physical gestures like scanning an RFID tag [115]). In contrast to many-to-one re-mappings of manipulation, many-to-one re-mappings of perception can be implemented by summaries. For example, a home automation app might contain many visual indicators of devices, which a screen reader manifests by serially scanning through each device and its status, but a many-to-one interaction re-mapping could add support for a summary of which devices are currently on.

3.6 Discussion

Apps and potential enhancements to those apps are composed of many interactions. In applying this design space to consider current and potential enhancements, it is helpful to consider enhancements as being composed of many re-mappings, each of which is characterized by one of these five patterns.

A simple case of composing re-mappings is when an enhancement applies the same type of re-mapping to multiple interactions (e.g., modifying screen reader metadata on multiple elements, modifying navigation order among multiple elements, disabling multiple elements as part of a stencils-based tutorial). But complex enhancements can also be understood as compositions of many re-mappings. Chapter 4 will discuss a proof-of-concept personalized interface shown in Figure 4.7, which was motivated by SUPPLE's generation of interface layouts adapted to an individual's motor abilities [30,105]. Considering such a complex enhancement in terms of its underlying re-

mapping of many interactions can provide a useful framework for approaching its design and implementation.

Most examples we used in this design space are (1) Android-based and (2) software-only. With proper implementation and permission within its security model, our conceptual design space could also be extended to iOS and other mobile platforms. In Chapter 6, we will introduce Interactiles, a general-purpose system that enhances tactile interaction on touchscreen smartphones. This system demonstrates extension of the design space to include hardware-based re-mapping of interactions.

This work was published at CHI 2017 as ***Interaction Proxies for Runtime Repair and Enhancement of Mobile Application Accessibility*** [112] with co-authors Anne Spencer Ross, Anat Caspi, James Fogarty, and Jacob O. Wobbrock. Please watch the demo video at: https://youtu.be/YXnbrVJ_8N0.

Chapter 4 INTERACTION PROXIES FOR RUNTIME REPAIR AND ENHANCEMENT OF MOBILE APPLICATION ACCESSIBILITY

To enable the interaction re-mappings described in Chapter 3, I introduce *interaction proxies*, an interaction re-mapping strategy that enables ***third-party runtime repair and enhancement of mobile app accessibility***. Conceptually, interaction proxies are inserted between an app's original interface and the manifest interface [18] that a person uses to perceive and manipulate that app (e.g., the interface exposed by a screen reader). As a strategy for third-party accessibility enhancements, interaction proxies contrast with both: (1) fixes in individual apps (which can be made only by an app's developer), and (2) platform-level enhancements (which can be made only by the platform developer). The strategy is analogous to web proxies [11,13,96], which modify a page between a server and a browser. However, it can be extended to address the design and implementation challenges of mobile app accessibility. This strategy allows third-party developers and researchers to modify an interaction without an app's source code or rooting the phone, while retaining all capabilities of the system.

This dissertation demonstrates interaction proxies on Android, modifying interactions without accessing an app's source code and without rooting the phone or otherwise modifying an app, while retaining all system capabilities (e.g., Android's full implementation of the TalkBack screen reader). Our implementations achieve this by modifying selected interactions using an accessibility service that creates overlays above an app while listening to and generating events to coordinate an interaction. We also summarize key abstractions to implement interaction proxies in Android.

We then present a set of interviews with blind and low-vision people interacting with prototype interaction proxies, discussing their experiences and thoughts regarding such enhancements. As we emphasize throughout this chapter, our contribution is not intended to be a specific accessibility enhancement. Rather, we aim to demonstrate the potential of interaction proxies as a ***strategy*** to support a variety of repairs and enhancements to the accessibility of mobile apps.

The specific contributions of our work include:

- Interaction proxies as a strategy for runtime repair and enhancement of mobile app accessibility.
- A set of techniques for implementing interaction proxies on Android without rooting the phone or otherwise modifying an app. Proof-of-concept implementations of interaction proxies demonstrate a range of technical techniques that developers and researchers can employ to implement, deploy, and evaluate accessibility improvements.
- Two sets of interviews with blind and low-vision people who use screen readers, first exploring accessibility barriers they encounter in apps and then exploring their reactions to using prototype interaction proxies as well as their thoughts on the potential of interaction proxies for enhancing mobile app accessibility.

4.1 Interaction Proxies

Effective interaction requires bridging both the gulf of evaluation (i.e., perceiving and understanding the state of an app) and the gulf of execution (i.e., manipulating the state of an app) [52]. These gulfs are bridged using an interface, and people commonly use different interfaces to access the same underlying app. Borrowing from Cooper [18], we refer to the interface a person directly perceives and manipulates as the manifest interface. We contrast it with the original interface created by an app developer. For many people, the manifest interface may be the same as the original interface. However, many people with disabilities elect an alternative manifest interface. Many blind or low-vision people choose a screen reader interface (e.g., Slide Rule [56], Android's TalkBack, iOS's VoiceOver), which re-maps touch to support navigation separate from activation and may completely disable the visual display of an app. Many people with motor impairments choose a scanning interface, which relies on visual perception but re-maps manipulation by scanning through potential targets that a person then activates using a switch. Even attaching an external keyboard or an additional display to a device can be considered a change

in the interface used for interaction. Conceptually, interaction proxies are inserted between an app's original interface and manifest interface, as illustrated in Figure 4.1.

We adopt the term "proxy" to be analogous to proxies on the web, which can modify webpages between a server and their rendering in a browser (including web proxies intended to improve web accessibility [11,13,96]). However, mobile app architectures do not allow the same approach to directly modifying the underlying representation of an interface prior to its rendering. Inserting an interaction proxy aims to improve interaction by modifying how an app is perceived or manipulated, without actually modifying the app. Key benefits are: (1) it does not require modifying the app, and (2) it does not require modifying the renderer of a manifest interface (e.g., a person continues to use Google's full implementation of Android's TalkBack screen reader). Figure 4.1 illustrates the insertion of an interaction proxy to repair the accessibility metadata shown in Figure 3.3. Using floating windows and other methods, the interaction proxy provides corrected accessibility metadata. It can be implemented by a third-party developer or researcher. The interaction proxy's modifications are then presented as part of the interface that is manifested by the platform's screen reader.

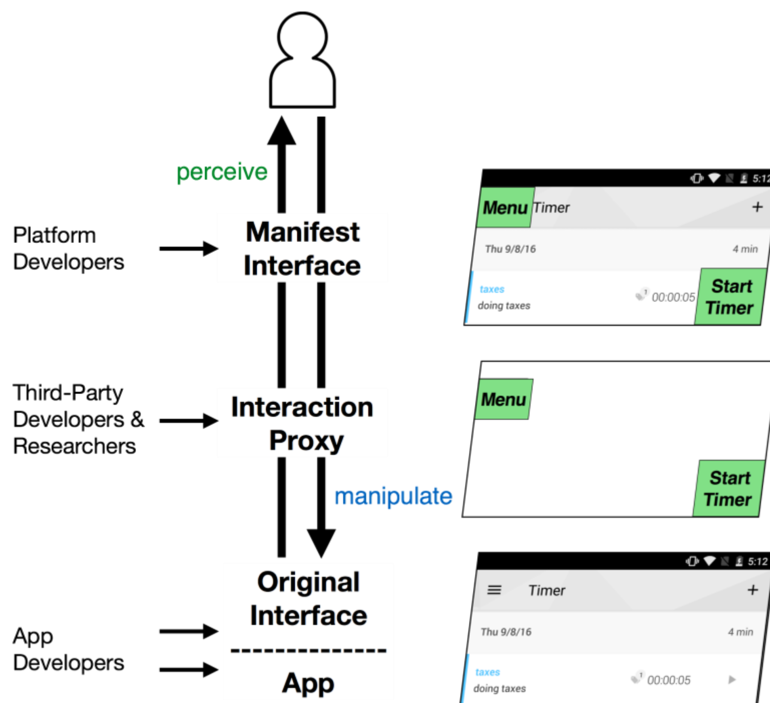


Figure 4.1. Interaction proxies are inserted between an app's original interface and the manifest interface to allow third-party modification of interactions.

4.2 Implementation in Android

Chapter 3 provided a framework for considering the design of interaction proxies. Here we present a set of implementation abstractions and describe how each is implemented in Android. Our abstractions can resemble prior examinations of runtime interface modifications [25,93], so we focus on details for interaction proxies in Android. Abstractions provide a higher-level language for discussing enhancements and convey what is needed to achieve these behaviors on other platforms. After introducing the primary abstractions, we discuss their composition to coordinate perception and manipulation over the course of modifying an interaction. Maintaining this coordination of the interaction is critical for an effective proxy.

Our interaction proxies are implemented as an Android accessibility service, which is allowed to inspect and manipulate an app using public Android APIs. Installing an accessibility service requires explicit consent, and none of our enhancements require rooting the phone or otherwise modifying the operating system since we believe it is inappropriate to require a person to compromise their devices' security to access an app.

4.2.1 Abstractions for Android Interaction Proxies

Our basic strategy is to minimize the scale and complexity of an interaction proxy by intervening as little as necessary in an interaction. We achieve this strategy using a combination of the following four abstractions for implementation in Android. We note the percentage of Android devices supporting each of the core requirements, as of December 2018 [37]. Some capabilities are limited to more recent versions of Android, which have more limited market share. These capabilities will become more available as people transition to newer devices.

Floating Windows: Our interaction proxies intervene between an app’s original interface and the manifest interface by creating floating windows that the interface manifests instead of the underlying app. Similar to overlapping windows on the desktop, floating windows sit above the app in z-order. We further define a *full overlay* as a floating window that occludes the entire underlying

app, and a *partial overlay* as occluding one or more elements while leaving other interactions unmodified. Enhancements composed of multiple re-mappings will often coordinate multiple partial overlays within an app. The floating windows capability has been included since Android 1.0 and is available on 100% of Android devices [37].

Event Listeners: Some enhancements require detecting events in the underlying app (e.g., to trigger an update in an overlay). Android allows an accessibility service to listen to interface accessibility events (e.g., a button click, a text field focus, a view update, an app screen switch). Although limited to accessibility events that are invoked by the app, many necessary events are implemented by the default Android APIs. This capability has been included since Android 1.6 and is available on 99.9% of Android devices [37].

Content Introspection: Some enhancements require knowledge of the content of elements in an underlying app. Proxies can inspect the app accessibility service representation. This provides a tree describing an app, where each node provides information about an element (e.g., content, size, state, possible actions). However, we have noted that apps do not always expose correct or complete information via this representation. This capability has been included since Android 4.1 and is available on 99.5% of Android devices [37].

Automation: Some enhancements require manipulating an underlying app. Accessibility service automation support allows programmatic invocation of common manipulations of elements exposed via the accessibility service representation (e.g., click, long press, select, scroll, or text input directed to a node in the tree). This capability has been included since Android 4.1 and is available on 99.5% of Android devices [37].

Screen Capture*:* Some enhancements require information or presentation details not available through content introspection. This information can include details of a view that the corresponding accessibility representation does not include in its model (e.g., pixel-precise positioning of content). The accessibility metadata provided by an app may also be incomplete or incorrect. Screen capture allows application of pixel-based methods developed in prior work (e.g.,

[23,24,109]), though some apps intentionally disable screen capture (e.g., banking apps). Screen capture has been included since Android 5.0 and is available in 88.9% of Android devices [37].

Gesture Dispatch: Some enhancements require simulating a gesture or touch because the automation events are not expressive enough for an enhancement to obtain the desired behavior, or because an enhancement needs to manipulate an element that is not properly exposed by the accessibility representation. Gesture dispatch allows simulating a gesture or touch at any screen location. This capability has been included since Android 7.0 and is available on 49.7% of Android devices [37]. This capability will become more common as people transition to newer devices.

Current Implementation Limitations: We focus on Android, but support for these abstractions in other mobile platforms should similarly enable interaction proxies. We have also identified a few limitations of Android's current support.

First, Android does not provide a robust unique identifier for each screen within an app. Similarly, the field used for differentiating among elements within a screen is optional and often not specified by developers. Reasoning about the state of an app can therefore be challenging. Our current proxies use signatures computed from the tree exposed for content introspection. Pixel-based methods for annotation of interface elements could also be valuable [24].

Second, although an accessibility service can observe events, it cannot consume them (i.e., cannot redirect or prevent an event from occurring). The next section presents one implication of this inability to consume events in coordinating perception and manipulation. Finally, we noted that accessibility services relied on app developers to provide necessary events and metadata. Default tools provide this information automatically whenever possible, but it is often missing. Our inclusion of screen capture and gesture dispatch provide additional options for interaction proxy implementation.

4.2.2 Coordinating Perception and Manipulation

A person perceives and manipulates the manifest interface, but that interaction must be re-mapped to the underlying original interface. Coordinating perception and manipulation in a re-mapping is critical to the illusion of the manifest interface remaining seamless (i.e., the interaction proxy being perceived as part of the interface, as opposed to itself being disruptive). The complexity of this coordination will vary according to the nature of the interaction and how it is re-mapped.

We found that direct interaction is generally most straightforward to coordinate, as illustrated in Figure 4.2. Proxy implementation using floating windows means an interface is composed of layers. Projecting these layers into the display (i.e., flattening them along the z-axis) naturally results in interaction proxies occluding anything below. Manipulation is similarly straightforward. Figure 4.2 (left) shows the selection of a button in the manifest interface and Figure 4.2 (right) shows that hit-testing finds nothing at that location in the proxy layer and so passes selection to the underlying original interface. This straightforward coordination similarly extends to other direct interaction in the flattened interface. For example, the expected elements are naturally presented by Android

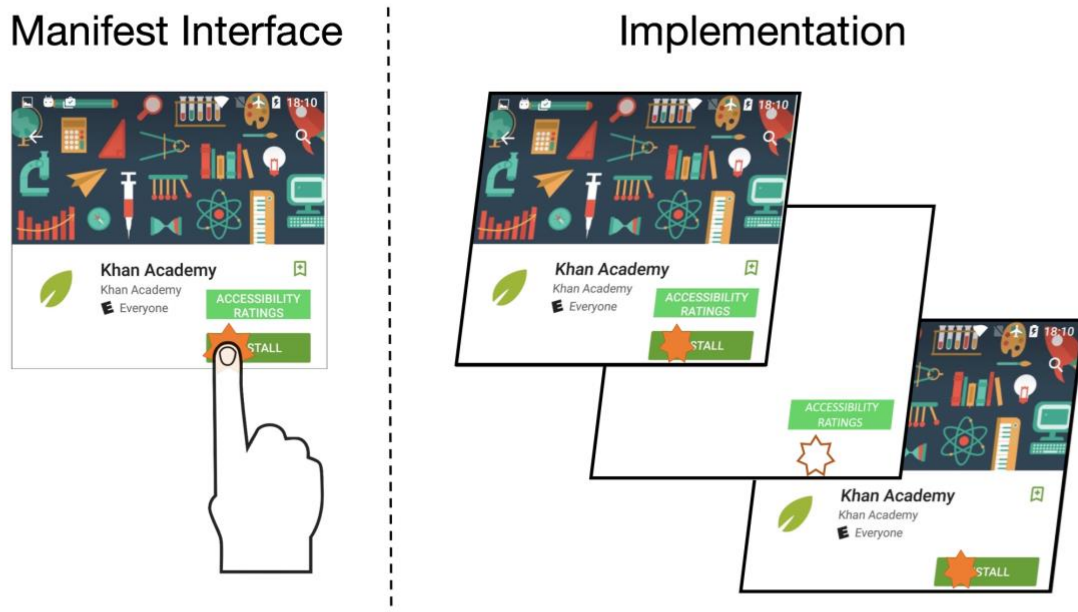
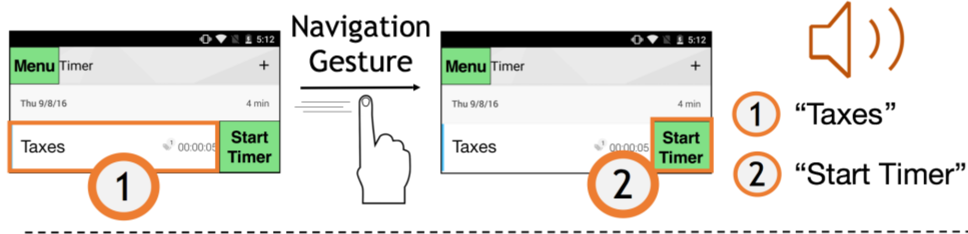


Figure 4.2. Direct interaction is generally straightforward to coordinate, with interface layers behaving as expected in their occlusion and in mapping input to the appropriate element.

Manifest Interface



Implementation

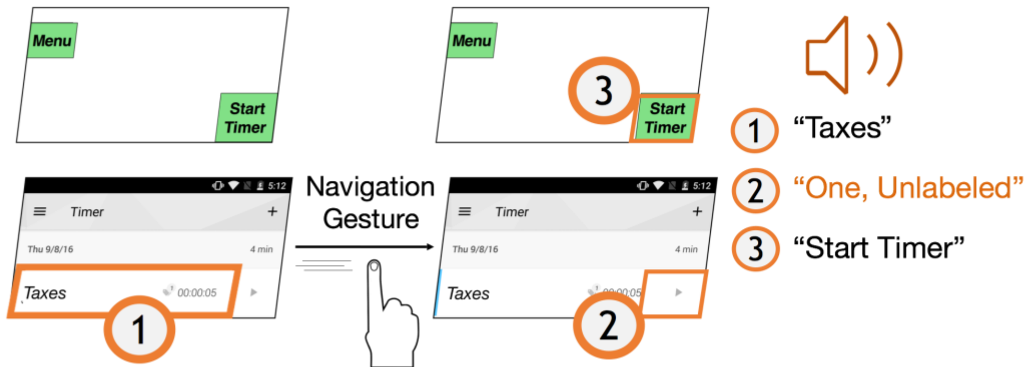


Figure 4.3. Indirect interaction can require more coordination. Here a gesture-based navigation requires an event listener to watch for "One, Unlabeled" to get focus and then immediately gives focus to "Start Timer". Perception in the manifest interface is seamless because the screen reader truncates the reading of "One, Unlabeled", but this example illustrates the type of coordination an interaction proxy may need to implement to remain seamless.

TalkBack's exploration mode (i.e., as a person moves a finger around the screen to browse an interface with the screen reader, the expected element is naturally presented).

Coordination can be more challenging for indirect interaction. Figure 4.3 shows a gesture-based interaction (i.e., swiping right to the next element in the navigation order). The manifest interface includes a button with correct metadata (i.e., "Start Timer") inserted to replace an original interface button that lacked metadata (i.e., "One, Unlabeled"). Figure 4.3 (top) illustrates the interaction experienced, swiping right to hear Android's TalkBack read "Start Timer". Figure 4.3 (bottom) shows that the navigation gesture is received by the original interface, which is unaware of the interaction proxy and gives focus to "One, Unlabeled". The proxy detects this using an event listener and immediately gives focus to "Start Timer", which is read aloud. In the instant that "One, Unlabeled" has focus, the screen reader begins to read it. However, this is imperceptible because focus moves to "Start Timer" and the screen reader aborts the prior reading (by design, ensuring prompt

perception of interaction state during rapid navigation). The illusion of the manifest interface is therefore maintained. Although we do not present all of our interaction proxy implementations in this same detail, this example is intended to illustrate a typical coordination of an indirect interaction.

4.3 Demonstration Implementations

Prior sections introduced several demonstration interaction proxies as part of conveying the strategy, design space, and implementation abstractions. This section presents additional details and demonstrations. All of our demonstrations were developed as proof-of-concept prototypes. By showing and explaining their key technical approaches, we aim to inform future development of the interaction proxy strategy. Details of these proof-of-concept implementations can also be found in their code, available at: https://github.com/appaccess.

4.3.1 Adding or Correcting Accessibility Metadata

Developers often fail to provide appropriate accessibility metadata for interface elements (e.g., labels for text fields). Although default tools provide this automatically whenever possible, people often encounter apps that have incomplete or incorrect metadata. Platforms have begun to support interactive correction, allowing a person to apply a custom label. But support is limited (e.g., Android only supports custom labels for elements with a `ViewIDResourceName`, which is itself optional and often not specified by app developers). Interaction proxies offer a strategy for third-

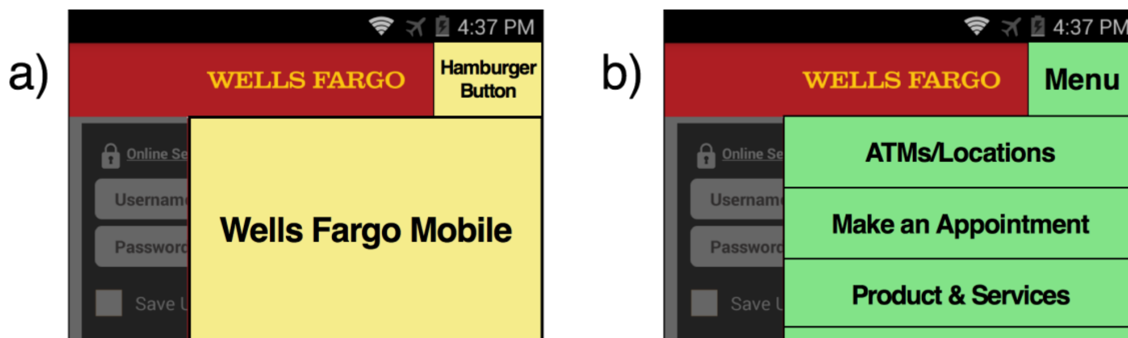


Figure 4.4. This interaction proxy corrects a "Menu" label and repairs interaction with the app's dropdown menu, which is otherwise completely inaccessible with a screen reader.

party developers and researchers to develop and explore new approaches (e.g., social annotation approaches that have been proposed and examined in web-based systems [88,97,98]).

