A Large-Scale, Multi-Factor Approach to Understanding and Improving Mobile Application Accessibility

Anne Spencer Ross

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington

2021

Reading Committee:

James Fogarty, Chair

Jacob O. Wobbrock, Chair

Jennifer Mankoff

Program Authorized to Offer Degree:

Computer Science & Engineering

University of Washington

Abstract

A Large-Scale, Multi-Factor Approach
to Understanding and Improving Mobile Application Accessibility

Anne Spencer Ross

Co-Chairs of the Supervisory Committee:
Professor James Fogarty
Computer Science & Engineering

Professor Jacob O. Wobbrock
The Information School
Computer Science & Engineering

Accessibility failures in mobile applications (apps) create barriers for disabled people and people who use assistive technologies. Given the growing role of apps in everyone's daily life, equitable access is imperative. Toward this goal, I created a conceptual framework for understanding and improving app accessibility at scale inspired by epidemiology. My epidemiology-inspired framework poses app accessibility failures as "diseases" in a "population" of apps. This perspective forefronts a population-level perspective within an ecosystem of factors that impact app accessibility (e.g., developer tools, guidelines, company culture, and many more). In this dissertation, I demonstrate my thesis that *applying my epidemiology-inspired framework, which emphasizes large-scale and multi-factor approaches, (1) can reveal population-level trends of accessibility failures, (2) can aid in identifying a range of factors that impact app*

accessibility, and (3) can inform the design of tools for identifying and repairing accessibility failures in apps.

To enhance our understanding of the state of app inaccessibility, I performed the first large-scale analyses of Android app accessibility. My results measured the prevalence of accessibility failures across apps and identified classes of elements that frequently had accessibility failures. Missing labels was one of the most prevalent failures; 23% of the 8,901 apps had more than 90% of their image-based elements missing labels. Reflecting a less frequent but severe accessibility failure, 8% of 9,999 tested apps were completely unusable with many assistive technologies, such as screen readers. Apps with such failures disproportionately came from the Education category.

Using a multi-factor assessment, partially guided by my large-scale analyses, I identified accessibility shortcomings in environmental factors such as programming tools, developer guidelines, and inter-team dynamics that could contribute to app inaccessibility at scale. For example, I identified a set of game engines and cross-platform tools (e.g., Unity, Adobe Air) that frequently produced apps that were unusable with many assistive technologies. Another factor I assessed was Android's implementation documentation, finding many instances of missing labels in the example code snippets.

Toward improving app accessibility, I explored techniques for enhancing developer tools, testing tools, and third-party repairs. In developer tools, I present novel designs and techniques that aim to improve the efficiency, effectiveness, and education of developers by tightening the runtime-implementation feedback loop, leveraging screen context to provide more specific repairs, and using new visualization and interaction tool interface designs. Within testing tools, I prototyped an extension to Google's Accessibility Scanner to allow human annotation of automated results.

Professional accessibility testers found the tool promising and discussed factors beyond tools, such as knowledge gaps and social dynamics with the developer teams, which affected their testing. Addressing repairs after an app is released, I present a proof-of-concept for third-party repair that people who use screen readers found useful while highlighting factors around trust, repair-production infrastructure, and co-creation that would impact real-word deployment.

Throughout my dissertation, I leverage my epidemiology-inspired concepts and language to highlight the value of my research and place it within the larger space of work on app accessibility. Together, this research aims to expand our understanding of app accessibility at scale and inform efforts to improve it.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

To my dad.

For cultivating my passion for research and sense of adventure.

I love you and miss you always.

ACKNOWLEDGEMENTS

I am eternally grateful for the support I received during my PhD.

Thank you to my family for their unwavering love and support. To my dad, who cultivated my research interest and lovingly took a red pen to every piece of writing. To my mom, always ready with words of support and a good laugh. To Samantha, my sister and academia buddy, for her continual mentorship through grad school and beyond. To Carl, my brother and tech buddy, for many great conversations and adventures. To Bryce, my amazing partner, for being a rock through the lowest lows and cheerleader through the highest highs. Not to mention being one of the only people to read my dissertation from cover to cover, just for me. To my whole family for always making time for one another, teaching me to prioritize those I love, and your unwavering support as I pursued my passions. Thank you with all of my heart; I could not have done it without you.

Thank you to my friends and colleagues who helped me grow as a researcher, navigate the challenges of grad school, and enjoy the experience. To my undergrad research group who introduced me to human-computer interaction research. To my grad school office mates of CSE 260 for your everyday energy, encouragement, laughs, and hot pot. To my dissertation writing group for your advice, celebration, and commiseration while making the final push. To my fogies and ACE lab mates for your mentorship, collaboration, and friendship. To my friends outside of grad school who enriched my life in innumerable ways; especially Radford-neighbors-turned-family and my cat Godson, Boney.

Thank you to the accessibility and HCI research communities—DUB, ASSETS, Access SIGCHI, CHI, UW accessibility groups—for teaching me the field, for the conversations, collaborations, and critiques, and for pushing me to develop as an accessibility advocate. To my undergraduate mentees for your enthusiasm, hard work, brilliance, and feedback. Thank you

to all my internship advisors and mentors who introduced me to new fields and supported my professional growth.

Thank you to my PhD committee members—James Fogarty, Jake Wobbrock, Jen Mankoff, Merrie Ringel Morris, Mark Harniss, and Leah Findlater—for your valuable feedback and support.

Thank you to my PhD advisors—James and Jake—for cultivating my skills and interests, for driving my professional and personal growth, and for your compassion, kindness, support, and mentorship throughout the years.

Thank you to the participants in my research who shared their personal experiences, spent their time giving feedback on my work, and are truly the foundation and drive of my accessibility work.

My work was funded in part by the National Science Foundation under awards IIS-0811063, IIS-1053868, IIS-1702751, DGE-1256082, and a Graduate Research Fellowship, the Wilma Bradley Endowed Fellowship in Computer Science & Engineering, a Google Faculty Award, and the Mani Charitable Foundation.

I thank each and every one of you who helped me reach this point.

Chapter 1. INTRODUCTION

"There's an app for that!" Over a decade later, Apple's 2009 iOS advertising slogan [104] still reflects the ever increasing and integral functionality provided by mobile applications (apps). Spanning finance, communication, transportation, education, and beyond, apps touch nearly every aspect of daily life.

Unfortunately, apps can fail to be fully accessible for people with disabilities or people who use assistive technologies[1]. Accessibility failures include buttons that are too small to tap accurately, images that lack labeling for screen readers, and apps that are completely unresponsive when used with assistive technology. Some legislation [105] and company statements [26,83] promote the importance of accessibility. Despite these efforts, accessibility failures continue to surface, including in essential apps for banking (e.g., Wells Fargo [100] and BECU [106]), social media (e.g., Instagram and Facebook [42,107]), health [64], and transportation (e.g., Greyhound [108]). The severity and impact of app inaccessibility has driven public outcry and lawsuits [106,108]. As one blind person noted, using an inaccessible app is "a constant feeling of being devalued. It doesn't matter about the stupid button that I can't press in that moment. It's that it keeps happening. … And the message that I keep receiving is that the world just doesn't value me" [42].

Each individual app with an accessibility failure can be a problem. However, apps do not exist in isolation. They are a product of the ecosystems in which they are designed, built, maintained, and used. Inaccessibility can propagate across apps through factors such as code reuse, the same developer or company making multiple apps, and popular design patterns

[1] To reflect a range of identities and preferences, I use person-first (people with disabilities) and identity-first (disabled person) language interchangeably. I mention both disabled people and people who use assistive technology to reflect that not everyone who identifies as disabled uses assistive technologies and *vice versa*.

being copied across apps. An approach to app accessibility that considers these factors and their impact across apps can enhance our understanding of the state of app accessibility and inform efforts to improve app accessibility.

Fortunately, other areas of study provide a model for such multi-factor, population-level analyses. One area is epidemiology, which examines health on a population level from a systems perspective. Where a physician treats a single patient, an epidemiologist considers the health of a population. Analogously, a designer or developer might address the inaccessibility of an individual app, but a population perspective on app accessibility might reveal the causes of systemic problems and suggest potential solutions. Example factors from my work include developer use of a game engine that creates inaccessible apps, app design patterns associated with accessibility failures, and social dynamics between app developers and professional accessibility testers.

My dissertation draws on my epidemiology-inspired framework to identify population-scale trends and ecosystem factors impacting app accessibility. Situated within this framework, my dissertation research builds our understanding of app inaccessibility and informs designing tools to improve app accessibility.

1.1 Thesis Statement

I pose the following thesis:

> *In mobile app accessibility, applying an epidemiology-inspired framework that emphasizes multi-factor and large-scale analyses can:*
>
> *(T1) reveal population-level trends of accessibility failures;*
>
> *(T2) aid in identifying a range of intrinsic to extrinsic factors that can impact app accessibility; and*

(T3) inform the design of tools for identifying and repairing accessibility failures.

In demonstration of this thesis, my dissertation makes theoretical, empirical, and artifact contributions [95]. I first created an epidemiology-inspired framework as a population-scale, multi-factor lens to apply to app accessibility. I then situate my artifacts and empirical results within this conceptual framework.

To enhance our understanding of the state of app inaccessibility, I performed large-scale analyses of Android apps. My results measure prevalence of accessibility failures across apps and identify classes of elements that frequently have accessibility failures. In a multi-factor assessment guided by large-scale analysis findings, I identified accessibility shortcomings in developer libraries and guidelines. These environmental factors could contribute to inaccessibility at scale.

Toward improving app accessibility, I explored tools for repairing accessibility during app development, testing, and after release by third parties. These projects contribute a set of artifacts including a tool prototype for a developer IDE, a testing tool that was later released as a product, and a proof-of-concept for third-party repair. I further performed empirical user studies on some of these tools to understand their value and the structural ecosystem in which they may be used.

Together, my dissertation projects used a range of methods to demonstrate my thesis and work toward my ultimate goal of understanding and improving app accessibility.

1.2 Dissertation Overview

In this dissertation, I discuss background in app accessibility, my epidemiology-inspired framework, findings from my large-scale analyses, technical details of my tool artifacts, and results from testing those tools with relevant stakeholders. I outline each chapter below.

Chapter 2 provides technical and conceptual background on app accessibility. I include an overview of assistive technologies, technical Android details, and accessibility failures.

Chapter 3 covers related research in accessibility and large-scale analysis. This includes systemic approaches to accessibility, web accessibility, app accessibility analyses, integrating accessibility in software engineering, and large-scale app analyses.

Chapter 4 details my epidemiology-inspired framework. I map key terminology and objectives from epidemiology to app accessibility. In particular, the framework emphasizes the value of large-scale and multi-factor analyses to understand and improve accessibility. I apply the concepts from my framework throughout my dissertation.

Chapter 5 presents my work toward understanding the state of Android app accessibility. In the first section, I present my large-scale analyses to identify the prevalence of a set of inaccessibility diseases (i.e., accessibility failures). Toward my thesis, these analyses reveal trends in inaccessibility across apps and within commonly used element classes. In the second section, I identify environmental factors that impact app accessibility at scale. For example, I present official Android example code that produced unlabeled buttons and identify cross-platform frameworks and game engines that produce highly inaccessible apps. The work in this chapter provides key contributions in (T1) revealing population-level trends in accessibility failures and (T2) aiding in identifying factors that can impact app accessibility.

Chapter 6 focuses on improvements to app accessibility through tools at various stages of app creation: development, testing, and third-party modification. I first explore designs for enhancing developer tools and prototype an extension to Android Studio. I then present my extension to the Android's Accessibility Scanner to support human annotation of automated testing and gather feedback from a professional accessibility testing team. Finally, I present interaction proxies, a proof-of-concept technique for third-party app accessibility enhancements. In addition to the tool artifacts, I apply my epidemiology-inspired framework

to highlight where the large-scale analyses informed tool design and the multi-factor lens highlights the social and institutional context in which these tools are used. These projects contribute to (T2) identifying factors that can impact app accessibility and (T3) informing the design of tools for identifying and repairing app accessibility failures.

In Chapter 7, I discuss how these projects support my thesis claim. I then outline a few avenues of future research suggested by my work. Finally, I reflect on my research experience, including discussing my experience applying the framework.

Chapter 8 concludes my dissertation.

Chapter 2. App Accessibility Background

A core goal of my research is to improve app accessibility, particularly for people with disabilities or who use assistive technology. I present a brief overview of assistive technologies, technical Android information, and a set of accessibility failures that result from incompatibility between technical Android implementations and assistive technologies. This overview is intended to provide a foundation to understand my dissertation research, especially for those unfamiliar with app implementations or accessibility.

2.1 Assistive Technology

Assistive technologies enable apps to support more types of interaction to better meet people's abilities and preferences. To complement the predominant interaction scheme of visually consuming information and interacting through taps on touch screens, assistive technologies use software, hardware, or both to enable additional interaction options. Interactions my use a range of modalities such as touch, audio, voice, eye gaze, external keyboards or buttons, or braille devices. These technologies can be highly personalized or general, including common features like pinch-to-zoom and voice assistants. I will detail a few specific Android assistive technologies that were central to my research but there is a wide range of assistive technologies.

2.1.1 TalkBack Screen Reader

Screen readers turn visual information into audio. This may be useful for people who are blind, have low vision, have learning disabilities, have situational impairments that prevent high visual engagement (e.g., working out), or otherwise benefit from audio feedback. Android devices ship with the TalkBack screen reader. The two main navigation techniques are Explore By Touch or Linear Navigation. In Explore By Touch, a screen reader will announce the content under the person's finger. With Linear Navigation, someone uses a swipe gesture

to advance TalkBack through the screen. Other gestures such as double tapping allow clicking buttons, entering text, and other interaction.

2.1.2 Switch Access

Switches are external devices, often shaped like large buttons, to supplement or replace interacting with a touch screen. These can be particularly helpful for people with fine motor impairments. Switches can be used in various numbers and configurations. For example, someone may press a switch to advance through a screen and then long-press to select. Switch Access is the software component on Android that supports switch-based interactions. In one setting, Switch Access may advance through elements in a Linear Navigation order (similar to TalkBack). Another setting creates a cross-hair cursor that pans the screen in two stages; selecting a location requires two clicks on a switch, stopping the cursor for a vertical and horizontal location selection. Specifically relating to my dissertation research, Switch Access relies on TalkBack-Focusable elements (explained below) and Linear Navigation.

2.1.3 Voice Access

Voice Access supports fine-grain, screen-by-screen app interaction through voice commands. In the Numbers setting, VoiceAccess visually labels each clickable element with a number (Figure 1). People can then speak a number aloud to interact with the corresponding element. For example, in the meal tracking app in Figure 1, someone could say “tap 27” to add a breakfast food log.

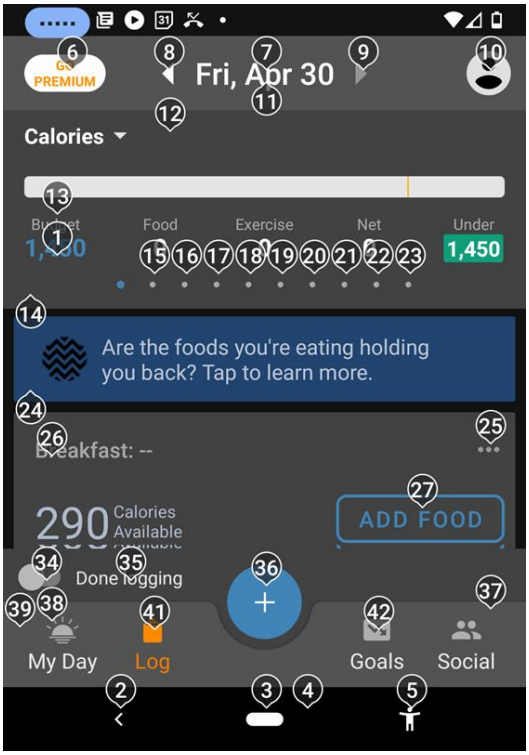


Figure 1: A cropped screenshot of a meal tracking screen in the Lose It! weight management app. In the Numbers setting, VoiceAccess displays numbers for all interactive elements on a screen. Speaking a command such as "tap 27" will activate the associated element, in this case clicking the Add Food button.

2.2 Android Background

This section presents a brief technical background on how Android apps are implemented and how they interact with assistive technologies. I define key terms used throughout my work: app *category*, *element*, *class*, *view hierarchy*, and *TalkBack-focusable*. When uploading an app to the Google Play Store, an app creator can choose one *category* from a pre-established list such as Education, Communication, or Weather. APKs are Android's file package for the installing and running apps. It is similar to an .exe file on Microsoft Windows.

2.2.1 Elements and Classes

Elements are the building blocks of app interfaces. Some types of elements are used to define the visual structure (i.e., layout) of a screen, such as the arrangement of child elements into

rows or columns. Other types of elements are the primary visual and interactive content of a screen, such as its buttons, text boxes, and images.

The *class* of an element refers to a packaged implementation of the element that can be re-used. For example, the Android API class `android.widget.ImageButton` provides an implementation of the basic visuals and skeleton functionality of an image button. The Android API provides many classes for commonly used layouts (e.g., `android.widget.ListView`) and visual elements (e.g., `android.widget.Button`). Developers can create their own custom classes built on the base API. Third parties can also supply classes for creating elements, such as Facebook's Login button (discussed in Section 5.3.4).

2.2.2 View Hierarchies and Screen Representations

Screens are composed of many elements, often nested within one another. In the source code implementation, screen elements can be statically defined using XML-style layout files or dynamically generated. For example, in a food ordering app, a screen's taskbar buttons (e.g., back, home, cart) and list layout element may be defined in the layout file while the list contents (e.g., the restaurants) are generated dynamically at runtime.

A *view hierarchy* is a representation of all of a screen's elements and their nesting, named after the base Android class `android.view.View`. The view hierarchy also captures *attributes* about each element. These attributes include the `class`, the location on the screen, if it is clickable, if it is visible, and its `contentDescription` (needed for labeling, as discussed in Chapter 2). Accessibility Services, such as assistive technologies, runtime scanners, and my developer tool (Section 6.1) rely on a runtime-constructed *accessibility view hierarchy* to provide necessary information about screen content to the service.

Minification techniques [109] reduce the size of an app by shortening the names of classes, functions, and other code. This practice obfuscates information captured in the view hierarchy. For example, a button element from the class `android.widget.Button` could have its class

attribute obfuscated to a value `z` in the view hierarchy. The mapping of obfuscated class names to original class names depends on obfuscation technique and cannot be determined from the app APK alone.

2.2.3 TalkBack-Focusable

Assistive technologies must determine which elements should receive focus (e.g., text to read, interactive buttons) versus which should be ignored (e.g., used for layout, in a hidden tab, behind a pop-up dialog box). I describe elements that are visited by Android's TalkBack screen reader as TalkBack-focusable. Many of my large-scale analyses use TalkBack-focusable elements as a proxy for elements important to assistive technologies. I chose TalkBack as the reference assistive technology because it is one of the core services offered by Android and the logic used by TalkBack is the foundation of many other assistive technologies (e.g., Switch Access).

2.3 Accessibility Failures

This section defines accessibility failures I explore in my dissertation research. Testing methods are detailed in each project section. Common sources of app accessibility testing criteria are the Web Content Accessibility Guidelines (WCAG), industry-released guidelines, and Section 508 of the Rehabilitation Act of 1973, legislation in the United States which legally mandates that certain government-related technology must be accessible to people with disabilities. Most of the accessibility failures I test for are based on similar guidelines.

I describe common accessibility failures referenced throughout my work: few TalkBack-focusable elements, missing labels, duplicate labels, uninformative labels, editable TextViews with `contentDescriptions`, fully overlapping clickable elements, undersized elements, and low color contrast. This section presents an overview of each failure. Details of how each failure was tested for or addressed are detailed in the future sections, as applicable.

2.3.1 Few TalkBack-Focusable Elements

Having one or zero TalkBack-focusable elements on a screen is likely an instance of the screen not properly exposing its elements to assistive technologies. An app screen with one or zero TalkBack-focusable elements is functionally unusable with many assistive technologies, equivalent to interacting with an unresponsive blank screen. Examples of screens with few TalkBack-focusable elements are presented in Figure 2.

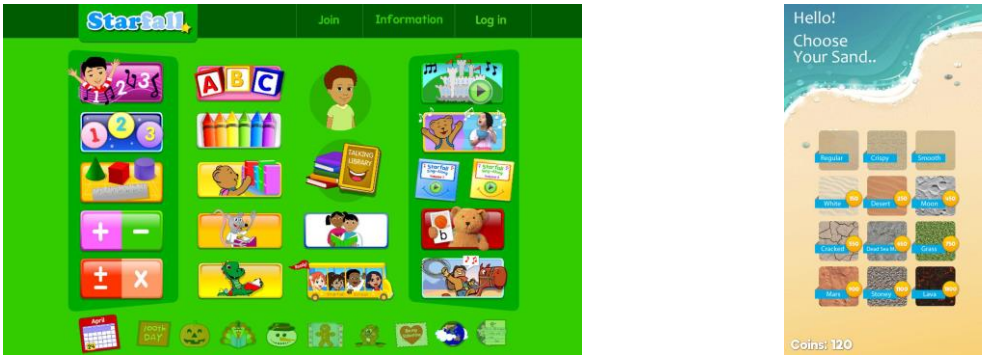


Figure 2: Example apps with the few TalkBack-focusable elements failure. (left) The Starfall app has zero focusable elements per screen despite being full of evident targets and information. (right) The Sand Draw app has one focusable element per screen despite showing text boxes and a grid of 12 targets.

Automated tests currently do not distinguish between problematic screens and screens that may legitimately have only one TalkBack-focusable element, such as a loading screen with only a progress bar or an empty list. However, most designs have more than one element and screens with one or zero TalkBack-focusable elements likely contain a set of non-focusable but essential elements. Future work could explore more robust testing techniques (e.g., computer vision or crowdsourcing) to more accurately identify apps that do not properly expose their screens.

2.3.2 Label-Based Inaccessibility

Screen readers use element labels to announce what an element says or represents. For text-based elements, TalkBack can directly use that text for the label. With image-based

elements, an additional label source is needed, akin to alt-text for images on the web. Most image-based elements are labeled with the `contentDescription` attribute or inherit a label from a child element.

I look at four label-based accessibility failures: *missing label*, *duplicate label*, *uninformative label*, and *editable TextView with contentDescription*.

2.3.2.1 Missing Label

Unlabeled elements create major accessibility problems in apps. When focusing on elements with missing labels, a screen reader may announce an unhelpfully vague label (e.g., "unlabeled button", "button") or nothing at all.

2.3.2.2 Duplicate Label

The presence of multiple clickable elements on a screen with the exact same label may be confusing to people using a screen reader. Examples of problematic duplicate labels are presented in Figure 3. There are instances of legitimate duplicate labels in apps (e.g., a list of songs in a music app where all the authors are labeled "Unknown"). My techniques do not distinguish between legitimate and problematic duplicate labels.

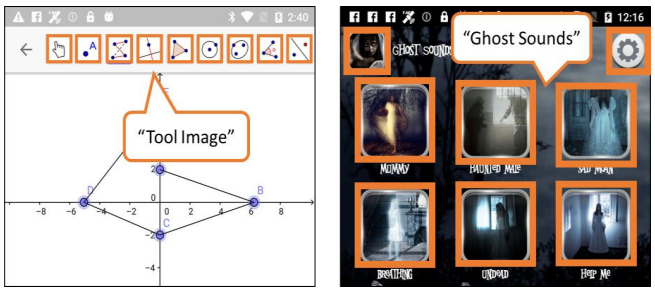


Figure 3: Two example app interfaces with duplicate labels on image-based buttons. (left) The Clickable Image buttons for drawing different types of figures in a graphing app are all labeled "Tool Image." (right) In the Ghost Sounds app, all of the Clickable Image buttons for playing different ghost sounds as well as the settings and home button are labeled "Ghost Sounds."

2.3.2.3 Uninformative Label

If a developer adds labels to elements, it is crucial the labels are meaningful; an element labeled "image" can be as significant of an accessibility barrier as an unlabeled element. Label accuracy is also important (i.e., whether a button labeled "back" actually functioned as a back button). Testing inaccurate labels was outside of the scope of my work due to the challenges in automatically determining the function of an image and comparing that to a text label.

2.3.2.4 Editable TextView with ContentDescription

Editable TextView elements allow a person to enter text (e.g., typing text into a search bar) and TalkBack should announce the entered text. A `contentDescription` label can interfere with that functionality; TalkBack may announce the `contentDescription` instead of any entered text. Therefore, adding a `contentDescription` for an editable TextView is an accessibility failure. Instead, the `hint` attribute can be used to label an empty editable TextView, akin to a visual text prompt appearing in a textbox before a person begins entering text.

2.3.3 Fully Overlapping Clickable Elements

Clickable elements that fully overlap make it challenging to activate an occluded target. If the fully overlapping elements perform different actions when clicked, it is impossible to use both functionalities. However, even fully overlapping clickable elements that perform the same functionality can cause problems for assistive technologies such as VoiceAccess. If two clickable elements are fully overlapping, they may both receive a distinct number in the Numbers setting. The extraneous number labels cause visual clutter and confusion. In the left example in Figure 4, someone could say "tap 21" or "tap 22" to go to the map view. The problem is further elevated when more than two clickable elements fully overlap. For example, in the right app in Figure 4, someone has seven numbers they can verbally "tap" to activate the same image. Fully overlapping elements with the same functionality can occur if a developer accidentally codes an element and its parent to be clickable with the same action.

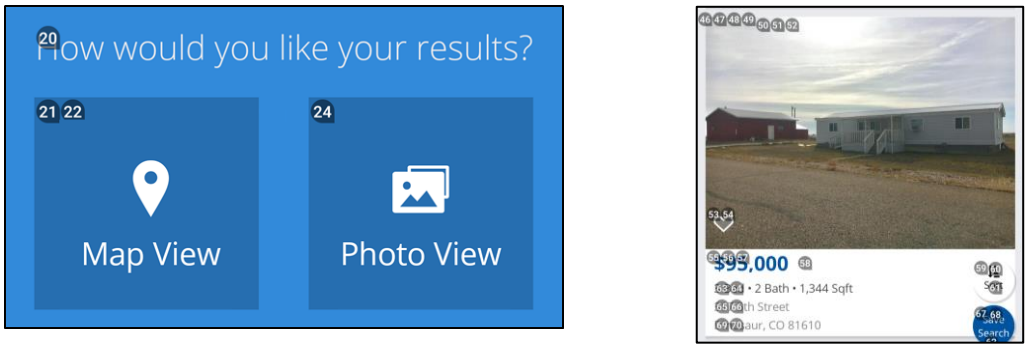


Figure 4: Apps with VoiceAccess turned on. Each number represents an interactive element. (left) Two fully overlapping clickable elements with the same functionality are notated as 21 and 22. (right) Many overlapping clickable elements add substantial visual clutter and confusion.

2.3.4 Size-Based Inaccessibility

Interactive elements can be too small for people to accurately touch. This failure can be particularly significant for people with motor impairments or for people using an Explore By Touch screen reader technique that announces elements as a person moves a finger around the screen. The Google Accessibility Guidelines for Android [45] suggests a minimum size of 48*dp* × 48*dp*.

I define the overall size-based failure (1) *too small in either* as elements that are not tall enough, not wide enough, or both. To explore nuances between failures to make elements large enough in different dimensions, I break down the barrier by dimension into (2) *too small in both*: elements that are not tall enough and not wide enough; (3) *too short only*: elements that are only not tall enough; and (4) *too narrow only*: elements that are only not wide enough. The prevalence of *too small in either* failure is the sum of the other three dimension-specific barrier prevalence measures.

Developers can use several techniques to create elements that meet the size recommendations. The most obvious approach is creating elements that are themselves large enough. However, enlarging elements may conflict with a desired visual appearance. An alternative is to place a visible element inside of a larger, invisible container and to set the

larger container as the clickable element [2]. In these cases, the invisible container is TalkBack-focusable, whereas the contained visible element would not be. Section 6.16.1 details resizing techniques in more detail.

2.3.5 Color Contrast

Certain colors and color combinations make screen content more readable and understandable. The effect of color may be particularly important for people with vision impairments or people in extreme lighting conditions such as direct sunlight. Guidelines provide suggested minimum color contrast between foreground color (e.g., text color) and background color. The exact contrast threshold depends on features of the visual content like text size, bold style, font, and outlining.

2.4 Chapter Summary

This overview of app accessibility cannot capture the complete, personal, and nuanced complexities of defining an accessible app experience that meets everyone's unique needs, abilities, preferences, and goals. Rather, I present it as a foundation for understanding the assessments and discussions of app accessibility presented in the following chapters.

Chapter 3. RELATED WORK

My dissertation contributes large-scale multi-factor analyses of app accessibility and explores techniques to improve app accessibility. I present related research that my dissertation work is situated within; both work that informed my approach and later work that complements my research. Key areas my dissertation fits within are approaches for evaluating accessibility, assessments measuring the state of accessibility in apps and on the web, explorations of the ecosystem of factors that impact accessibility (e.g., software engineering practices, development APIs), efforts to improve app accessibility, and work on using apps as a tool to increase the accessibility of the physical world.

3.1 Evaluating Accessibility

My app accessibility assessments are informed by prior work evaluating the accessibility of apps and websites. I overview how prior work defined accessibility failures and comparisons of evaluation techniques.

Most prior app accessibility assessments based their accessibility tests on adapted versions of Web Content Accessibility Guidelines (WCAG) [27,29,33,79,80] industry-related guidelines [64], and Section 508 [64]. A range of techniques have been used to test app accessibility. including manual testing [29,64], automated tools [12], and hybrid approaches combining multiple techniques [97]. The accessibility failures I test for are largely based on Google's Accessibility Test Framework for Android, which covers similar accessibility considerations to prior work and is tailored to Android apps.

Some work evaluated the quality of evaluation techniques. Vigo and Branjnick [92] surveyed and assessed different metrics for evaluating web accessibility and provided a framework to assess the value of evaluation metrics. Studies analyzing the success of Web guidelines suggest that guidelines are not sufficient for ensuring full accessibility due to a range of factors

including lack of developer knowledge, difficult to implement recommendations, difficulties testing for adherence, or the mismatch between actual user concerns and guideline recommendations [29,63,64]. Haleas [50] describes the impact of poor guideline applicability centering the challenge for companies to avoid lawsuits related to inaccessible websites given the ambiguity in guidelines and when they are legally applicable. While Haleas acknowledges the societal good of creating accessible websites, the piece focuses on the negative impact on companies ADA "lawsuit chasers" have when legislation provides no concrete criteria for compliance. This perspective further emphasizes the need for robust, accessibility evaluation that captures accessibility and usability, culture shifts in companies, and tools that integrate accessibility into development practices. Legislation is a powerful tool and guidelines can strengthen that power. However, legislation-based approaches to improving accessibility can be challenging due to potentially creating an adversarial attitude toward accessibility and legislation being slow to adapt to rapidly-changing technologies.

Similar limitations were found in mobile app guidelines. Milne *et al.* [64] found many failures they identified in their study of nine iOS health apps that were not covered in Apple's accessibility guidelines, which focus primarily on individual elements versus interactions between elements. Clegg-Vinell *et al.* [29] found app failures identified by app accessibility consultants with disabilities were either not fully covered in WCAG or the severity of the failures did not correspond to guideline accessibility levels. Antônio Mateus *et al.* [12] compared two automated accessibility testing tools (e.g., MATE [110] and Android's Accessibility Scanner [44]) with assessments by people who were blind or had low-vision. They found some overlap in failures identified but each technique found accessibility failures other techniques did not. Acknowledging the limitation of automated tools, Silva *et al.* [81] found only 12.5% of accessibility features in guidelines were covered by all automated app accessibility assessment tools combined.

This work emphasizes the importance of acknowledging limitations in accessibility testing techniques and maintaining the role of human testers, both accessibility experts and people with disabilities. In my dissertation research, I ground my understanding of accessibility failures through user studies, reading blog posts, and consulting experts. I primarily used automated techniques in my analyses to support the large scale of my app accessibility assessments (presented in Chapter 5). Toward improving assessment tools, I explore combining human expertise and automated assessment in Chapter 6.

3.2 Measuring the State of Accessibility

The web has a long history of accessibility analyses and enhancement. Hanson *et al.* [51] performed a longitudinal study of 100 top government and commercial websites over 14 years. Findings include that, overall, accessibility improved over time. In follow-up work, Richards *et al.* [70] discussed potential contributing factors such as changes in web coding practices. Kane *et al.* [60] performed manual and automated accessibility analyses of 100 university websites. Their results indicate the continued impact of inaccessibility on the web as well as potential contributing factors that include the university's country and legislation. Such work highlights the value of characterizing accessibility through large scale and longitudinal analyses and exploring influential factors in accessibility through data.

Other research assessed mobile app accessibility. Some work focused on specific types of apps such as health [37,64,98], smart cities [27], and government engagement [79]. Other work used more generalized samples of apps [1,12,29,69,91,97]. Prior studies primarily assessed Android apps [1,12,27,91,97] with some coverage of iOS [64,69,99] or mixed platform [29,79]. My analyses sample Android apps due to their high use [111] and the openness of the platform. Future work comparing the accessibility of the iOS and Android apps would be beneficial in identifying platform-based factors that can impact accessibility.

Many studies had smaller sample sizes in the four to ten apps range [27,64,79]. My large-scale analysis of 10,000 apps [74,75] was the first app accessibility assessment of that scale published, these studies complement my work. More recently, a few large-scale assessments have been published studying in the range of 479-13,000 apps [1,91,97]. Early, scoped studies informed my choice of accessibility failures to focus on and motivated the continuing need to study app accessibility. Later work complements my assessments by validating some failures as highly prevalent, using different metrics to provide additional insights, and focusing on different subsets of apps.

App assessments identified frequent and problematic accessibility failures including elements with missing or poor labels [1,27,29,37,64,79,91,97,99], elements not focusable with assistive technology [27,29,69,97,99], elements too small in size [1,29,37,97], and elements with colors too low in contrast [1,27,29,79,97].

More recent large-scale analyses complement my analyses to enrich our population-scale understanding of app accessibility [1,91,97]. Yan and Ramachandran [97] assessed 479 free Android apps for a set of accessibility barriers. Their analysis contributes a detailed description of an app accessibility assessment tool, metrics for capturing the state of app accessibility, and the frequency of a set of accessibility barriers. Vendome *et al.* [91] tested 13,000 Android apps hosted on Github for use of accessibility APIs and for labeling of images and image buttons. They found missing labels were highly prevalent and only 3% of apps imported accessibility APIs. Alshayban et. al [1] assessed approximately 1,000 top Android apps for the prevalence of 11 accessibility failures, trends in accessibility of apps over time, and relationships between accessibility and app characteristics such as rating, popularity, or category. I compare the results of these studies to my large-scale analysis in more detail in Section 5.4.

3.3 Factors Affecting App Accessibility

In addition to measuring how frequently accessibility failures occur, it is important to understand why failures happen at individual and systemic levels. My research fits into a space of understanding the wide range of factors that can impact app accessibility.

Software engineering is one field that has explored a range of factors impacting accessibility in software engineering practices. In a systemic literature review of accessibility and software engineering practices, Pavia *et al.* [68] found most work focused on design and testing phases while accessibility and establishing software processes is a more newly emergent space. Considering systemic influences, Bi *et al.* [15] interviewed and surveyed software practitioners on how their personal understanding, work characteristics, and organizational factors impacted mobile and web app accessibility. They found, overall, accessibility is not well integrated into projects; the focus is often on lightweight criteria like color contrast and only 30% of practitioners had limited accessibility-related work experience. The authors noted barriers to improving accessibility included a belief that accessibility practices were challenging or needed high expertise, a lack of organizational factors (e.g., time, budget, access to experts, company support), a belief that underlying platforms and frameworks manage accessibility, difficulty collecting requirements, competing priorities and timelines, and a belief their company is too small to have the resources to prioritize accessibility. The authors distill these insights into recommendations for addressing challenges to improving accessibility at the organizational, people, process, and practice levels. The range of factors Bi *et al.* put forth fits well within my epidemiology-inspired framework.

Vendome *et al.* [91] analyzed 366 StackOverflow posts of developers asking questions related to Android Accessibility. Insights indicated factors that impact developer ability to follow accessibility practices. The authors noted posts predominantly focused on vision impairments and screen readers; the authors theorized this may be due to many automated tests and

resources focused on vision-related accessibility features. This work reported other challenges developers face working within the Android ecosystem including not understanding the output from Lint tests and seeking techniques to disable or interpret results; challenges with encoded accessibility breaking with updates to the Android API; and compatibility problems between third-party frameworks and mobile web apps. The authors suggested future work needs to integrate more accessibility testing into IDEs to improve awareness, that tools should provide additional scaffolding for accessibility-novice developers, and that focus needs to expand to forefront more diverse accessibility needs.

Alshayban *et al.* [1] explored a variety of factors impacting app accessibility. Based on their large-scale app accessibility analysis, they found developers tended to have similar types of failures across multiple apps they build; overall, inaccessibility failures were similar between apps from the well-known and lesser-known companies; and app ratings were not associated with accessibility. Surveying 66 Android developers revealed lack of awareness as a top reason for introducing accessibility failures and a perception that the developer's company did not value accessibility as importantly as other attributes like security.

Neves da Silva [82] examined the relationship between app accessibility failures and the Android accessibility API, such as the `contentDescription` element attribute and the `setAccessibilityLiveRegion` method. Demonstrating a lack of accessibility infrastructure on the platform, they found only 27% of WCAG accessibility criteria are covered by the Android API. When looking at developer use of the API, they found 31% of apps used no accessibility API code. When apps did use accessibility-related API code, it tended to be general attributes, such as text size, that impact but are not strictly accessibility focused versus accessibility-specific code.

This set of work illustrates the complexity of factors that impact developers attempting to integrate accessibility into their practice and fits well within my epidemiology-inspired multi-factor framework of app accessibility. The research further complements my work

identifying environmental factors and supports the necessity of improving tools and IDEs, such as my work presented in Chapter 6.

3.4 Improving Accessibility

Repairs for web accessibility failures can leverage the DOM, a readily available and modifiable site representation (e.g., by a browser extension). Some repairs are automated, such as attempting to automatically identify sources of alternative text for web images [18] or automatically increasing font size to improve readability [16]. Other approaches emphasize social annotation, wherein people contribute accessibility repairs that benefit additional people who encounter the same accessibility failure [86]. Such research has examined collaborative authoring of missing image alternative text and other metadata [78,86,87], sharing of scripts to implement site-specific repairs [19], and crowdsourcing contributions [52].

To move web accessibility tools from compliance to usability, Takagi *et al.* [86] created a tool called Accessibility Designer for accessibility-novice designers and developers. The tool combined error notifications with usability visualizations that overlayed the website with colors to indicate the reachability of site content with a screen reader. Putting the tool to practice when designing an internal website, the authors noted the tool helped accessibility-novice designers and developers more easily grasp accessibility problems, contribute to repairs more efficiently, and reduced the demand on accessibility experts; this overall reduced the time and cost of improving website accessibility. The resulting website received positive feedback from blind people. I explore visualizations and testing techniques for app accessibility in Chapter 6.

For people with motor impairments, an accessibility repair may require modifying how they manipulate an interface. Some approaches re-reorganize interface elements according to personal motor abilities [39,94]. Other systems apply various techniques to ease pointing. Examples include making targets "sticky" [53,96], dynamically modifying cursor behavior

according to a person's movement profile [88], or breaking a pointing interaction into multiple interactions that disambiguate the intended target [39,55,102]. Other research has explored alternative pointing in physically large devices [59].

As discussed above, hardware and software assistive devices can support a range of interaction techniques. Slide Rule introduced techniques for re-mapping gestures to support navigating and exploring the screen separately from activating targets in an interface to support screen reader navigation on touch screens [57]. These techniques contributed to the development of screen readers for major mobile platforms (e.g., Android's TalkBack, iOS's VoiceOver), and now improve accessibility of apps on those platforms. Zhang *et al.* [101] also explored improving interaction with screen readers through a physical overlay. Other techniques for supporting more accessible mobile interaction include pointing enhancements [102] and macros [71]. The Action Blocks app from Google uses macros to support people with cognitive disabilities [112]. The SWAT framework examines system-level instrumentation of content and events to support developer creation of accessibility services [72], but requires rooting a device, which is a significant security risk and presents a technical expertise barrier. In more recent work, Zhang *et al.* [99] created Screen Recognition, an extension to the iOS VoiceOver screen reader that attempts to automatically repair accessibility failures on apps at runtime based on screenshots. Based on a model trained on annotated app screenshots, Screen Recognition exposes elements that were not focusable (akin to the few TalkBack-focusable elements failure explained in Section 2.3) and generates metadata to support VoiceOver (e.g., navigation order, grouping elements, UI content). Our work on third-party accessibility services (Section 6.3) fits within this space of runtime repair that supports existing assistive technologies.

Mobile devices can also be tools for improving access in the physical world [58]. VizWiz examines opportunities for supporting blind people in using a mobile device to take a picture and ask a crowd a question about that picture [17,23]. Extensions have examined

conversational interaction with the crowd [61] and soliciting volunteer assistance through social networks [23,24]. Related insights have also been applied to supporting deaf students through real-time captioning for classroom activities [61]. Google's Live Caption on Pixel devices [113] similarly attempts to provide real-time captioning for in-person conversations. Commercial apps such as Microsoft's SoundScape [114] provide rich audio feedback during eyes-free navigation (such as announcing points of interest and intersections). Microsoft's Seeing AI [115] and Google's Lookout [116] use artificial intelligence to provide audio feedback on visual surroundings such as reading signs, identifying money, and describing scenes.

Chapter 6 of my dissertation presents how I contribute to improving app accessibility through developer tools, testing tools, and third-party repair.

3.5 Large-Scale App Analyses Beyond Accessibility

Beyond accessibility, large-scale analyses have helped us understand features of apps including security vulnerabilities, use patterns, and popular designs. This prior work demonstrates insights that can be gained from such analyses and provides further motivation for designing and leveraging population-level analyses for accessibility.

Large-scale analyses of app usage reveal patterns of usage in terms of duration, app category, and context (e.g., time of day, location) [22,38]. The results of these studies aim to inform app design [38] or to provide a basis for an app recommendation system [22]. This type of analysis has yet to be applied with an accessibility focus but could be insightful. For example, difference in time spent in communication apps between those who use a screen reader and those who do not could suggest an accessibility failure.

Population-level analyses have also been used to explore the interactions between apps and more extrinsic factors through analyzing code reuse [65], design reuse [36], and widget and

layout popularity [76]. For example, the insights from a study of code reuse [65] support the interdependence of apps by highlighting the high frequency of code reuse between apps. The ERICA project [36] similarly analyzed interface designs and user traces with the aim of informing future designers. App interdependence is a key component of my epidemiology-inspired framework and motivates extending analyses to include the impact of this code reuse and popular design on accessibility.

Computer security uses large-scale studies to understand the prevalence of security vulnerabilities, their sources, and how they spread. In work that is perhaps most similar my epidemiology-inspired framework, Gil et. al. [40] propose a "genetic epidemiology approach to cyber-security" using large-scale, automated analyses to create tools to determine the probability of a network being susceptible to a threat. The authors focus on computer networks and genetic mutation detection concepts from genetic epidemiology. My epidemiology-inspired framework utilized many more concepts from general epidemiology applied to mobile app accessibility.

Chapter 4. EPIDEMIOLOGY-INSPIRED FRAMEWORK

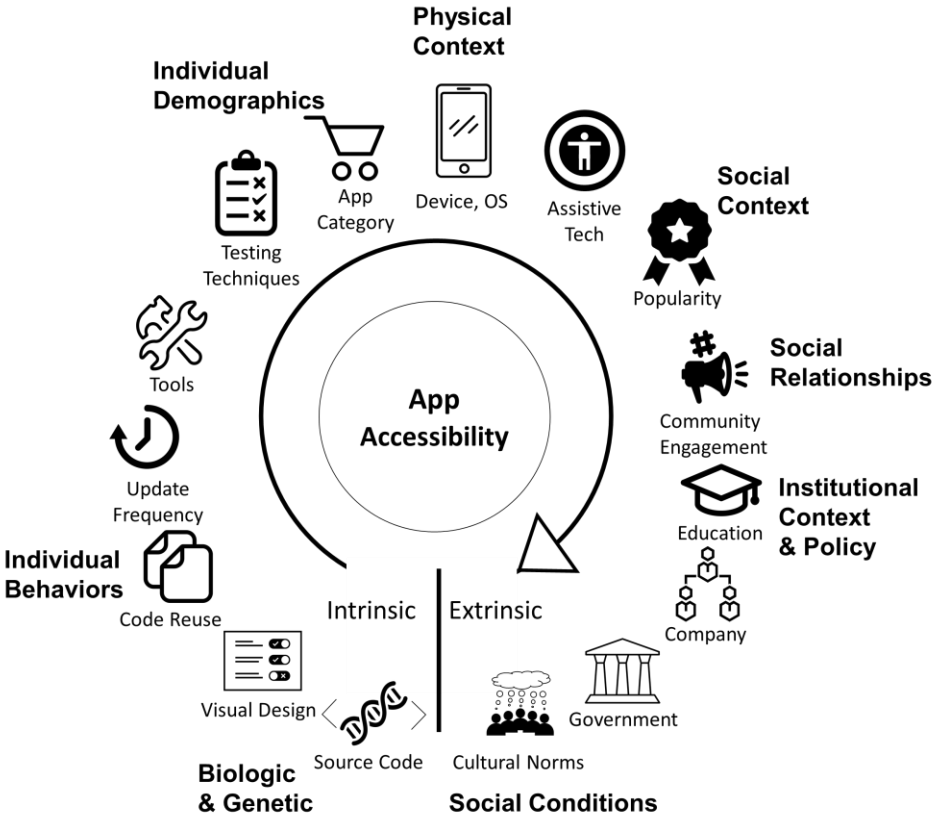


Figure 5: As a systems science, epidemiology can serve as a metaphor that changes the way we think and work with mobile app accessibility. Here, I apply the concept of a multi-factor ecosystem from epidemiology to mobile app accessibility. An app's accessibility is a product of many factors ranging from individual and intrinsic to population-level and extrinsic. These factors include: source code and design, behaviors, demographics, physical context, social context and relationships, institutional context and policies, and cultural norms. Accessibility is affected by factors at all levels. Figure inspired by [93]**.**

In this chapter, I introduce an epidemiology-inspired framework for app accessibility. I emphasize how my model reflects the social model of disability [67]; app accessibility is a community responsibility. My framing places the responsibility of accessibility on the app and guides how different stakeholders can collectively contribute to app accessibility. Moreover, I emphasize the language of my framework is applied to apps, not people. I discuss tensions with applying medical language to accessibility work in Section 7.4.1.