Figure 4.1 shows an example from Toggl, a popular time-tracking app. The "Start Timer" button is missing metadata (i.e., the screen reader announces it as "One, Unlabeled") and the menu button has metadata resulting from an implementation artifact (i.e., is read as "Navigate Up"). Figure 4.4 shows a similar example in the Wells Fargo banking app (i.e., the label "Hamburger Button" is an implementation term better presented as "Menu"). Our interaction proxies identify these failures through content introspection, then obtain an image of the element using screen capture. Captured images can be used to obtain content descriptions (e.g., our proof-of-concept prototype uses a local database, envisioning social annotation mechanisms in future systems). Each failure is then repaired using a floating window to create a partial overlay that replaces the element in the manifest interface. Any manipulation of the element in the floating window is proxied to the original interface, using automation to activate the appropriate element.

4.3.2 Restoring Missing Interactions

Figure 4.4 also illustrates the repair of a similar but more severe failure in the Wells Fargo banking app. The original interface's dropdown menu includes several important functions but does not correctly present itself to accessibility services. It therefore manifests as a single large element, is read as "Wells Fargo Mobile", and activates whatever menu item is in its physical center (e.g., "Make an Appointment"). Figure 4.5 (left) shows a similar flaw in the Yelp app, which exposes its

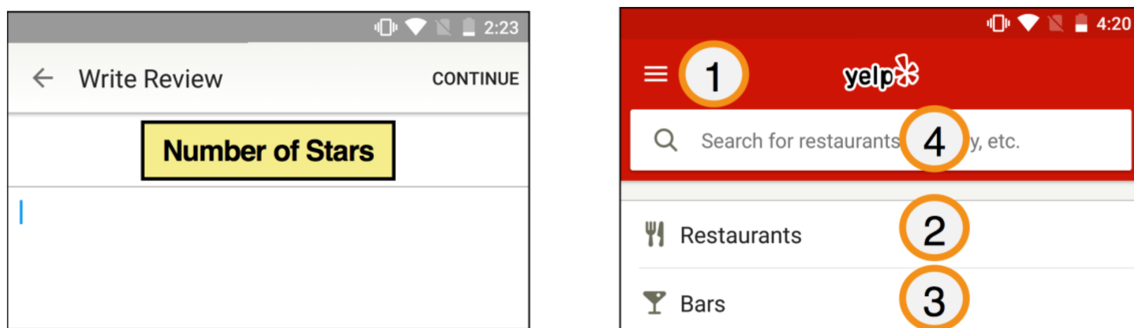


Figure 4.5. The Yelp app manifests its five-star rating system as a single element that cannot meaningfully be manipulated using a screen reader, and its navigation order using a screen reader makes it needlessly difficult to access "Search".

five-star rating system as a single element that cannot be meaningfully manifested by a screen reader. An interaction proxy repairs this issue using screen capture, a floating window, and content introspection. The proxy cannot use automation to manipulate underlying items since this will again activate whatever item happens to be in the physical center of the erroneously monolithic element. The proxy instead activates the correct underlying item using gesture dispatch (i.e., sending a two-finger touch to the correct screen coordinate, which is consumed by the screen reader, generating a touch in the underlying original interface).

4.3.3 Modifying Navigation Order

Navigation order is a significant aspect of an interface, and optimal navigation may differ for different manifest interfaces (e.g., parallel visual scanning, touch-based exploration using a screen reader, serial navigation using a switch interface). Inappropriate orders can also result from an implementation failure, similar to other incorrect accessibility metadata. Figure 4.5 shows an example where an implementation error means the Yelp search box visually appears before the scrolling list of businesses but is manifested after that list in a screen reader (i.e., making it difficult to access via serial navigation). Toggl similarly includes a "Create Timer" button that visually floats above the list of existing timers for easy and prominent access but is manifested to a screen reader at the end of the list of existing timers. Our interaction proxies modify these navigation orders, moving the appropriate elements to the beginning of the manifest interface, using content introspection, event listeners, and automation, coordinating interaction similarly to how it was done in Figure 4.3.

4.3.4 Fully-Proxied Interfaces

Our prior interaction proxies have been minimal, emphasizing the ability to repair or enhance the accessibility of an interaction without needing to re-implement unrelated portions of the original interface. Such targeted re-mappings also highlight challenges of coordinating an interaction so that it blends in to the surrounding interface. Our same abstractions can also be applied in

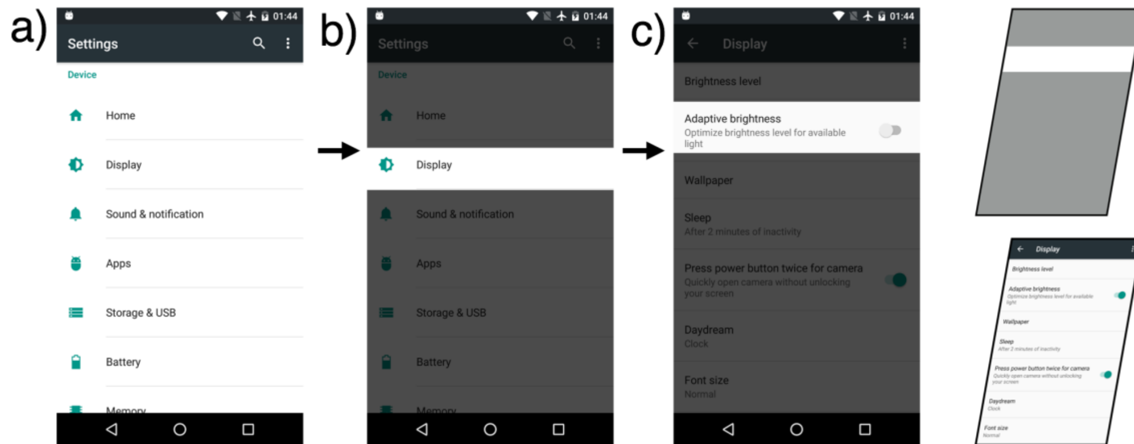


Figure 4.6. This interaction proxy implements a stencils-based tutorial. At each tutorial step, only the appropriate element is available. All other elements are obscured and disabled.

enhancements that proxy the entire interface, more completely changing the interface's manifestation.

Figure 4.6 shows an example of re-mapping an interaction to implement stencils-based tutorials. This technique was designed to guide a person through an interaction using translucent colored stencils containing holes that direct the user's attention to the correct interface component and prevent the user from interacting with other components [46,62]. Stencils could complement features like iOS Guided Access, provide additional capabilities, and enable third-party development of different potential guides for varying needs (e.g., stencils guide a person to a device setting). Our proxy is implemented using a floating window to create a full overlay, displayed as a translucent overlay and capturing all input. For each tutorial step, it uses content introspection to determine the bounds of elements that should be visible through the overlay (i.e., holes in the stencil), then uses automation to proxy manipulation of those elements. An event listener ensures the proxy's prompt response to interface changes, as when the interface advances between each step in the tutorial.

Figure 4.7 shows a sample personalized interface layout, motivated by SUPPLE's approach to arranging entire interfaces to match an individual's motor abilities [30,105]. Although such personalization is promising for many accessibility needs, adoption of such methods is limited by a need for interfaces to be re-written as abstract specifications [72]. We instead propose an

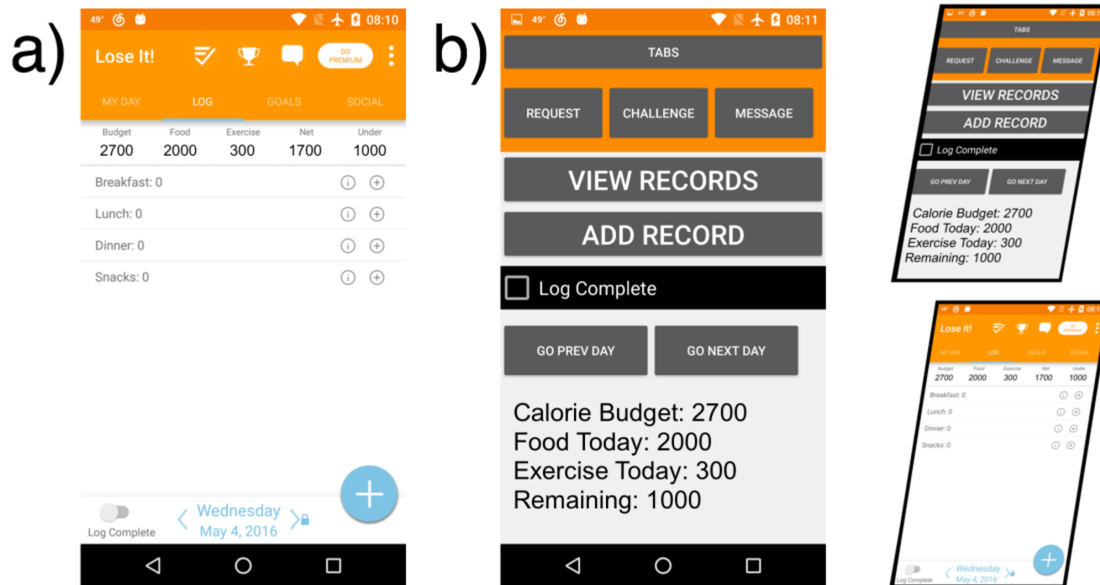


Figure 4.7. Motivated by research in personalized interface layout, this interaction proxy creates a completely new personalized layout for interacting with the underlying app.

interaction proxy could re-map an original interface into an abstract specification that is then used to generate a personalized interface. As a proof-of-concept, our interaction proxy pictured in Figure 4.7 re-maps the original interface for Lose It! (a popular food journaling app) into model-level variables and then manifests those in a new interface. The new interface uses a different layout strategy to support larger text and buttons, and it presents underlying functionality differently (e.g., replacing a small sliding widget toggle with a larger checkbox element). Full implementation of this strategy is future research, but our proof-of-concept demonstrates the necessary combination of a floating window as a full overlay, perceiving the original interface using content introspection with event listeners and manipulating it using automation.

4.4 Interviews Regarding Interaction Proxies

We conducted two sets of interviews with blind and low-vision people who use screen readers. Interviews focused on: (1) accessibility barriers and the contexts in which they are encountered, (2) the experience of using an interface with an interaction proxy, (3) usefulness and potential of interaction proxies, and (4) potential for adoption of such enhancements.

4.4.1 Method

Our first set of interviews included eight people who are blind or low-vision and use a screen reader. We discussed what types of accessibility barriers these participants encounter in apps, how they navigate the barriers, how barriers could be addressed, and specific apps in which barriers are encountered. These interviews informed many of the interaction proxies developed in this paper. Specifically, participants identified three common barriers that could be addressed with interaction proxies: mislabeled elements, inaccessible functionality, and challenging navigation. Participants also identified three major categories of app of interest: community engagement, productivity, and banking apps. Our proof-of-concept demonstrations and our second set of interviews therefore focused on these needs and opportunities.

Our second set of interviews included six people who are blind or low-vision and use a screen reader (including two from the first set). We developed three proof-of-concept enhancements to present to participants in support of these interviews: (1) Yelp: As illustrated in Figure 4.5, we repaired the stars on the business rating page to make it possible for a person using a screen reader to rate a business and repaired the navigation order to make the search box easier to reach. (2) Toggl: As illustrated in Figure 4.1, we repaired missing screen reader labels for elements associated with each existing timer. We also repaired navigation order to make the "Start New Timer" button easier to reach. (3) Wells Fargo: As illustrated in Figure 4.4, we repaired the items in the dropdown menu to make them accessible with a screen reader and repaired the label of the menu button.

Interviews with these participants focused on the potential of interaction proxies to support accessibility enhancements. We asked each participant to use the Android Talkback screen reader to interact with the above three apps (on a Google Nexus 6P with Android 7.0). For each app, participants first completed simple tasks while our interaction proxy applied its repairs, then again with the screen reader manifesting the original interface. We chose this approach (e.g., instead of counterbalancing) in part because tasks generally could not be completed without the interaction proxy. Our focus was therefore on qualitative reactions rather than task metrics. In addition, asking participants to begin with a task we knew was impossible would have undermined the interview.

Participants discussed the feeling of the interaction with the interaction proxy active, how well it addressed accessibility barriers, and their ideas for the potential of accessibility enhancements. Interviews were transcribed and then analyzed using open coding.

Participants primarily reported using an iPhone, the more popular choice for people in the United States who use accessibility services [71]. Two reported using Android. Interaction with iOS's VoiceOver screen reader is similar to Android's TalkBack, and the barriers within apps are similar between platforms. We therefore believe these participants provided useful insight into the interaction proxy strategy.

4.4.2 Results

Interview participants reported our proof-of-concept prototype interaction proxies worked well for improving the accessibility of interactions. P3 said *"I think in every case [the enhancement] made it a much better experience than it would normally be"*, while P5 said *"I think the enhancements have made it better".*

One goal for participant interaction with our prototypes during interviews was to examine seamlessness of the interaction (i.e., an interaction proxy being perceived as part of the interface as opposed to itself being disruptive). Interviews explicitly probed this, in part by having participants first interact with the enhanced interface and then the app with its underlying accessibility failures. Participants ideally would not be able to distinguish between a natively accessible interface versus an interface that had been repaired or enhanced by an interaction proxy, and several participants commented that interactions were seamless. P3 said *"[the enhancements] made it behave as I would expect it to. I think, when the enhancements were on, I generally didn't have any trouble completing the tasks, which definitely means it's working"*, while P2 said *"[the enhanced Yelp app] acted the way I would expect it to act"*.

Two participants commented on swipe-based navigation when using a proxy that repairs metadata by changing screen reader focus (as discussed with Figure 4.3). P1 described "*lagginess*", and we did note the app and enhancement were unusually slow for this participant. P2 described

"*oversensitivity*" that made it more difficult to use swipe-based navigation to select a target without skipping it. However, P2 also explicitly noted that the value provided by the enhancement was sufficient to outweigh "oversensitivity". Even when prompted, participants did not mention any other unusual or bothersome interactions.

Participant responses to the specific enhancements we showed were varied. For example, all participants agreed that it was an improvement for the Wells Fargo app menu to be accessible, but all felt the "Forgot Password" item was still difficult to find because the app hid it in the dropdown menu. Even when an interaction proxy improves accessibility of an interface, usability barriers due to poor interface design choices can remain.

All participants described how the types of enhancements we demonstrated could be made to address barriers they encounter, and all expressed interest in using such enhancements. P1 said *"In email…forward and reply all are at the very bottom of the message and if it's a really long message, it's really a pain to have to scroll all the way to the bottom of the message"*. P5 said *"[an enhancement] would definitely be a value to be able to get the Greyhound app accessible so that I could be able to purchase tickets and look at the schedules and so forth"*.

In discussions regarding the potential for broad deployment, all participants said they would be willing to submit apps needing repair or enhancement if they thought it was likely the enhancement would be created. P3, a software developer, indicated that he would be willing to create enhancements if good tools were available. P3 and P5 also expressed concern over whether enough people would contribute enhancements.

P6 was more optimistic about participation, saying *"I do think that people would be very interested in it, and I think people would want to help contribute to the actual programming, and then people would also be interested in making suggestions."* Finally, participants discussed their trust of third-party enhancements. They reported that primary factors in whether they would trust an enhancement enough to download it are the reputation of the source, endorsements from known organizations (e.g., the National Federation for the Blind), and feedback from other people who use screen readers.

4.5 Discussion and Conclusion

We introduced interaction proxies as a strategy for runtime repair and enhancement of the accessibility of mobile apps. Inserted between an app's original interface and a manifest interface, an interaction proxy allows third-party developers and researchers to modify an interaction without an app's source code and without rooting the phone or otherwise modifying an app, while retaining all capabilities of the system. We examined the interaction proxy strategy by defining key implementation abstractions and implementing them in Android proof-of-concept interaction proxies. Details of these proof-of-concept implementations can also be found in their code.

These technical contributions are our primary contributions, and our interviews with blind and low-vision people who use screen readers provide support for further developing this strategy. Participants were enthusiastic about the strategy, based on our proof-of-concept prototypes repairing accessibility failures in popular real-world apps. Including these prototypes in our interviews provided a real-world context for discussing the potential of interaction proxies, and participants used this as a starting point for discussing other apps in which they have encountered accessibility barriers that might be addressed. Participants saw a need and potential for third-party enhancements, expressing interest in the value they could provide even if a repaired interaction was not quite seamless.

Our ultimate goal is to help catalyze advances in mobile app accessibility. Interaction proxies should not replace the effort of app developers to correctly implement the platform accessibility support. Instead, interaction proxies open an opportunity for third-party developers to create and deploy accessibility repairs and enhancements into widely used apps and platforms, in contrast to how contributions have previously been limited to developers of individual apps or the underlying mobile platform. P1 supported a multi-faceted approach by saying *"I mean I think what you did is great, to make some more improvements, but also how we can work with community people and ideally Google and [app] developers"*. Interaction proxies therefore provide an additional tool that complements efforts to educate and support developers in improving the accessibility of their apps, as well as improvements in platform accessibility support. Interaction proxies also enable the work

in the next two chapters to (1) repair mobile app accessibility issues with a robust metadata annotation system and (2) enhance mobile touchscreen tactile feedback with 3D printed interfaces.

This work was published at CHI 2017 as ***Interaction Proxies for Runtime Repair and Enhancement of Mobile Application Accessibility*** [112] with co-authors Anne Spencer Ross, Anat Caspi, James Fogarty, and Jacob O. Wobbrock. I developed the initial idea, summarized the implementation abstractions, and implemented the prototype interaction proxies used in the interview. All co-authors provided valuable ideas in the demonstration examples and paper. Anne also took a lead in the interviews. Please watch the demo video at: https://youtu.be/YXnbrVJ_8N0.

Chapter 5 ROBUST ANNOTATION ON MOBILE APP INTERFACES

Previous chapters introduced accessibility barriers on mobile platforms and the potential of third-party accessibility repair. Interaction Proxies enable proof-of-concept demonstrations, but it is infeasible to scale demonstrations beyond a handful of elements in a handful of apps due to the lack of methods for determining where and how to apply a runtime repair.

The web platform has a robust method for determining where to apply an annotation, typically using the combination of a URL, which indicates the context to apply an annotation, and an XPath, which indicates the element to annotate within the URL's context. With this robust identification method, social annotation techniques have demonstrated compelling approaches to accessibility concerns on the web. However, it is difficult to apply these techniques in mobile apps because they lack ways to specify an annotation context (i.e., a robust screen identifier, the equivalent of a URL in web annotation).

To address this difficulty, I developed methods for ***robust annotation on mobile app interfaces***, implemented in screen identifiers, element identifiers, and screen equivalency heuristics. By integrating annotation-based techniques, proof-of-concept accessibility repairs in Chapter 4 can become more reliable and scalable. This offers an important step toward large-scale runtime accessibility repair in mobile apps.

The research in this chapter pursues a vision of opportunities for social annotation in mobile accessibility. Social annotation has been a powerful tool in web accessibility (e.g., [12,48,87,88,97]), and I believe it can also benefit mobile accessibility. As a demonstration of the feasibility of social annotation in mobile apps, I implemented initial developer tools for annotating mobile app accessibility metadata, evaluated key screen equivalence heuristics, presented case studies of runtime repair of common accessibility issues, and examined repair of real-world accessibility issues. My current work limits the scope of contributors to developers because they have the necessary technical skills to handle edge cases in annotation tools (e.g., manually authoring selectors to repair errors in screen identification). The underlying methods implemented in this initial research could

be used to build new annotation tools that do not require developer-level expertise, including tools to support crowdsourcing or friendsourcing approaches. In my vision, a crowdsourcing-powered annotation tool can divide the task of repairing an inaccessible screen (e.g., a screen may contain many mislabeled buttons) to several simple microtasks (e.g., crowd workers see the image of a button and write its description). Therefore, the annotation process might be driven by volunteers and paid crowd workers. By ensuring the accessibility of crowdsourcing tasks [94], people with disabilities may also make their own contributions in reporting accessibility issues and providing appropriate metadata to address some issues.

The specific contributions include:

- Development of methods for robust annotation of mobile app interface elements. Designed for use in runtime interface modification, these methods combine a novel approach to screen identifiers and screen equivalence heuristics with familiar techniques for Android element identification.
- Implementation of initial developer tools for annotating mobile app accessibility metadata, including tools for authoring annotations and applying annotations at runtime.
- Evaluation of our current screen equivalence heuristics in a dataset of 2038 screens collected from 50 mobile apps.
- Three case studies that demonstrate the implementation of runtime repair of common accessibility issues, each using the robust annotation methods developed in this research.
- Examination of repairing real-world accessibility issues in 26 apps, including popular Android apps, apps with accessibility issues reported in online forums, and apps identified through an in-person interview with a person who regularly uses the Android TalkBack screen reader.

5.1 Android Background

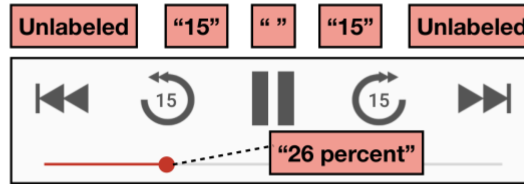
This section reviews several Android capabilities. I first discuss why existing capabilities are inappropriate for robust annotation and then provide background on accessibility services and Android's existing limited repair capabilities.

Android's accessibility services expose a *WindowId* for each *View*. Intended to support input interactions across multiple processes, *WindowId* is not stable (i.e., it changes each time an app is launched). Therefore, it is an inappropriate identifier for storing annotations that will be used in future sessions.