As accessibility continues to surface as a priority in research and industry, it becomes increasingly beneficial to have a conceptual framework to guide thought and action. Conceptual frameworks provide a shared vocabulary to ground discussion, guide efforts to improve accessibility with known strategies, and illuminate previously unconsidered opportunities. I believe the large scope of concepts in my framework is indicative of the framing's richness and of its potential to inspire and inform thought and action.

Adapting a model from epidemiology [93], Figure 5 illustrates many factors that influence app creation, distribution, maintenance, and use. These factors range from *intrinsic factors* that are tightly tied to an individual app to *extrinsic factors* that indirectly impact larger app populations. Example factors, listed from intrinsic to extrinsic, include source code, visual design, development and testing tools, operating systems, assistive technologies, app popularity, company and government policies, and public opinion. As this framing exemplifies, apps do not exist independently of one another or of their environments. A natural extension is to recognize that neither do their accessibility strengths or weaknesses. Understanding how factors interact and influence app accessibility can inform treatments to improve app accessibility.

We need multiple levels of analyses to understand how a range of factors affect app accessibility, from individual entities to populations at a specific moment and over time. Many well-established scientific disciplines have benefitted from longitudinal, population-level analyses, such as ecology [89], oceanography [62], and computer security [21]. I chose epidemiology [49] as the core metaphor for my app accessibility framework. While no metaphor is perfect, I claim app accessibility can benefit from epidemiology's well-developed approach to collecting, analyzing, and acting on longitudinal, multi-factor, and population-based data.

This chapter introduces my novel epidemiology-inspired conceptual framework for monitoring, analyzing, and acting on longitudinal, multi-factor, and large-scale app accessibility data. I

first published framework in the ACM SIGACCESS Conference on Computers and Accessibility (ASSETS) in 2017 [73] with co-authors Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock and significant feedback from Anat Caspi. I led conceptually constructing the framework and collaborated on the writing the paper, which received a Best Paper Nomination.

My framework highlights wide-ranging intrinsic and extrinsic factors that influence app accessibility, motivates the collection and analysis of large-scale data, and guides opportunities for improving app accessibility. This framework provides a structural foundation to my dissertation work. In this chapter, I present language, objectives, and models mapped from epidemiology to app accessibility. In later chapters, I use this framing to position my other dissertation work.

4.1 Real-World/Grounding Examples

To ground my introduction of the epidemiology-inspired framework, I present two real-world examples of population-level factors that influence app accessibility. The examples present resources that are *carriers* of inaccessibility within Android Studio [117] and Android's Floating Action Button design tutorial [118]. These resources exemplify how factors apart from developer-written source code can affect the likelihood of an accessibility failure.

4.1.1 Android Studio App Designer

Android Studio is the development environment released by Google for coding Android apps. Due to its widespread use, the accessibility of the Android features it provides have a large impact on the accessibility of a large population of Android apps.

Android Studio [117] includes a drag-and-drop Layout Editor. The editor provides core widgets and icons for common functionality, (e.g., Image Button widgets; a star icon for "favoriting"). When an Image Button is dragged onto the design interface, Android Studio generates the XML layout code to define the button and basic attributes such as its size and icon. Notably,

the generated code does not include a `contentDescription` attribute, the field a screen reader uses to describe an image. If a label is not added, the app will have an inaccessible button for screen reader users. In Android Studio, when an image button has a missing or empty `contentDescription` attribute, the Lint tests [3] will issue an "image missing label" warning that describes the failure, provides documentation on how to repair the problem, and presents an option to ignore all warnings of this type.

This example illustrates the *transmission* of an inaccessible button *disease* from a *carrier*, the image button, to a *host* app. The *determinant*, or cause of the disease, is the missing `contentDescription`. My epidemiology-inspired framework then motivates an analysis to *evaluate the existing preventative treatment* of the warning to determine if it is sufficient at preventing the *spread* of the disease.

4.1.2 Android Floating Action Button Design

Floating action buttons (FABs) are a part of Google's Material Design for Android, a guide for a more unified design in Android apps [119]. FABs appear in popular apps such as Skype, Gmail, Facebook Messenger, and Dropbox. FABs are potential carriers of inaccessibility. These buttons are typically separate from standard menu bars, "floating" in a visually prominent location such as the bottom-right corner of the screen, and represent the most important action on the screen. Android provides design guidelines for using FABs including how the button should look, act, animate, and function. Since FABs are not anchored to traditional menus, their location in linear navigation orders may be unexpected or inconvenient. Intuitive and efficient linear navigation order is essential when using assistive technologies such as switches, screen reader linear navigation, or accessing a screen with an external keyboard. Moreover, the guidelines recommend designs where FABs animate, move, or switch functionality with changes in app state, which could be difficult or impossible to track non-visually.

This example illustrates the transmission of a second inaccessible button *disease* from a *carrier*, the FAB design, to the *host* app. One *determinant* of the disease is the lack of intuitive integration of the button in linear navigation order. Another *determinant* is the lack of feedback for FAB state transitions. This *carrier* lives in the *reservoir* of Google's Material Design guides.

4.2 Epidemiology-Inspired Framework

Epidemiology regards health as a holistic physical, emotional, and social well-being, not just the absence of illness. Epidemiology acknowledges that an individual's health cannot be understood in isolation but rather is the product of continuous interaction with environmental factors [25]. I use terminology, concepts, and techniques from epidemiology to frame app accessibility in a similarly holistic fashion. A healthy app is an accessible and enjoyable experience for everyone who uses it, not just an app that has no obvious accessibility failures. My framework defines a single app as a potential *host* for one or more inaccessibility *diseases*. A *population* consists of a large group of apps, for example, all apps in existence, all Android apps, all apps that use the Google Map library, all shopping apps, or the Top 100 downloaded apps.

Inspired by epidemiological concepts and the five objectives of epidemiology presented by Gordis [49], I offer a conceptual framework to guide thought and action concerning app accessibility. I present framework terminology in Tables 1-4, drawing analogies between epidemiology and accessibility, with examples.

Table 1: Epidemiology-Inspired Terminology — Describing an App

Term	Epidemiology	Accessibility	Examples
Health	State of complete physical, social, and mental well-being, not just the absence of disease	State of complete accessibility and usability, not merely the absence of obvious accessibility or usability problems	An app has all its buttons labeled but the labels so poorly describe the button functions that the app is almost impossible to use
Disease	A condition that interferes with a vital physiological process	An accessibility barrier	Missing label, low contrast
Host	An organism that can be infected	An app that can have an accessibility barrier	A specific app (e.g., the Yelp app)
Case	An instance of a particular condition	A single instance of an app with inaccessibility disease	An instance of the Toggl app with an unlabeled button
Carrier	An entity that carries and transmits a disease	A component that carriers or transmits disease	The image button from Android Studio (Section 4.1)
Determinant	A factor (entity, characteristic, behavior, or event) that directly influences disease occurrence	The root cause (element, characteristic, code, or design) of an accessibility barrier	The missing content description within the button's source code
Factor	An aspect of behavior, lifestyle, environment, or inherited characteristic that is associated with increased occurrence of a disease	A characteristic of an app or of the ecosystem an app is developed, maintained, and used in, that impact the likelihood of an app having an accessibility disease. Can be *risk* or *protective.*	See Factors and Causation Section 4.2.1
Usual Source of Care	The place a patient usually goes when sick or needing advice about health	The way an app is normally tested for accessibility	Automated tests Blindfolded developer
Diagnosis	The process of determining by examination the nature and circumstances of a disease	The process of determining the existence and cause of a disease	By hand exploration Google Accessibility Scanner
Life Expectancy	Average number of years of life remaining based on an individual, population, and environment characteristics	How long before an app is abandoned based on its risk and protective factors, environment, and characteristics. Can be of development or use	How long app is maintained. Time between download and abandonment

Table 2: Epidemiology-Inspired Terminology — Describing an Inaccessibility Disease

Term	Epidemiology	Accessibility	Examples
Reservoir	The habitat in which an infectious agent normally lives, grows, and multiplies	A harbor for accessibility barriers	Toolkits Design guides
Contagiousness	How capable a disease is of being transmitted by contact or close proximity	The ease at which an accessibility barrier can be transmitted given its host and environment	Highly contagious: An accessibility barrier within core library source code
Natural History of a Disease	The temporal course of disease from onset to fatal termination, remission, relapse, or recovery	The process of an accessibility barrier being introduced, encountered, fixed or ignored, and perpetuated or permanently remedied. May be of use or development	See Section 4.3
Incidence	Measure of the frequency of a new case of the disease occurring in a population over time	A measure of the frequency of new occurrences of an accessibility barrier in a population over time	Number of new cases of buttons with missing labels in the Top 100 apps released in a month
Prevalence	The number or proportion of cases of a disease in a given population	The number or proportion of apps with a particular determinant of the disease in a given population	Number of apps in Top 100 with an unlabeled button
Lethality	How likely is a disease to cause death or complications	How likely is an app to be abandoned due to accessibility barriers	Highly lethal: An log-in button that can’t be activated with a screen reader
Transmission	Any mode or mechanism by which an agent is spread	How an accessibility barrier enters an app	Copy-paste repository code Using a drag-and-drop tool

Table 3: Epidemiology-Inspired Terminology — Describing an App Population

Term	Epidemiology	Accessibility	Examples
Population	A particular group of people (e.g., all doctors, UW students, all humans)	The apps or a group of apps under consideration	Google Play Store Top 100, Banking apps
Census	The enumeration of a population with details including residence, occupation, age, etc.	Enumeration of apps including versions, APK, platform, health status, etc.	The Rico dataset of app screens, metadata, and view hierarchies [34]
High-Risk Group	A group in the population with an elevated risk of disease	A group of apps at elevated risk of having a particular accessibility barrier	Android apps are more at risk for inaccessibility than iOS apps
Outbreak	The occurrence of more cases of a disease than expected in a given area or group over a particular period of time	Occurrence of more cases of accessibility barriers or a particular determinant than expected in a period of time	Significant increase in number of unlabeled buttons in a week
Mortality Rate	The measure of frequency of death in a population during a specified time interval	A measure of how often apps are abandoned, for any reason, during a specified time interval	70% of apps are abandoned within a week of downloading
Herd Immunity	Resistance to an infection of because of a substantial proportion of the population is immune thereby reducing the likelihood an infected person will encounter a susceptible one.	An app's resistance to an accessibility barrier because its ecosystem is dominated by factors that are accessible	Minimizing the number of widgets in Android Studio that introduce accessibility barriers
Health Indicator	A measure that reflects, or indicates, the state of health of people in a population	A measure that reflects, or indicates, the state of accessibility within a population of apps	The number of apps with unlabeled buttons
Diagnostic Criteria	A metric used to determine if someone has a disease	A metric used to determine if an app has an inaccessibility disease	Testing for color contrast using an automated runtime scanner
Detection Bias	Can occur when people with a risk factor are more likely to have a disease detected because of intense follow-up	Can occur when certain apps are more likely to have accessibility barriers detected because of closer scrutiny	Apps built by accessibility-educated developers might be more likely to have early diagnosis of accessibility barriers
Common Source Outbreak	An outbreak that results from a group of persons being exposed to a common disease agent	When there is a common source for an increased incidence of an inaccessibility disease	An OS update that causes widespread inaccessibility

Table 4: Epidemiology-Inspired Terminology — Taking Action to Improve Accessibility

Term	Epidemiology	Accessibility	Examples
Public Health	Systematic collection, analysis, interpretation, and dissemination of ongoing health data to gain knowledge of disease patterns, and to control and prevent disease	Systematic collection, analysis, interpretation, and dissemination of ongoing app accessibility data to gain knowledge of accessibility patterns, and to control and prevent barriers to access	Community reporting by and for people with disabilities about the accessibility of certain apps
Treatment	Techniques to combat a disease	An intervention designed to reduce or eliminate an accessibility barrier or its impact	App developer tools that aid in the detection and remedy of accessibility barriers
Prevention	Treatment measures to prevent disease (e.g., immunization, limiting exposure to risk factors)	Treatment measures that prevent an app from having an accessibility barrier	Screening toolkits Thorough testing
Therapeutic	Measure to treat a contracted disease, reduce its impact on health, or reduce its spread	A treatment that repairs an existing inaccessibility disease	Adding custom labels to buttons
Universal Precautions	Recommendations issued to minimize the risk of transmission of pathogens by health care and public safety workers.	Population-based prevention by giving good structure that all apps should follow to reduce inaccessibility.	Accessibility guidelines Integration of accessibility testing into testing

4.2.1 Identify Factors and Causation

Addressing a disease requires understanding what causes it and what factors make an entity more (*risk factor*) or less (*protective factor*) likely to contract it. The same is true for various *inaccessibility diseases* that arise in apps. Effective *treatments* can be informed by understanding where inaccessibility diseases come from, how they spread, and what factors affect an app's risk.

I define a *factor* as characteristics of an app or of the ecosystem in which an app is developed, maintained, and used, that impacts the likelihood of an app having an inaccessibility disease. There are *risk factors* that increase the likelihood of a disease and *protective factors* that reduce the likelihood. Figure 5 presents a structure for understanding the many factors within the *ecosystem* that impact accessibility. Much of the language and structure of this framing are inspired from the "model for analysis of population health and health disparities" presented in epidemiology [93].

Epidemiology elements move from *intrinsic* to *extrinsic* factors. Intrinsic factors include the core of an individual app (i.e., its source code and design). At the other end of the spectrum are highly extrinsic factors, or those that impact a large number of apps.

Starting at the intrinsic end of the factor spectrum (bottom left of Figure 5), there are metaphorical *biologic* and *genetic* factors—an app's source code and design. Progressing around toward extrinsic factors, we continue into factors that directly impact the biologic characteristics. These factors include *individual behavior* such as code reuse through libraries, copying from repositories or tutorials, frequency of updates, testing techniques, and tools used. Factors such as tools, testing, and code reuse not only reflect what app building strategies are used but also the trust in those strategies. Having high trust in a tool might reduce a developer's personal sense of responsibility for investigating accessibility [15]. An app's *individual demographics* play into these factors as well including app age and category (e.g., travel, shopping, entertainment).

The next section of the spectrum is more extrinsic than intrinsic. Within *physical context*, there is the device an app is running on, the operating system and version, and any accessibility software or hardware being used. These elements have fewer direct integrations with the app's biologic and genetic factors. Yet, an app's accessibility is impacted by how the physical context, source code, and design interact with and support one another. For example,

different versions of a screen reader might vary in their interaction with an app's source code, resulting in different levels of accessibility.

Social context encompasses the popularity of an app and how that popularity can impact the accessibility standards to which the app is held. *Social relationships* cover how active people in the community are in discussing and demanding accessibility and how an app responds to critical feedback.

The most extrinsic factors include those on the institutional and societal level. In *institutional context*, there are education, company, and government influences. Education influences include the education of app creators on accessibility practices, of users on existing accessibility support, of the community on the importance of advocating for accessibility, and of institutional leaders on the importance of prioritizing accessibility.

Company factors can improve app accessibility by dedicating resources to accessibility, choosing internal tools and resources that promote accessibility, creating policies that enforce accessibility in their organization, and influencing popular opinion around accessibility. *Government* factors are similar, but with a cross-organization, regional impact. A government's role in funding allocation, public initiatives, policies, lawmaking, enforcement, and advocacy all contribute to app accessibility. At the extrinsic end of the spectrum is *social condition* which covers cultural norms and public expectations. For example, is accessibility overall viewed as a bonus or essential requirement? Does society support the allocation of resources for achieving better accessibility?

Factors throughout the intrinsic-to-extrinsic spectrum interact with one another to shape the ecosystem in which an app is created, maintained, and used. Changes in any one factor can impact other factors up and down the spectrum, potentially affecting accessibility. Structuring our understanding of how these factors affect an app's risk for acquiring an inaccessibility disease can guide accessibility enhancing treatments.

4.2.2 Determine the Extent of a Disease

Epidemiologists determine the extent of a disease in a community—through measures such as incidence and prevalence—to plan health services such as facilities, trainings, and resource distribution. App accessibility could benefit from similar metrics. *Prevalence* metrics capture how frequently inaccessibility diseases occur in an app population. For a given inaccessibility disease, other important metrics include the causes of the inaccessibility disease (defined as the *determinants*) and measures of severity such as *lethality*, defined as the likelihood an app will be abandoned due to an accessibility failure. Finally, metrics that capture rates of inaccessibility diseases over time (e.g., *incidence*) can help identify new risk factors or protective factors that impact many apps. An example would be an improvement in accessibility after an operating system update. The objective of *determining the extent of a disease* gives a data-driven focus to addressing app inaccessibility.

4.2.3 Study Natural Histories

The next inspiration from epidemiology is the study of the progression of a disease in a host. Epidemiologists map disease progression through the stages of: exposure to a *risk factor* or *infectious agent*, early *disease onset*, appearance of *symptoms*, *diagnosis*, to *outcome*. This progression model, known as the *natural history of the disease*, informs what risk factors and symptoms to be alert for, what impact the disease will have if untreated, and where in the timeline opportunities exist for preventative or therapeutic treatments [49]. Rather than modeling the natural history of an inaccessibility disease, I use similar concepts to model the progression of an app through two important components of its existence: (1) its creation and maintenance, and (2) its usage. These models can guide our thinking in when a host app might be exposed to a risk factor or infections agent, at what stages in the app lifecycle a disease might manifest and be diagnosable, and what impact the disease might have. The natural histories also inspire when a treatment, *preventative* or *therapeutic*, could be applied most effectively.

4.2.3.1 Natural History of App Development

The "Natural History of App Development" (Figure 6) outlines the stages of app creation and maintenance. Pre-birth, or before an app is released, the app goes through iterative steps of conceptualization, design, implementation, and testing. Interventions within this period are preventative, aimed at treating an inaccessibility disease before the app is released for use.

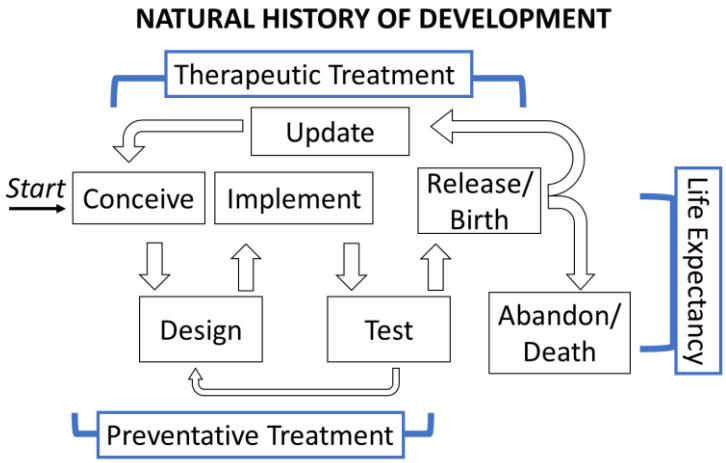


Figure 6: The natural history of app development model represents the design and implementation process an app goes through pre-release and post-release. It serves as a framework for where new treatments might be introduced.

As with epidemiology, preventative treatment is preferred to therapeutic treatment (after release) as it reduces the burdens put on the people trying to use the app. However, inaccessibility diseases do occur in released apps. We then consider the next stages of the app's post-birth development lifecycle, which is an iteration on pre-birth stages through updates [28]. App updates might be driven by bug fixes, end user feedback, adding or extending features, or any number of other factors. Updates occur at varying frequencies on the order of days to years. The update stage is an opportunity to apply *therapeutic treatments* to address accessibility failures or vulnerabilities in released apps. It is also unfortunately possible that updates can introduce or worsen inaccessibility diseases, motivating monitoring app accessibility over time.

The last important milestone in the natural history of app development is death. In this model, *death* of development occurs when an app is no longer being maintained. A "dead" app will not benefit from treatment aimed at developers or maintainers, such as guidelines or source code testing tools. Instead, accessibility diseases must be addressed with other, more external forms of remediation, such as third-party repairs.

4.2.3.2 Natural History of App Usage

The "Natural History of App Usage" (Figure 7), focuses on the cycle of an app's use on a device by an end user. The beginning of active use is defined as the birth of use. Pre-birth stages include finding and downloading an app. Preventative treatment can be introduced at these stages. For example, information on known accessibility failures or features could be presented on an app's download page in an app store. Such preventative treatments allow people to understand the health of an app and potentially avoid trying apps with accessibility diseases.

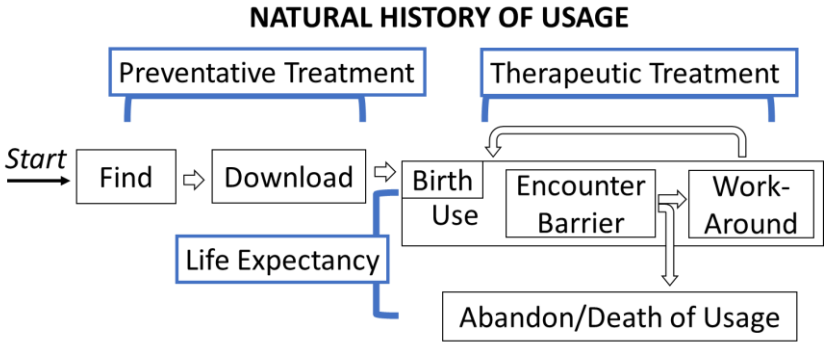


Figure 7: The natural history of usage model represents the process by which an end-user finds, downloads, uses, and abandons an app. The usage stage includes first usage, or birth of usage. Within usage, someone might cycle through the stages of encountering a barrier and trying to work around it. The progression could ultimately end when a user abandons an app and usage dies.

An app's usage-birth happens when it is first opened on a device. During use, barriers caused by inaccessibility diseases might be encountered and workarounds attempted. Usage-death occurs when the person discontinues using an app entirely. Therapeutic treatments could be introduced during use or after abandonment. Screen readers that support end users creating

custom labels for unlabeled screen elements is an example of an existing therapeutic treatment. A post-death treatment could prompt someone to submit feedback on why they discontinued using an app.

4.2.4 Evaluating Existing and New Treatments

Epidemiologists collect information to guide intervention methods and modes of health care delivery. Evaluating the effectiveness of interventions is a key component of revising strategies and focusing efforts on the most promising techniques.

App accessibility enhancement efforts would benefit from expanding evaluation techniques, such as those motivated by my epidemiology-inspired framework. Existing techniques for improving app accessibility include preventative treatments (e.g., developer guidelines [13,45], automated testing tools [44,110,120]) and therapeutic treatments (e.g., adding custom labels with a screen reader, online accessibility forms where people can search for assistance [14,121]]). Population-scale, longitudinal, and multi-factor analyses of these treatments could provide insights into their effectiveness and highlight opportunities for improvement. Epidemiology-inspired analyses complement more focused evaluations such as user studies and case studies.

Metrics from epidemiology that can guide evaluations include: (1) tracking the prevalence or lethality of different disease determinants, or causes, in the population (e.g., how many Android apps have an unlabeled image button or how many apps are abandoned because of those failures?); (2) comparing those measures before and after a treatment is introduced (e.g., did accessibility failures reduce after an accessibility review was added to the process of uploading an app to the Google Play Store?); and (3) examining whether treatments influence factors as expected (e.g., logging how frequently the missing label warnings in Android Studio are muted). Population-level, longitudinal, and multi-factor data can support the evaluation and evolution of approaches for improving app accessibility.

4.2.5 Breaking the Chain of Infection

The Chain of Infection [31] (Figure 8) is another model for understanding how multiple factors interact in the spread of disease. The chain portrays the links between a disease carrier, a susceptible host, and how the disease spreads from one to the other. I apply this conceptual model to the spread of inaccessibility diseases through *carriers* (e.g., source code, design), *reservoirs* (e.g., toolkits, libraries), and *host* apps.

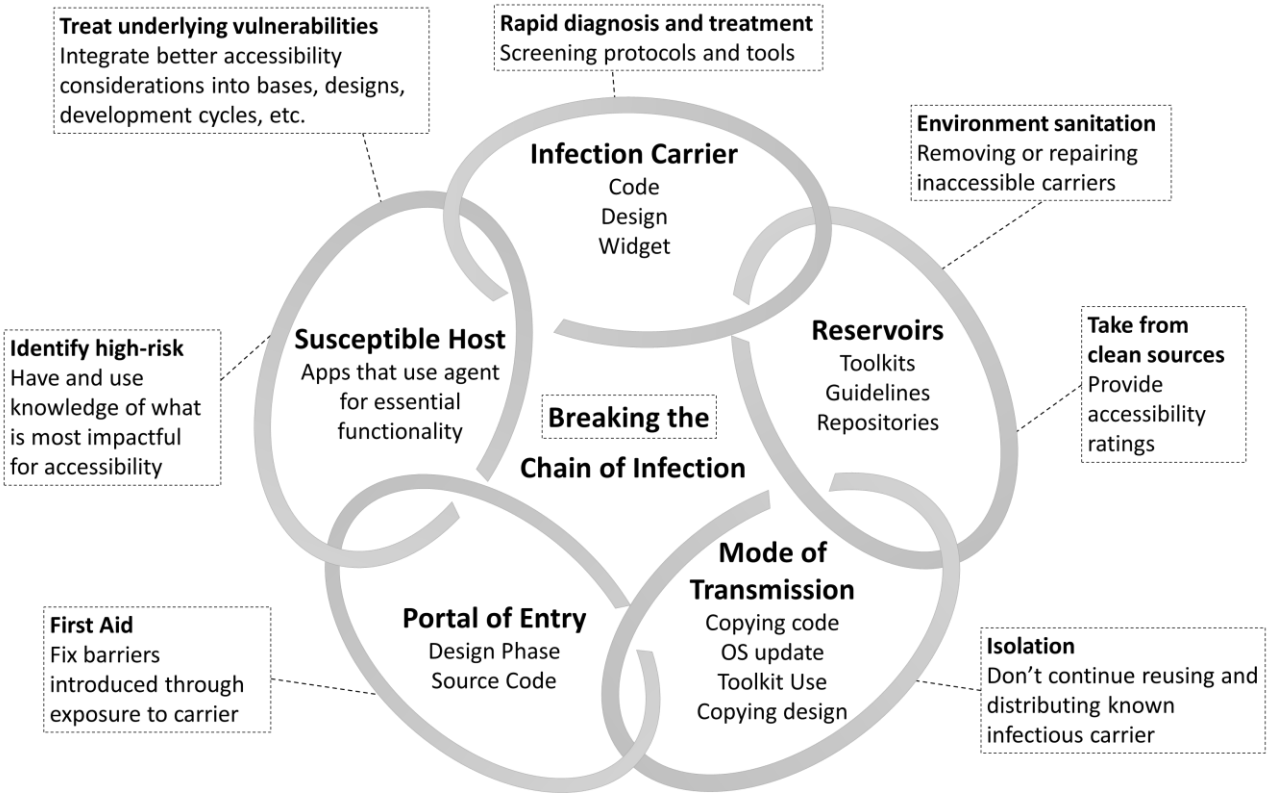


Figure 8: The Chain of Infection helps visualize where an accessibility barrier originates and how it spreads into host apps. Working to break the chain at any one of the major links—infectious agent, reservoir, transmission, portal of entry, or susceptible host— can guide where treatments might be introduced. Inspired by [31]**.**

Breaking the Chain of Infection [31] extends this model to highlight that, metaphorically, if any link of the chain is broken then the disease cannot spread. Each component in the chain is an opportunity to disrupt the flow of a disease. Using this model, we can consider where

interventions could be introduced, what those interventions might look like, and what resources would be needed for success.

Let us walk through the chain of infection in our Android Studio unlabeled image button example (Section 4.1.1, above). We start by defining the app as a susceptible *host* for the inaccessible button *disease*. The image button element is the *agent* with its missing `contentDescription` being the disease *determinant*. The Android Studio design interface acts as a *reservoir* for the agent. The agent is *transmitted* through the drag-and-drop design interface with a *port of entry* at the source code implementation stage.

We can use the breaking the chain of infection model to think about possible interventions by considering how each link of the chain could be broken. In our Android Studio example, the tool provides a warning for the missing `contentDescription`, a metaphorical *first aid* (the warning) in the *portal of entry* (the source code). The existing accessibility guidelines for Android act to *treat underlying vulnerabilities* in the host app by addressing the lack of developer knowledge. A potential *treatment* at the *agent* link (the image button element) could be to create a default `contentDescription` for common Android icons (e.g., "like" for a thumbs-up image). Framing accessibility within the Chain of Infection provokes thinking about causes of inaccessibility in greater granularity and can inspire new techniques for addressing this problem.

4.2.6 Inform Public Policy and Regulation

Mirroring Gordis's [49] final objective for epidemiology, *Informing Public Policy and Regulation*, I consider how population-scale and multi-factor approaches to app accessibility can change the app environment to enhance app health. Changes could include improving and enforcing legislation, companies vetting apps, initiatives to inform developers, or education for people on available treatments for their apps. Large-scale data can contribute to motivating and informing these efforts. For example, the percentage of people in the world

with disabilities if often used to motivate accessibility work. Similar data around the prevalence and lethality of app accessibility failures can contribute to policy change movements. My epidemiology-inspired framework helps inform what data collection, analyses, and presentation might look like.

4.3 Chapter Summary

In this chapter, I have shown how epidemiology's objectives, language, techniques, and models transfer to the challenge of mobile app accessibility. I acknowledge the sheer size and complexity of this framework but believe it proportional to the problems and opportunities associated with improving the accessibility of the entire mobile app ecosystem. In the following chapters, I use the language and models presented in this chapter to discuss my research findings and situate them within the larger ecosystem of app accessibility.

Chapter 5. UNDERSTANDING APP ACCESSIBILITY AT SCALE

My epidemiology-inspired framework promotes using population-scale and multi-factor analyses to better understand the state of app accessibility. Such analyses contribute to the objectives of determining the extent of a disease in a population, identifying potential risk factors, and evaluating existing treatments. Data-driven analyses complement human-centered techniques by providing information on what the most common accessibility failures are, identifying trends in failure occurrence across apps, and highlighting factors that may influence when and how often barriers occur.

In this chapter, I present two app accessibility analyses I performed on 100 and 10,000 Android apps. Guided by the results of those analyses, I identify accessibility shortcomings in a range of factors including Android documentation, example code repositories, and popular frameworks. I first summarize key findings from my work. I then present more extensive descriptions of my analyses and findings. I intend the thoroughness of the presentation to aid readers in understanding my analyses, to support my conclusions, and to allow others to draw their own insights from the data.

I published this research in ASSETS 2017 [73], ASSET 2018 [75], and the ACM Transactions on Accessible Computing 2020 [74] with my co-authors Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. I led developing our large-scale testing tools, designing the large-scale and multi-factor assessment protocols, running the app assessments, and writing the submissions.

In my epidemiology-inspired framework, this research works toward the objectives of determining the extent of inaccessibility diseases in the app population, identifying environmental risk and protective factors, and evaluating existing treatments. In demonstration of two components of my thesis, this research (T1) reveals population-level

trends of accessibility failures and (T2) aids in identifying a range of intrinsic to extrinsic factors that can impact app accessibility.

5.1 Key Findings

I present key finding from across my analyses and point to dissertation sections where further details can be found. These results highlight accessibility failures are prevalent across apps, discrepancies in failure prevalence between element classes, patterns in the contexts in elements tend to have accessibility failures, and shortcomings in Android developer resources.

Key findings from my large-scale and multi-factor analysis include:

- In my 100-app analysis, all apps had at least one of the nine types of accessibility failures tested for; most apps (72%) had 6-7 different types of accessibility failures (Section 5.2.1.2).
- In the large analyses, 8% of the 9,999 apps tested had the few TalkBack-focusable elements failure (Section 5.2.2.4—Few TalkBack-Focusable Elements). Apps with this failure were disproportionately from the Education category. Apps implemented using game engines (e.g., Unity) or cross-platform tools (e.g., Adobe Air) frequently had this failure (Section 5.3.3).
- Missing labels was one of the most prevalent accessibility failures. In the 100-app analysis, 94% of apps had at least one occurrence (Section 5.2.1.2). In the larger analysis, missing labels was particularly prominent in `contentDescription`-dependent elements (`contentDescription`-dependent elements approximately capture elements that do not use visual text as their label). Out of 8,901 apps, 23% were missing labels on more than 90% of their `contentDescription`-dependent elements (Section 5.2.2.4).
- I explored coverage of labeling practices in developer and designer resources (Section 5.3.1). Accessibility-specific Android documentation provided some

coverage of labeling. However, much of the general documentation, example code, and automated tests did not provide sufficient coverage of element labeling practices. For example, Android's Quality Guidelines do not mention accessibility. In technical-focused resources, code snippets from Android's general implementation documentation produced unlabeled elements and the Android Lint static analysis tests do not catch missing labels in Floating Action Buttons. For design resources, the general Material Design guides for images do not address labeling. In 2019 I collaborated with a Google accessibility team to improve accessibility practices in some of this documentation.

- Undersized elements was another prevalent accessibility failure. In the 100-app analysis, 95% of apps had at least one undersized element (Section 5.2.1.2). In the larger analysis, 24% of the 9,650 apps tested had over 50% of their clickable elements too small in either height, width, or both. Considering size-based failures by dimension, elements were most likely to be too short only (Section 5.2.2.4—Size-Based Inaccessibility).
- Checkboxes and Radio Buttons were commonly used classes that were frequently undersized; 82% of Checkboxes and 62% of Radio Buttons were too small in either dimension (Section 5.2.2.5—Size-Based Inaccessibility). Patterns emerged between the element's design purpose and whether it was undersized. For example, Radio Buttons used for page tabs were more likely to be large enough while Radio Buttons used as page indicators (e.g., in a tutorial sequence) were more likely to be too small. Other design details that differentiated Checkbox and Radio Button elements that were big enough from those that were undersized included using padding, grouping with text labels, and using visually large icons (Section 5.3.4.3).
- Third parties may influence the accessibility of their plug-in elements through defaults or use guides. For example, the Google+ and Facebook Login elements were similarly sized and styled across apps. Google+'s visually smaller icon was too small

in both height and width in 43% of uses and too short only for an additional 56% of uses. The Facebook Login elements were too small in both dimensions in only 0.3% of uses but too short only in 82% of uses (Section 5.3.4).

- I tested for low contrast in the 100-app analysis; 94% of apps had at least one low text contrast failure and 85% of apps had at least one low image contrast failure (Section 5.2.1.2).
- The remaining failures had low prevalence across apps. I list the percentage of apps from the ~10,000 app analyses with fewer than 10% of their relevant elements with each accessibility failure: duplicate labels, 89% of apps; uninformative labels, 99%; editable TextView with `contentDescription`, 96% of apps; fully overlapping clickable elements, 84%. Details of the results for these failures can be found in Section 5.2.1.2 for the 100-app analysis and Section 5.2.2.4 for the ~10,000-app analyses.
- Results from more recent app accessibility assessments published by other researchers validate, complement, and extend my large-scale, multi-factor analyses (Section 5.4).

5.2 Population-Scale Assessment

Understanding how often different accessibility failures occur (i.e., the failure *prevalence*) is one metric for understanding the impact of those barriers. It can also inform how to allocate resources for enhancing app accessibility or how to develop new interventions. Large-scale analysis alone cannot prove causation between treatments and the resulting accessibility of apps. However, it is a powerful component to guide complementary work such as interviews and user studies.

5.2.1 Failure Prevalence in 100 Top Android Apps

In a preliminary assessment of the prevalence of inaccessibility diseases in Android apps, I tested 100 apps for nine failures: overlapping clickable elements, editable TextView with

`contentDescription`, duplicate label, missing label, redundant label, undersized element, text contrast, image contrast, and invalid link. This 100-app analysis laid the foundation for my larger analyses (discussed in the next section). In demonstration of my epidemiology-inspired framework, I use framework terminology throughout this section.

5.2.1.1 Method & Data

I took a stratified sample of 100 apps from the *population* of top, free Android apps. Apps were selected from the "top downloaded, free" lists in the Google Play Store in each of ten categories (my strata): Business, Communication, Education, Entertainment, Health and Fitness, Maps and Navigation, Medical, Productivity, Shopping, and Social. I excluded apps that required a specialized log-in (e.g., a bank account or subscription) or blocked automated scanning (e.g., banking apps often block screenshots). Ten apps from each category were analyzed, totaling 100 apps. For each app, my co-authors and I identified 4-8 primary tasks. For example, in the "Indeed Job Search" app, the tasks were: recover forgotten password, log-in, search for jobs, apply for jobs, and access settings. Google's Accessibility Scanner was the *diagnostic tool* used to test each screen involved in the tasks for *determinants* of nine *inaccessibility diseases*: overlapping clickable elements, editable TextView with `contentDescription`, duplicate label, missing label, redundant label, undersized element, low text contrast, low image contrast, and invalid link. Descriptions of most of these failures can be found in Section 2.3. Redundant labels occur when an element is labeled with its type, (e.g., a button with a `contentDescription` of "Play Button"). TalkBack automatically announces item type, so information is redundant.

5.2.1.2 Results

Figure 9 presents the number of apps that presented at least one occurrence of each accessibility failure. The most prevalent determinants are undersized elements (95% of apps), missing labels (94%), and low text contrast (94%). Slightly less prevalent are duplicate labels (85%), low image contrast (85%), and overlapping clickable elements (57%). The least

prevalent failures were redundant labels (20%), editable TextView with `contentDescription` (10%), and invalid link (1%).

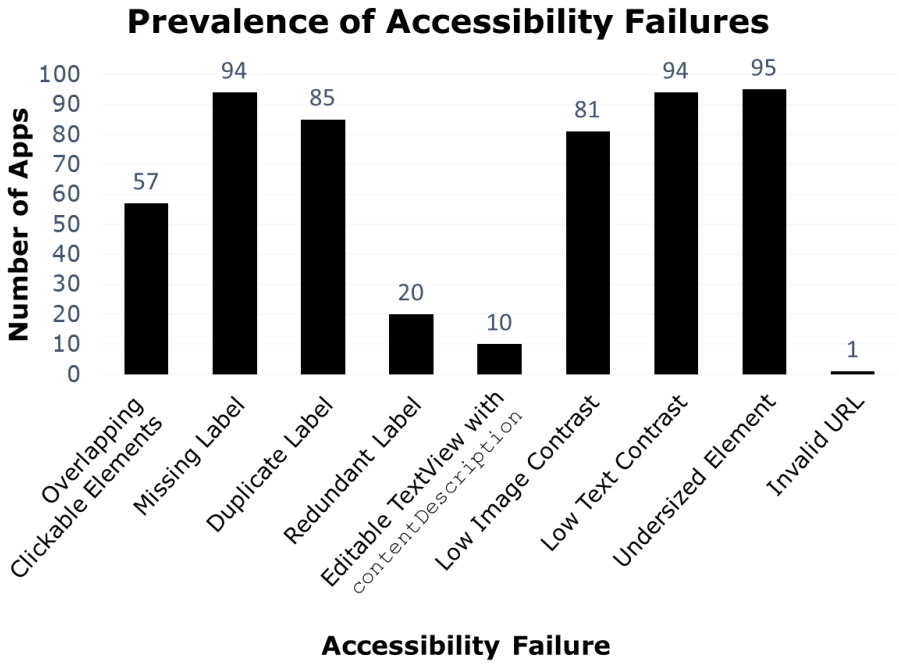


Figure 9: The high prevalence of accessibility failures in the tested apps highlights that apps are still largely inaccessible, even for well-documented accessibility features like labeling.

Reflecting the co-occurrence of failures, Figure 10 presents the distribution of number of unique types of accessibility failures per app. In the language of my framework, all apps exhibited symptoms of at least one of the nine disease determinants. Seventy-two percent of apps were diagnosed with five or six determinants (36% each). The remaining distribution skewed slightly with 3% of apps presenting one determinant, 2% with three, 9% with four, 10% with seven, 3% with eight, and no apps presenting a case with all determinants. The limitations of the Accessibility Scanner diagnostic tool impacted these results. For example, all three apps with a single determinant presented a single missing label failure on all tested screens. In each case, the single failure actually represented a screen with no TalkBack-focusable elements (i.e., the entire interface was presented as a single item to the accessibility API, see Section 2.3).

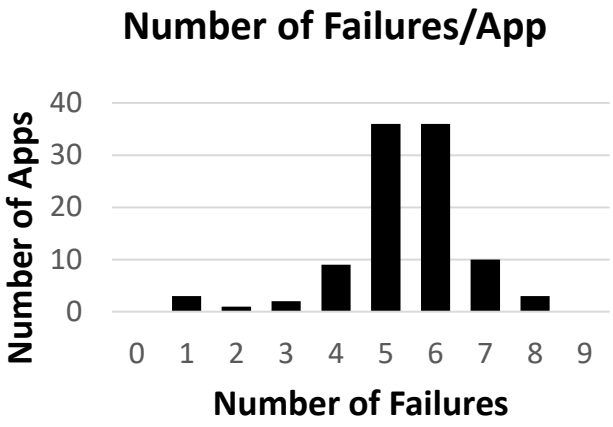


Figure 10 The distribution of the number of errors in each app explores co-occurrences of different determinants. Co-occurrence might suggest different underlying influential factors.

5.2.2 Failure Prevalence in 10,000 Android Apps

Building on the preliminary 100-app analysis, I tested 10,000 free Android apps containing approximately 14,000 classes of screen elements for seven accessibility failures: few TalkBack-focusable elements, missing labels, duplicate labels, uninformative labels, editable TextView with `contentDescriptions`, fully overlapping clickable elements, and undersized elements. Due to code obfuscation (Section 2.2), some of the 14,000 unique class names may represent the same actual class.

I first present the prevalence of accessibility failures of all apps and all relevant classes of elements appearing in those apps. I then present the prevalence of the most common accessibility failures within the five elements classes that are most used among apps. Finally, I perform scoped assessments on a few key classes of elements. The choice of element classes was based on the results of the broader-scoped analyses, on metrics such as class use, and on existing knowledge of common accessibility failures. In addition to those findings, I present my data filtering methods. Specifically, I chose classes of elements based on prior knowledge about common accessibility failures, the use of classes across apps, and the failure proneness of the classes. These filtered analyses highlight patterns in accessibility failures.

5.2.2.1 Data

My app dataset is a subset of the Rico repository, downloaded April 17, 2018 [34]. I filtered the 10,477 free Android apps in the repository based on exclusion criteria described below. I analyzed 9,999 apps that contained elements from approximately 14,000 distinct element class names. For each app, the dataset included the app's metadata, a set of screenshots, and view hierarchies associated with each screenshot. The metadata captured app attributes such as its category. Details of this dataset and its collection can be found in the paper by Deka *et al.* [35].

Exclusion Criteria

Every app in the dataset has a unique `package_name`. Each view hierarchy file had an `activity_name` field with a value in the form `<package_name>/<activity>` to indicate what app was in focus when the screen was captured. If the package in the `activity_name` field of the view hierarchy did not match the package of the app being assessed, that specific screen was marked as invalid and excluded. This criterion eliminated screens captured outside of the app, such as the Android home screen, lock screen, or a web browser redirect. If a view hierarchy file was `null`, the associated screen was discarded as invalid. If an app had no valid screen, and therefore zero captured elements, the entire app was excluded.

A portion of my analyses focused on subsets of element classes. In such analyses I only included apps with at least one element from the class of interest. The class of an element was determined by its `class` field in the view hierarchy. Captured class names were inexact representations of the class used to create the element due to obfuscation. For example, the classes `ztu` and `yei` were both mutated names for the Google+ button shown in Figure 30. As is typical with obfuscation, I did not know the mapping from original class name to obfuscated name. I therefore treated each class name in the dataset as distinct. I discovered the mapping

from these classes to the Google+ button by manually inspecting screenshots that contained `ztu` and `yei` classes, a non-scalable approach for all potentially obfuscated class names.

Limitations

The Rico repository was collected by Deka *et al.* [35] to analyze design patterns; data capture was not motivated by accessibility analysis. Using a dataset outside of its intended purpose often adds limitations. For example, the captured view hierarchies did not include some element attributes that would have improve accessibility assessment (see Appendix A for details on missing attributes).

The Rico dataset was also limited to Android apps. Research on iOS apps [64,69,99] identified many similar accessibility failures, such as missing labels. However, the prevalence of those failures found in this work may not reflect the state of iOS app accessibility. A future comparison of iOS and Android apps could yield insights into how factors that differ between the two platform ecosystems may have a strong impact on app accessibility.

Large datasets of Android app screens are difficult to collect, especially when collection tools do not have access to the source code. Determining if two screenshots are of the same screen is one challenge because app screens have no robust, unique identifier. Due to this challenge, some apps in the Rico dataset had very little of their functionality captured. For example, the dataset contained over 185 screens for the WhatsApp Messenger app. However, those screens cover a very narrow range of its functionality. Specifically, all of the captured screens are of the country selection and phone number verification steps of registration. The Rico crawler may have failed to identify these duplicates because of pixel-level differences in the screenshots from the visible country names changing after scrolling the selection list. Barriers to logging into apps (e.g., needing a bank account) or the need for human input (e.g., a login screen) are additional data collection challenges. Further work in data collection methods could improve future large-scale app assessment.

Despite the limitations of the Rico repository, it contains a significant amount of useful information that is otherwise difficult to collect. I believe this data gives meaningful insights into the state of Android app accessibility.