When available, Android's accessibility services also expose a *ViewIDResourceName* for each *View*. *ViewIDResourceName* is Android's primary approach to a robust identifier (e.g., to be used in automated testing). Unfortunately, it is optional and often not specified by an app developer. When specifying an app's layout in an XML layout file, including a *ViewIDResourceName* allows developers to obtain a reference to that element at runtime (i.e., similar to web programming practices of accessing an element according to an *id* attribute). However, app developers commonly create interface elements directly in code, obtain direct references to those elements, and therefore see no reason to specify a *ViewIDResourceName.* Further, *ViewIDResourceName* is not required to be unique, and the same value may be used by multiple elements in different contexts (e.g., elements in different screens of an app). It is therefore not an adequately available and robust identifier for annotating Android elements.

Android allows an accessibility service to capture a screen image if a person grants screenshot permission to the service. A person may refuse this permission. Apps can also specify a *FLAG_SECURE* to disable screen capture, a common practice in apps that contain sensitive information (e.g., banking apps). Prior research has examined pixel-based analysis and annotation (e.g., [21,24,109]), but the application of those techniques in mobile apps is limited by the potential lack of access to screenshots and concerns about mobile performance in pixel-based analysis.

710 ESPN app



This interface includes 6 elements with missing or misleading labels for use by a screen reader.

TalkBack allows end-users to add custom labels to only 2 of the elements (shown in green).

We develop new annotation methods that allow developers to repair all 6 elements.

Figure 5.1. Missing and misleading labels are a common and important accessibility issue that can be addressed by new approaches to robust annotation for accessibility repair.

Our approach focuses on using information available via standard Android accessibility APIs. Each interface element is represented as an *AccessibilityNode* that exposes properties of that element (e.g., *ClassName*, *AvailableActions, Text, ContentDescription*). Each *AccessibilityNode* can also access its parent and any children, allowing us to obtain and consider the tree of all interface elements in a screen.

As shown in Figure 5.1, Android's TalkBack screen reader lets users add custom labels to elements. When users navigate to an element without alternative text, they may perform a gesture to open the local context menu, then find the "add label" option, and finally enter a label for that element. Current support for interactive correction has several limitations: 1) People with visual impairments may have difficulty learning the correct label for an unlabeled element, which can require trial and error or seeking assistance from another person. 2) TalkBack's support for correction applies only to *ImageButton* or *ImageView* elements, which must have a `ViewIDResourceName` assigned by app developers. 3) TalkBack does not let people replace an existing label, even if it is incorrect or

misleading. Improved support could enable significant advances in these areas, such as a greater ability to annotate elements and a new ability to share robust annotations among many app users.

5.2 Approach and System Overview

This work aims to develop an approach to robust annotation of mobile app interface elements. Figure 5.2 explains our approach, currently implemented in a set of related tools. Later sections discuss details of our approach and our current tools in more detail, while also emphasizing that new tools can be composed to implement the overall approach.

Beginning on the left of Figure 5.2, a developer implementing accessibility repairs first captures screens they would like to annotate within an app. To do this, we developed a capture tool, which is implemented as an Android accessibility service. With the tool running in the background, the developer visits each screen that will be annotated. A software button added by the tool then allows capturing a current screen, including a screen image and a snapshot of the current *AccessibilityNode* hierarchy.

Collection will often include multiple captures of the same screen, as illustrated by color and shape in Figure 5.2. This can occur when a screen is visited multiple times during collection, or when a screen is captured with different content (e.g., the same Yelp rating screen captured for different restaurants). The developer of a repair identifies template screens that correspond to unique screens in the app. A template tool displays unique template screens in a row, with captured variations displayed in the column beneath each template. The tool applies our current screen

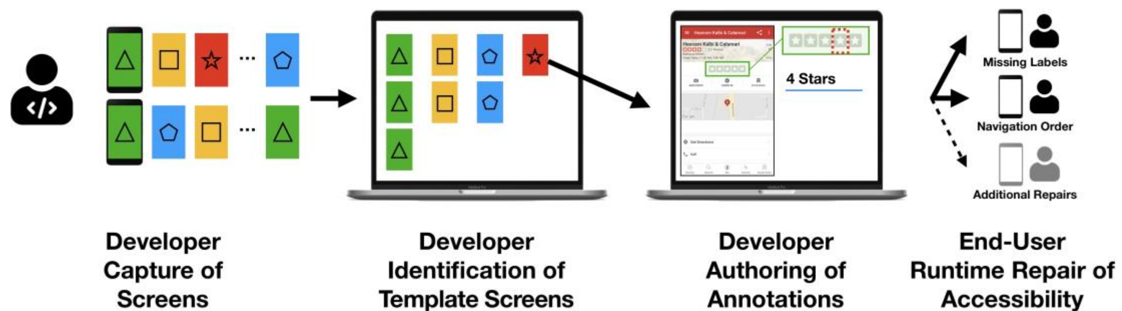


Figure 5.2. We develop and evaluate new methods for robust annotation of mobile app interface elements appropriate for runtime accessibility repair, together with end-to-end tool support for developers implementing accessibility repairs.

equivalence heuristics, as discussed and evaluated in later sections, to automatically identify templates. A developer therefore only needs to inspect and potentially correct identified templates.

A developer then authors annotations using a combination of (1) a screen identifier for the template screen to which an annotation applies, (2) an element identifier for the annotated element within that screen, and (3) the metadata to be associated with that element. We developed an annotation tool to support authoring of annotations. This tool displays the screen image, uses *AccessibilityNode* data to generate an element identifier when a developer selects an element in the image, provides highlighted feedback on elements that will be selected as a developer edits an element identifier, and allows inclusion of JSON-formatted metadata input in an annotation.

A developer can then create an accessibility service that uses annotations for the runtime repair of accessibility issues on end-user devices. We developed a runtime library that supports comparing the current screen of an app to template screens for that app. If the current screen matches a template, the library further supports testing element identifiers of annotations against the current screen. The accessibility service can then use matching annotations when applying its runtime repairs. Later sections describe three example services we implemented using this approach.

5.2.1 Supporting a Range of Accessibility Repair Scenarios

Our approach and tools support varied scenarios for developers implementing annotation-based accessibility repair. Two example scenarios include: 1) targeted repair for one or two screens of an app, or 2) more general-purpose repair for a class of error across many different apps.

In the first scenario, a developer might decide to modify the navigation order within a specific screen of a specific app (e.g., in the "file explorer" screen of the Dropbox app). The developer can open the app, visit the screen to be repaired, and capture its data. The developer can then inspect that data using our annotation tool, obtain a screen identifier for the "file explorer" screen, and obtain element identifiers for elements to be modified. The developer could then implement a custom accessibility service that: 1) uses our runtime library to detect when the app’s context

matches the "file explorer" screen identifier, 2) obtains references to interface elements in the screen using their element identifiers, and 3) uses the references to re-order navigation in that screen.

In the second scenario, the developer may want to extend their repairs to other apps in which users report the same type of accessibility issue. Instead of developing many such specialized repair services, the developer can generalize their repair services. They can remove the use of specific screen identifiers and element identifiers, instead defining annotation types and modifying their code to repair navigation according to any annotations available for current screens. This might be sufficient for their needs, they might extend our annotation tool to make it easier to author such annotations, or they might examine new approaches to supporting a community of people interested in annotating many apps.

5.3 Implementing Robust Annotation

Annotation is implemented using a combination of a screen identifier and an element identifier. A screen identifier corresponding to a *template screen* and a set of *screen equivalence heuristics* are used in both: 1) defining template screens (i.e., determining whether a screen is a *variation* of an existing template screen or distinct from existing templates), and 2) runtime identification of screens (i.e., determining whether the current screen matches a screen identifier). This section discusses each of these key components.

5.3.1 Screen Identifier

A screen identifier corresponds to a template screen and any variations of that screen, where a variation informally has the same screen structure with minor differences in content (e.g., images, text, number of items in a list). Annotations applied to a template also apply to any variations, which both minimizes effort that might otherwise be spent annotating many different versions of a screen and allows our approach to be used for screens containing dynamic content that could not otherwise be feasibly annotated.

A set of template screens is initialized with the first captured screen (i.e., a single template with no variations). Each screen is then compared to the set of current templates using *screen equivalence heuristics*. If a screen is equivalent to an existing template, it is added as a variation. Otherwise, the screen is used as a new template. Although a capture includes both a screen image and accessibility data, the image is used only for developer inspection and annotation. Screen equivalence heuristics must be based in the accessibility data because the image is not available at runtime. Because all variations are equivalent, a developer may also use any variation as the representative screen for a template.

At runtime, an accessibility service can capture accessibility data for a screen and use our runtime library to compare that screen to the set of template screens for that app. Doing so uses the same screen equivalence heuristics. If a match is found, annotations associated with that template screen are considered relevant to the app's current screen.

Our template tool generates a unique and random screen identifier for each template screen (e.g., "screen_1520907"). A developer may also associate a human-readable identifier with the screen identifier (e.g., "file explorer"), while ensuring human-readable names are unique within an app. Identifiers can then be used with our runtime library to detect a screen (e.g., for a repair to the "file explorer" screen).

5.3.2 Element Identifier

A developer can then reference elements in a template screen using techniques familiar in Android testing frameworks (i.e., *UiSelector* and *XPath* selectors). Our implementation of these selectors also differentiates between *stable* and *dynamic* properties. Stable properties are unlikely to change between screen content updates (i.e., between screen variations), including *ClassName, Depth, IsLeaf,* and *ViewIDResourceName.* Dynamic properties are more likely to change, including *ContentDescription*, *Location*, *NumberOfChildren, Size,* and *Text*. The current set of properties could be extended if necessary or if future versions of the Android Accessibility API expose additional properties of interface elements.

Our annotation tool also automatically generates a unique and random element identifier for each element in a template screen (e.g., "element_59401"). Each default element identifier corresponds to a selector that includes the element's path in the hierarchy and its stable properties. A developer can verify the default selector by using the annotation tool to inspect how it applies in each variation. If necessary, they can edit the default selector, again inspecting how it applies in each variation. They may also associate a human readable name with an element identifier (e.g., "menu button"). At runtime, an accessibility service can use our runtime library to obtain a reference to an interface element using either an element identifier or a supported selector.

5.3.3 Screen Equivalence Heuristics

Annotation requires screen equivalence heuristics for determining a set of template screens for annotation and determining whether the active screen of an app matches one of those templates. As previously noted, we rely only on information available via standard Android accessibility APIs so that our runtime library requires neither: 1) rooting end-user devices, nor 2) pixel-based analysis of screen images, which may be unavailable or present performance challenges in a mobile context.

Our heuristics are instead based on two key insights regarding identification of template screens. First, contexts where Android identifiers fail to correspond to a meaningful notion of a screen are not random (i.e., are not well described by ignoring any one `ViewIDResourceName` or by treating them as noise in a similarity metric). Instead, they are often systematic, resulting from developer behaviors (e.g., failing to provide an identifier, copy-pasting code resulting in non-unique identifiers) or standard toolkit behaviors (e.g., widgets that dramatically change what is presented in a screen with only subtle indications of that change in the accessibility API information for the screen). We develop a set of heuristics based on such systematic behaviors, and we evaluate our heuristics in a later section. We note that these heuristics can be updated and extended as we gather additional data or as toolkit behaviors evolve (e.g., by introducing new widgets that require adjustments).

Second, the two types of error in screen equivalence have different implications. We define a *FalseSame* error as incorrectly determining that two screens are the same. This can result in what

should be a distinct template screen instead being considered a variation of an existing template that requires developer correction. Alternatively, it can also result in a runtime screen matching an incorrect template and retrieving incorrect annotations. We define a *FalseDifferent* error as incorrectly determining that two screens are different. This can result in additional annotation overhead through the creation of spurious templates that could be combined. Alternatively, it can result in a screen not being annotated at runtime. Our techniques allow developers of an accessibility repair to correct either form of error, but we design our default screen equivalence heuristics to minimize *FalseSame* errors. This corresponds to preferring a need for greater annotation effort over the possibility of annotations being incorrectly applied at runtime.

Our current screen equivalence is implemented using eight heuristics, each based on a specific app developer practice or toolkit behavior. Heuristic 1 makes an early determination based on explicit app developer indication that screens differ. Heuristics 2 to 5 account for common interface structures that require special consideration, transforming the accessibility API representation to better support comparison. Heuristic 6 then *filters* items that should not be considered in comparison. Given these special-case checks and adjustments, Heuristic 7 then makes the primary comparison based on values of `ViewIDResourceName` in the two screens. Heuristic 8 then further reduces *FalseSame* errors by comparing values of *ClassName* on the two screens. After describing each heuristic, we discuss how a repair developer can correct any errors.

1. Compare *ActivityName*: If two screens both have an *ActivityName* value that was specified by the developer, but not the same value, the screens are considered different. This heuristic is intended to reduce *FalseSame* errors.
2. Check for Navigation Drawer: This common Android element presents a menu above an interface by dimming and preventing interaction with elements under the menu. When this heuristic detects an open navigation drawer, it transforms the representation of the interface so that remaining heuristics apply only to contents of the menu (i.e., by ignoring the occluded background elements). If one screen contains an open navigation drawer but

the other does not, the screens are considered different. This heuristic is intended to reduce *FalseDifferent* errors.

3. Check for a Floating Dialog: This common Android element also occludes elements underneath it. This heuristic similarly detects a floating dialog, transforms the representation so remaining heuristics apply only to contents of that dialog, and considers two screens to be different if only one contains a floating dialog. This heuristic is intended to reduce *FalseDifferent* errors.
4. Check for Tab Layout: Android's tab layout preloads the content of each tab, presenting the same tree to the Android accessibility APIs regardless of which tab is selected. When this heuristic detects a tab layout, it uses an active tab's binary *Selected* property to transform the representation so remaining heuristics apply according to that tab's content. It also considers two screens to be different if only one contains a tab layout. This heuristic is intended to reduce *FalseSame* errors.
5. Check for Radio Button Group with a Multi-Page View: This alternative approach to tab-like functionality similarly results in an Android accessibility API tree structure that does not adequately correspond to the selected radio button. It uses a binary *Checked* property of the active radio button to transform the representation so remaining heuristics apply according to content of the active view. This heuristic is intended to reduce *FalseSame* errors.
6. Visibility Filter: Common Android container elements expose elements in their accessibility API structure that are outside the bounds of the screen (e.g., *WebView*), so we transform the representation by filtering to include only visible elements (i.e., elements with *BoundsInScreen* values that correspond to non-zero areas on the screen). This heuristic is intended to reduce *FalseSame* errors.
7. Compare `ViewIDResourceName`: This stable property of each element will not change when an element's content is modified. If the set of `ViewIDResourceName` values is not the same, the screens are considered to be different. This heuristic is the primary comparison based on any transformations applied in the previous heuristics.

8. Compare *ClassName*: As with `ViewIDResourceName`, this stable property will not change when an element's content is modified. We consider this additional stable property to help address situations where `ViewIDResourceName` is not informative. If the set of *ClassName* values is not the same, the screens are considered different. This heuristic is intended to reduce *FalseSame* errors.

Our evaluation shows that these heuristics are highly effective, and they can be extended as additional data suggests new heuristics. However, any approach will sometimes require developers to correct a repair. For a *FalseSame* error, a developer can write an element selector that differentiates the two screens. Any future screens that match the original template will then be separated into two templates based on whether they match the selector. For a *FalseDifferent* error, a developer can combine the two template screens and their variations. Any future screens will then be considered equivalent if they match either of the original templates. Although we have not found it necessary, we note that multiple such corrections could be composed as needed.

5.3.4 Annotation Storage

The tasks of inspecting, editing, and using annotations require: 1) collections of template screens, each including a screen image, associated accessibility data, and a screen identifier used for referencing that template screen, 2) variations associated with each template screen, 3) element identifiers for each element in each template screen, and 4) annotations, defined as a combination of a screen identifier, an element identifier, and the annotation metadata to be associated with that element of that screen. Our current implementation stores this data in Google's Firebase.

5.4 Data Collection and Annotation Tools

Our core methods for screen identifiers, element identifiers, and screen equivalence can be applied in a variety of tools. We created an initial set of tools to support development of repairs based on these methods. This section introduces these tools and briefly describes potential alternatives.

5.4.1 Capture Tool

Implemented as an Android accessibility service, the capture tool runs in the background to allow a developer to capture screens. A developer browses to the screen they want to capture, then presses a software button on the navigation bar. The tool plays a confirmation sound, captures a screen image with associated accessibility data, and uploads them to the database. The capture tool therefore requires screenshot permission, but our runtime tools do not (i.e., captured images are used only to support annotation, and our runtime tools do not use pixel-level data). If a developer wants to capture an app that has disabled screenshot permission, they can use a rooted device or emulator [108]. Although a requirement to root a device is inappropriate for end-user accessibility tools, it is more appropriate for a developer and is the only way to circumvent FLAG_SECURE. Typical capture will include a developer navigating through an app, using the tool to capture different screens, interacting with the app, and capturing variations of screens.

5.4.2 Template Screen Tool

This web application supports a developer to inspect and potentially correct identified template screens in each app. Images of template screens are shown in the top row, with any variations displayed in a column below each template. Template screens and their variations are automatically and reliably identified using screen equivalence heuristics, so the tool is primarily used to inspect the results, obtain screen identifiers, make occasional corrections, and access the annotation tool by clicking on a screen. To make corrections, the tool supports authoring a selector or combining templates, as discussed in Screen Equivalence Heuristics section.

5.4.3 Annotation Tool

This web application supports a developer to author annotations on a template screen. It is currently accessed by clicking on a screen image in the template screen tool. The tool shows the screen image with its screen identifier and uses captured accessibility API data to highlight elements when a developer clicks on them. A developer can also author a custom selector and receive

feedback by highlighting one or more elements. For each highlighted element, its identifier and properties are shown in a list. An annotation can be authored as JSON-formatted metadata, or a developer can extend the annotation tool with custom functionality for a particular class of annotation.

5.4.4 Runtime Library

Our runtime library supports annotation-based accessibility services by providing key functions for obtaining accessibility data, identifying a screen by comparing it to a library of templates, identifying elements in a screen, and retrieving annotations. The library also supports listening for *ViewClicked* and *WindowStateChanged* events, which can lead to a change of screen structure that requires identification of the new screen. Our library therefore supports overall management of relevant annotations, letting a developer focus on the functionality of their accessibility repair service.

5.4.5 Alternative Collection and Annotation Tools

Our current tools support an end-to-end annotation process for developers, chosen as our first primary audience. We envision future research exploring complementary approaches. For example, an extension of our tools might support end-users to capture and annotate directly on their phones (e.g., requiring screenshot permission during capture, but allowing end-users to directly collect and annotate data for a repair). Future research might also examine how to scale annotation, perhaps drawing upon crowdsourcing and friendsourcing techniques developed in other contexts [88,97,98]. Our approach to screen equivalence could be included in tools for automated exploration of mobile apps [7,45,69], and such tools could also benefit the capture of data for accessibility repair.

5.5 Evaluation of Screen Equivalence Heuristics

To evaluate the effectiveness of our current screen equivalence heuristics, we recruited 5 developer participants to capture screens and identify templates in a dataset of real-world mobile apps. Our mobile app sample included 5 of the top free apps in each of 10 categories. 5 participants were

Table 5.1. Improvements in error rates resulting from adding each screen equivalence heuristic.

		Error Rate (%)	
	Heuristic	***FalseSame***	***FalseDiff***
-	Only *ViewIdResourceName*	3.10	2.28
1	*ActivityName*	2.51	2.28
2	Navigation Drawer	2.51	1.06
3	Floating Dialog	2.51	0.83
4	Tab Layout	1.55	0.83
5	Radio Button Group	1.48	0.83
6	Visibility Filter	0.44	0.83
7	*ViewIdResourceName*	as above	as above
8	*ClassName*	0.09	0.92

recruited from our computer science department, because our primary criterion was to recruit experienced developers familiar with mobile apps.

Each session began with simple training, showing participants how to capture a screen and how to use the template screen tool to examine identification of template screens in an app. We then asked each participant to capture screens for all of the major features in 10 apps, and, if possible, to capture one or more variations for each screen. After completing capture for each app, the participant was asked to use the template screen tool to examine the identification of template screens in the capture of that app and to correct any errors. Because our focus was on data collection, participants used a simplified version of the tool that allowed the dragging of screens to re-arrange them, without the need to identify a selector that could allow the templates to be used with our runtime tools. When a participant completed capture and identification of template screens for the 10 assigned apps, we asked them to examine template screens in another 10 apps captured by other participants. We therefore obtained 2 developer judgments regarding the template screens and variations within each app, and the lead researcher resolved the limited number of disagreements (a total of 12 disagreements in 9 apps). Participants were compensated with a $20 gift card. Data collection took about 5 to 10 minutes per app.

Participants collected a total of 2,038 screens from 50 apps. Following the same procedure used in [19], we examined equivalence in the 42,504 pairs of screens (i.e., from all screens in an app, consider each pair of screens). Table 1 summarizes the improvement associated with each heuristic. Because our primary heuristic compares values of `ViewIDResourceName`, we report the effectiveness of other heuristics in terms of improvement relative to this. Considering only `ViewIDResourceName` in our dataset resulted in a *FalseSame* error rate of 3.10% and a *FalseDifferent* error rate of 2.28%. Adding each heuristic reduced these, and the comparison of `ViewIDResourceName` following all previous heuristics resulted in a *FalseSame* error rate of 0.44% and a *FalseDifferent* error rate of 0.83%. Comparison of *ClassName* then further reduced the *FalseSame* error rate to 0.09% but slightly increased the *FalseDifferent* error rate to 0.92%.

Overall, we saw a 97% reduction in *FalseSame* errors and a 60% reduction in *FalseDifferent* errors, consistent with our goal of prioritizing the minimization of *FalseSame* errors. Remaining errors could be addressed by the developer of a repair (i.e., by specifying a selector or merging templates), a capability lacking in prior approaches to screen equivalence [19,45,67]. Error rates were well below the 6% *FalseSame* and 3% *FalseDifferent* error rates in [19], though we must take care when comparing these rates because those numbers were based on a different, much smaller dataset: 1044 pairs of screens from 12 apps that tuned the equivalence thresholds used in that work. Robust screen identifiers should also let developers author an element identifier for any element in a screen. In contrast, the TalkBack screen reader's reliance on `ViewIDResourceName` made it apply a custom label to only 13.6% of TalkBack-visited elements in this data.