5.2.2.2 Accessibility Evaluation Method

This section defines my test for each accessibility failure together with the criteria for elements that were tested. I tested apps for seven accessibility barriers: *few TalkBack-focusable elements*, *missing labels*, *duplicate labels*, *uninformative labels*, *editable TextViews with* `contentDescriptions`, *fully overlapping clickable elements*, and *undersized elements*. All tests except the tests for uninformative labels and few TalkBack-focusable elements were based on the Google-released Accessibility Test Framework for Android [46]. Details on the use of the Accessibility Test Framework for Android to implement the tests can be found in Appendix C. Tests for uninformative labels and few TalkBack-focusable elements were operationalized by the first author based on prior work and known accessibility failures. As with all automated accessibility evaluation tools, our accessibility tests have limitations in their coverage and accuracy. I note these limitations for each test below.

Few TalkBack-Focusable Elements

I identified TalkBack-focusable elements by leveraging heuristics used by TalkBack's implementation (see Appendix A for details). Not all element attributes used by TalkBack heuristics were available in the dataset, such as the `checkable` property, so those heuristics were skipped. Although this approach likely misses some important elements or includes some elements that are not of interest, it mirrors the Accessibility Test Framework and provides a meaningful characterization of app accessibility.

Apps with one or fewer TalkBack-focusable elements per screen, averaged over all captured screens, tested positive for the few TalkBack-focusable elements failure. I tested all 9,999

valid apps (i.e., app that had at least one element captured, even if it was not TalkBack focusable) for few TalkBack-focusable elements.

One limitation of this test is its inability to distinguish between problematic screens and screens that may legitimately have only one TalkBack-focusable element, such as a loading screen with only a progress bar or an empty list. However, most designs have more than one element and screens with one or fewer TalkBack-focusable elements likely contain a set of non-focusable but essential elements. Future work could explore more robust testing techniques (e.g., computer vision or crowdsourcing) to more accurately identify apps that do not properly expose their screens.

Label-Based Inaccessibility

I tested for four label-based accessibility failures: *missing label*, *duplicate label*, *uninformative label*, and *editable TextView with* `contentDescription`. Labels are primarily used by screen readers, therefore, I only tested TalkBack-focusable elements. Because TalkBack uses different logic for WebView elements than it uses for native Android elements, I did not test WebViews for label-based errors. I did not test editable TextViews for missing, duplicate, or uninformative labels since TextViews have unique label requirements, as captured in the editable TextView with `contentDescription` failure.

I determined the label, or lack thereof, of each element using logic from the TalkBack implementation [122] and the Accessibility Test Framework for Android (see Appendix A). The Rico dataset did not include the `labelFor` element attribute, which allows a developer to explicitly use one element as a label for another (e.g., using a text box with visible text as the announced label for an editable text field). Thus, there may be some elements labeled using the `labelFor` technique that our tests classify as unlabeled. However, the accessible development documentation [4,45] and Android Studio Lint test recommendations [3] do not cover using the `labelFor` attribute, they only mention the use of a `contentDescription`.

Moreover, more recent research identified low usage of the `labelFor` attribute in open-source Android projects [82]. Therefore, I believe my element label identification technique provided a reasonable evaluation.

The testing criteria for each label-based accessibility failure is detailed below.

Missing Label

I implemented the missing label failure in Python based on the `SpeakableTextPresent` test in the Accessibility Test Framework for Android 2.1. I tested the 9,677 apps that had at least one TalkBack-focusable element for missing labels.

Duplicate Label

I test for the duplicate labels failure by comparing the labels of all clickable, TalkBack-focusable elements on a single screen. If two or more elements have the exact same label, they are all counted as having the duplicate label failure. This criterion is based on the `DuplicateSpeakableText-ViewHierarchyCheck` from the Accessibility Testing Framework for Android 2.1, looking only at clickable elements.

To avoid an overpowering effect of missing labels, I only tested labeled elements for duplicate labels. I tested the 8,869 apps that had at least one screen with at least two labeled clickable elements.

Uninformative Label

To develop a list of uninformative labels, I reviewed the labels of clickable Image Views, Image Buttons, and Floating Actions Buttons in the dataset. I noted labels whose content was only a reflection of the class or the contentDescription attribute name (i.e., the label was composed of only these words or their abbreviations: button, image, content, description, icon, or view). The resultant set of "uninformative labels" is: alt image, button, Button, contentDescription, desc, Desc, Description, Description Image, icon desc, [image], image, Image, images,

Images, image description, Image Des, image description default, Icon, Image Content, ImageView, and View.

Note this criterion does not test whether labels were accurate (e.g., whether a button labeled "back" actually functioned as a back button). Future work could expand uninformative label detection (e.g., using crowdsourcing to expand and assign severity to the list of uninformative labels). I tested the 9,650 apps that had at least one labeled clickable element, excluding elements with missing labels.

Editable TextView with contentDescription

My test defined editable TextViews as elements that are TalkBack-focusable and have an ancestor class of `android.widget.EditText`. Because the `hint` attribute was not available in the Rico view hierarchies, I tested that editable TextViews did not provide a `contentDescription` but could not test whether they appropriately provided a `hint`. I tested the 2,919 apps that had at least one editable TextView element.

Fully Overlapping Clickable Elements

The Accessibility Test Framework for Android tests if elements that have the `clickable` and `importantForAccessibility` attributes set to True are fully overlapping. The Rico dataset, however, did not include the `importantForAccessibility` element attribute. I therefore approximated the Accessibility Test Framework's approach by testing whether each clickable, TalkBack-focusable element fully overlapped with another clickable, TalkBack-focusable element on the same screen. I determined an element's location on the screen using its `bounds` attribute. We tested the 9,171 apps with at least two clickable elements on a screen.

Size-Based Inaccessibility

Following the Google Accessibility Test Framework for Android, I tested elements that were clickable and TalkBack-focusable for the four size-based accessibility failures: *too small in*

either dimension, *too small in both* dimensions, *too short only* and *too narrow only*. I chose 48dp as the size threshold based on the Android guidelines [45].

The Android guidelines uses the density-independent pixels unit (*dp*) in its size recommendation to account for varying screen resolutions (which are measured in dots per inch, or *dpi*). I calculated the size of an element in the dataset using the `bounds` attribute from the view hierarchy, measured in pixels (*px*). I then converted the pixel-unit measurement (*px*) to density-independent pixels (*dp*) using the formula $dp = (px \times 160)/dpi$ [123].

The pixel density (*dpi*) value is determined by device resolution, but the Rico publication [35] does not specify the capture device. I therefore reverse engineered a likely *dpi* for our calculations, deciding on a value of 560 *dpi*. I based this value on the Google Nexus 6P and additionally verified that it yielded behaviors consistent with the Rico dataset. For example, I compared the size of a variety of elements from the Rico dataset to the size of that same element in a current version of the same app running live on a Nexus 6P. I also used the Google Accessibility Scanner], based on the Accessibility Test Framework for Android, to test the minimum size of several elements in the live version of an app, comparing results to those obtained for the associated element in the Rico dataset. The consistency of the test results with the measures of live apps confirmed that the pixel density assumption was reasonable.

Developers can use several techniques to create elements that meet the size recommendations. The most obvious approach is creating elements that are themselves large enough. However, enlarging elements may conflict with a desired visual appearance. An alternative is to place a visible element inside of a larger, invisible container and to set the larger container as the clickable element [2]. In these cases, the invisible container is TalkBack-focusable, whereas the contained visible element would not be. My tests, which were applied to TalkBack-focusable elements, capture both implementation techniques.

I tested the 9,650 apps with at least one clickable TalkBack-focusable element for undersized elements.

5.2.2.3 Filtering

Analyses of focused subsets of data can reveal trends in app accessibility that may be obscured in analyses of the full dataset. I used knowledge-based filters and metric-based filters during my analyses. *Knowledge-based filters* are based on existing knowledge about accessibility failures in apps, including my preliminary app analyses. *Metric-based filters* are based on the use of classes and the likelihood of accessibility failures in those classes.

My analysis of label-based accessibility failures in image-based buttons (Section 5.2.2.6) is one example of a knowledge-based filtered analysis. Prior work [1,27,29,37,64,79,91,97,99] and existing accessibility guidelines identified missing labels as an important accessibility failure. However, the significance of missing labels is not immediately clear from the prevalence analysis of my full dataset (Figure 11, left). Apps predominantly fall in the low prevalence distribution, with only 2.9% of apps having over 90% of their elements missing labels. Using this metric, the infrequency of the failure could suggest missing labels might best be addressed by focusing on specific, poor-performing apps.

Most image-based elements need to be explicitly labeled by a developer (e.g., using the `contentDescription` attribute) whereas text-based elements do not. Using that knowledge, I filtered the data to focus on three popular classes of image-based buttons. In contrast to the unfiltered analysis, missing labels are highly prevalent in image-based buttons; 44% of apps have over 90% of their image-based buttons missing labels (Figure 11, right). This filtered distribution suggests a more systemic problem involving factors beyond specific apps. Section 5.2.2.6 presents a more detailed analysis of image-based button labeling.

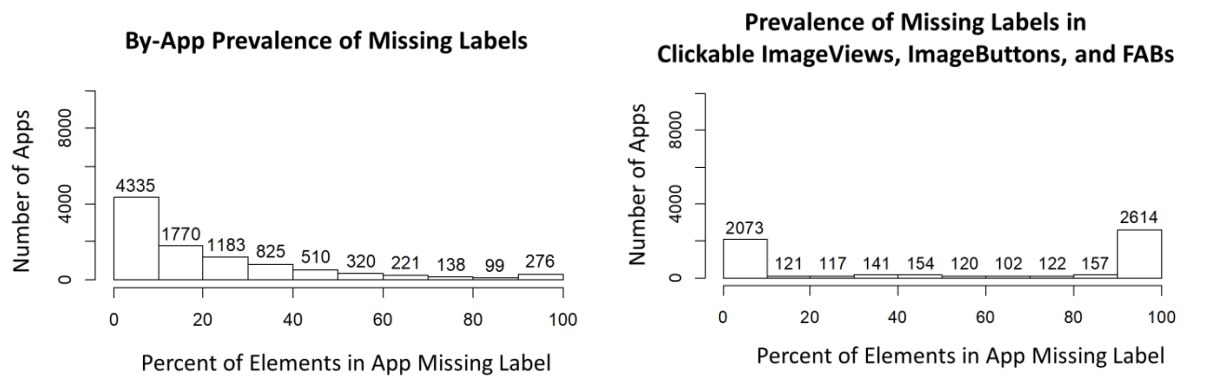


Figure 11: A comparison of the distribution of percent of elements within an app that are missing labels between (left) considering all tested classes of elements in the apps (9,677 apps total) and (right) focusing on Clickable Images, Image Buttons, and Floating Action Buttons in apps (5,721 apps total). Considering only the more relevant elements highlights a significant problem that is not apparent from the analysis of all elements.

Metric-based filters complement knowledge-based filters by revealing trends that may not be apparent from current knowledge. I used two metrics for filtering: (1) class *use* (i.e., the number of different apps that have at least one element of that class captured and tested in the dataset), and (2) class *failure-proneness* (i.e., the number of elements of that class with a given accessibility failure out of all tested elements of that class). In epidemiology-inspired concepts, assessing relationships between code use and failure-proneness could indicate the contagiousness of inaccessibility diseases being transmitted through code reuse.

Analyzing any region of the space defined by these two metrics (as visualized in Figure 12) yields different app accessibility insights. For example, classes with low use may include more custom classes developed for a specific app or small set of apps. Trends in low-use classes could reflect risk factors associated with creating and using custom or uncommon classes. Such insights could suggest opportunities to improve documentation or tools for creating custom classes. In my dataset, however, some of the low use class names were due to code obfuscation. Investigating the accessibility of low use classes is an opportunity for future work.

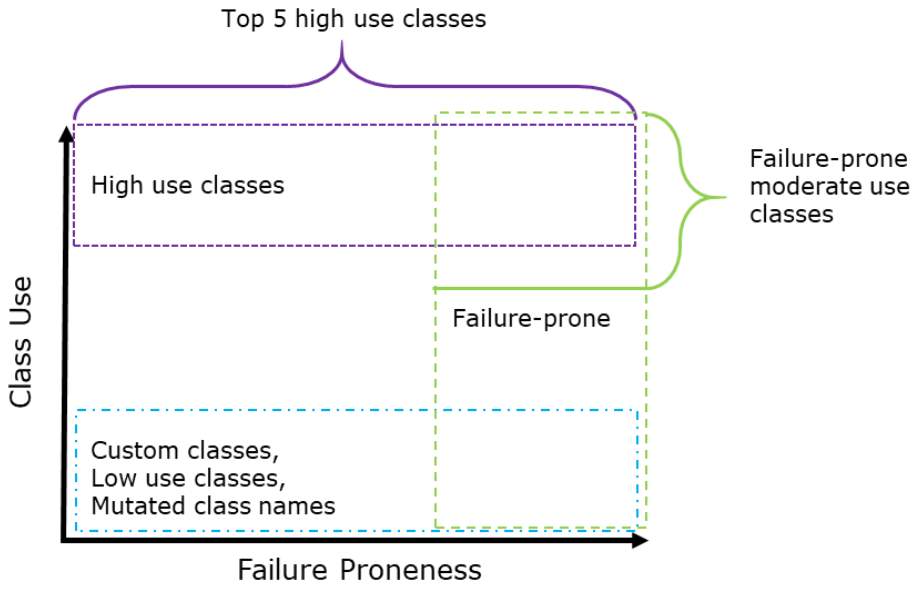


Figure 12: Two dimensions of impact when considering classes of elements. Class use was defined as the number of apps that contained at least one tested element of that class. Failure-proneness was defined as (the total number of elements of a class across all apps that had a given accessibility failure) / (the total number of elements of a class tested for that failure). Parts of the space are labeled with the types of classes the region likely contains. My analyses focused on two sections of the space indicated with brackets: high use classes and failure-prone moderate use classes.

I focus on two groups of classes: (1) *high use classes* and (2) *failure-prone moderate use classes*, as indicated by the brackets in Figure 12. I define *high use classes* as the top five most used classes, regardless of their failure proneness. I define *moderate use* as classes in the top 1% of use out of the approximately 14,000 total unique class names.

The accessibility of high use classes can have a widespread impact on the overall accessibility of the population of apps. The *failure-prone moderate use classes* filter captures classes that may not necessarily be the most used but are more likely to have an accessibility failure when they are used. In epidemiology-inspired terms, these classes are *high-risk groups*. Investigating extreme cases of classes with high failure prevalence can guide and complement other investigations into what creates worst-case classes (e.g., documentation, tool support, awareness, education).

Although understanding failure-prone classes regardless of use (i.e., the entire right side of Figure 12) might also provide valuable insights, I focus on classes with at least moderate use due to their impact across many apps in the population. This definition filters out classes that are used in very few apps, including the approximately 9,000 class names that appeared in only one app. Moderate use of use captures a range of use. For example, the classes in the top 1% of use tested for missing labels include 140 classes that are each used in 36-6,233 apps. High use classes may not appear in failure-prone moderate use classes if they have low failure rates. Tables of moderate use classes (in the top 1% of use) for each accessibility failure I tested for can be found in Appendix B. I present analyses of a subset of *high use classes* and *failure-prone moderate use classes* in Section 5.2.2.5.

5.2.2.4 All-Class, By-App Analysis

I first present the prevalence of each accessibility failure over all classes of tested elements in all apps. The results are one indication of the spread and impact of each type of failure.

Few TalkBack-Focusable Elements

For the *few TalkBack-focusable elements* failure, I consider apps with an average of exactly 0, exactly 1, and between 0-1 focusable elements per screen (Table 5). The experience of using a screen with zero TalkBack-focusable elements is likely similar to that of using a screen with one focusable element (e.g., a screen reader or switch cannot access any content or give any indication of available functionality). I divide this failure into three groups–apps that average exactly 0, exactly 1, and between 0-1 focusable elements per screen– because the factors influencing the number of focusable elements per screen may be different (e.g., different developer tools or libraries).

Of 9,999 tested apps with at least one element, focusable or not, 791 (8%) had an average of no more than 1 focusable element per screen. This included 253 apps (3%) with an average

of exactly 0, 98 apps (1%) with an average between 0 and 1, and 440 apps (4%) with an average of exactly 1 element per screen.

Table 5: Prevalence of apps with few TalkBack-focusable elements, broken down by how many elements per screen on average the app had.

average # TalkBack-focusable elements per screen	# apps	% apps
0	253	3%
(0,1)	98	1%
1	440	4%
[0,1]	791	8%

Although the prevalence of *few TalkBack-focusable elements* is lower than some other accessibility failures, the lethality of the barrier may be notably higher (i.e., not being able to interact with any of an app's functionality). I discuss a subset of apps with this failure and potential causes in Section 5.3.3.

Label-Based Inaccessibility

I present the prevalence of four types of label-based accessibility failures: *missing label*s, *editable TextViews with contentDescriptions*, *duplicate labels*, and *uninformative labels*.

Missing Labels

I tested 9,677 apps with at least one TalkBack-focusable element for *missing labels*. There was a median of 124 TalkBack-focusable elements tested per app (*M:* 232, *SD*:338, *R*: 1-7,916)[2]. In the distribution of missing label failures over all tested elements in all apps, most apps were in the lower prevalence ranges (Figure 13, left). In the low prevalence range, 4,335 apps (45%) were missing labels in less than 10% of their elements. At the lowest prevalence, 2,191 (22%) of apps had exactly zero tested elements missing labels. The apps

[2] In the statistical reporting, *"M"* stands for arithmetic mean, "*SD"* stands for standard deviation, and "*R*" stands for range.

with exactly zero elements missing labels had a median of 28 tested elements (*M*: 65, *SD*: 113, *R*: 1-1,830).

Looking at apps with a high prevalence of the failure, 276 apps (3%) had 90% or more of their elements missing labels. Within the rest of the distribution, 4,288 apps (44%) were missing 10-50% of their element labels and 778 apps (8%) were missing 50-90% of their element labels.

For text-based elements, labels can be generated from the visual text. In general, developers must add a `contentDescription` to label elements without text. To explore if non-text elements were at higher risk of having missing labels, I filtered the data to exclude elements that had a label but did not have a `contentDescription`. I refer to these as *contentDescription-dependent elements*. This knowledge-based filter approximates excluding elements that automatically have labels from visible text.

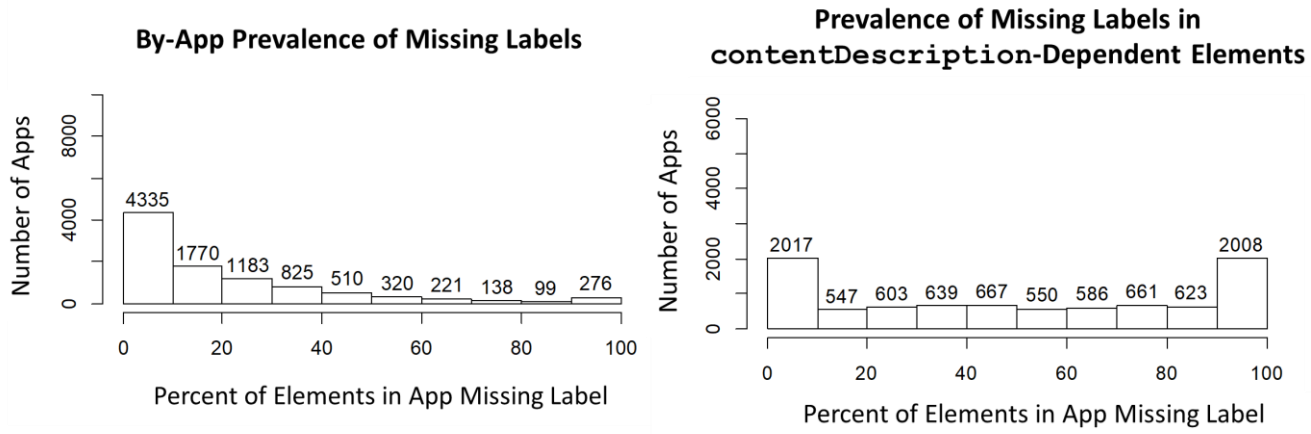


Figure 13: Distribution of the missing label accessibility failure per app. (left) considering all TalkBack-focusable elements, showing 9,677 apps. (right) considering all elements that depended on a `contentDescription`, showing 8,901 apps. Focusing only on elements that use `contentDescriptions` highlights a high prevalence of missing labels.

The resulting distribution of `contentDescription`-dependent elements was bimodal, with peaks at the left and right extremes (Figure 13, right). There was a median of 50 tested `contentDescription`-dependent elements per app (*M*: 99, *SD*: 161, *R*: 1-5,666). At the lower

prevalence end of the distribution, 2,017 apps (23%) had 0-10% of their elements missing labels. In that 0-10% interval, 1,415 apps (16%) had exactly zero elements missing labels. The apps that had exactly 0% of their `contentDescription`-dependent elements missing labels had a median of 18 elements tested (*M*: 55, *SD*: 129, *R*: 1-1,942).

At the high prevalence end of the distribution, 2,008 apps (23%) had 90-100% of their `contentDescription`-dependent elements missing labels, with 1,535 apps (17%) at exactly 100%. The apps with exactly 100% had a median of 19 `contentDescription`-dependent elements tested for missing labels (*M*: 55, *SD*: 129, *R*: 1-1,942).

The bimodal nature of the distribution reflects two prominent groups of apps. These groupings may reflect some apps existing in environments that invest in accessibility while other apps do not. Additionally, the 4,876 apps (54%) between the two extremes indicate a third interesting group of apps with inconsistent element labeling. The existence of this group points to the richer ecosystem encompassing apps. Factors such as different developers, different tools, external components, competing priorities, or the evolution of new features may explain these sometimes-accessible apps. More work is needed to uncover what interactions between factors exist.

Duplicate Labels

I tested the screens of 8,869 apps that contained at least two labeled clickable elements for the *duplicate labels* failure. There was a median of 99 tested elements per app (*M*: 185, *SD*: 279, *R*: 1-7.714). The distribution shows low prevalence of duplicate labels across apps (Figure 14). In the low prevalence end of the distribution, 7,902 apps (89%) had 0-10% of their tested clickable elements with duplicate labels. There were 6,812 apps (77%) with exactly 0% of tested elements with duplicate labels. The apps at 0% had a median of 70 tested elements (*M*: 132, *SD*: 200, *R*: 1-6,777). The remaining apps tended to have low

prevalence of duplicate labels: 929 apps (9.3%) had 10-50% of tested clickable elements with duplicate labels.

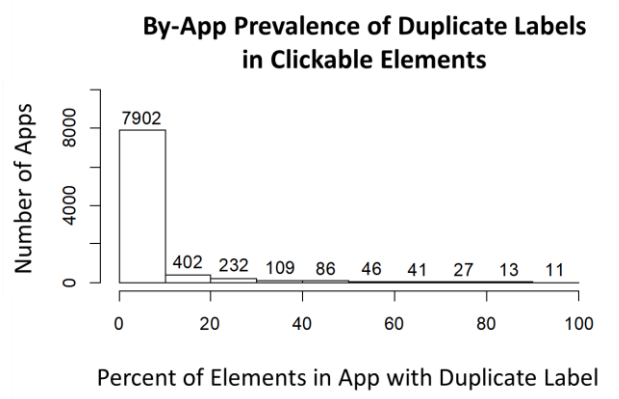


Figure 14: Distribution of clickable elements with duplicate labels per app out of 8,869 apps.

Uninformative Labels

I tested 9,121 apps with at least one labeled TalkBack-focusable element for *uninformative labels*. This app population had a low prevalence of this failure (Figure 16). I tested a median of 98 elements per app for uninformative labels (*M*: 184, *SD*: 277, *R*: 1-7,714). Showing low prevalence of this failure, 9,003 apps (99%) had 0-10% of their tested elements with uninformative labels. The 8,796 apps (96%) with exactly 0% of tested elements with this failure had a median of 94 elements tested (*M*: 178, *SD*: 273, *R*: 1-7,714).

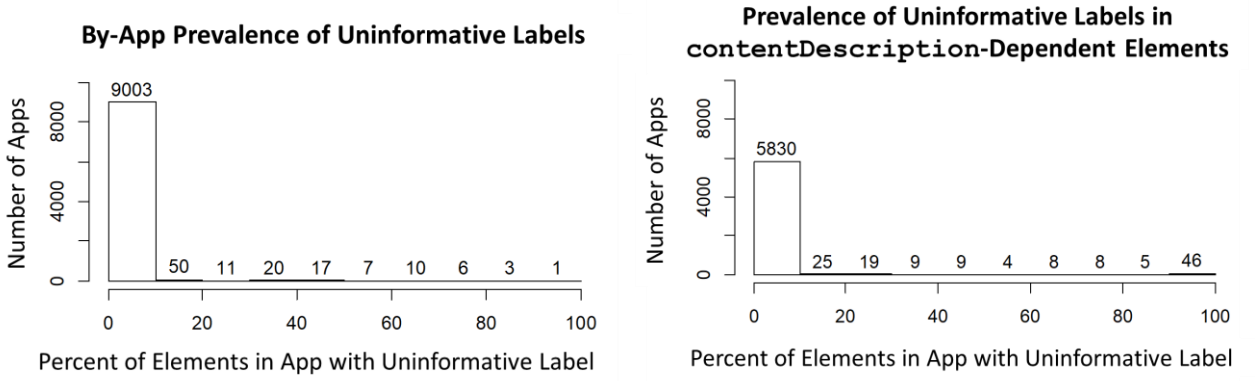


Figure 15: Distribution of percent of uninformative element labels per app over (left) all tested elements per app, out of 9,121 apps; (right) contentDescription-dependent elements per app, out of 5,963 apps.

Informative labels may be automatically obtained from the visible test of an element. Therefore, I again filtered the data down to elements that depended on a `contentDescription`.

Tested apps had a median of 23 elements that obtained their label from a `contentDescription` (*M*: 48, *SD*: 79, *R*: 1-1,455). Within this population, uninformative labels still had low prevalence (Figure 15, right); 5,787 apps (97%) had exactly 0% of their tested elements with the failure. Apps with 0% failure rate had a median of 23 `contentDescription`-dependent elements tested (M: 47, SD: 77, R: 1-1,455). Only 133 apps (2%) had more than 10% of their `contentDescription`-dependent elements with uninformative labels.

Editable TextView with ContentDescription

I tested 2,919 apps with at least one editable TextView. Apps had a median of 9 editable TextView elements tested (*M*: 20, *SD*: 33, *R*: 1-596). The *editable TextView with contentDescription* failure had low prevalence. A total of 2,806 apps (96%) had 0-10% of their editable TextViews with the failure (Figure 16: Distribution of percent of editable TextViews per app that incorrectly had a contentDescription, out of 2,919 apps.). The 2,800 apps (96%) with exactly 0% of their elements with this failure had a median of 9 elements tested (*M*: 20, *SD*: 33, *R*: 1-596). There was a small spike at the most prevalent extreme of the distribution with 65 apps (2%) having 90-100% of their editable TextViews having `contentDescriptions`.

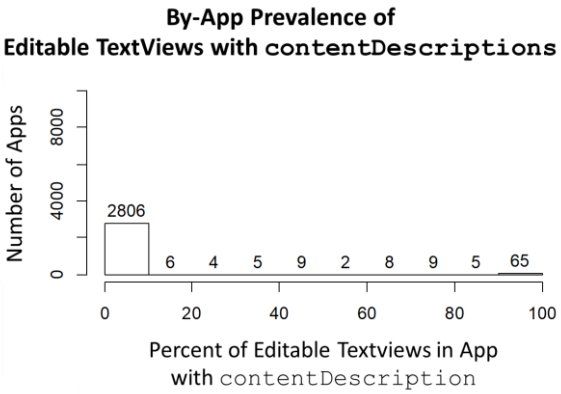


Figure 16: Distribution of percent of editable TextViews per app that incorrectly had a contentDescription, out of 2,919 apps.

Fully Overlapping Clickable Elements

I tested 8,886 apps with at least two clickable TalkBack-focusable elements for the *fulling overlapping clickable elements* failure. This accessibility failure had a low prevalence across apps (Figure 17). A median of 110 elements per app were tested (M: 200, SD: 297, R 2-7,792). At the lowest prevalence end of the distribution, 7,446 apps (84%) had 0-10% of their clickable elements fully overlapping; 6,007 apps (68%) had exactly 0%. Apps with exactly 0% of tested elements with this failure had a median of 88 elements tested (M: 164, SD: 249, R: 2-6,711). The remaining apps tended to have a low prevalence of this failure; 1,231 apps (14%) had 10-50% of their tested clickable elements fully overlapping.

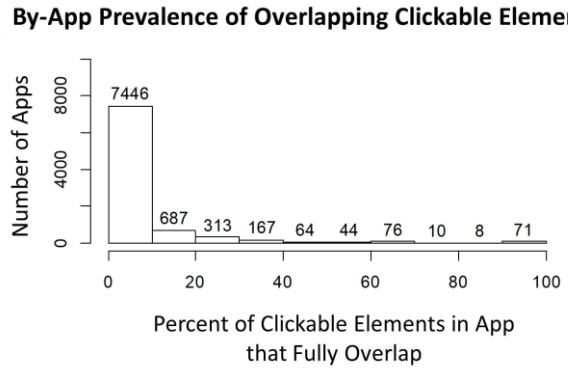


Figure 17: Distribution of percent of fully overlapping clickable elements per app, out of 8,886 apps.

Size-Based Inaccessibility

The 9,650 apps tested for sized-based accessibility failures had a median of 98 clickable, TalkBack-focusable elements tested (M: 188, SD: 290, R: 1-7,792). Figure 18 presents the prevalence distributions for the four types of sized-based accessibility failures: *too small in either* dimension, *too small in both* dimensions, *too short only*, and *too narrow only*. Too small in either is the sum of the other three size-based failures. An element that is wide enough and tall enough (i.e., does not have a size-based accessibility failure) is *big enough*.

Apps had a notable prevalence of undersized elements, regardless of dimension; 6,875 apps (71%) had more than 10% of their elements too small in either. Decomposing by dimension, apps seemed most at risk of having elements that were too short only; 5,289 apps (55%)

had more than 10% of their elements too short only. In the next most prevalent failure, 2,753 apps (29%) had more than 10% of their elements too small in both dimensions. Apps appeared least likely to have elements that were too narrow only; 706 apps (7%) had more than 10% of their tested elements with this failure.

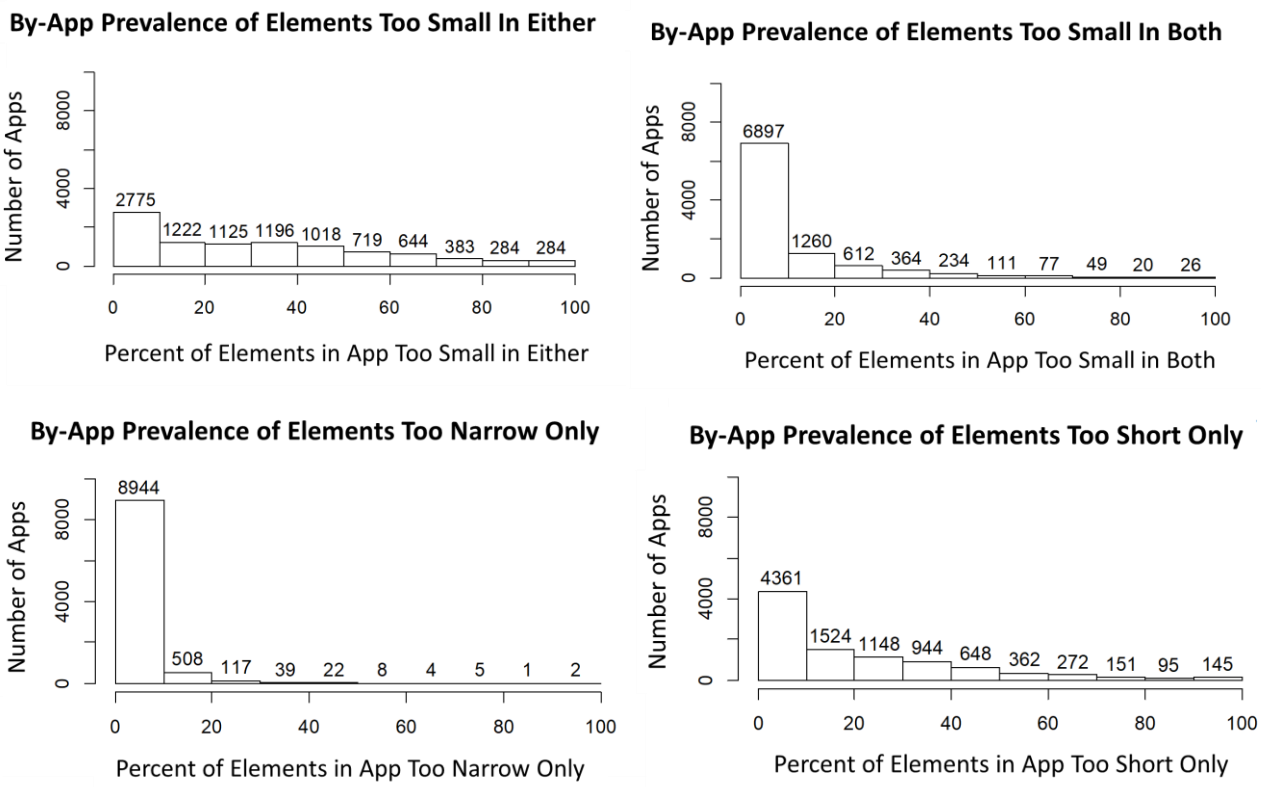


Figure 18: Distributions of the percent of elements in each app with (top left) the size-based accessibility barrier *too small in either* and the dimension-based subcategories of: (top right) *too small in both*, (bottom left) *too short only*, and (bottom right) *too narrow only*. 9,650 apps were tested.

At the left end of the distribution, 1,925 apps (20%) had exactly 0% of their elements *too small in either*, with a median of 19 elements tested per app in that extreme (*M*: 42, *SD*: 83, *R*: 1-1,853); 4,486 apps (46%) had exactly 0% of their elements *too small in both*, with a median of 41 elements tested per app (*M*: 88, *SD*: 139, *R*: 1-1,853); 2,627 apps (27%) had exactly 0% of their elements *too short only*, with a median of 26 elements tested per app (*M*: 60, *SD*: 119, *R*: 1-2,483); 6,901 apps (72%) had exactly 0% of their elements *too narrow*

only, with a median of 66 elements tested per app in that extreme group (*M*: 138, *SD*: 227, *R*: 1-7,792).

The number of apps within the mid-range of undersized elements (10-90%) indicates within-app variation on element sizing. Section 5.3.4 explores how the use context of different classes of elements may affect sizing and contribute to the within-app sizing variation.

5.2.2.5 High Use Classes

The by-app distributions presented in the last section contribute high-level insights into how frequently apps have accessibility failures. I now explore the prevalence of accessibility failures in different classes of elements, compiled across apps. Enhancing the accessibility of classes that are more susceptible to failures can improve accessibility across apps. To focus on this breadth of impact, I analyze the high use classes for the two most prevalent accessibility failures: *missing label* and *size-based failures*. I measure *class use* as the number of apps that had a least one element of a given class.

Missing Labels

The high use classes of elements tested for *missing labels* are presented in Table 6. The list includes three layout or container classes (i.e., `LinearLayout`, `ScrollView`, and `ListView`) with very low failure rates; layout and container classes can inherit labels from their contained children elements. `TextView` text-based class has similarly low prevalence (1%); this is likely because these elements get labels from their visible text. The only high prevalence among the high use classes of elements tested for missing labels is the `ImageButton` class at 53%.

Consistent with the by-app analyses of missing labels, I applied a knowledge-based filter to focus on elements directly dependent on `contentDescriptions` for labels. The high use `contentDescription`-dependent classes are primarily for images (Table 7). The `android.widget.ImageView` and `android.support.widget.AppCompatImageView` classes are nearly synonymous; `AppCompatImageView` subclasses `ImageView` to support older versions of

Android [124]. The same relationship of subclassing for legacy support applies to the `android.widget.ImageButton` and `android.support.v7.widget.AppCompatImageButton` classes. These high use, image-based elements have high prevalence of missing labels. Approximately 67% of elements from both classes of Image Button and approximately 90% of tested elements of both Image View classes were missing labels. I explore missing labels in image-based buttons in more detail in Section 5.3.1.

Table 6: Prevalence among the high use classes of elements of missing labels. All elements that were TalkBack-focusable were considered.

Class	# apps using element	# elements	# missing label	% missing label
`android.widget.LinearLayout`	6,233	208,614	2,304	1%
`android.widget.TextView`	5,597	186,161	1,925	1%
`android.widget.ScrollView`	5,299	43,943	612	1%
`android.widget.ListView`	4,189	41,446	2,293	6%
`android.widget.ImageButton`	4,038	126,675	66,458	53%

Table 7: Prevalence of missing labels in the high use classes with contentDescription-dependent elements. Elements that had a label but not a contentDescription were excluded.

Class	# apps using element	# elements	# missing label	% missing label
`android.widget.ImageButton`	4,038	247,015	163,008	66%
`android.widget.ImageView`	2,976	635,228	580,975	92%
`android.support.v7.widget.AppCompatImageView`	1,845	157,827	133,346	85%
`android.support.v7.view.menu.ActionMenuItemView`	1,842	49,529	2,537	5%
`android.support.v7.widget.AppCompatImageButton`	1,725	108,049	73,175	68%

Size-Based Inaccessibility

The high use classes tested for size-based accessibility failures included three layout classes (Table 8 and Table 9). Of those, `LinearLayout` and `RelativeLayout` had notable prevalence of size-based accessibility failures, being *too small in either* dimension for 27% and 29% of their

uses. Considering the dimension-specific failure breakdown, the Linear and Relative Layout elements were predominantly *too short only*.

Both button classes appearing in the high use classes (i.e., `Button` and `ImageButton`) have high size-based accessibility failure prevalence with almost 40% of tested elements being too small in either. However, the distribution of the dimension in which elements were undersized differs. `ImageButtons` were about twice as likely to be *too small in both* (24%) compared to *too short only* (13%). `Buttons` had the reverse distribution, being twice as likely to be *too short only* (23%) than *too small in both* (12%). I further explore size-based failures in these classes in Section 5.3.4.

Table 8: The number of elements in each of the high use classes with size-based accessibility failures.

Class	# apps using element	# elements	# too small in either	# too small in both	# too short only	# too narrow only
android.widget. LinearLayout	4,342	150,631	40,636	1,898	36,479	2,259
android.widget. ListView	4,185	41,355	346	0	254	92
android.widget. ImageButton	4,030	124,605	49,631	30,424	16,124	3,083
android.widget.Button	3,912	129,054	49,321	16,060	30,022	3,239
android.widget. RelativeLayout	2,989	97,608	27,843	3,076	21,997	2,770

Table 9: The percentage of elements in each of the high use classes with size-based accessibility failures.

Class	# apps using element	# elements	% too small in either	% too small in both	% too short only	% too narrow only
android.widget. LinearLayout	4,342	150,631	27%	1%	24%	2%
android.widget. ListView	4,185	41,355	0.8%	0%	0.6%	0.2%
android.widget. ImageButton	4,030	124,605	40%	24%	13%	3%
android.widget.Button	3,912	129,054	38%	12%	23%	3%
android.widget. RelativeLayout	2,989	97,608	29%	3%	23%	3%

5.2.2.6 Label-Based Inaccessibility in Image-Based Buttons

Image-based elements are a substantial subset of contentDescription-dependent elements that must be explicitly labeled by developers. Delving into these types of elements in more detail, I assessed three classes of image-based buttons for label-based accessibility failures: Clickable Images, Image Buttons, and Floating Action Buttons (FABs).

Clickable Images and Image Buttons are in the high use contentDescription-dependent classes and are mentioned in Android's element labeling guidelines. I included the FAB class to explore how label-based failures affected more recently released class with less use.

In this section, I present the prevalence of missing labels, duplicate labels, and uninformative labels in these image-based button by-app and by-class.

Image-Button Class Details

In the follow subsections, I detail the use of each class and how the three classes relate to one another, including the date each class of button was added to the Android platform.

Figure 19 shows examples of each of the three types of image-based buttons. Table 10 presents the use of each class across the population of apps.

Table 10: The number of apps from our population of 9,999 that have image-based buttons of each class.

Class	# apps	% of app population
Clickable Image	2,976	30%
Image Button	4,038	40%
FAB	590	6%

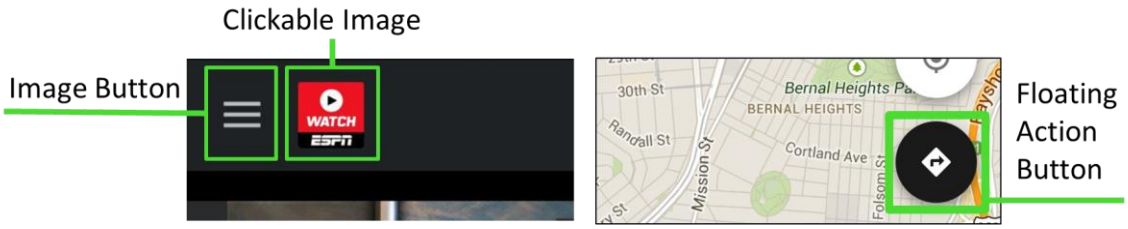


Figure 19: Examples of elements from the (left) Image Button, (center) Clickable Image, and (right) Floating Action Button classes. In practice, Image Buttons and Clickable Images are used interchangeably. Floating Action Buttons are primarily icon-based and denote the most important action on the screen.

Clickable Image

Image elements can be added to an app using the Android API class android.widget.ImageView [5]. If the clickable attribute is set to true, the image functions as a button (Figure 19). I call such elements Clickable Images.

Clickable Images have slightly different defaults and rendering than Image Buttons (which are discussed below). Non-decorative images should be labeled with a `contentDescription`. The `ImageView` class has been in the Android API since Android 1.0, released in September 2008.

Decorative images can be labeled with a `null` string to properly be ignored by screen readers. The interactivity of Clickable Images indicates a non-decorative functionality. I therefore classify a `null` string label in a Clickable Image as a *missing label accessibility failure*.

Image Button

Image Buttons are from the Android API base class `android.widget.ImageButton` [6]. This is a sub-class of the Clickable Image's `ImageView` class. As the name suggests, Image Buttons are buttons that visually present an image rather than text. Image Buttons were released in Android 1.0 in September 2008.

Floating Action Button (FAB)

FABs are visually prominent buttons that "float" above an underlying interface (Figure 19). According to Google's Material Design guidelines [118], "a floating action button represents

the primary action in an application." FABs rarely have visual text labels, usually simple icons convey the functionality (e.g., a pencil icon for a draft new email FAB). Given the importance of FABs for key functionality and their image-based style, proper labeling with a `contentDescription` is imperative for accessibility. Missing a FAB label would likely be highly lethal for app use.

FABs are from the class `android.support.design.widget.FloatingActionButton` [7]. It is a sub-class of `ImageButton` and a sub-sub-class of `ImageView`. FABs were added in Android 22.2.0 in May 2015, much later than the other two classes of image-based button. This more recent introduction to the API and the official guideline suggestions to use at most one FAB per screen may both contribute to the relatively lower number of apps using FABs (Table 10).

Prevalence of Missing Labels

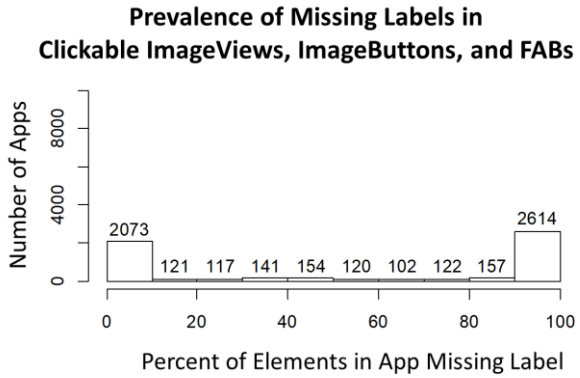


Figure 20: Distribution of prevalence of missing labels in Clickable Images, Image Buttons, and FABs in the 5,721 apps tested.

I tested 5,721 apps with at least one TalkBack-focusable, image-based button tested for missing labels. The distribution of the proportion of buttons in an app missing labels is bimodal (Figure 20). At the positive extreme, 2,073 apps (36%) have less than 10% of their image-based buttons missing labels. On the negative extreme, 2,614 apps (46%) have at least 90% of their image-based buttons missing labels. The remaining 1,034 apps (18%) are relatively uniformly distributed between the two extremes (i.e., have 10-90% of their elements

missing labels). The bimodal nature of the distribution unsurprisingly mirrors the distribution of missing labels in `contentDescription`-dependent elements.

Considering the by-class prevalence analysis presented in Table 11, we note all three of the tested image-based button classes had a high prevalence of missing labels. The use, number of elements with the barrier, and prevalence varied between the classes despite their technical similarities. Image Buttons occurred in the most apps with 40% of apps having at least one. This high class use indicates a significant likelihood that someone encounters Image Buttons in many apps and about half of those elements were missing labels.

While Clickable Images were found in fewer apps, the class had the highest number of elements missing labels; this class produced the most problematic elements in the population that someone may encounter. FABs were fewer in number of apps using them and in number of elements missing labels but had the highest prevalence of missing labels at 93%. Thus, although FABs are less frequently encountered, when they are used, they are extremely likely to be unlabeled. The differences in prevalence of missing labels in the classes despite their technical similarities and release dates suggest other factors impact the likelihood of missing labels. Potential factors include Android guidelines and testing suites, as they can be used in the creation of many apps. I explore how some of these tools address image-based button labeling in Section 5.3.1.

Table 11: The total number of elements tested is in the # elements column. The number of image-based buttons with a missing label and the percent out of all tested image-based buttons of that class is presented in the # missing label and % missing label columns.

Class	# elements	# missing label	% missing label
Clickable Image	115,667	99,825	86%
Image Button	126,675	66,458	53%
FAB	6,389	5,932	93%

Duplicate Labels

I tested for duplicate labels in 3,398 apps that had at least one labeled image-based button to avoid an overpowering effect of missing labels. Figure 21 shows the distribution of proportion of labeled buttons with duplicate labels per app. In 2,961 (87%) of those apps, less than 10% of their tested image-based buttons have duplicate labels. On the least accessible end of the distribution, 174 apps (5%%) have over 90% of their buttons with duplicate labels. The remaining 263 apps (8%) are distributed relatively uniformly over the remaining spectrum of 10-90% error rates.