Examining this data, we observe a practice of obfuscating `ViewIDResourceName`. For example, the Facebook Messenger app sets `ViewIDResourceName` to "name_removed" for all of its elements. Considering only `ViewIDResourceName` resulted in 84 *FalseSame* errors in this app, while our heuristics used *ActivityName*, interface structure, and *ClassName* to reduce this to only 2 *FalseSame* errors. These errors could then be corrected by developer specification of an appropriate selector. We also note that approaches based entirely on `ViewIDResourceName`,

including the TalkBack screen reader's support for adding custom labels, will be completely ineffective in such an app (i.e., because all elements have the same *ViewIDResourceName).*

Heuristic 8 reduced *FalseSame* errors by checking *ClassName*, but it slightly increased *FalseDifferent* errors. Examining this, we find that advertising banners were a common cause of increased *FalseDifferent* errors. For example, the Abs Workout app included advertising elements that had different *ClassNames* before and after an advertisement was loaded. This suggests that a future heuristic might filter advertising elements, perhaps by blacklisting their *ClassNames*.

We also observed a small number of cases that likely could not be resolved using our current techniques due to an app's complete failure to implement a meaningful representation via the accessibility APIs. For example, the TopBuzz app included a custom-implemented tab layout that did not expose any indication of what tab was active, unlike the *Selected* or *Checked* properties in our current heuristics. It also did not properly expose elements of all tabs to the accessibility APIs, but instead exposed contents of the first tab regardless of which tab was currently active. Resolving such a complete failure could require pixel-based techniques [21,23,24]. Although this will require screenshot permission at runtime, performance implications might be addressed by limiting pixel-based analysis to only such special-case scenarios where accessibility data fails.

5.6 Case Studies of Runtime Accessibility Repair

This section demonstrates the repair of three common types of accessibility issues, with all repairs implemented using our approach to robust annotation. These case studies are implemented using interaction proxy techniques and correspond to proof-of-concept demonstrations in Chapter 4. However, it was previously infeasible to scale demonstrations beyond a handful of elements in a handful of apps due to the lack of methods for determining where and how to apply runtime repairs. Integrating annotation-based techniques into these demonstrations is an important step toward runtime accessibility repair in mobile apps, which the next section further examines in a set of 26 real-world apps.

5.6.1 Missing or Misleading Labels

As illustrated in Figure 5.1, many apps contain both unlabeled elements (e.g., elements lacking a *ContentDescription* that will therefore be read as "unlabeled") and elements with misleading labels (e.g., Figure 5.1 has two buttons labeled "15").

We implemented an accessibility service that uses annotations to repair elements with missing or misleading values of *ContentDescription*. A developer uses the annotation tool to identify elements in need of label repair (e.g., by clicking it in the image, by writing a custom selector), then uses a text field to enter an appropriate *ContentDescription*, which the tool stores as an annotation. At runtime, the accessibility services detect whether the current screen includes any annotations and then use interaction proxy techniques to repair how annotated elements are read by the screen reader.

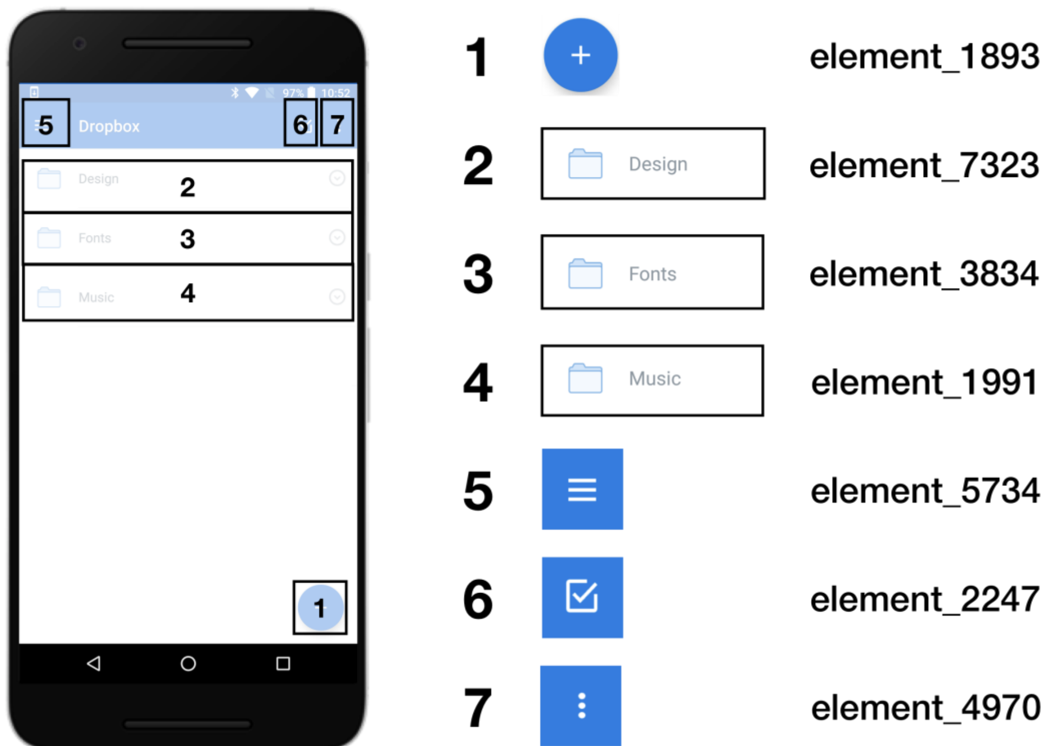


Figure 5.3. Illustration of annotation interface to correct navigation order. 7 elements are listed in the navigation order determined by Talkback.

5.6.2 Navigation Order Issues

The navigation order of interface elements is important to many people (e.g., a person using swipe gestures to navigate interface elements with a screen reader, a person using a switch interface). However, many apps have navigation orders that can make them difficult to use. For example, the navigation order for the Dropbox app begins with the "add" button and then requires navigating through all files in the current folder (i.e., a list of arbitrary length) before accessing the "menu", "select", or "more" buttons.

We implemented an accessibility service that uses annotations to repair navigation order on a screen. Figure 5.3 illustrates our annotation tool for this enhancement, an interface that shows the current navigation order and lets developers modify it by moving elements in a list. The resulting navigation order is stored as an annotation associated with the screen, which our accessibility service detects at runtime and uses to correct the navigation order.

5.6.3 Inaccessible Customized Widgets

When a developer creates a custom interface element, they also need to write additional code to expose appropriate accessibility hierarchy and context [39]. Unfortunately, many developers fail to do this, so these custom elements can be difficult or impossible to use with an accessibility service. For example, custom rating widgets found in Yelp and many other apps are often inaccessible (e.g., the Yelp rating widget is exposed as a *TextView* and does not allow a person using a screen reader or switch interface to enter a rating).

We implemented an accessibility service that uses annotations to repair some forms of inaccessible customized widgets. Figure 5.2 illustrates our enhancement of the annotation tool that supports rubberband selection to define clickable areas within an element, storing a list of these areas with a *ContentDescription* for each as an annotation on the element. At runtime, the accessibility service uses these annotations to create the missing accessibility API representations. This approach can repair only relatively simple custom elements, but it suggests an approach for more sophisticated repairs.

5.7 Evaluation of Runtime Repair

Our case studies build upon prior demonstrations of accessibility repairs that have received positive feedback, including feedback from two rounds of studies with 14 people with visual impairments who use screen readers [112]. The end-user experience with repair was the same as in this prior research, but prior demonstrations were limited to a handful of apps chosen by the research team and implemented using custom code for each repair. Our current evaluation therefore focused on examining the application of our selected categories of repair to accessibility issues in real-world apps. We first worked with a participant who used a screen reader, repairing accessibility issues in a set of apps he identified. We then collected and repaired issues in a larger set of apps.

5.7.1 Participant Feedback on Accessibility Repairs

To gather initial feedback on accessibility repairs implemented in our case studies, we interviewed a blind user (male, 38-year-old) with six years of experience with an Android screen reader. Via email prior to the interview, we described the three types of accessibility issues addressed in our case studies and asked if he found these issues in apps he frequently used. He replied to report issues in 6 apps. We then spent an hour capturing screens and authoring annotations to repair the accessibility issues he reported, followed by an additional hour examining the apps to find and repair issues he had not mentioned. We note the runtime repair of accessibility issues in 6 different apps would be infeasible in prior approaches requiring custom code to repair to specific elements in specific apps (e.g., prior repair demonstration in interaction proxies [112]).

During the interview, we first asked the participant to show how he normally used each app and how it was inaccessible. We then enabled our accessibility repair service and asked him to revisit the interactions he had showed us. After he experienced all repairs to the accessibility issues he reported, we disabled our repair service and guided him to screens with additional accessibility issues he had not mentioned. We then re-enabled the repair service, so he could experience the difference. After each app, we asked him: 1) to what extent the accessibility issues are a barrier; 2) if a repair service would change how he uses the app; 3) whether the repair service addressed

all accessibility issues he mentioned. At the end of the interview, we asked for his overall opinion and thoughts regarding our approach and its potential.

Overall, the participant expressed frustration with accessibility issues: "*These (accessibility) barriers make me not want to use them (apps). I'm a customer, just happened to be blind, but I'd like to use these services.*" He confirmed that our repairs addressed the issues reported, as well as additional issues in the same apps, and described how repairs would change app use and might help more people: "*Having the annotation available and making the app accessible make me more likely to use the app. I'd like to be able to use more stuff and do more. Enhancing (the apps) to be more usable and accessible…that makes it better for everybody.*"

One app he identified was *BECU*, a local credit union. The app is implemented with *cocos2dx*, a game engine that was probably chosen for its ability to provide high-quality animations. It unfortunately exposes very limited information to the Android accessibility APIs. On the login screen, TalkBack cannot move focus to the input fields for the account name or password. This app did include support for Android's voice assistant, which speaks a list of available options (e.g., "enter the password"), and then requires double tapping and speaking an option. The participant objected that this solution did not meet accessibility expectations: "*that's not what I want, and it is not the way it should be working…I should just be able to double tap on the username and type it.*" He also noted that speaking introduces privacy concerns: "*I often wear headphones and (keep) the screen off so that nobody could hear what's going on.*" We repaired the inaccessible login screen by defining both a clickable area and a description for each input field. We did not continue repairs beyond the login screen because we did not have credentials to use during capture and did not want to ask the participant to expose his personal information in testing.

The participant also identified the *At Bat* app, which features listening to live streams of baseball games. However, after paying for a subscription, the participant could not access the streams. The "play" button is unlabeled, and a feed source must be selected to enable the unlabeled "play" button. Without instruction, this interaction is extremely difficult for a person using a screen reader. The participant was frustrated by the player: "*it's a big barrier that I am not able to really use that*

app; it makes me frustrated, and I don't understand why they are unlabeled…I don't want to open some random buttons." We annotated the unlabeled buttons with appropriate labels, repaired the navigation order to more easily move to the audio player, and added an instruction to select a feed. The participant described how these repairs would make the app useful: "*I would actually use it, and I paid for it…Right now, I'm not using it at all.*"

5.7.2 Repairs in Additional Mobile Apps

As a complement to our in-depth exploration with this participant, we repaired 20 additional apps. 10 were identified as having accessibility issues by participants in accessibility-related forums [40,53,92], and 10 were selected from the top downloaded apps in the Google Play Store.

Table 5.2 summarizes the accessibility issues we repaired in a total of 26 apps. Across 24 apps, we found and repaired a total of 115 missing labels and 46 misleading labels. Across 18 apps, we found and repaired 29 navigation order issues. For 11 apps, we found and repaired 12 inaccessible custom widgets. We include repair examples at: https://youtu.be/oLMjy2IwYbY.

Because runtime repair of mobile accessibility issues is a relatively new capability and prior methods required custom code specific to each repair [82,112], we believe this is the largest existing set of runtime repairs of mobile app accessibility issues, thereby providing support for the potential of annotation-based accessibility repair.

Table 5.2. The number of accessibility issues repaired in each of the 26 apps used in our evaluation of runtime repair.

App			# of Issues Repaired	
	Missing Label	Misleading Label	Navigation	Widget
Identified by Participant				
710 ESPN	5	3	3	0
Amazon Music	11	4	2	0
At Bat	4	2	0	1
BECU	0	0	0	1
Yelp	3	1	2	2
Youtube Music	0	2	1	0
Identified in Accessibility Forums				
Astro	1	2	1	0
BBC iPlayer	0	1	1	1
BBC Media Player	0	0	0	1
Checkout 51	8	1	1	0
Dropbox	9	2	2	0
EMusic	9	3	2	0
Etsy	4	1	0	1
Postmate	11	0	0	0
Timely	10	1	3	1
WD My Cloud	3	1	1	1
Top Downloaded Apps				
Fitbit	4	2	2	0
Flipboard	6	1	0	1
GroupOn	3	3	1	0
NBC News	2	5	0	0
Paypal	2	4	0	0
Sephora	3	2	2	1
Starbucks	3	2	1	0
Tinder	9	1	2	1
The Weather Channel	4	1	1	0
Youtube	1	1	1	0

5.8 Current Limitations

Our evaluation found that a relatively small number of apps expose an accessibility API representation that fundamentally lacks vital information (e.g., screens in the TopBuzz and BECU apps). Our current screen equivalence heuristics cannot be effective in such circumstances. Careful authoring of selectors based on available information might let motivated developers differentiate screens and author repairs, but other approaches may also be beneficial. For example, we have generally avoided pixel-based analysis, but we might make limited use of such techniques in situations like these, which cannot otherwise be addressed.

We currently examine capture and annotation of an app within a single version of that app running on a single phone (i.e., at a single screen resolution and in portrait orientation). We are not aware of any prior research in screen equivalence that has addressed this limitation, but future research toward large-scale deployment of annotation-based repair will need to consider different versions and renderings of the same app. Our approaches should be promising because they do not rely upon element location or size, and large-scale changes can likely be modeled as additional template screens. Scrolling, animation, and dynamic introduction of new elements are also classic difficulties in runtime interpretation and modification. Our runtime tools currently address this by identifying a screen when it first appears, then monitoring events that might indicate a change in the active screen. This has been effective, but additional approaches may be necessary.

Our current implementation is for Android. Although it is not the most popular mobile platform among screen reader users, its open platform both enables our annotation techniques and allows direct deployment of advances in accessibility services. Our overall strategy (i.e., identifying components and patterns that lead to screen equivalence errors) is likely to generalize to additional mobile platforms.

5.9 Conclusion

This chapter has introduced an approach to robust annotation of mobile apps, using techniques appropriate for runtime accessibility repair. We presented our underlying methods in terms of screen identifiers, element identifiers, and screen equivalence heuristics. We developed an initial set of tools based on these methods, focused on developer implementation of accessibility repair services. We then evaluated our screen equivalence heuristics, presented our case studies applying annotation in runtime accessibility repair, and examined these case study implementations in repairing real-world accessibility issues. Supporting materials (e.g., code and screen data) are available at: https://github.com/appaccess.

We demonstrated an initial set of annotation tools, but there are many more possibilities. For example, our approach might be integrated directly into Android's core accessibility services, the TalkBack screen reader and Switch Access. Annotation could address limitations of these tools in relying upon `ViewIDResourceName`. Future research could also explore tools that do not require developer-level expertise, including crowdsourcing or friendsourcing approaches. Robust approaches to mobile app screen equivalence and annotation can also have applications beyond accessibility, including interface testing, large-scale collection and analysis of mobile apps [19,45,85], and task automation [65]. Overall, we believe many new tools can be developed using the methods developed in this initial research.

This work was published at UIST 2018 as ***Robust Annotation of Mobile Application Interfaces in Methods for Accessibility Repair and Enhancement*** [113] with co-authors Anne Spencer Ross and James Fogarty. I built the system, implemented the runtime repairs in case studies, and conducted the evaluations. All co-authors provided valuable opinions regarding the scope of this project, evaluations, and paper writing. Please watch the demo video at: https://youtu.be/oLMjy2IwYbY.

Chapter 6 INTERACTILES: A SET OF 3D-PRINTED TACTILE INTERFACES TO ENHANCE MOBILE TOUCHSCREEN ACCESSIBILITY

The absence of tactile cues such as keys and buttons makes touchscreens difficult to navigate for people with visual impairments. Tactile feedback and tangible interaction can improve accessibility [8,43,59], but previous approaches have either required relatively expensive customization [14], had limited compatibility across multiple apps [47,70,100], consisted of many hardware components that were difficult to organize or carry [8], or have not focused on mobile systems [8,59]. Compared to larger touchscreens, mobile touchscreens pose additional challenges, including smaller screen targets and the necessity of using the device on-the-go. Static physical overlays on mobile devices include raised areas and/or cutouts to provide tactile feedback and guidance. However, prior research has limited each such overlay to a specific screen in a single app [47,70,100].

There is an untapped opportunity to add tactile feedback on mobile touchscreens for multiple apps. Therefore, we create Interactiles, ***an inexpensive, unpowered, general-purpose system that improves tactile interaction on touchscreen smartphones***. It consists of two major parts: (1) a set of 3D-printed hardware interfaces that provide tactile feedback, and (2) software that understands the context of a currently running app and redirects the hardware input to manipulate the app's interface. The hardware material costs less than $10. Without any circuitry, our system does not consume power and therefore preserves phone battery life. Our validation shows that its functionality is compatible with most of the top 50 Android apps. In our usability study, Interactiles showed promise in improving task completion time and increasing participant satisfaction for certain interactions. Study results support the value of a hybrid software-hardware approach for providing tangibility while maximizing compatibility across common apps.

The specific contributions include:

- Designing and building Interactiles, an inexpensive, unpowered, general-purpose software-hardware system that enhances tactile interaction on mobile touchscreen devices.
- Validating the compatibility of Interactiles with 50 Android apps from Google Play Store, which makes it the first tactile enhancement system on mobile platform that is compatible with many existing apps.
- Conducting three user studies. We first interviewed two people using screen reader to learn their preferred improvements. We then pilot tested our prototype with people with visual impairments and received feedback for improvement. Finally, we conducted a usability study that compared Interactiles with Android TalkBack (built-in screen reader). This study collected qualitative reactions to Interactiles and compared task completion times and participant preferences.

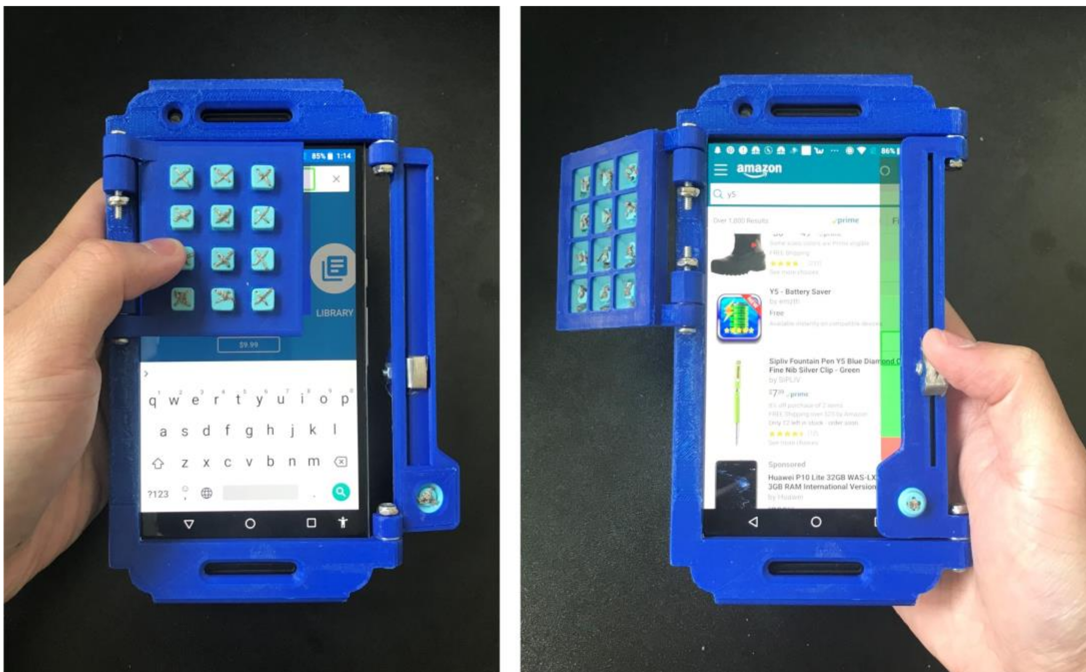


Figure 6.1. Interactiles allows people with visual impairments to interact with mobile touchscreen phones using physical attachments, including a number pad (left) and a multi-purpose physical scrollbar (right).

6.1 Use Case Scenario

The Interactiles system is designed to work with mobile screen readers while providing tactile feedback. The hardware base is a 3D-printed plastic shell that snaps on a phone, which contains three hardware components: a number pad, a scrollbar, and a control button at the bottom of the scrollbar, as seen in Figure 6.1. Users may flip in and out hardware components for different tasks. When all hardware components are flipped out and the software is off, a person has full access to the touchscreen as in normal usage. These hardware components work with our software to perform five main functions: element navigation, bookmarking, page scrolling, app switching, and number entry.

Consider a use case scenario when Jane, a person with visual impairments, purchases a backpack on Amazon with Interactiles and screen reader (demonstrated in https://youtu.be/Fd6zmfJ9DOk).

> Jane turns on the software and flips in the scrollbar. When the software is active, a floating window appears on screen to receive touch events on the scrollbar. She browses the Amazon mobile app to buy a backpack. Starting with the element navigation mode, she moves the scroll thumb to hear information about each product on a page. When she navigates through all products on the current page, the scroll thumb arrives at the bottom of the scrollbar. She then long presses the scroll thumb to move to the next page.
>
> While browsing, Jane finds a backpack with good reviews and a reasonable price. She would like to browse through all the products before purchasing, but she also wants to return to this backpack later. Thus, she long presses the control button to bookmark this backpack.
>
> After reviewing a long list of backpacks, she likes the bookmarked one best. She presses the control button once to activate the page scrolling mode. She slides up the scroll thumb to scroll up pages. Her phone vibrates to announce when it has scrolled to the bookmarked page.
>
> While Jane is shopping, she notices vibration from a new message notification. She presses the control button to activate app switching mode and hears "Mode: App" as confirmation. She

moves the scroll thumb to hear the recently opened apps. When she hears "Messages," she double-taps the scroll thumb to open the Messages app.

Jane presses the control button again to switch back to element navigation mode. Moving the scroll thumb, she hears the subject line for each message (e.g., "What’s the address for coffee on Monday?") and then navigates to the reply text field. She double-taps the scroll thumb to activate the text field for input. When the phone’s soft keyboard pops up, a floating window for the number pad appears on the screen. She flips in the physical number pad for number entry. She enters numbers using the number pad and enters letters using the phone’s soft keyboard, as she normally would.

6.2 Design and Implementation of Interactiles

Interactiles is an inexpensive, unpowered, general-purpose system that increases tactile interaction on touchscreen phones. Its interactive hardware provides tactile feedback, and its software receives touch input on hardware and invokes the corresponding action on the currently running app. The system is designed to work with built-in screen readers and mobile apps without modification to them. Currently, our software supports devices running Android 5.0 and above (88.9% of Android devices as of Dec 2018 [37]).

Before designing Interactiles, we spoke with two people who use screen reader to learn what improvements they wanted to experience in mobile touchscreen interactions. Their feedback motivated us to design the selected functions. They expressed disappointment at the disappearance of physical phone buttons. They also cited interactions and situations where they had particular difficulty, such as losing their place while using explore-by-touch on a moving bus. This motivated us to design the element navigation feature because the scroll thumb keeps a person's place on the screen. Neither of those interviewed liked entering numbers, especially in a time-sensitive situation. They expressed enthusiasm for physical buttons that could help them avoid this difficulty.

6.2.1 Design Goals

Taking into account the challenges highlighted in previous tangible accessibility research and our own interview input, we identified the following design goals.

Mobile focus: Prior research [8,59] shows the positive impact of tangible interaction, but most such work has been designed for large touchscreens. The prevalence of mobile devices and apps support our targeting of mobile touchscreens. Further, these devices present a different set of challenges than larger interfaces.

Portable, contained, and non-blocking: Because our design target was mobile devices, our solution had to be sufficiently portable to carry on a daily basis. All hardware components had to be contained in one piece so that a person did not have to carry and assemble additional pieces. Furthermore, the system had to work with built-in screen readers and not block normal touchscreen interaction when not in use.

Compatible with various mobile apps: In previous research [47,70], a tactile overlay is limited to one screen in a single app. Carrying a piece of hardware for each screen or app is not practical on a daily basis. We wanted to design common functionalities that are useful in different mobile apps.

Low-cost: Several commercial products provide tactile feedback (e.g., Braille phone [79]). However, their cost could pose a problem for people with disabilities, a population more likely to face socio-economic barriers [2]. Therefore, we wanted to design a less expensive solution to improve deployability.

Unpowered: Many rich tactile interactions have been enabled through electrical [9] and electrical-mechanical experiences [8]. However, these systems increase the cost of assistive technologies and decrease portability, which is important for mobile platforms. Therefore, we wanted to design an unpowered solution.

6.2.2 Hardware Components

The unpowered and portable Interactiles hardware provides tactile feedback. Inspired by prior research on capacitive touch hardware components [17], our hardware leverages conductive material to connect a person's finger to an on-screen contact point, thus registering a touch event.

Our prototype cost less than $10 in materials. Most parts of the hardware were 3D-printed using inexpensive PLA plastic. Some parts were handmade from silicone, conductive fabric, conductive thread, and standard fasteners (e.g., nuts, bolts, and washers). Printing took about 10 hours, and assembly required about 2 hours. Because it is a one-time print, these times are unlikely to be prohibitive. In addition, a person may receive printing and other help from volunteer communities. For example, e-NABLE volunteers successfully adapted and assembled assistive devices (e.g., prosthetic limbs) for people with disabilities.

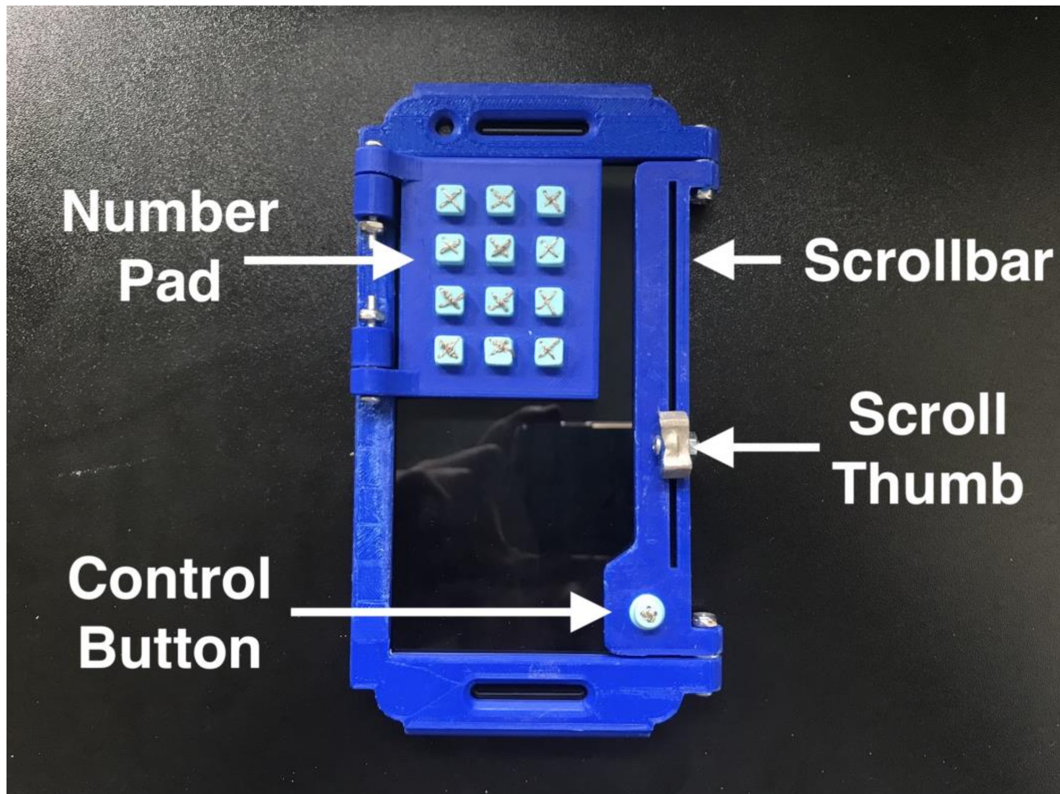


Figure 6.2. The Interactiles hardware base is a 3D-printed plastic shell that snaps on a phone, and three hardware components (Scrollbar, Control Button, and Number Pad) are attached to the shell.

As shown in Figure 6.2, the hardware base is a 3D-printed plastic shell that snaps on a phone, with three hardware components attached to the shell.

The **Scrollbar** is attached to the right side of the phone shell using a hinge. It can be flipped out from the screen when not in use. It consists of a 3D-printed PLA frame and a scroll thumb. The scroll thumb is a piece of metal bent at a right angle, encased in PLA, and covered with conductive fabric. We chose fabric because it transmits capacitive touch reliably, feels comfortable, and does not scratch the touchscreen.

The **Control Button** is attached to the bottom of the scrollbar. It is a silicone button sewn through with conductive thread to transmit touch. The button was cast using a 3D-printed mold to control its size and shape. Silicone material was chosen to ensure that the button was comfortable to press. Because there is currently no castable and affordable conductive material, we sewed the button on with conductive thread.

The **Number Pad** is attached to the left side of the phone shell. It is a 3D-printed PLA plate that contains a 4x3 grid of silicone buttons. Like the scrollbar, the number pad is hinged and can be flipped out when not in use.

6.2.2 Software Proxies

The Interactiles software is an accessibility service that runs in an app-independent fashion. To increase deployability and generalizability, our software was implemented with standard Android accessibility APIs. It does not require rooting a phone or access to app source code and is compatible with Android TalkBack. Our implementation approach was inspired by interaction proxies [112], a strategy for modifying user input and output on the phone, useful for accessibility repairs such as adding alternative text or modifying navigation order.

Interaction proxies consist of one or more floating windows, which are visible on top of any currently running app. As these proxies sit above the app in z-order, they can modify both the output and input of the underlying app. Output is modified by drawing inside the floating windows. Input is modified using event listeners, which intercept all touch events on the floating window

using the Android accessibility API and consume them before they reach the underlying app. Interaction proxies can further leverage accessibility APIs for content introspection of the underlying app as well as for automation of actions.

Using these abstractions, interaction proxies were used to implement proof-of-concept accessibility repairs that are compatible with various mobile apps.

The Interactiles service captures touch events generated when a person touches the conductive portion of a hardware component, interprets them, and takes appropriate action. It captures events using floating windows with attached event listeners and delivers events using a combination of content introspection and automation.

Event listeners monitor AccessibilityEvents [41] generated in the currently running app and floating windows to perform actions. An interaction with TalkBack generates events that differ from standard touch events, so we cannot use the standard gesture listener. We implemented a custom listener to recognize the following events that occur in the floating windows: tap, double tap, slide (press and move scroll thumb), and long press with different durations.

The floating windows in Figure 6.3 sit beneath the relevant piece of hardware and thus do not interfere with interactions on uncovered screen areas. The presence/absence of each floating

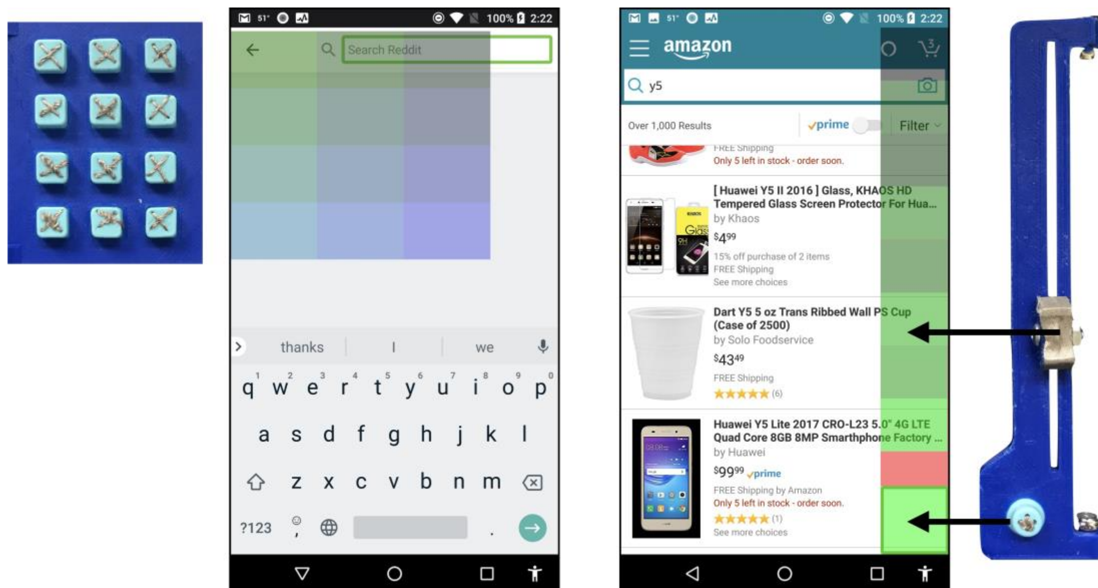


Figure 6.3. Floating windows created for number pad (left), scrollbar (right) and control button (right bottom). The windows can be transparent; we use colors for demonstration.

window and its content are determined by the current Interactiles mode, along with the state of the current app or apps that are running. In particular, our software assumes the scrollbar is always active until the phone keyboard pops up for text entry. Whenever the keyboard is active, the number pad floating window appears, and the scrollbar floating window is temporarily removed to avoid blocking the keyboard. A person flips in the physical number pad and flips out the scrollbar, accordingly. When the keyboard is deactivated, the number pad floating window disappears, and the scrollbar reappears. When all hardware components are flipped out and the software is off, the person has full access to the touchscreen as normal.

The scrollbar floating window is implemented using a dynamic list of buttons representing the items to be scrolled through. When the scroll thumb slides to a button, TalkBack announces the alternative text of the button. The scrollbar has three modes: App Switching, Element Navigation, and Page Scrolling. A short press on the control button at the bottom of the scrollbar switches modes, announces the new mode, and updates the buttons in the floating window under the scrollbar. The number and size of buttons are dynamically determined by the number of elements on the current screen for Element Navigation mode and are constant in the other two modes. The scrollbar floating window also reserves an active button for the control button. For Number Entry, the number pad floating window is implemented using a static 4x3 grid of buttons, which becomes active when a text box is in focus.

App Switching: People with visual impairments typically switch between apps using a physical button or a software button at the bottom of the screen, and then navigate through a list of opened apps. However, locating this button and switching through apps can be difficult. For example, TalkBack requires three swipes to navigate to the next app (i.e., announce app name, open app info, close app). In our App Switching mode, a person simply slides the scroll thumb to hear the name of each opened app (represented as a button in the floating window). Double-tapping the scroll thumb opens the app. To minimize required precision, Interactiles defaults to switching between the four most recently opened apps, a number that can be customized as desired.

Element Navigation: People using screen readers can swipe left/right to navigate between all interface elements on the screen in a linear order. They can swipe quickly without listening to the

full text of each item (e.g., for a long list). Alternatively, they can use explore-by-touch, where they move one finger around the screen to hear about on-screen elements. This is a faster way to skip content in a long list, but it risks missing elements with small or hard-to-reach targets. Interactiles uses the physical scrollbar to navigate through app interface elements; each element creates a button with its alternative text in the floating window. Users slide the scroll thumb to move the focus between elements and hear the content of the focused element. A quick slide can skip elements. Double-tapping the scroll thumb clicks on the focused element. When the scroll thumb arrives at the top/bottom of the scrollbar, users can long press the scroll thumb to move to the previous/next page.

Page Scrolling: People using screen readers may scroll pages instead of elements with a two-finger or three-finger swipe on the screen. They can also assign a two-stroke gesture to scroll pages with one finger. TalkBack announces page location after scrolling (e.g., "showing items 11 to 20 of 54"). Interactiles supports a similar function with tactile feedback on the physical scrollbar. If the current app screen is scrollable, users can slide up/down the scroll thumb to scroll to the previous/next page and hear the page number. When the scroll thumb arrives at the top/bottom of the scrollbar, users can quickly slide down/up the scroll thumb without scrolling screen. Thus, users can keep scrolling in an infinite list.

Bookmarking: It can be challenging for people using TalkBack to relocate an interface element in an app. Interactiles uses the physical control button to bookmark an interface element in an app. In Element Navigation mode, after users navigate to an element of interest, they can long press the control button and hear "Bookmarked" with the alternative text of that element. Later, when users move to a screen that contains the bookmark, the phone vibrates and announces "Bookmark found" with the alternative text of the bookmarked element. Currently, our system allows one bookmark per app.

Number Entry: People using TalkBack enter numbers by switching to the symbol mode of the soft keyboard. However, locating the symbol keyboard and then a specific number on it can be challenging. In addition, typing a combination of letters and numbers requires frequent keyboard mode switching. Interactiles uses the physical number pad to enter numbers; as seen in Figures 6.1

(left) and 6.3 (left), users can type letters on the soft keyboard at the same time. The number pad floating window contains a 4x3 grid of buttons, matching the position of the physical buttons. The number pad uses a layout similar to a phone keypad: the first three rows have numbers 1 to 9, and the bottom row has "read the entered text," number 0, and "backspace." When a button is pressed, our software updates the content of the active text field by appending a number or removing the last character.

6.2.3 System Improvements Based on Pilot Study Input

After we developed a fully working prototype of the software and hardware, we conducted a pilot study with a blind participant (a 38-year-old male who has used TalkBack for 7 years) and iterated based on his feedback. Originally, to go to the next page of elements when in element navigation mode, the participant had to switch to page mode, scroll down a page, and switch back to element mode. Based on our pilot study, we changed the system to allow a "next page" action while in element mode. We also added more verbal feedback to help the participant. The final system announces the page number while page scrolling, the bookmark content when found, and the entered characters in a text box. In the final system, pages are not directly mapped to scrollbar locations; instead, the participant goes to the next page when at the bottom of the scrollbar by moving the scroll thumb quickly up (faster than 10 cm/s) to get more room, and then down slowly again.

6.2.3 System Validation in Android Apps

To test robustness, we validated the compatibility of Interactiles with 50 Android apps from the Google Play Store. Our samples were the top 5 free apps in each of 10 categories: Book,

Table 6.1. The number of apps in which Interactiles features worked as expected, out of 50 sample apps. One app did not include any text field to test number entry.

Page Scrolling	Element Navigation	Bookmark	Number Entry	App Switching
49/50	42/50	49/50	49/49	50/50

Communication, Entertainment, Fitness, Navigation, Medical, News, Productivity, Shopping, and Social. For each app, we tested page scrolling, element navigation, and bookmarking on a screen that allowed scrolling. We tested number entry on a screen that allowed text entry. When possible, we tested the main screens of apps (e.g., the News Feed screen in the Facebook app). The results are summarized in Table 6.1.

Page scrolling worked as expected in all but one app. In the main screen of Google Play Book, our system did not scroll the page vertically; instead, it scrolled the large horizontal banner at the top. As future work, we may allow developers to annotate which element should be scrolled in specific apps or may allow scrolling in multiple scrollable elements.

Element navigation did not work as expected in 8 apps. In 3 apps, there are more than 30 interface elements on a screen so that each corresponding button on the floating window was too small for the scroll thumb to individually select. In 5 apps, our software ignored important interface elements (e.g., bottom tab buttons in Reddit). These apps did not provide alternative text on those interface elements, and therefore our software ignored them. As future work, we may also let developers annotate the elements to ignore or include. We can also design new hardware to support infinite scrolling (e.g., a scroll wheel) so that each element will have a larger, constant-size selection area.

Bookmarks could be created in all apps and found in all but one app. To display a list of products, the Amazon app used a WebView, which exposed out-of-screen content to accessibility APIs. Thus, our system incorrectly found the bookmark even if it was not visible on the current screen. We solved this by checking element visibility. As future work, we may examine other interface components that expose out-of-screen content.

Number entry worked as expected in all apps except Rosa, which lacked any text field to test. The number pad floating window appeared when the keyboard popped up and the pressed number was correctly updated in the active text field. The backspace and announcement functions worked as well.

App switching worked in all apps since this feature did not rely on specific app screen content.

6.3 Usability Study

To complement the feedback that informed our design of Interactiles, we conducted a study comparing Interactiles with TalkBack (the built-in Android screen reader). This study collected qualitative reactions to Interactiles and compared task completion times and participant preferences.

6.3.1 Participants

We recruited participants (N = 5) through word of mouth, university services, and mailing lists of organizations of blind people. 3 participants were blind, and 2 had some level of impaired vision. All participants used mobile screen readers, primarily iOS VoiceOver. Table 6.2 shows their background.

6.3.2 Comparative Study Method

We conducted a comparative study in a university usability lab and in public libraries. We employed a within-subjects design to examine task completion time, accuracy, and participant preference between two interaction methods: a Nexus 6P phone with TalkBack (the control condition) and the same phone with TalkBack and Interactiles (the experimental condition). Participants had the option of setting up the phone with their preferred TalkBack volume and speed settings.

Table 6.2. Information on study participants, all of whom were VoiceOver users. Proficiency was self-rated as basic, intermediate, or advanced.

ID	Age	Gender	Vision	Screen Reader Proficiency
P1	24	Male	Blind	Intermediate
P2	58	Female	Blind (L) Low Vision (R)	Basic
P3	29	Male	Blind	Advanced
P4	31	Female	Low Vision	Advanced
P5	43	Female	Blind	Intermediate

Participants were asked to complete four tasks that isolated specific functionality, followed by a more open-ended task to explore the system as a whole. At the beginning of each task, we explained the corresponding Interactiles feature(s) and gave the participant a training task to gain familiarity with Interactiles. Participants were also given time to do the training task with TalkBack.

For each participant and each task, we randomized the ordering of conditions to counterbalance order effects. Participant feedback was audio recorded by the researchers. After each task, the participant was asked to give qualitative feedback and answer Likert scale questions about the speed, ease, intuitiveness, and confidence while using Interactiles and TalkBack. Upon task completion, the participant was asked to provide general feedback about Interactiles and TalkBack and their preference.

6.3.3 Tasks

The specific tasks in the usability study were designed to test each Interactiles feature. Tasks were chosen by considering the difficulties faced in using common apps and how Interactiles might be used in such situations. These tasks covered target acquisition (locate), browsing speed (locate, relocate, app switch), data entry (mixed text/number entry), and spatial memory (relocate).

Locate: Participants were asked to find a specific song in a Spotify playlist. In Interactiles, they were encouraged to use element navigation to search through the current page of songs. They could slide to navigate elements and long press at the bottom of the scrollbar to move to the next page. When participants found the song in the Interactiles condition, we encouraged them to bookmark it for the relocate task described next.