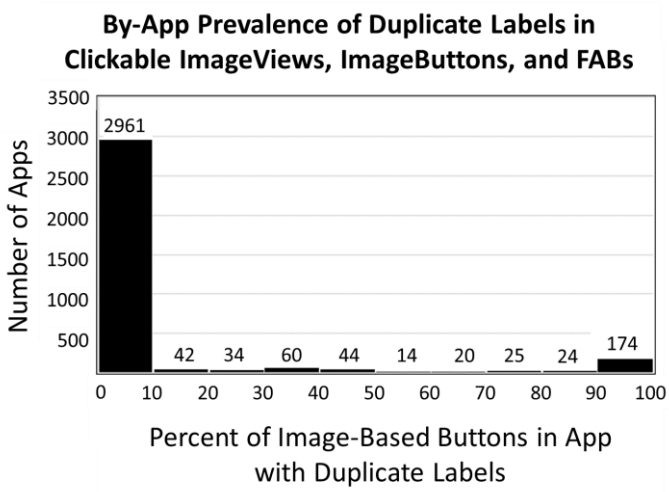


Figure 21: The distribution of the proportion of labeled image-based button elements in an app that have a duplicate label. A total of 3,398 apps were tested. Most apps have a very low proportion of their image-based buttons with the failure. The more inaccessible extreme of having 90%-100% of elements with the failure has a small spike as well.

Table 12 shows the by-class prevalence of the duplicate label failure. Clickable Images had the highest prevalence (47%) and the most elements with duplicate labels. Image Buttons and FABs had lower prevalence (8% for both) and a lower number of labeled elements with duplicate labels. Through manual inspection of apps with the failure, I noticed one pattern of image-based button labels whose duplicate labels were general text, such as "Tool Image"

(Figure 22). Possible contributors to this pattern are copy-and-paste errors or a lack of understanding of the function of a `contentDescription`.

Table 12: The number of labeled image-based buttons with a duplicate label and the percent out of all tested elements are presented per class in the # duplicate label and % duplicate label (of labeled) columns.

Class	# labeled elements	# duplicate label	% duplicate label (of labeled)
Clickable Image	15,531	7,250	47%
Image Button	55,617	4,536	8%
FAB	454	36	8%

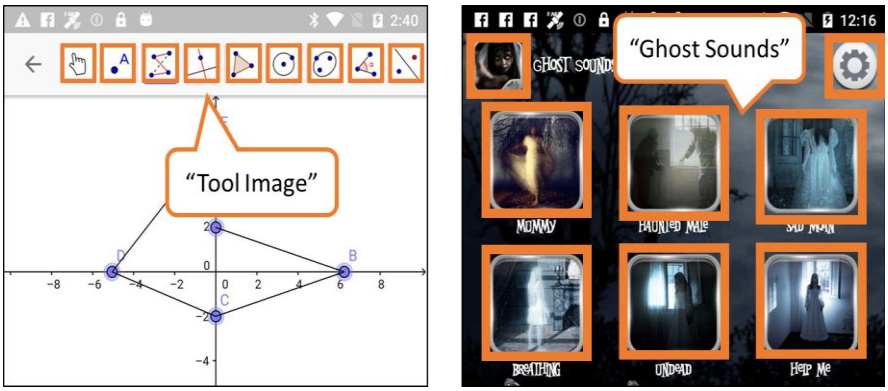


Figure 22: Two example app interfaces with duplicate label errors on image-based buttons. (left) The Clickable Image buttons for drawing different types of figures in a graphing app are all labeled "Tool Image." (right) In the Ghost Sounds app, all of the Clickable Image buttons for playing different ghost sounds as well as the settings and home button are labeled only "Ghost Sounds."

Not all screens with multiple elements with the same label are erroneous. For example, our dataset contained a music app with a list of songs in which all artists were "Unknown." In this case, labeling all artist information buttons with "Unknown" was appropriate. As this example illustrates, more nuanced methods are needed to distinguish between valid and invalid duplicate labels.

Uninformative Label

Out of 3,398 total apps with at least one labeled image-based button tested for uninformative labels, 3,342 apps (98%) had fewer than 10% of their image-based buttons with the failure.

The low prevalence of uninformative labels is mirrored in the by-class analysis presented in Table 13.

Table 13: The total number of labeled image-based buttons tested (# labeled elements), the number of labeled image-based buttons with an uninformative label (# uninformative label), and the percent of all tested elements with the barrier (% uninformative label (of labeled)) are presented per class. Uninformative labels are much less prevalent compared to missing and duplicate labels.

Class	# labeled elements	# uninformative label	% uninformative label (of labeled)
Clickable Image	18,372	1,802	10%
Image Button	62,306	1,278	2%
FAB	524	0	0%

Compared to missing labels, uninformative label failures were not as prevalent. This comparison suggests that if an image-based button is labeled, it tends not to be blatantly uninformative. Further work is needed to determine label quality at a more nuanced level (e.g., the "Tool Image" label discussed as a duplicate label in Figure 22 is also an uninformative label, but not captured by my analysis).

5.3 Identifying Environmental Factors

The large-scale analyses presented above measure the of accessibility failures in the app population, contributing to the epidemiology-inspired objective of measuring the *extent of disease in a population*. I now present work toward the objective of *identifying environmental factors* that may impact app accessibility at scale and *evaluating existing treatments*. My approach is two-fold: (1) assess how existing tools address prevalent accessibility failures and (2) seek out patterns and relationships between accessibility failures and other characteristics of apps. In this section, I present: (1) the coverage of image-based button labeling in existing developer resources; (2) apps with the few TalkBack-focusable elements failure, exploring their category and the tools used to create the apps; (3) the relationship

between app rating and missing labels; and (4) visual design patterns and failure prevalence in a set of interactive and third-party elements.

These multi-factor analyses can inform efforts to improve these specific factors. The insights gained within and across these analyses can also have broader impact, such as identifying systematic problems in app development tools. Finally, the techniques used to scope and assess these case studies can be applied to find other patterns of interest.

5.3.1 Image-Button Labeling in Current Android Resources

The explicit or implicit emphasis that app development and testing tools put on accessibility (e.g., in their default settings, in tests they perform, in the guidance they give) can impact the accessibility of apps created with that tool. If a tool is widely used, that can amount to a large influence.

In this section, I evaluate the coverage of image-button labeling in a set of existing resources, including general Android development and design guides [48,119], accessibility-specific guidelines [8,45], and accessibility testing tools [3]. Android also has Quality Guidelines [9] to articulate what key components an app should have and what fundamental tests an app should pass to be ready for distribution. Categories include visual experience, privacy & security, building for different form factors (i.e., screen sizes, wearables, TV), and build for billions (i.e., localization, diverse devices, reduce power consumption). As of a February 10, 2021 update, the Quality Guidelines still do not explicitly mention accessibility either in its own category or within the criteria of existing categories.

The focus of this evaluation was informed by the large-scale prevalence of label-based accessibility failures in image-based buttons presented in above. In addition to identifying factors that potentially contribute to those failure trends, this evaluation also demonstrates how the large-scale analysis can be used as a guide in the evaluation of tools.

5.3.1.1 General Resources

FABs [7], Buttons [125], and Images [5] have a general developer and design guideline pages. From at least April 2018 (when I first evaluated the pages) to July 2019, the guidelines did not mention the need to label images and the embedded code samples did not have `contentDescriptions`. FABs additionally had a more extensive sample implementation, "FloatingActionButtonBasic" [10] that had missing labels. The copyright date on the FAB code sample is 2014. During my internship at Google in the summer of 2019, I worked with an accessibility team to update some of their documentation. As of May 2021, the embedded code samples in the developer reference for FABs and `ImageViews` have `contentDescriptions`. The last updated dates for those resources are in April and May of 2021. The Button reference no longer has an `ImageButton` code sample and the "FloatingActionButtonBasic" example is now archived.

While the updated developer references are an improvement, Google has been working to centralize their documentation under their Material Design resource. Newer Material Design pages combine design and implementation guidelines. This evolution appears to include attempts to forefront accessibility. For example, the general guidelines for FABs [47,118] and Buttons [125] have subsections on making these widgets accessible through labeling and the 2021 Google I/O conference had a session on "Designing A11y with Material Design" [41]. This improvement accessibility coverage in documentation is a promising indicator of more extrinsic changes. Since developers, especially Android novice developers, are likely to encounter general guidelines, the accessibility of the resources can have a widespread impact. Further work is needed to understand how documentation improvements ultimately affect app accessibility at scale.

5.3.1.2 Accessibility-Specific Resources

The Accessibility subsection of Android Material Design's Usability Guide [8] indicates the necessity of supporting screen readers by "add[ing] audible descriptions to input controls and

other elements." This practice is echoed in the Android Accessibility Development Guidelines "Labeling UI Elements" section [45]. Although the guidelines discuss the need to label all graphical elements, they explicitly use Image Views (the class encompassing Clickable Images) and Image Buttons as examples. Image Buttons are further used in the guideline's sample code snippet demonstrating how to add a `contentDescription` to an element. FABs are not explicitly mentioned in the accessibility development guides.

Beyond addressing the need for labels, the guides provide some indication of what a description should be, such as "provide useful and descriptive labels that explain the meaning and purpose of each interactive element to users" [2]. There is also guidance on practices to avoid when labeling, such as "Note: Many accessibility services, such as TalkBack and BrailleBack, automatically announce an element's type after announcing its label, you shouldn't include element types in your labels. For example, 'submit' is a good label for a Button object, but 'submitButton' isn't a good label" [2]. At the time of my first investigation in April 2018, avoiding duplicate labels is not explicitly mentioned in the guides. In the most current version (updated October 2020), the "Describe each UI element" subsection [2] does note "each description should be unique."

Android provides a repository of code samples that demonstrate using different elements or techniques. The "Basic Accessibility" code sample [11] has an Image Button element and a non-clickable Image View element with `contentDescription` labels. A comment within the code describes the need for `contentDescriptions`. Another comment notes that if the Image is decorative, it does not need a `contentDescription`. This is also no longer the correct practice for decorative images; instead, the `contentDescription` should be `null` or the `importantForAccessibility` attribute should be set to `no` [126]. FABs are not represented in the "Basic Accessibility" code sample.

Regarding label quality in the "Basic Accessibility" Android code sample, each element with a `contentDescription` has a unique label. Comments within that code sample offer guidance on

what a label should be: "Since the contentDescription is read verbatim, you may want to be a bit more descriptive than usual, such as adding 'button' to the end of your description, if appropriate." Note this advice is counter to current best practices; adding the element type of "button" to a content description will cause a redundant label because TalkBack automatically announces the element type. Duplicate labels are not mentioned in the code sample. The copyright date on the sample is 2013 and the sample was deprecated in October 2019 [127].

In April 2018, Android Lint v23.0.0 [3] had a set of accessibility warnings including "Image without `contentDescription`." This warning triggers for an unlabeled Clickable Image or Image Button. It will not trigger for an unlabeled FAB. Android Lint v23.0.0 scans do not warn about duplicate labels. As of May 2021, the Android Linter in Android Studio 4.0 still does not identify missing labels in FABs. Further, the Basic Activity project template has an unlabeled FAB.

The Android Test Framework for Accessibility [43] and the Android Accessibility Scanner v1.1.2 [44] based on that framework detect missing `contentDescriptions` in Clickable Images, Image Buttons, and FABs. These tools also tested for duplicate label but did not cover any type of uninformative label.

5.3.1.3 Summary

We cannot conclude that the discrepancies in representation of the three classes within these resources cause, or even impact, the discrepancy in failure prevalence between the classes seen in my large-scale analysis. However, the omissions within the guidelines combined with the high prevalence, suggest an opportunity to test the success of guidelines. Improving the guide and then testing the impact of guideline usage on app accessibility could give insight into the effectiveness of guidelines. Such testing may include comparing the disease prevalence of apps known to have been made by consulting the guidelines against the disease prevalence within the general population.

5.3.2 Rating & Missing Labels in Image-Based Elements

App ratings can inform an app creator of people's satisfaction with an app and can help people find "good" apps. Accessibility is an important component of apps that needs to be conveyed. Understanding the relationship between rating and failure prevalence, if one exists, can help us understand whether the current rating systems capture app accessibility.

I conducted a Spearman rank-order correlation test between an app's rating and the app's proportion of image-based buttons that were missing labels. I found a statistically significant relationship ($\rho = -0.05$, $p = .001$). Although statistically significant, the correlation coefficient is very low, suggesting that if a relationship exists, it is extremely weak. Looking at the distributions of proportion of missing labels by rating (Figure 23), I note there is high variability of missing labels over the entire range of app ratings.

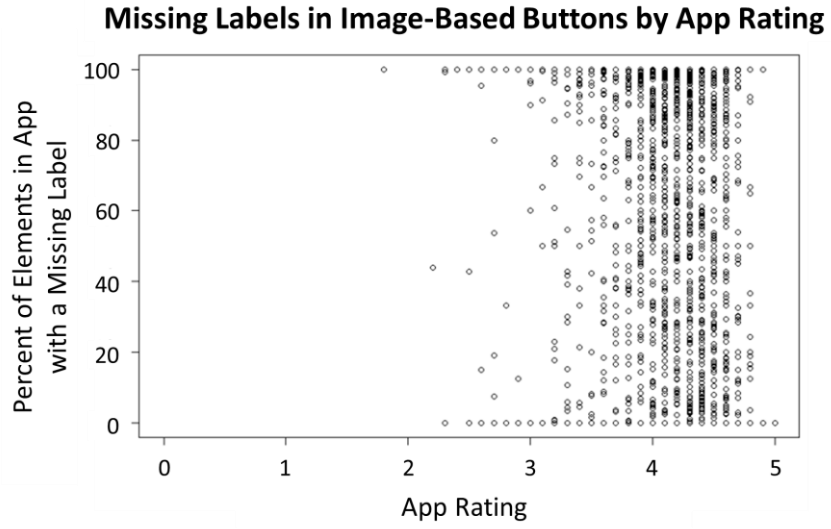


Figure 23: There is high variability in the relationship between an app's rating and its proportion of image-based buttons with missing labels. A statistically significant, but very weak correlation exists between the two factors ($\rho = -0.05$, $p = .001$). The weakness of the relationship suggests current ratings do not reflect the missing labels component of app accessibility.

The weakness of the relationship between app rating and missing label error rates suggest that the current app rating system may not sufficiently capture the missing labels component of accessibility. More recent work by Alshayban *et al.* [1] further demonstrate there is no

strong association between accessibility and app rating or popularity. Given the importance of capturing and presenting end-user satisfaction with an app's accessibility, it may be beneficial to give apps an additional accessibility rating, better factor labeling into existing ratings, or otherwise highlight missing label failures which may be encountered in that app. Having access to a rating that reflected how well labeled an app is could allow an individual to more easily find apps that support their assistive technologies. Presenting these accessibility ratings as prominently as existing ratings may also draw broader attention to the state and importance of accessibility, potentially inspiring public pressure to improve labeling practices. Finally, accessibility ratings provide another avenue to inform app developers on the state of labels in their app in a form that emphasizes its importance.

5.3.3 Frameworks Contributing to the Few Talk-Back Focusable Elements Failure

I consider what categories of apps are most susceptible to the *few TalkBack-focusable elements* failure. The app category is defined by the app developer and found in the app metadata of the Rico dataset used for this analysis.

Figure 24 shows the distribution of percent of apps per category overall and of apps with the few TalkBack-focusable elements failure per category. Of the 815 apps with the failure, 162 apps (19%) are in the Education category; this is disproportionate to the 645 Education apps (7%) in the overall population of 9,999 apps tested. In contrast, Communication apps and Weather apps seem to be underrepresented in apps with few TalkBack-focusable elements.

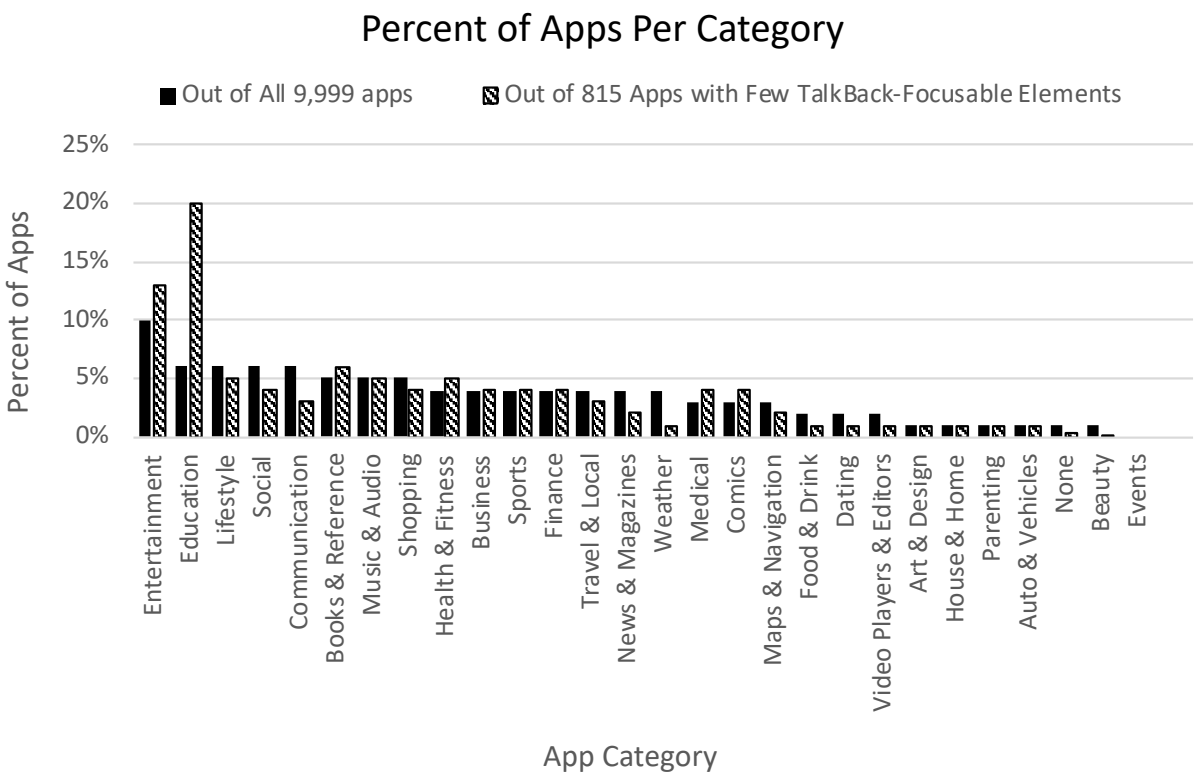


Figure 24: The distribution of the categories of the 815 apps with the *few TalkBack-focusable elements* failure and of the 9,999 apps. Education apps are disproportionately likely to have few TalkBack-focusable elements.

Investigating potential factors that contribute to the few TalkBack-focusable elements failure, I examine classes of elements that frequently appeared in the apps with the failure. Two notable factors were: (1) the use of multi-platform and hybrid web/native app creation tools (e.g., Adobe Air), and (2) the use of game engines (e.g., Unity). I determined tool and plug-in use from the class names within the view hierarchies of affected apps. For example, the `com.unity3d.player.UnityPlayer` class indicated use of the Unity game engine [128]. Appendix D presents the frequency that classes I identified in apps with few TalkBack-focusable elements occurred in apps with and without the failure. The appendix table presents the tool associated with the classes, the number of elements of each class captured, the number of apps each class appears in, and the frequency of the failure occurring in apps with an average of exactly zero and (0,1] TalkBack-focusable elements per screen.

Multi-platform and hybrid web/native Android app plug-ins allow developers to re-use code across mobile, desktop, and web implementations. In my examination of apps with few TalkBack-focusable elements, I found common multiplatform plugins including Apache Cordova [129], Adobe Air [130], and Crosswalk [131]. These plug-ins convert HTML, CSS, Javascript, and Flash implementations into an Android app using an alternative of Android's `WebView` class. As detailed in Appendix D, 64% of Adobe Air elements, 69% of Crosswalk elements, and 75% of Apache Cordova elements occurred in apps with few TalkBack-focusable elements.

I also identified game engine classes in apps with few TalkBack-focusable elements including Unity [128] and Cocos2d-x [132]. Similar to hybrid web/native tools, game engines can create cross-platform experiences. Game engines also enable more expressive styles and functionality (e.g., placing a greater emphasis on 3D graphics and animation) as compared to general app creation tools like Android Studio. Games engine classes were highly associated with the few TalkBack-focusable elements failure; 53% of identified Cocos2d-x and 100% of identified Unity elements were found in apps with few TalkBack-focusable elements (see Appendix D).

5.3.4 Design Patterns in Sizing and Labeling

As a complement to the quantitative analyses of accessibility failure prevalence, I explored how design might influence accessibility by examining patterns in the visual design and use of element classes. I focus on *missing labels* and *sized-based* accessibility failures in three groups of interactive element classes: high use buttons; failure-prone moderate use Checkboxes and Radio Buttons; and failure-prone moderate use third-party social media plug-ins for Facebook and Google+.

5.3.4.1 Overview of Classes

The high use Button (`android.widget.Button`) and Image Button (`android.widget.ImageButton`) classes had notable prevalence of missing labels and size-based accessibility failures (see Tables 6, 7, 8, and 9). The Button class is intended to be used for text-based buttons while the Image Button class is optimized for image-based content (e.g., the cog icon button commonly used for "Settings"). In practice, both classes were used to create image-based buttons. Choice of class for implementing image-based buttons may reflect developer experience or company development practices, additional factors potentially impacting accessibility.

Checkboxes (`android.widget.CheckBox`) and Radio Buttons (`android.widget.RadioButton`) were failure-prone moderate use classes, identified among the highest in prevalence for size-based accessibility failures and in the top 1% of use (Appendix B). Checkboxes and Radio Buttons were often used for selecting between options, choosing a setting, favoriting, implementing a tabbed interface, or implementing a paging indicator (Figure 25).

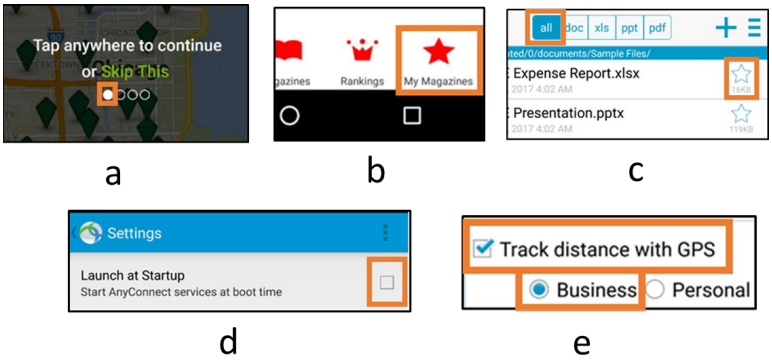


Figure 25: Example uses of Radio Buttons and Checkboxes. (a) Radio Buttons used as pagination indicators, (b) Radio Buttons used as tabs, (c) a Radio Button tab and Checkbox favorite button, (d) a settings Checkbox, (e) a settings Checkbox and Radio Button option selection.