Relocate: After the participant located the song in the locate task, the playlist was scrolled back to the top, and participants were asked to find the song again. In the Interactiles condition, we encouraged participants to use page scrolling to find the bookmark from the locate task. Because they were novices with respect to Interactiles, we explicitly encouraged them to use Interactiles' functionality.

App switch: With four apps open, participants were asked to switch from one of the apps to another. This was repeated four times in total.

Mixed text/number entry: Participants were asked to compose a message in the default Messages app, which required entering both numbers and text. The message dictated to participants was: "My number is [phone number]. Meet me at [address]."

Holistic: This task consisted of three steps. Participants were first asked to find a specific product by name (a water bottle using Talkback and a backpack using Interactiles) on pre-curated Amazon shopping lists without using the search bar. When they found the product, they were asked to switch to Messages to enter a contact phone number and a shipping address. Finally, participants were asked to switch back to Amazon and add the product to their shopping cart. This task was designed to encourage participant use of all Interactiles features.

6.3.4 Data Collection and Analysis

Our data included times for each task, accuracy for each task, task-specific Likert scales, general Likert scales, and qualitative feedback. Time data was analyzed to compare participant performance using Interactiles against the control condition; however, the data could not be directly compared across participants because each participant used TalkBack at a different speed.

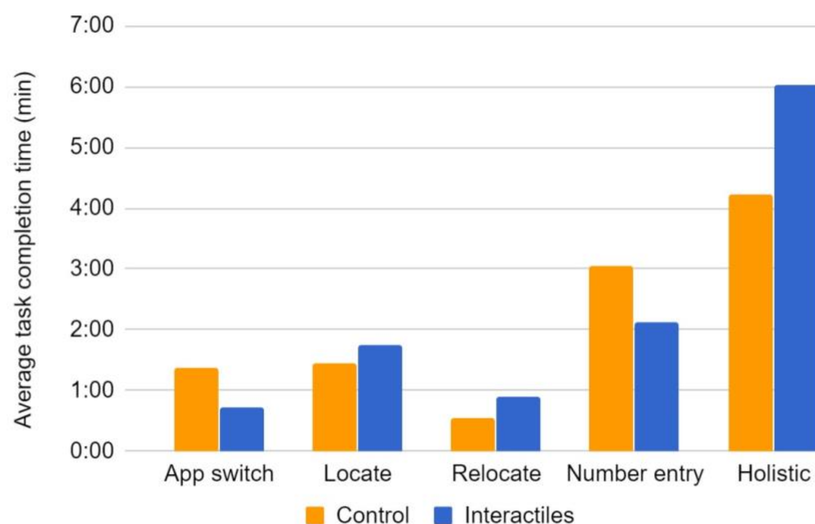


Figure 6.4. Average task completion time for each task in the study. P4 did not complete app switching on the control condition, and P5 did not complete the holistic task.

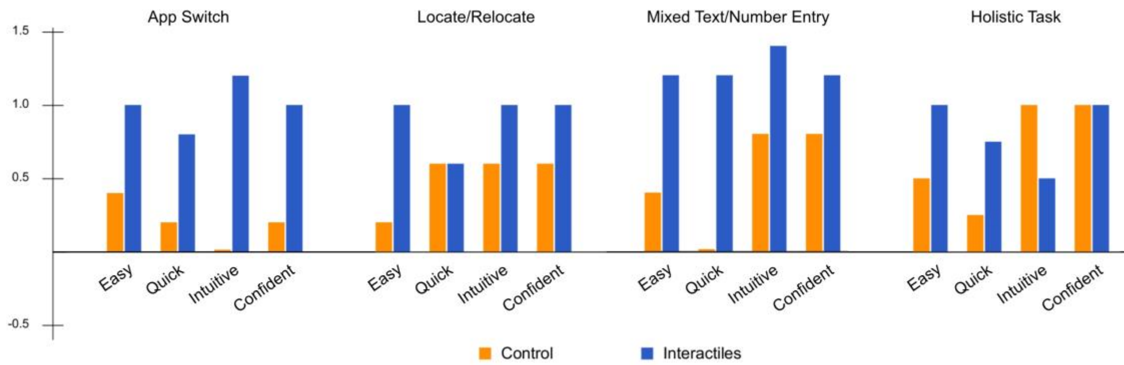


Figure 6.5. The average Likert scale rating (strongly disagree = -2, strongly agree = 2) given by participants for the study tasks. Participants were asked how easy, quick, intuitive, and how confident they felt completing each task with the control condition (only TalkBack) and Interactiles.

We use bar charts to show trends in the Likert scale data. However, our sample was too small to calculate statistics for either timing or Likert scale data. The qualitative data was organized into themes by one researcher and discussed as a group until agreement was reached on what was learned.

6.3.5 Results

In terms of speed, Interactiles improved performance times for the app switching and number entry tasks, as shown in Figure 6.4. Participants were uniformly positive about the number pad but were mixed on the usefulness of the scrollbar and control button even though the scrollbar resulted in a better task completion time for the previously mentioned task. Average Likert scale ratings for each condition and task can be seen in Figure 6.5.

App switching: As seen in Figure 6.4, participants performed app switching with Interactiles almost twice as quickly as they did with just TalkBack. They also had a positive response to this task. *"Compared to hunting down the overview button every time, it was relatively quick and painless, especially when I figured out how to operate the slider with the thumb."*

Locating/Relocating: The scrollbar had slower task completion times for 3 out of 5 participants than Talkback for the locating task, in which participants used element navigation. For relocating, 3 out of 5 participants had faster task completion times using Interactiles. Figure 6.6 shows individual task times. Qualitative feedback from participants confirmed that the scrollbar did not provide a performance advantage over the default Talkback element navigation methods (i.e., swipe

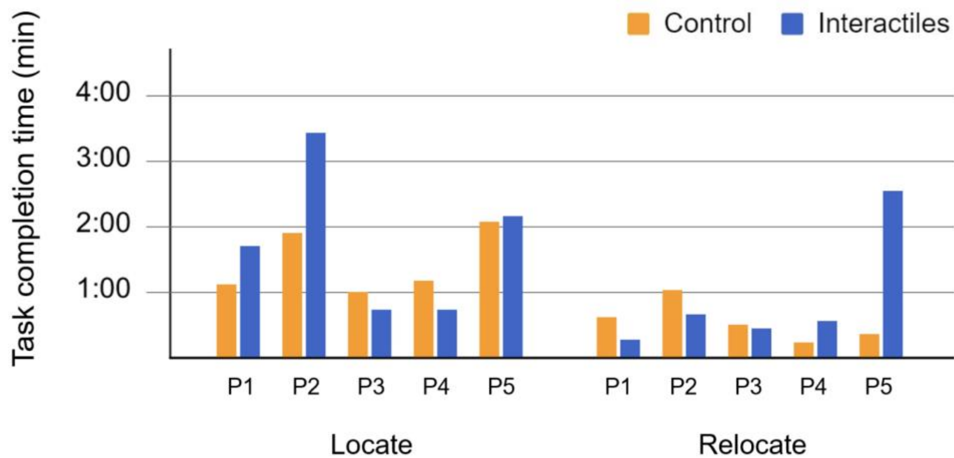


Figure 6.6. Individual task completion time of locate and relocate tasks.

navigation or explore-by-touch). Though participants used physical location to find elements (P1 said, *"This song should be in the middle of the scrollbar…"*), they were unsure about the exact scrollbar location of the element and slid the scroll thumb slowly to avoid skipping elements. We observed that participants varied in how they touched the scroll thumb (with their index finger, thumb, or a two-finger pinch) and how hard they pressed the scroll thumb against the screen. P1 said, *"I feel like the scrollbar with the way it scrolls between pages isn't quite as intuitive as I would think,"* while P2 had trouble knowing when to use element navigation versus page scrolling. The confusion of where and how to use the scroll thumb was also noted by P3, who said, *"I would have kind of hoped that I could use my thumb on the scrollbar without having to lift and put it back down [to move to the next page]."* Additionally, some behaviors registered unintended touch events on the screen (e.g., a long press was interpreted as two short presses because the touch contact was lost in the middle). However, participants saw potential value in the scrollbar. Though P5 felt that using the scrollbar required her to uncomfortably hold the phone, she said, *"It's just nice being able to touch something,"* and she later followed up with *"I think for beginners on the phone, that scrollbar would be better."*

For 3 out of 5 participants, the *relocate* task was faster using Interactiles than TalkBack with the help of the *bookmark* feature. However, participants were skeptical of its value due to the current implementation. They expressed the desire to be able to automatically go to a bookmark. P3 said,

"I question the usefulness of bookmarking mostly because digitally when I think of bookmark I can immediately tap a button and go to this control versus having to scroll around to find it."

Mixed text/number entry: Using the number pad was much faster than the keyboard alone for all participants. Participants also showed a strong preference for the number pad, as shown in Figure 6.5. For example, P1 said, *"Where I would use this piece of hardware the most is the number pad… [to] enter the extension 156 during a phone call… I can't do that fast enough [with the default keyboard]."* Similarly, P3 liked the number pad *"for entering long strings of numbers... It saves time switching to the number/symbol keyboard and is more intuitive because the layout of the numbers on the symbol keyboard is horizontal and that takes a bit of getting used to."* P4, quite enthusiastic, said *"This I love. This is genius."* but also suggested *"rather than the key that reads aloud what's on the screen, I might change that to something that's more commonly used, maybe a pound sign or perhaps a period or a hyphen."*

Holistic value: Functions overloaded to a single component created confusion for participants. For example, there were two ways to go to the "next page" (i.e., a long press of the bottom at the scrollbar in element navigation or a slide in page scrolling). These two ways confused our participants due to the mode switch required to go between them. In the words of P4, *"It felt a little clumsy figuring out up/down, when to push, how long to hold it."* On the other hand, when the mode switches and associated functionality were clear, Interactiles was valued. To use the number pad, participants had to make a conscious effort to flip the number pad onto the screen and the scrollbar off the screen. Although it required a mode switch of sorts, the number pad feature resulted in high participant satisfaction.

6.3.6 Customizability

P1, P2, and P3 expressed the desire to personalize Interactiles to meet their own needs. For example, P1 liked the number pad but did not feel he needed the scrollbar. P2 felt the hardware was uncomfortable to hold and wished she could move components around. P3 was used to holding his phone by his ear; it was difficult to use the scrollbar in that position. All participants interacted with the scroll thumb a little differently (index finger, thumb, or a two-finger pinch). Feedback from

participants suggests that a tactile approach to mobile touchscreen accessibility should be physically customizable for both the types and locations of components.

6.4 Discussion

Our results demonstrated that Interactiles was particularly useful for app switching and number entry, tasks that currently require a mode switch, but may not be as useful for tasks that are already quick even without tangibility, such as locate and relocate. Our analysis also explored interactions that may be more helpful to map to a scrollbar and provided design recommendations for future work in tangible mobile touchscreen accessibility.

As shown in Figure 6.4, Interactiles did not provide a major speed advantage for all tasks. However, it did improve task completion times for app switching and number entry. Because participants were beginners with both Interactiles and TalkBack, this suggests that Interactiles may be of value for novice users. Given more time to learn, participants might also be more comfortable with Interactiles and find greater value in its functionality.

Interactiles was least helpful for locate and relocate tasks. It failed to serve as a memory aid and was not sufficiently reliable to be trusted by participants. A secure clip for holding the scrollbar to the screen to maintain screen contact would help to reduce the uncertainty that resulted from inconsistent touch events. The scrollbar may be more useful for known mappings (e.g., a menu) than unknowns (e.g., infinite scroll). For the relocate task, although Interactiles improved task performance for 3 out of 5 participants, participants wanted an additional feature to automatically arrive at bookmarks. Given the speed benefit of bookmarking, this could be of great value. A future implementation might include a short strip of buttons that could be used as bookmarks, similar to saved radio station buttons on cars.

Interactiles was slower for element navigation than Talkback's swipe navigation or explore-by-touch. Because of space limitations in the mobile platform, many apps use linear layouts to deliver content. Although swipe navigation and explore-by-touch lack tactility, they work fast enough to help participants form a coherent understanding of the app, especially when content is linear. One

reason may be the common use of one-dimensional layouts in many small screen mobile apps. Even though swiping and explore-by-touch do not have tactility or much of a physical mapping, they work fast enough to help participants form a coherent understanding of the app, especially if content is linear. We believe this is the reason the scrollbar did not rate highly with participants, even though it showed faster completion times for all participants in the app switching task and was faster for 3 of 5 participants in the relocate task. Participants still provided positive feedback on having tangible feedback on the physical scrollbar.

One of the most difficult challenges for tangible or screen reader interaction on mobile platforms is infinite scroll. Ideally, there should be no distinction between elements and pages. The Interactiles approach of chunking elements into discrete pages, which requires participants to stop processing elements to go to the next page, adversely affects their memory and content understanding. However, software implementations of infinite scroll not only load the next page on an as-needed basis, but they also may change the order of elements each time when scrolling begins. Overlapping pages may help prevent this, but it may simply be the case that swiping is a better model for interacting with infinite scroll content.

Interactiles was most valuable both in task completion times and participant ratings for app switching and number entry. This suggests that the interactions to target on mobile phones might be those that currently require a mode switch, particularly difficult ones, such as opening the symbol keyboard to enter symbols and numbers.

6.5 Conclusion

We presented Interactiles, a system for enhancing screen reader access to mobile phones with tangible components. The success of Interactiles for app switching and number entry lends argues for a hybrid hardware-software approach. Supporting materials (e.g., 3D-printing files) are available at https://github.com/tracyttran/Interactiles.

Interactiles enhances accessibility by enabling mobile touchscreen tactility across multiple existing apps. It is the first mobile system compatible across apps, as opposed to a static overlay that works

with only one screen configuration. As shown in our technical validation, Interactiles functions effectively in most of the top 50 Android Apps. Another major advantage is its low cost and assembly from readily available materials. Both advantages indicate that Interactiles is highly deployable.

This work was originally published at ASSETS 2018 as ***Interactiles: 3D Printed Tactile Interfaces to Enhance Mobile Touchscreen Accessibility*** [114] with co-authors Tracy Tran, Yuqian Sun, Ian Culhane, Shobhit Jain, James Fogarty, and Jennifer Mankoff. I proposed the initial idea in a 3D printing course (i.e. "adding tactile feedback on mobile phones with hardware overlay and software overlay") and implemented all software components. I also designed the user study with undergraduate researchers, Tracy Tran, and Yuqian Sun. Tracy joined my course group and took a lead role in designing 3D-printed hardware. She also recruited study participants and accompanied me to all user studies. Please watch the demo video at: https://youtu.be/Fd6zmfJ9DOk.

Chapter 7 ADDITIONAL RESEARCH THEMES

This chapter highlights additional research themes that I pursued but did not directly include in this dissertation. Although they are not core contributions of my dissertation, each of these three research themes has had an impact on a chapter of this dissertation. At the end of this chapter, I will summarize their impacts on my dissertation.

7.1 Mobile Interactions

In situ self-report is widely used in human-computer interaction, ubiquitous computing, and for assessment and intervention in health and wellness. Unfortunately, it remains limited by high burdens. We examined unlock journaling as an alternative. Specifically, we built upon prior work that introduced single-slide unlock journaling gestures appropriate for health and wellness measures. We then presented the first field study comparing unlock journaling with traditional diaries and notification-based reminders in self-report of health and wellness measures. We found

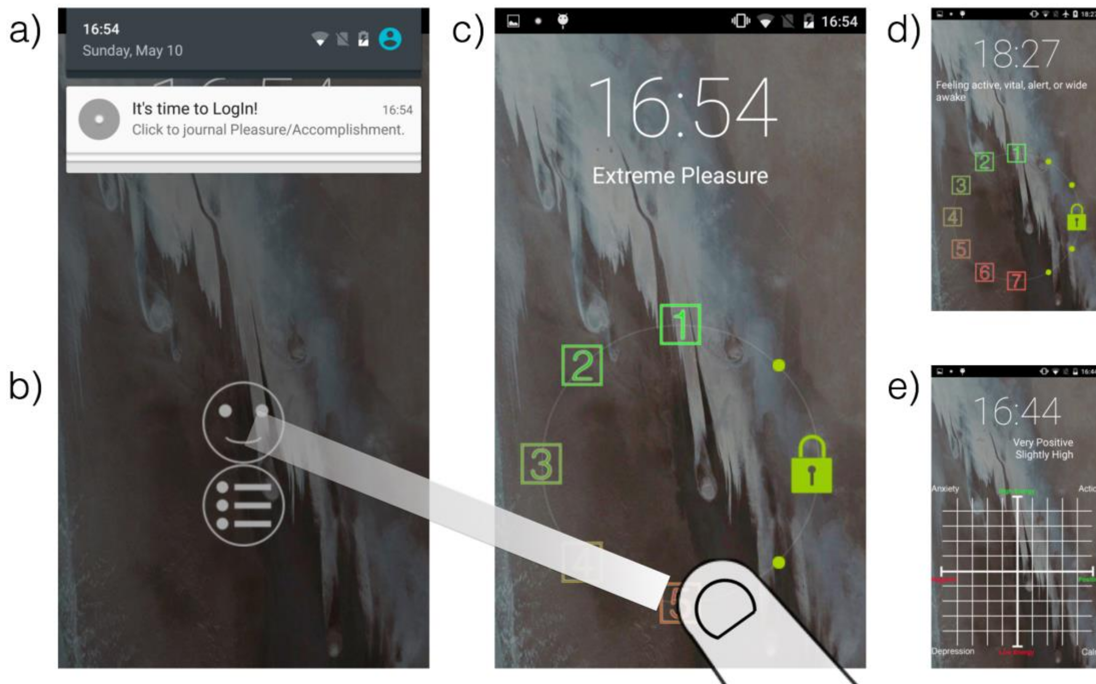


Figure 7.1. We examined unlock journaling, in which unlocking a phone also journals an in situ self-report. We presented the first study comparing it to diaries and notification-based reminders.

unlock journaling is less intrusive than reminders, dramatically improves frequency of journaling, and can provide equal or better timeliness. Where appropriate to broader design needs, unlock journaling is thus an overall promising method for in situ self-report.

The specific contributions of this work include:

- Expansion of the single-slide unlock journaling gestures design to support (1) selection among multiple measures to journal, and (2) journaling a two-dimensional measure, as shown in Figure 7.1.
- The first field study comparing unlock journaling to traditional diaries and notification-based reminders for the self-reporting of health and wellness measures.

This work was originally published at CHI 2016 as ***Examining Unlock Journaling with Diaries and Reminders for In Situ Self-Report in Health and Wellness*** [111] with co-authors Laura R. Pina and James Fogarty. Please watch the demo video at: https://youtu.be/XauaChOS9y4.

7.2 Large-Scale Mobile App Accessibility Analysis

We conducted the first large-scale analysis of the accessibility of mobile apps, examining what unique insights this could provide into the state of mobile app accessibility. We analyzed 5,753 free Android apps for label-based accessibility barriers in three classes of image-based buttons: Clickable Images, Image Buttons, and Floating Action Buttons. We used an epidemiology-inspired framework to structure the investigation [85]. We assessed the population of free Android apps for label-based inaccessible button diseases. Figure 7.2 shows the distribution of the proportion of image-based buttons within an app with a missing label. We considered three determinants of the disease: missing labels, duplicate labels, and uninformative labels. Then, we examined the prevalence (i.e., frequency of occurrences of barriers) in apps and in classes of image-based buttons. In the app analysis, 35.9% of analyzed apps had 90% or more of their assessed image-based buttons labeled, 45.9% had less than 10% of assessed image-based buttons labeled, and the remaining apps were relatively uniformly distributed along the proportion of elements that were labeled. In the class analysis, we found 92.0% of Floating Action Buttons have missing labels, compared to 54.7% of

Image Buttons and 86.3% of Clickable Images. Finally, we discussed how these accessibility barriers are addressed in existing treatments, including accessibility development guidelines.

The specific contributions of this work include:

- The first large-scale analysis of image-based button labeling for accessibility in 5,753 free Android apps.
- An analysis demonstrating a concrete application of concepts from an epidemiology-inspired framework [85] and the value of large-scale app accessibility analysis.

This work was originally published at ASSETS 2018 as ***Examining Image-Based Button Labeling for Accessibility in Apps through Large-Scale Analysis*** [86] with co-authors Anne Spencer Ross (first author), James Fogarty, and Jacob O. Wobbrock.

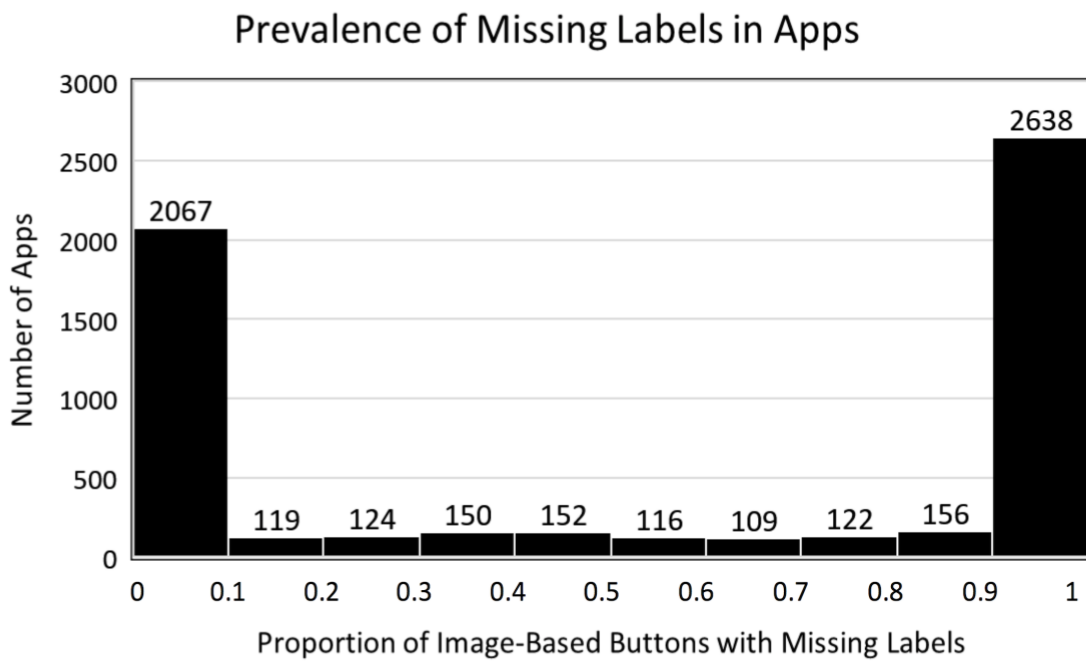


Figure 7.2. The distribution of the proportion of image-based buttons within an app with a missing label. A total of 5,753 apps were tested. A higher proportion indicates an app with more errors. The high number of apps at the extremes along with the uniform, non-zero distribution between the extremes hints at a rich ecosystem of factors influencing whether an app's image-based buttons are labeled.

7.3 Mobile Eye Tracking for People with Motor Impairments

Current eye-tracking input systems for people with ALS (Amyotrophic Lateral Sclerosis) or other motor impairments are expensive, not robust under sunlight, and require frequent re-calibration and substantial, relatively immobile setups. Eye-gaze transfer (e-tran) boards, a low-tech alternative, are challenging to master and offer slow communication rates. To mitigate the drawbacks of these two status quo approaches, we created GazeSpeak, an eye gesture communication system that runs on a smartphone. GazeSpeak is designed to be low-cost, robust, portable, and easy-to-learn, with a higher communication bandwidth than an e-tran board. Figure 7.3 shows the setup of GazeSpeak. It can interpret eye gestures in real time, decode these gestures into predicted utterances, and facilitate communication. In our evaluations, GazeSpeak performed robustly, received positive feedback from participants, and improved input speed with respect to an e-tran board. We also identified avenues for further improvement to low-cost, low-effort gaze-based communication technologies.

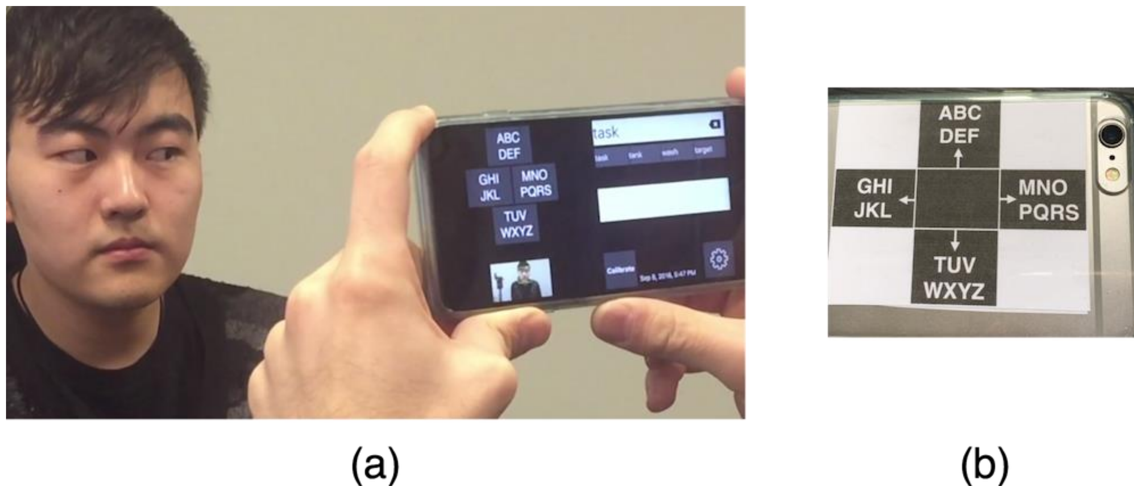


Figure 7.3. The communication partner of a person with motor disabilities can use the GazeSpeak smartphone app to translate eye gestures into words. (a) The interpreter interface is displayed on the smartphone's screen. (b) The speaker saw a sticker depicting 4 groupings of letters affixed to the phone's case.

The specific contributions of this work include:

- The GazeSpeak system, which includes algorithms to robustly recognize eye gestures in real time on a hand-held smartphone and decode these gestures into predicted utterances, and interfaces to facilitate the speaker and interpreter communication roles.
- Study results demonstrating GazeSpeak's error rate and communication speed, as well as feedback from people with ALS and their communication partners on usability, utility, and directions for further work.

This work was originally published at CHI 2017 as ***Smartphone-Based Gaze Gesture Communication for People with Motor Disabilities*** [110] with co-authors Harish Kulkarni and Meredith Ringel Morris. Please watch the demo video at: https://youtu.be/bHwIIVDoZoM.

7.4 Conclusion

These additional research themes influenced the research my dissertation. The first theme, ***mobile interactions***, prompted me to explore interaction patterns and their implementation on mobile platforms. This exploration prepared me to categorize the interaction re-mapping design space (Chapter 3) and to implement interaction proxies (Chapter 4). The second theme, ***large-scale mobile app accessibility analysis***, provided insights into the widespread inaccessibility of apps. The large number of accessibility issues motivated me to develop a more robust and scalable method of repair (Chapter 5). The third theme, ***mobile eye tracking for people with motor impairments***, demonstrated the feasibility of a low-cost and portable alternative to commercial eye tracking systems enabled by a commodity mobile phone. This demonstration inspired me to build Interactiles (Chapter 6) as a low-cost and portable system to provide tactile feedback.

Chapter 8 CONCLUSION

Mobile apps have become ubiquitous, used to access a wide variety of services online and in the physical world. However, many apps remain difficult or impossible to access for people with disabilities. This dissertation proposes the opportunity for third-party repair and enhancement of mobile app accessibility at runtime. In this dissertation, I demonstrated my thesis statement:

> ***An interaction remapping strategy can enable third-party developers and researchers to robustly repair and enhance the accessibility of mobile applications at runtime, while preserving the platform's security model and accessibility infrastructure.***

In support of this statement, I developed interaction proxies as a strategy to enable developers and researchers to modify app interactions at runtime (Chapters 3 and 4). This strategy does not require an app's source code or rooting the phone and remains compatible with built-in assistive technologies. To scale interaction proxies for repairing real-world accessibility issues, I built a robust annotation system with developer tools to create accessibility repairs for mobile apps (Chapter 5). To enhance the accessibility of touchscreen mobile phones for people with visual impairments, I created the Interactiles system, which provides tactile feedback on mobile phones using low-cost 3D printed widgets (Chapter 6). All thesis projects preserve the mobile platform's security model and accessibility infrastructure.

The next sections summarize the contributions made in my dissertation work, discuss some opportunities for future work, and conclude this dissertation.

8.1 Contributions

The specific contributions of my dissertation research are presented below, including concepts, artifacts, and experimental results.

8.1.1 Concepts

- The **design space of interaction re-mapping** (Chapter 3) provides examples of re-mapping existing inaccessible interactions into new accessible interactions. It supports researchers and developers who design accessibility repairs and enhancements for mobile apps.
- The **interaction proxy strategy** (Chapter 4) enables runtime repair and enhancement of app accessibility while preserving mobile security model and accessibility infrastructure. It supports the implementation of examples in the interaction re-mapping design space.
- **Screen equivalence heuristics** (Chapter 5) enable robust annotation on mobile app interfaces. The heuristics examine systematic behaviors that cause screen equivalence errors, based entirely on information available from standard Android accessibility APIs.

8.1.2 Artifacts

- **Demonstration accessibility repairs** (Chapter 4) present the use cases of interaction proxies and provide implementation abstractions on Android. Their key technical approaches inform future development of the interaction proxy strategy.
- **Mobile app interface annotation tools** (Chapter 5) support developers implementing repairs of mobile app accessibility. The *capture tool* uploads current screen information to the database. The *template tool* identifies and displays unique template screens. The *annotation tool* supports authoring annotations on a template screen. The *runtime library* compares a runtime screen against template screens and applies annotations on its interface elements if a match is found.
- The **Interactiles system** (Chapter 6) enhances tactile interaction on touchscreen smartphones with 3D-printed hardware and a software service. Interactiles is portable, low-cost, unpowered, and compatible with existing Android apps.

8.1.3 Experimental Results

- The **interviews regarding interaction proxies** (Chapter 4) focused on: (1) app accessibility barriers, (2) the experience of using an interface with an interaction proxy, (3) the usefulness of interaction proxies, and (4) their potential for adoption. The interviews about accessibility barriers informed many of the interaction proxies developed. Participants reported the interactions in repaired apps were seamless. The interaction proxies were perceived as being part of the interface and helped participants complete tasks. Finally, participants raised forward-thinking questions for broad deployment.
- The **evaluation of screen equivalence heuristics** (Chapter 5) examined equivalence in 42,504 pairs of screens. Participants collected a total of 2,038 screens from 50 apps. The heuristics prioritized *FalseSame* error reduction to avoid applying annotations to a wrong screen. Applying all heuristics reduced 97% of *FalseSame* errors and 60% of *FalseDifferent* errors in screen equivalence from the baseline method.
- The **case studies of annotation-based runtime repair** (Chapter 5) gathered individual feedback and implemented larger-scale repairs. The participant confirmed our repairs addressed the issues he reported, as well as additional issues. He also described how repairs would change how he uses apps and might help more people. The set of runtime repairs for 26 apps supports the potential of annotation-based accessibility repair.
- **System validation of Interactiles** (Chapter 6) verified its compatibility across 50 Android apps. The main features functioned effectively in most of the apps: *page scrolling* worked in 49 apps; *element navigation* worked in 42 apps; *bookmark* worked in 49 apps; *number entry* worked in 49 apps; *app switching* worked in all 50 apps.
- The **Interactiles usability study** (Chapter 6) evaluated task performance and participant preference. Compare to Talkback, Interactiles was faster for all participants in app switching and number entry tasks, slower for all participants in the holistic task, and faster for some participants in locating/relocating tasks. Participants were uniformly positive about the number pad but were mixed on the usefulness of the scrollbar.

8.2 Future Directions

This dissertation surfaces numerous opportunities for future work. It may enable future research, and there are opportunities to improve upon the research done in this dissertation:

8.2.1 Deploying New Accessibility Repairs and Enhancements

The ultimate goal of interaction proxies (Chapter 4) is to help catalyze advances in mobile app accessibility. Where contributions have previously been limited to developers of individual apps or the underlying mobile platform, interaction proxies open an opportunity for third-party developers and researchers to develop and deploy accessibility repairs and enhancements into widely used apps and platforms. In addition, the annotation techniques (Chapter 5) will make these new accessibility repairs and enhancements more robust and scalable.

8.2.2 Scaling Mobile App Interface Annotation by Crowdsourcing

The annotation system (Chapter 5) currently requires developer-level expertise, which limits scalability. Future research could explore methods that allow contribution from volunteers to scale annotation, including crowdsourcing and friendsourcing techniques developed in other contexts (e.g., [88,97,98]). Researchers could build a capture tool that lets end users collect screens while using the app. A slight modification to the current annotation web interface could support crowd workers in creating accessibility metadata at scale. Furthermore, it is worth investigating whether future annotation tools could support contribution from people with disabilities.

8.2.3 Applying Robust Annotation Approaches Outside Accessibility

Robust approaches to screen equivalence and annotation (Chapter 5) could have applications that extend beyond accessibility. A few examples include:

Mobile app interface testing: Our approach to screen equivalence could be included in mobile automation testing tools [3,35,89] to determine current progress in the graph of app states.

Large-scale collection and analysis of mobile apps: An automated crawler for apps needs to determine whether an interaction leads to a new interface state or one that is already captured [19,20]. Our approach to screen equivalence reduces the error in interface state comparison and therefore avoids unnecessary and repetitive data collection.

Task automation in mobile apps: APPNITE combines natural language instructions with demonstrations to create more generalized GUI automation scripts [66]. Its authors indicate the potential of annotation: "For graphics in GUIs, especially for those without developer-provided accessibility labels, we can use runtime annotation techniques [113] to annotate their meanings", referring to our contribution in Chapter 5.

8.2.4 Making More Tasks Tangible

There is an opportunity to explore the value of Interactiles (Chapter 6) for additional tasks (e.g., menu access, copy/paste, text editing/cursor movement). It is also possible to enable T9 text entry [75] using the number pad and explore its design choices (e.g., how to effectively present word prediction candidates).

For future work, our study suggests the importance of creating a toolchain for hardware customization. Advances needed here would include: (1) an ability to create models for "plug and play" components that could be snapped in and out of the phone shell and (2) a facility for configuring mappings between components and tasks (e.g., bookmarking, app switching, text entry).

Support for adding new types of components and invoking their associated modes would also add flexibility to Interactiles. A solution could be to place a physical component on the screen and move a finger around its edges, using a gesture recognizer to identify the component according to its shape. If components were not all of a unique shape, they could include small conductive strips that create a unique pattern when a finger is swiped over them, which could then similarly be recognized to identify the component.

8.3 Limitations

I now summarize several major limitations in this dissertation.

Limited types of impairments in participants: Although this dissertation explored and implemented potential accessibility repairs and enhancements for people with different disabilities, we mainly recruited people with visual impairments in our user studies. In the evaluation of ***Mobile Eye Tracking for People with Motor Impairments*** (Section 7.3), we also received valuable feedback from people with motor impairments. I believe insights shared by people with a greater variety of impairments would also have benefited my dissertation research, and further exploring the approaches developed in this dissertation with people with a greater variety of disabilities remains an important opportunity.

One-dimensional design space: Our current interaction re-mapping design space uses only the quantity of interactions to categorize potential accessibility repairs and enhancements. A richer design space could consider many other dimensions, including the forms of interactions (e.g., swipe versus tap), the types of disability (e.g., hearing impairments versus motor impairments), or the nature of re-mappings (e.g., virtual versus physical). Such a multi-dimensional design space could offer further inspiration and structure to third-party accessibility developers and researchers.

Support for media content: In this dissertation, I mainly considered the tree representation of an interface hierarchy during screen content introspection. We may gain additional insights by making use of media content (e.g., images and animation) or conducting pixel-level analyses (e.g., automatically generating semantic annotations for mobile app interface elements [68]).

8.4 Reflections

Before concluding, I would like to share some observations I made during system development and when interacting with participants and other researchers.

I would like to have renamed "interaction proxies" to "interaction re-mappers". "Proxy" is an acceptable and widely used term in the field of computer networking, and we chose it at the time

because of the technical analogy for our interception and re-mapping of input and output. However, when I talked with accessibility researchers and industry executives, some of them pointed out the negative implication of the word "proxy" (i.e., people with disabilities require assistance from it). In addition, web proxies aim at providing an accessible rendering of the inaccessible website, instead of making the *original* website accessible to all. The disability community rejects a "separate but equal" philosophy, and I would prefer we had chosen a name that avoided such an implication.

Third-party developers may not be the best stakeholder to repair app accessibility issues, as this responsibility should primarily lie with the original app developer. I wish I could have explored more about app developer education and developer tools in building accessible apps. For example, repairing the accessibility issues of the BECU app in Chapter 5 would require extensive effort from third-party developers. In practice, the third-party repair may fail to cover some issues. Mobile apps will be more accessible when their developers consider accessibility in the first place and have more actionable accessibility suggestions from developer tools.

Automation can play a critical role in making accessibility repairs and enhancements more scalable. I wish I could devote more time to improving automation in my thesis work. Supported by techniques like those in SUPPLE [30], it may be possible to evaluate a person's abilities and automatically generate personalized interfaces for people with various forms and levels of impairments. I also wish I could have integrated crowdsourcing with my mobile app interface annotation methods and could have explored research questions in a crowd-powered annotation system (e.g., security issues arising from crowd-contributed accessibility metadata). I think such extensions to my work are important to realizing its full potential.

8.5 Conclusion

This dissertation offers: (1) a new perspective into third-party runtime mobile accessibility improvement, (2) potential of developing new tools using methods for robust annotation of mobile app interface, and (3) a vision for future research in tangible accessibility on mobile touchscreen devices.

Interaction proxies (Chapter 4) enable third-party interaction modification. Inserted between an app's original interface and a manifest interface, an interaction proxy allows third-party developers and researchers to modify an interaction at runtime, without an app's source code, without rooting the phone or otherwise modifying an app, while retaining all capabilities of the system. Participants saw the potential for third-party enhancements and were enthusiastic about the strategy, based on our proof-of-concept prototypes repairing accessibility failures in popular real-world apps. Including these prototypes in our interviews provided a real-world context for discussing the potential of interaction proxies, and participants used this as a starting point for discussing other apps in which they have encountered accessibility barriers that might be addressed.

Robust annotation (Chapter 5) allows developers to annotate mobile interfaces with metadata appropriate for runtime accessibility repair. We presented our methods in terms of screen identifiers, element identifiers, and screen equivalence heuristics. We developed an initial set of tools based on these methods, focused on developer implementation of accessibility repair services. Our current screen equivalence heuristics effectively reduce screen identification error. The evaluation proves the system's potential for repairing real-world accessibility issues. We received positive feedback on accessibility repairs from the participant and used our system to repair issues in 26 real-world apps. Because runtime repair of mobile accessibility is a relatively new capability and prior methods have required custom code specific to each repair, we believe this is the largest existing set of runtime repairs of mobile app accessibility, thereby providing support for the potential of annotation-based accessibility repair.

Interactiles (Chapter 6) enhances screen reader access to mobile phones with tangible components. The success of Interactiles for app switching and number entry provides support for a hybrid hardware-software approach. A major advantage of Interactiles is that the system is general and compatible across apps, as opposed to a static overlay that works only with one screen configuration. Another major advantage is its low cost and assembly from readily available materials. Both advantages indicate that Interactiles is highly deployable. The evaluation demonstrates the system's potential to provide tactile feedback and enhance task completion speed. Our results show that Interactiles is particularly useful for app switching and number entry,

which currently require a mode switch. But it may not be as useful for tasks that can be done quickly without tangibility (e.g., locate and relocate). This suggests that we should target the interactions that require a mode switch such as opening the symbol keyboard to enter symbols and numbers.

8.6 Final Remarks

This dissertation documents my research in runtime repair and enhancement of mobile app accessibility. This work started with the interaction re-mapping design space to characterize potential accessibility repairs and enhancements. I then built interaction proxies, which enable third-party developers and researchers to create such repairs and enhancements at runtime. Next, my mobile app interface annotation methods made these repairs more reliable and scalable. Finally, Interactiles demonstrated that accessibility enhancements can be extended to work with hardware. The strategies and tools I developed in this dissertation may suggest future research in new accessibility repairs and enhancements, crowd-powered mobile interface annotation, applications of robust annotation methods, and additional tangible tasks.

Developer education in accessibility guidelines and tools remains crucial to preventing app accessibility issues in the first place. Third-party developers and researchers cannot replace the original developers of apps in the effort to make those apps accessible. However, when app developers fail to meet the accessibility standard, it is beneficial to allow motivated third-party developers to repair accessibility issues. Moreover, this dissertation enables researchers to deploy and evaluate advanced accessibility enhancements. My work demonstrates that third-party developers and researchers can play an important role in mobile app accessibility repair and enhancement with software and low-cost hardware.

REFERENCES

1. AChecker. IDI Web Accessibility Checker: Web Accessibility Checker. Retrieved from http://achecker.ca/checker/index.php

2. Afb.org. Statistical Snapshots from the American Foundation for the Blind. Retrieved from http://www.afb.org/info/blindness-statistics/2

3. Appium. Appium. Retrieved from http://appium.io/

4. Apple. Accessibility. Retrieved from http://www.apple.com/accessibility/osx/voiceover/

5. Apple. Use Guided Access with iPhone, iPad, and iPod touch. Retrieved May 5, 2016 from https://support.apple.com/en-us/HT202612

6. Shiri Azenkot, Cynthia L. Bennett, and Richard E. Ladner. 2013. DigiTaps: Eyes-Free Number Entry on Touchscreens with Minimal Audio Feedback. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2013)*, 85–90. https://doi.org/10.1145/2501988.2502056

7. Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2013)*, 641–660. https://doi.org/10.1145/2544173.2509549

8. Mark S. Baldwin, Gillian R. Hayes, Oliver L. Haimson, Jennifer Mankoff, and Scott E. Hudson. 2017. The Tangible Desktop. *ACM Transactions on Accessible Computing (TACCESS)* 10, 3: 1–28. https://doi.org/10.1145/3075222

9. Olivier Bau, Ivan Poupyrev, Ali Israr, and Chris Harrison. 2010. TeslaTouch: Electrovibration for Touch Surfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2010)*, 283–292. https://doi.org/10.1145/1866029.1866074

10. Jeffrey P. Bigham. 2014. Making the Web Easier to See with Opportunistic Accessibility Improvement. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2014)*, 117–122. https://doi.org/10.1145/2642918.2647357

11. Jeffrey P. Bigham, Ryan S. Kaminsky, Richard E. Ladner, Oscar M. Danielsson, and Gordon L. Hempton. 2006. WebInSight: Making Web Images Accessible. In *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2006)*, 181–188. https://doi.org/10.1145/1168987.1169018

12. Jeffrey P. Bigham and Richard E. Ladner. 2007. AccessMonkey: A Collaborative Scripting Framework for Web Users and Developers. In *Proceedings of the Web for All Conference (W4A 2007)*, 25–34. https://doi.org/10.1145/1243441.1243452

13. Jeffrey P. Bigham, Craig M. Prince, and Richard E. Ladner. 2008. WebAnywhere: a Screen Reader On-the-Go. In *Proceedings of the International Workshop on Web Accessibility (W4A 2008)*, 73–82. https://doi.org/10.1145/1368044.1368060

14. Blitab. Blitab | Feelings Get Visible. Retrieved from http://blitab.com/

15. Jim A. Carter and David W. Fourne. 2007. Techniques to Assist in Developing Accessibility Engineers. In *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2007)*, 123–130. https://doi.org/10.1145/1296843.1296865

16. Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. 2011. Associating The Visual Representation of User Interfaces with Their Internal Structures and Metadata. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2011)*, 245–256. https://doi.org/10.1145/2047196.2047228

17. Tzu-wen Chang, Neng-Hao Yu, Sung-Sheng Tsai, Mike Y. Chen, and Yi-Ping Hung. 2012. Clip-on Gadgets: Expandable Tactile Controls for Multi-touch Devices. In *Proceedings of the International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI 2012)*, 163–166. https://doi.org/10.1145/2371664.2371699

18. Alan Cooper. 1995. *About Face: The Essentials of User Interface Design*. John Wiley & Sons, Inc., New York, NY.

19. Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2017)*, 845–854. https://doi.org/10.1145/3126594.3126651

20. Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction Mining Mobile Apps. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2016)*, 767–776. https://doi.org/10.1145/2984511.2984581

21. Morgan Dixon and James Fogarty. 2010. Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2010)*, 1525–1534. https://doi.org/10.1145/1753326.1753554

22. Morgan Dixon, Gierad Laput, and James Fogarty. 2014. Pixel-Based Methods for Widget State and Style in a Runtime Implementation of Sliding Widgets. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2014)*, 2231–2240. https://doi.org/10.1145/2556288.2556979

23. Morgan Dixon, Daniel Leventhal, and James Fogarty. 2011. Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2011)*, 969–978. https://doi.org/10.1145/1978942.1979086

24. Morgan Dixon, A. Conrad Nied, and James Fogarty. 2014. Prefab Layers and Prefab Annotations: Extensible Pixel-Based Interpretation of Graphical Interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2014)*, 221–230. https://doi.org/10.1145/2642918.2647412

25. James R. Eagan, Michel Beaudouin-Lafon, and Wendy E. Mackay. 2011. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2011)*, 225–234. https://doi.org/10.1145/2047196.2047226

26. W. Keith Edwards and Elizabeth D. Mynatt. 1994. An architecture for transforming graphical interfaces. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 1994)*, November: 39–47. https://doi.org/10.1145/192426.192443

27. W. Keith Edwards, Ian Smith, Scott E. Hudson, Joshua Marinacci, Roy Rodenstein, and Thomas Rodriguez. 1997. Systematic Output Modification in a 2D User Interface Toolkit. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 1997)*, 151–158. https://doi.org/10.1145/263407.263537

28. Yasmine N. El-Glaly, Francis Quek, Tonya Smith-Jackson, and Gurjot Dhillon. 2013. Touch-screens Are Not Tangible. In *Proceedings of the International Conference on Tangible, Embedded and Embodied Interaction (TEI 2013)*, 245–253. https://doi.org/10.1145/2460625.2460665

29. Leah Findlater, Alex Jansen, Kristen Shinohara, Morgan Dixon, Peter Kamb, Joshua Rakita, and Jacob O. Wobbrock. 2010. Enhanced Area Cursors: Reducing Fine-Pointing Demands for People with Motor Impairments. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2010)*, 153–162. https://doi.org/10.1145/1866029.1866055

30. Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. 2010. Automatically Generating Personalized User Interfaces with SUPPLE. *Artificial Intelligence* 174, 12–13: 910–950. https://doi.org/10.1016/j.artint.2010.05.005

31. Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. 2008. Improving the Performance of Motor-Impaired Users with Automatically-Generated, Ability-Based Interfaces. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2008)*: 1257–1266. https://doi.org/10.1145/1357054.1357250

32. Mayank Goel, Jacob O. Wobbrock, and Shwetak N. Patel. 2012. GripSense: Using Built-In Sensors to Detect Hand Posture and Pressure on Commodity Mobile Phones. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2012)*. https://doi.org/10.1145/2380116.2380184

33. Google. Make Apps More Accessible. Retrieved from https://developer.android.com/guide/topics/ui/accessibility/apps.html

34. Google. Get Started on Android with TalkBack. Retrieved from https://support.google.com/accessibility/android/answer/6283677?hl=en

35. Google. Monkeyrunner. Retrieved from https://developer.android.com/studio/test/monkeyrunner/index.html

36. Google. UiSelector. Retrieved from https://developer.android.com/reference/android/support/test/uiautomator/UiSelector.html

37. Google. Distribution Dashboard. Retrieved from https://developer.android.com/about/dashboards/index.html

38. Google. Accessibility Scanner. Retrieved from https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor

39. Google. Building Accessible Custom Views. Retrieved from https://developer.android.com/guide/topics/ui/accessibility/custom-views.html#virtual-hierarchy

40. Google. Eyes-free Forum. Retrieved from https://groups.google.com/forum/#!forum/eyes-free

41. Google. AccessibilityEvent. Retrieved from https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent.html

42. Tiago Guerreiro, Hugo Nicolau, Joaquim Jorge, and Daniel Gonçalves. 2009. NavTap: a Long Term Study with Excluded Blind Users. In *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2009)*, 99–106. https://doi.org/10.1145/1639642.1639661

43. Anhong Guo, Jeeeun Kim, Xiang "Anthony" Chen, Tom Yeh, Scott E. Hudson, Jennifer Mankoff, and Jeffrey P. Bigham. 2017. Facade: Auto-generating Tactile Interfaces to Appliances. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2017)*, 5826–5838. https://doi.org/10.1145/3025453.3025845

44. Vicki L. Hanson and John T. Richards. 2013. Progress on Website Accessibility? *ACM Transactions on the Web* 7, 1: 2:1-2:30. https://doi.org/10.1145/2435215.2435217

45. Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys 2014)*, 204–217. https://doi.org/10.1145/2594368.2594390

46. Kyle J. Harms, Jordana H. Kerr, and Caitlin L. Kelleher. 2011. Improving Learning Transfer from Stencils-Based Tutorials. In *Proceedings of the International Conference on Interaction Design and Children (IDC 2011)*, 157–160. https://doi.org/10.1145/1999030.1999050

47. Liang He, Zijian Wan, Stacy Biloa, Leah Findlater, and Jon E. Froehlich. 2017. TacTILE: a Preliminary Toolchain for Creating Accessible Graphics with 3D-printed Overlays and Auditory Annotations. In *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2017)*, 397–398. https://doi.org/10.1145/3132525.3134818

48. Yun Huang, Brian Dobreski, Bijay Bhaskar Deo, Jiahang Xin, Natã Miccael Barbosa, Yang Wang, and Jeffrey P. Bigham. 2015. CAN: Composable Accessibility Infrastructure via Data-Driven Crowdsourcing. In *Proceedings of the Web for All Conference (W4A 2015)*, 1–10. https://doi.org/10.1145/2745555.2746651

49. Humanware. Victor Reader Stream. Retrieved from https://store.humanware.com/hus/victor-reader-stream-new-generation.html

50. Amy Hurst, Scott E. Hudson, and Jennifer Mankoff. 2010. Automatically Identifying Targets Users Interact With During Real World Tasks. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI 2010)*, 11–20. https://doi.org/10.1145/1719970.1719973

51. Amy Hurst, Jennifer Mankoff, Anind K. Dey, and Scott E. Hudson. 2007. Dirty Desktops: Using a Patina of Magnetic Mouse Dust to Make Common Interactor Targets Easier to Select. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2007)*, 183–186. https://doi.org/10.1145/1294211.1294242

52. Edwin Hutchins, James Hollan, and Donald Norman. 1985. Direct Manipulation Interfaces. *Human-Computer Interaction* 1, 4: 311–338.

53. InclusiveAndroid. App and Game Categories. Retrieved from https://www.inclusiveandroid.com/?q=app-and-game-categories

54. Yeongjin Jang, Chengyu Song, Simon P. Chung, Tielei Wang, and Wenke Lee. 2014. A11y Attacks: Exploiting Accessibility in Operating Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS 2014)*, 103–115. https://doi.org/10.1145/2660267.2660295

55. Alex Jansen, Leah Findlater, and Jacob O. Wobbrock. 2011. From The Lab to The World: Lessons from Extending a Pointing Technique for Real-World Use. In *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems (CHI 2011)*, 1867–1872. https://doi.org/10.1145/1979742.1979888

56. Shaun K. Kane, Jeffrey P. Bigham, and Jacob O. Wobbrock. 2008. Slide Rule: Making Mobile Touch Screens Accessible to Blind People using Multi-touch Interaction Techniques. In *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2008)*, 73–80. https://doi.org/10.1145/1414471.1414487

57. Shaun K. Kane, Chandrika Jayant, Jacob O. Wobbrock, and Richard E. Ladner. 2009. Freedom to Roam: A Study of Mobile Device Adoption and Accessibility for People with Visual and Motor Disabilities. In *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2009)*, 115–122. https://doi.org/10.1145/1639642.1639663

58. Shaun K. Kane, Meredith Ringel Morris, Annuska Z. Perkins, Daniel Wigdor, Richard E. Ladner, and Jacob O. Wobbrock. 2011. Access Overlays: Improving Non-Visual Access to Large Touch Screens for Blind Users. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2011)*, 273–282. https://doi.org/10.1145/2047196.2047232

59. Shaun K. Kane, Meredith Ringel Morris, and Jacob O. Wobbrock. 2013. Touchplates: Low-cost Tactile Overlays for Visually Impaired Touch Screen Users. In *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2013)*. https://doi.org/10.1145/2513383.2513442

60. Shaun K. Kane, Jacob O Wobbrock, Mark Harniss, and Kurt L. Johnson. 2008. Truekeys: Identifying and Correcting Typing Errors for People with Motor Impairments. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI 2008)*, 349. https://doi.org/10.1145/1378773.1378827

61. Shinya Kawanaka, Yevgen Borodin, Jeffrey P. Bigham, Darren Lunn, Hironobu Takagi, and Chieko Asakawa. 2008. Accessibility Commons: A Metadata Infrastructure for Web Accessibility. In *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2008)*, 153–160. https://doi.org/10.1145/1414471.1414500

62. Caitlin Kelleher and Randy Pausch. 2005. Stencils-Based Tutorials: Design and Evaluation. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2005)*, 541–550. https://doi.org/10.1145/1054972.1055047

63. Richard E. Ladner. 2015. Design for User Empowerment. *Interactions* 22, 2: 24–29. https://doi.org/10.1145/2723869

64. Jonathan Lazar, Daniel F. Goldstein, and Anne Taylor. 2015. *Ensuring Digital Accessibility through Process and Policy*. Retrieved from http://www.elsevier.com/books/ensuring-digital-accessibility-through-process-and-policy/lazar/978-0-12-800646-7

65. Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2017)*, 6038–6049. https://doi.org/10.1145/3025453.3025483

66. Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M. Mitchell, and Brad A. Myers. 2018. APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2018)*, 105–114. https://doi.org/10.1109/VLHCC.2018.8506506

67. Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. 2014. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI 2014)*, 57–70.

68. Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018. Learning Design Semantics for Mobile Apps. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2018)*. https://doi.org/10.1145/3242587.3242650

69. Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2016)*, 94–105. https://doi.org/10.1145/2931037.2931054

70. David McGookin, Stephen Brewster, and WeiWei Jiang. 2008. Investigating Touchscreen Accessibility for People with Visual Impairments. In *Proceedings of the Nordic Conference on Human-Computer Interaction: Building Bridges (NordiCHI 2008)*, 298–307. https://doi.org/10.1145/1463160.1463193

71. James Morris and Jörg Müller. 2014. Blind and Deaf Consumer Preferences for Android and iOS Smartphones. In *Inclusive Designing*. Springer International Publishing, Cham, 69–79. https://doi.org/10.1007/978-3-319-05095-9_7

72. Brad Myers, Scott E. Hudson, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction 7*, 3–28. https://doi.org/10.1145/344949.344959

73. Elizabeth D. Mynatt. 1997. Transforming Graphical Interfaces Into Auditory Interfaces for Blind Users. *Human-Computer Interaction* 12, 1: 7–45. https://doi.org/10.1207/s15327051hci1201&2_2

74. Elizabeth D. Mynatt and W. Keith Edwards. 1992. Mapping GUIs to Auditory Interfaces. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 1992)*: 61–70. https://doi.org/http://doi.acm.org/10.1145/142621.142629

75. Nuance. T9 Text Input. Retrieved from https://www.nuance.com/mobile/mobile-solutions/text-input-solutions/t9.html

76. Dan R. Olsen Jr., Scott E. Hudson, Thorn Verratti, Jeremy M. Heiner, and Matt Phelps. 1999. Implementing Interface Attachments Representations Based on Surface. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 1999)*, 191–198. https://doi.org/10.1145/302979.303038

77. World Health Organization. 2011. World Report on Disability. Retrieved from http://www.who.int/disabilities/world_report/2011/en/

78. OWASP. 2017. Clickjacking. Retrieved from https://www.owasp.org/index.php/Clickjacking

79. OwnFone. Make Your OwnFone. Retrieved from https://www.myownfone.com/make-your-ownfone

80. Elaine Pearson, Chrstopher Bailey, and Steve Green. 2011. A Tool to Support the Web Accessibility Evaluation Process for Novices. In *Proceedings of the Conference on Innovation and Technology in Computer Science Education (ITiCSE 2011)*, 28–32. https://doi.org/10.1145/1999747.1999758

81. André Rodrigues. 2015. Breaking Barriers with Assistive Macros. In *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2015)*, 351–352. https://doi.org/10.1145/2700648.2811322

82. André Rodrigues and Tiago Guerreiro. 2014. SWAT: Mobile System-Wide Assistive Technologies. In *Proceedings of the International BCS Human Computer Interaction Conference (British HCI 2014)*, 341–346. Retrieved from https://dl.acm.org/citation.cfm?id=2742991

83. Franziska Roesner, James Fogarty, and Tadayoshi Kohno. 2012. User Interface Toolkit Mechanisms for Securing Interface Elements. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2012)*, 239–250. https://doi.org/10.1145/2380116.2380147

84. Mario Romero, Brian Frey, Caleb Southern, and Gregory D. Abowd. 2011. BrailleTouch: Designing a Mobile Eyes-free Soft Keyboard. In *Proceedings of the International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI 2011)*, 707–709. https://doi.org/10.1145/2037373.2037491

85. Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. 2017. Epidemiology as a Framework for Large-Scale Mobile Application Accessibility Assessment. In *Proceedings of the ACM SIGACCESS Conference on Computers and Accessibility (ASSETS 2017)*, 2–11. https://doi.org/10.1145/3132525.3132547

86. Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. 2018. Examining Image-Based Button Labeling for Accessibility in Android Apps through Large-Scale Analysis. In *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2018)*, 119–130. https://doi.org/10.1145/3234695.3236364

87. Daisuke Sato, Masatomo Kobayashi, Hironobu Takagi, and Chieko Asakawa. 2009. What's Next? A Visual Editor for Correcting Reading Order. In *Proceedings of the International Conference on Human-Computer Interaction (INTERACT 2009)*, 364–377. https://doi.org/10.1007/978-3-642-03655-2_41

88. Daisuke Sato, Hironobu Takagi, Masatomo Kobayashi, Shinya Kawanaka, Chieko Asakawa, and Asakawa Chieko. 2010. Exploratory Analysis of Collaborative Web Accessibility Improvement. *ACM Transactions on Accessible Computing (TACCESS)* 3, 2: 5. https://doi.org/10.1145/1857920.1857922

89. Selendroid. Selendroid. Retrieved from http://selendroid.io/

90. Square. Flow Github Repository. Retrieved from https://github.com/square/flow

91. Statista. 2016. Number of Available Applications in the Google Play Store. Retrieved from http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

92. StrangelyTyped. My Brief Experiences with Android Talkback/Accessibility. Retrieved from https://www.reddit.com/r/Android/comments/3uqs6z/my_brief_experiences_with_android/

93. Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Phillips, and Nicolas Roussel. 2006. User Interface Façades: Towards Fully Adaptable User Interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2006)*, 309–318. https://doi.org/10.1145/1166253.1166301

94. S. Swaminathan, K. Hara, and J.P. Bigham. 2017. The Crowd Work Accessibility Problem. In *In Proceedings of the Web for All Conference (W4A 2017)*. https://doi.org/10.5194/bgd-4-3409-2007

95. Symantec. 2016. Accessibility Clickjacking – Android Malware Evolution. Retrieved from https://www.symantec.com/connect/blogs/accessibility-clickjacking-android-malware-evolution

96. Hironobu Takagi and Chieko Asakawa. 2000. Transcoding Proxy for Nonvisual Web Access. In *Proceedings of the ACM Conference on Assistive Technologies (ASSETS 2000)*, 164–171. https://doi.org/10.1145/354324.354371

97. Hironobu Takagi, Shinya Kawanaka, Masatomo Kobayashi, Takashi Itoh, and Chieko Asakawa. 2008. Social Accessibility: Achieving Accessibility Through Collaborative Metadata Authoring. In *Proceedings of the ACM SIGACCESS Conference on Computers and Accessibility (ASSETS 2008)*, 193–200. https://doi.org/10.1145/1414471.1414507

98. Hironobu Takagi, Shinya Kawanaka, Masatomo Kobayashi, Daisuke Sato, and Chieko Asakawa. 2009. Collaborative Web Accessibility Improvement: Challenges and Possibilities. In *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2009)*, 195–202. https://doi.org/10.1145/1639642.163967

99. Desney S. Tan, Brian Meyers, and Mary Czerwinski. 2004. WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. In *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems (CHI'EA 2004)*, 1525–1528. https://doi.org/10.1145/985921.986106

100. Brandon Taylor, Anind Dey, Dan Siewiorek, and Asim Smailagic. 2016. Customizable 3D Printed Tactile Maps as Interactive Overlays. In *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2016)*, 71–79. https://doi.org/10.1145/2982142.2982167

101. Mosaic Design Team. 1993. Group Annotations in NCSA Mosaic. Retrieved from https://www.math.utah.edu/~beebe/support/html/Docs/group-annotations.html

102. Daniel Trindade, André Rodrigues, Tiago Guerreiro, and Hugo Nicolau. 2018. Hybrid-Brailler: Combining Physical and Gestural Interaction for Mobile Braille Input and Editing. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2018)*. https://doi.org/10.1145/3173574.3173601

103. W3C. W3C Web Annotation Working Group. Retrieved from https://www.w3.org/annotation/

104. Jacob O. Wobbrock, James Fogarty, Shih-Yen (Sean) Liu, Shunichi Kimuro, and Susumu Harada. 2009. The Angle Mouse: Target-Agnostic Dynamic Gain Adjustment Based on Angular Deviation. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2009)*, 1401–1410. https://doi.org/10.1145/1518701.1518912

105. Jacob O. Wobbrock, Shaun K. Kane, Krzysztof Z. Gajos, Susumu Harada, and Jon E. Froehlich. 2011. Ability-Based Design: Concept, Principles and Examples. *ACM Transactions on Accessible Computing (TACCESS)* 3, 3: 1–27. https://doi.org/10.1145/1952383.1952384

106. Aileen Worden, Nef Walker, Krishna Bharat, and Scott E. Hudson. 1997. Making Computers Easier for Older Adults to Use: Area Cursors and Sticky Icons. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 1997)*, 266–271. https://doi.org/10.1145/258549.258724

107. xda-developers. 2017. Google is Threatening to Remove Apps with Accessibility Services from the Play Store. Retrieved from https://www.xda-developers.com/google-threatening-removal-accessibility-services-play-store/

108. Xposed. DisableFlagSecure. Retrieved from http://repo.xposed.info/module/fi.veetipaananen.android.disableflagsecure

109. Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2009)*, 183–192. https://doi.org/10.1145/1622176.1622213

110. Xiaoyi Zhang, Harish Kulkarni, and Meredith Ringel Morris. 2017. Smartphone-Based Gaze Gesture Communication for People with Motor Disabilities. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2017)*, 2878–2889. https://doi.org/10.1145/3025453.3025790

111. Xiaoyi Zhang, Laura R. Pina, and James Fogarty. 2016. Examining Unlock Journaling with Diaries and Reminders for In Situ Self-Report in Health and Wellness. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2016)*, 5658–5664. https://doi.org/10.1145/2858036.2858360

112. Xiaoyi Zhang, Anne Spencer Ross, Anat Caspi, James Fogarty, and Jacob O. Wobbrock. 2017. Interaction Proxies for Runtime Repair and Enhancement of Mobile Application Accessibility. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2017)*. https://doi.org/10.1145/3025453.3025846

113. Xiaoyi Zhang, Anne Spencer Ross, and James Fogarty. 2018. Robust Annotation of Mobile Application Interfaces in Methods for Accessibility Repair and Enhancement. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2018)*, 609–621. https://doi.org/10.1145/3242587.3242616

114. Xiaoyi Zhang, Tracy Tran, Yuqian Sun, Ian Culhane, Shobhit Jain, James Fogarty, and Jennifer Mankoff. 2018. Interactiles: 3D Printed Tactile Interfaces on Phone to Enhance Mobile Accessibility. In *Proceedings of the ACM Conference on Computers and Accessibility (ASSETS 2018)*, 131–142. https://doi.org/10.1145/3234695.3236349

115. Jason Chen Zhao, Richard C. Davis, Pin Sym Foong, and Shengdong Zhao. 2015. CoFaçade: A Customizable Assistive Approach for Elders and Their Helpers. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2015)*, 1583–1592. https://doi.org/10.1145/2702123.2702588

116. Yu Zhong, Astrid Weber, Casey Burkhardt, Phil Weaver, and Jeffrey P. Bigham. 2015. Enhancing Android Accessibility for Users with Hand Tremor by Reducing Fine Pointing and Steady Tapping. In *Proceedings of the Web for All Conference on (W4A 2015)*, 29:1-29:10. https://doi.org/10.1145/2745555.2747277