The Facebook Login (`com.facebook.login.widget.LoginButton`) and Google+ (`ztu, yei`) element classes are examples of custom elements from third-party companies released for app creators to integrate with company services. The Facebook Login and Google+ elements

were identified through metric-based filtering; both classes appeared on the list of failure-prone moderate use classes for size-based accessibility failures (see Appendix B). I manually identified the classes `ztu` and `yei` were obfuscated class names of the Google+ elements shown in Figure 30.

5.3.4.2 Missing Labels in Design and Use Patterns

The social network third-party plug-in buttons for Facebook Login and Google+ had very low prevalence of missing labels (Table 14). Only four of the Facebook Login buttons (1%) and zero Google+ buttons had the failure. Moreover, the labels on these elements were consistent across apps and of high quality (see Appendix E for full label list). There were only four unique labels for Google+ buttons, two phrasings of "Touch to add to plus one" and two labels indicating the number of +1's an app had. Across all 309 Facebook Login buttons, there were 28 unique labels that were slight variations of "sign in with Facebook" in different languages. The consistency of labels across apps suggests these labels may have been added by the third-party element creators and rarely modified by individual app developers using the plug-in. These components exemplify how the accessibility of a well-created element that is used by many apps can promote accessibility across the population. Due to a lack of missing labels in these third-party social media plug-ins, I focus on Buttons, Image Buttons, Checkboxes, and Radio Buttons for the remainder of this section.

Table 14: Prevalence of missing label barriers in Image Buttons, Buttons, Checkboxes, Radio Buttons, Google+, and Facebook Login elements.

Class	# apps using class	# elements	# missing label	% missing label
`android.widget.ImageButton`	4,038	126,675	66,458	52.5%
`android.widget.Button`	3,916	130,195	34,140	26.2%
`android.widget.CheckBox`	711	10,636	6,907	64.9%
`android.widget.RadioButton`	402	11,315	3,024	26.7%
Google+ (`ztu` and `yei` combined)	90	687	0	0.0%
`com.facebook.login.widget.LoginButton`	133	309	4	1.3%

The Button, Image Button, Checkbox, and Radio Button classes had a notable prevalence of the missing label failure (Table 14). Checkboxes had the highest prevalence of the failure at 64% followed by Image Buttons at 52%. Radio Buttons and Buttons had lower but still notable failure prevalence at 26% each.

I visually assessed screenshots of each type of element with and without the failure looking for patterns in visual design and use. Similar stand-alone icons (i.e., without accompanying visible text) appeared in elements from the two buttons classes that both were labeled and missing labels. For example, the back arrow and menu icon displayed in Figure 26. This similarity suggests the icon itself may not be the primary factor affecting whether a button class element is labeled.

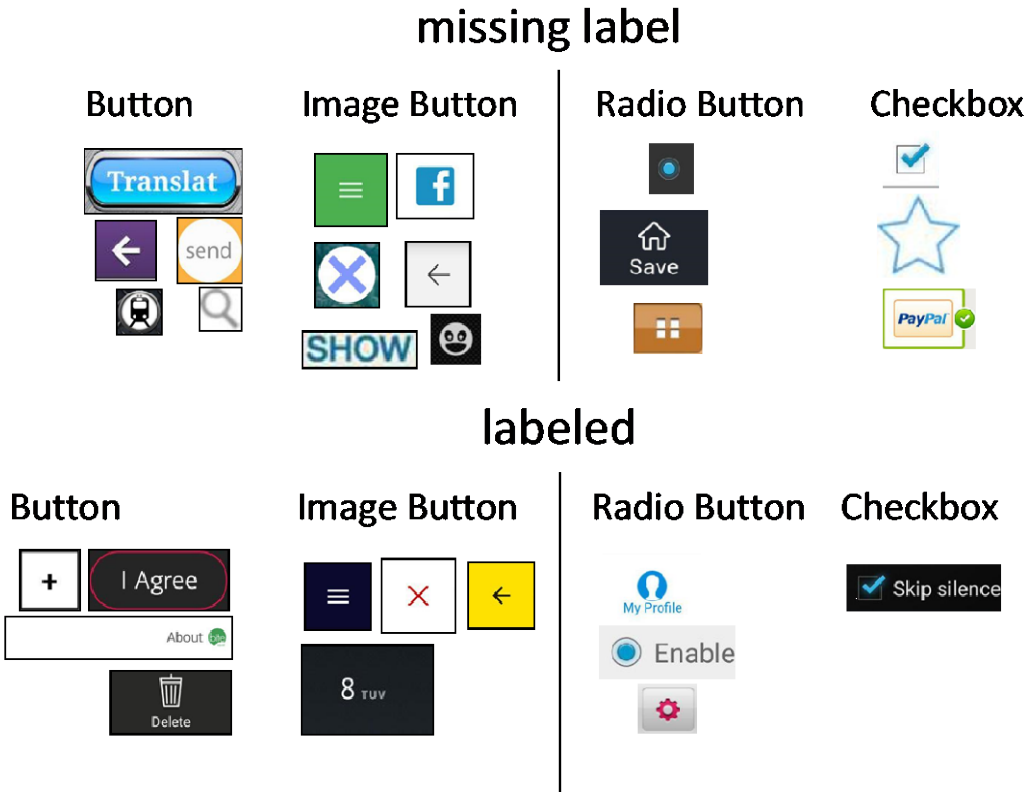


Figure 26: Example Buttons, Image Buttons, Radio Buttons, and Checkboxes that were: (top) missing labels or (bottom) labeled. Most unlabeled elements use only icons or image-based text. Including a text label in the clickable region created labeled element. Labeled elements with image-based text or no text had `contentDescriptions`.

The traditional circular Radio Button and square Checkbox icons occurred in elements of those classes with and without labels (Figure 26). However, unlike the button classes, there was a difference in visual design between labeled and unlabeled Radio Buttons and Checkboxes. When the circle or square icon was stand-alone, the element tended to be unlabeled. When the circle or square icon was grouped with a visual text-based label, it tended to be labeled. A similar trend occurs in labeled Buttons having visible text grouped with an icon (e.g., the "I Agree" Button in Figure 26, bottom left).

However, we also see unlabeled elements with visual text. This is likely caused by image-based text where an element visually looks like text but is implemented only in pixels (like a screenshot). For example, Figure 26 shows image-based text on the "Translat" and "send" Buttons, the "SHOW" Image Buttons, and the "Save" Radio Button. Image-based text cannot automatically be used as a label and instead needs a `contentDescription`. Figure 26 shows an example Image Button of a keypad "8" implemented as an image-based-text element that was appropriately labeled with a `contentDescription`.

5.3.4.3 Size-Based Inaccessibility in Design and Use Patterns

For the size-based accessibility failures, both the Button and Image Button classes had almost 40% of tested elements *too small in either* dimension (Table 9). Image Buttons were more likely to be *too small in both* (24%) than *too short only* (13%.) Button elements had the reverse trend being more likely to be *too short only* (41%) than *too small in both* (21%).

As presented in Table 15, sized-based accessibility failures were more prevalent in Checkboxes (82% *too small in either*) and Radio Buttons (62% *too small in either*) than the Button and Image Button classes. Checkboxes were more likely to be *too small in both* (47%) than *too short only* (28%). Radio Buttons were the opposite, more frequently being *too short only* (41%) than *too small in both* (21%). None of the four classes were prone to having elements that were *too narrow only*.

Table 15: Prevalence of size-based accessibility failures in Checkbox and Radio Button classes.

Class	# elements	# apps using class	% too small in either	% too small in both	% too short only	% too narrow only
`android.widget.CheckBox`	10,176	686	82%	47%	28%	8%
`android.widget.RadioButton`	11,124	393	62%	21%	41%	0.3%

Elements from the Button, Image Button, Radio Button, and Checkbox classes with the same size-based accessibility failure had similar visual design and use patterns (Figure 28 and Figure 29). Elements of these classes that were too small in both tended to be composed of a tightly cropped icon. A few too small in both elements had short visible text (e.g., a single word or number). Among elements that were too short only, Button, Radio Button, and Checkbox elements had longer text-based content while Image Buttons had additional horizontal padding around icons. Styles that gave too narrow only elements sufficient height included using taller icons or adding vertical padding around icons. I did not note any text-based elements that were too narrow only. Vertical and horizontal padding was a key component of Button and Image Button classes that were big enough. Checkboxes and Radio Buttons often achieved this padding by matching the height or width of container elements. Figure 27 illustrates this technique and Figure 29 shows examples from the data such as "Help" and "5 Stars" options that were rows in a menu list.

Figure 27: A Checkbox element that, stand-alone, is *too small in both* dimensions and *missing a label* if without a contentDescription. If the clickable region includes the text label, the element is *too short only* and *labeled*. Expanding the region to the full width and height of the row layout makes the element *big enough*. No modifications affect the visual appearance.

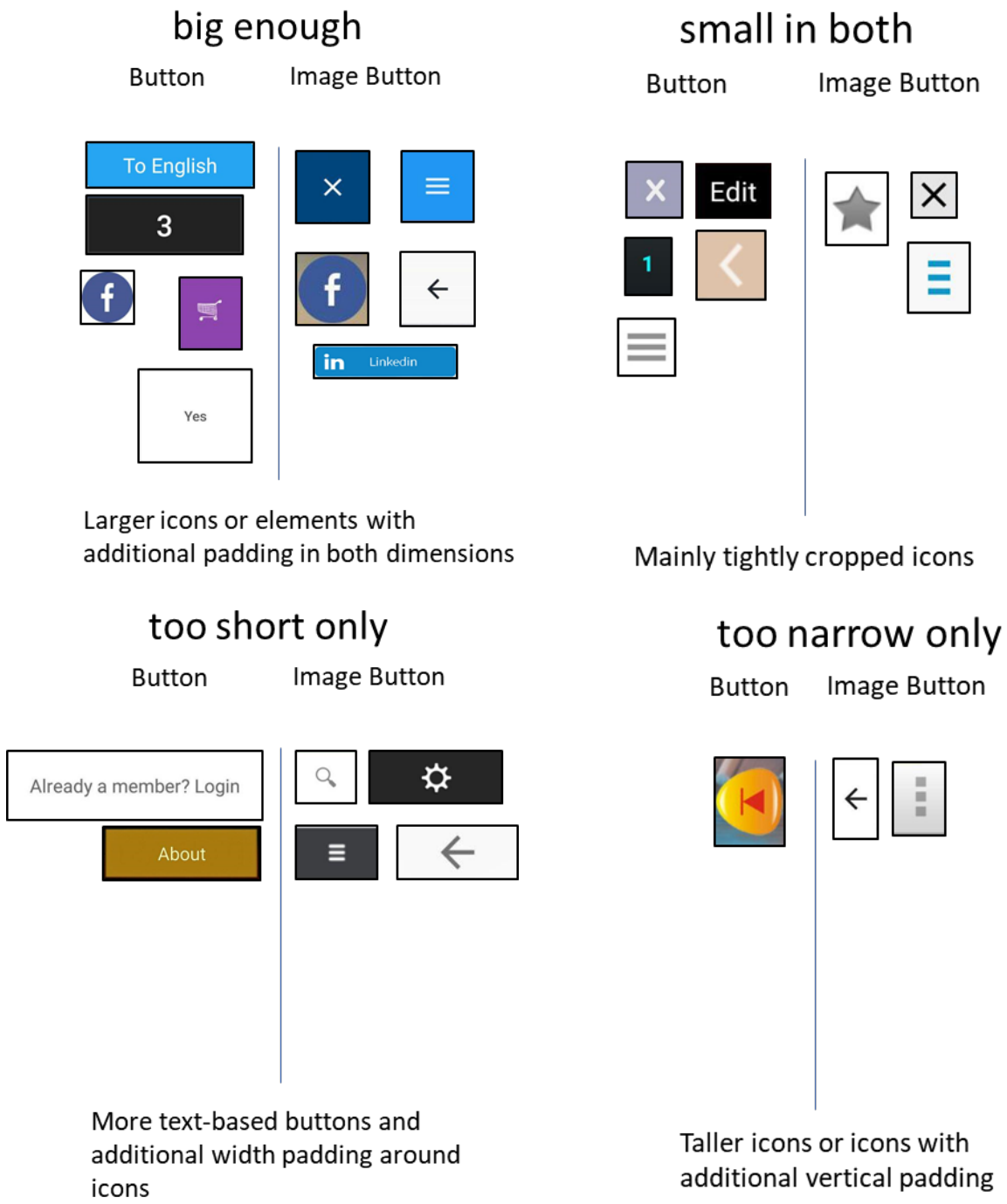


Figure 28: Example Buttons and Image Buttons that were big enough or had size-based accessibility failures in different dimensions: too small in both, too short only, too narrow only.

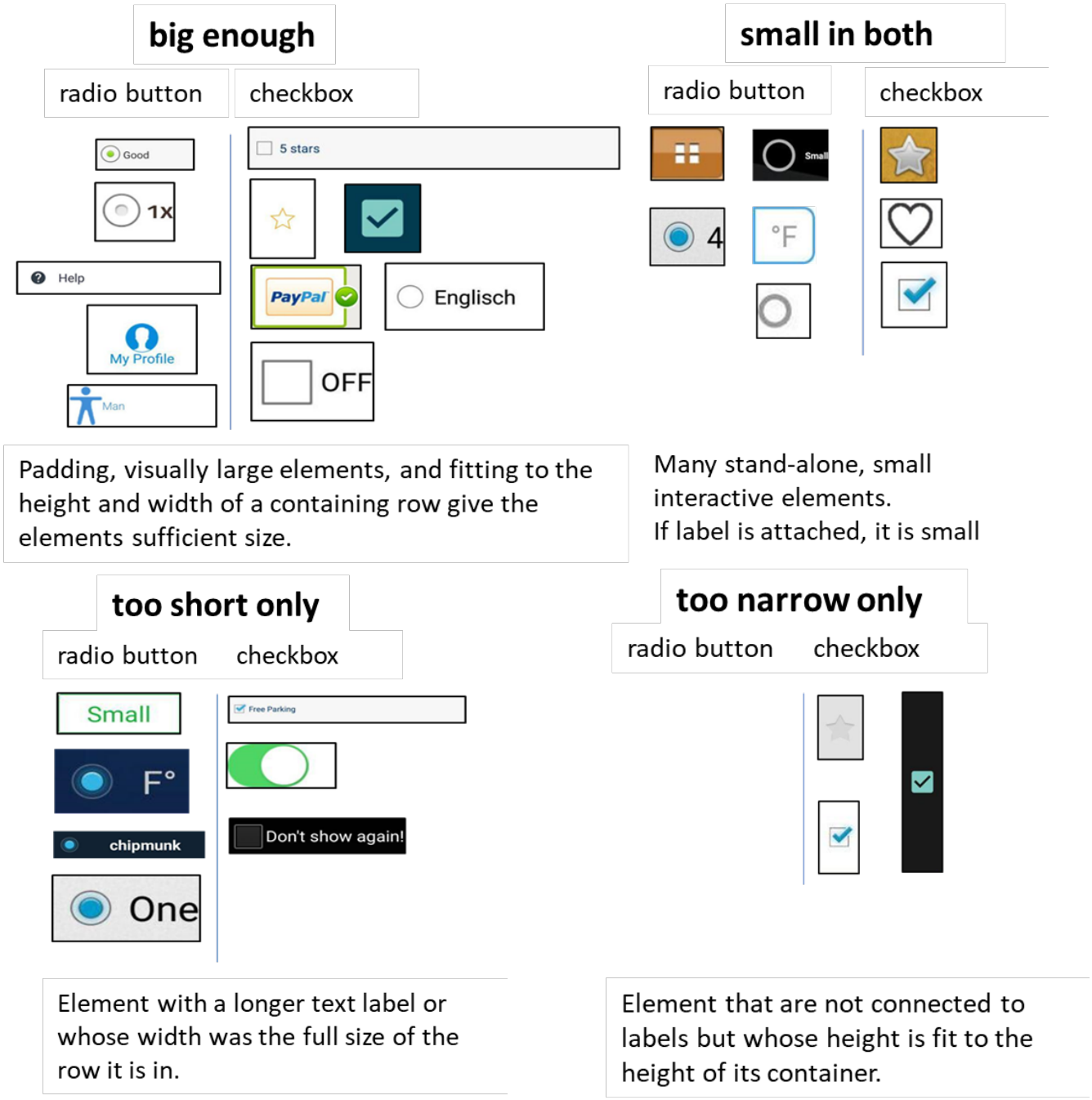


Figure 29: Example Radio Buttons and Checkboxes that were too small in both, too short only, too narrow only, or big enough. What components are included in the clickable element affects the likelihood the element is big enough.

Facebook Login and Google+ elements also had high prevalence of the size-based accessibility failures, as presented in Table 16. Google+ elements were too small in both when the element was comprised of only the g+1 logo (Figure 30). The Google+ elements that were wide enough but too short had a status indicator placed next to the logo (e.g., showing the number of +1's, see Figure 30). In one case, the status indicator was placed above the logo, which resulted in an element that was big enough.

Table 16: Prevalence of size-based accessibility failures in Google+ (`yei` and `ztu`) and Facebook Login elements.

class	# elements	# apps using class	% too small in either	% too small in both	% too short only	% too narrow only
`yei`	318	37	100%	63%	37%	0%
`ztu`	369	56	99%	27%	72%	0%
combined	687	90	99%	43%	56%	0%
`com.facebook.login.widget.LoginButton`	306	132	83%	0.3%	82%	0.7%

The Facebook login elements were almost entirely too short only. The length of the visible text of the button gave the elements width; the few elements that were too narrow only were just icons. However, the text and padding were rarely sufficient to make the element big enough. Figure 30 illustrates these differences.

Figure 30: Sample Google+ (g+1) and Facebook Login elements that had the size-based errors of being *too small in both*, *too short only*, *too narrow only*, or *big enough*. The comparison highlights how design choices made by third-party providers can impact many apps.

5.4 Comparison to Recent Large-Scale App Accessibility Analyses

Since my analyses, additional large-scale analyses have been published that complement my work. I compare some of our findings. Complementing my analysis of 100 apps for seven accessibility failures (Section 5.2.1), Neves da Silva *et al.* [82] tested 111 apps for 14 accessibility failures. They found 12,108 accessibility failures across apps with a median of 44 failures per app (mean: 104, range: 3-1,115).

Missing labels was one of the most prevalent accessibility failures in my analysis. Other work identified missing labels was a similarly problematic failure. Alshayban *et al.* [1] listed missing labels as a top failure with apps having a mean rate of 11% of elements missing labels. Yan and Ramachandran [97] reported 20% of their violations and 78% of their potential violations were from their Description error, a measure of label-based inaccessibility.

As in my analysis, image-based elements were identified as particularly susceptible to missing labels. Yan and Ramachandran [97] identified elements that had Description errors primarily came from the Button, ImageView, and View classes. Vendome *et al.* [91] tested the ImageViews, ImageButtons, and editable TextViews from 13,000 Android apps on Github for missing labels. They found 50% of apps labeled all tested elements; 46% of apps labeled less than half of their elements; and 74% apps were missing labels on all of their tested elements. Their distribution follows the bi-modal trend I found in elements missing labels (Figure 13 and Figure 20) with more extreme peaks at both ends of the distribution. The difference may be due to their app population (Github Android apps versus my free Android apps from the PlayStore) or evaluation measures. Future work comparing techniques and populations could provide insights into additional factors affecting app accessibility. These analyses strengthen my conclusion that missing labels is a widespread and frequently occurring accessibility failure in apps.

In my analysis, the few TalkBack-focusable elements disease had low prevalence, with 8% of apps having the failure, but this problem has high severity. Yan and Ramachandran [97] found a related focus-based accessibility barrier to be prominent in their analyses. Their focus-based accessibility failure tested whether any element on a screen from a predefined set of classes was not focusable. This metric differs from my few TalkBack-focusable elements test, thus the prevalence values reported by Yan and Ramachandran are not directly comparable to my measures. However, Yan and Ramachandran's findings support the importance of focus-related failures. They found that 53% of their measured violations were focus errors. As part of their work to improve app accessibility, Zhang *et al.* [99] manually annotated screen elements from approximately 80,000 screens from around 4,000 iOS apps. They reported discrepancies between manually identified screen elements and elements exposed in the accessibility view hierarchy, similar to my TalkBack-focusable elements. They found 59% of apps had manually annotated screen elements did not have a corresponding element in the accessibility view hierarchy; 4% of apps did not expose any elements in the accessibility view hierarchy. Reflecting the prominence of the failure across apps, they found 84% of apps had at least one un-exposed element.

Fitting within my multi-factor lens, recent studies have also considered the interaction of between app characteristics, development resources, and accessibility. I identified missing labels in some Android documentation and Lint tests. Alshayban *et al.* [1] found 5 of the 10 new project templates provided in Android Studio had low text contrast, undersized elements, and missing labels failures. Similar to my results in Section 5.3.2, they found no strong association between an app's accessibility and rating. Considering app category, Alshayban *et al.* [1] also found Finance apps had the lowest prevalence of missing labels at 4% of apps; Design and Beauty apps had the highest rate of missing labels at 16%.

Ongoing large-scale and multi-factor analyses of app accessibility enrich our population-level understanding of the state of app inaccessibility.

5.5 Chapter Summary

In the language of my epidemiology-inspired framework, my large-scale analyses contribute to a *census* measuring the prevalence of accessibility failures in the population of top Android apps. My multi-factor analysis guided additional insight into what may have impacted the difference in disease prevalence between the classes. Looking to existing treatments, we see differences in accessibility representation in tools for design, development, and testing. The prevalence measures and patterns identified in these analyses informed some of my exploration of improving app accessibility testing and development tools, discussed in the next chapter.

Chapter 6. INFORMING TOOLS TO IMPROVE APP ACCESSIBILITY

In the language of my epidemiology-inspired framework, population-level and multi-factor approaches can inform the creation of new treatments for improving app accessibility. To demonstrate this concept, I present my research on improving app accessibility at three stages of an app's natural history of development and use (Section 4.2.3): development, testing, and post-release, third-party repair.

I first explore novel designs for developer tools that are informed by the trends seen in my large-scale data analyses. I then present my improvements to a professional app accessibility testing tool and my investigation of the larger organization environment professional testers work within. This work was done in collaboration with Google's Accessibility Engineering Team during an internship in the Summer of 2019. Finally, I apply the epidemiology-inspired framework to our runtime repair proof-of-concept technique and reflect on the take-aways the framework highlights. The third-party repair technique was published at CHI 2017 [100] led by Xiaoyi Zhang and co-authored by myself, James Fogarty, and Jacob O. Wobbrock. I suggested a few technical techniques for the proof-of-concept implementation, and collaborated with Xiaoyi Zhang to create the design framework, design and perform the user studies, and write the publication. This project was based on foundational interviews performed by Catherine Feng, Jennifer Niedlerländer, Xiaoyi Zhang, and myself.

Combined, these projects demonstrate the third claim of my thesis (T3) that my epidemiology-inspired framework that emphasizes multi-factor and large-scale analyses can inform the design of tools for identifying and repairing accessibility failures.[94]. Combined, these projects demonstrate the third claim of my thesis (T3) that my epidemiology-inspired framework that emphasizes multi-factor and large-scale analyses can inform the design of tools for identifying and repairing accessibility failures.

6.1 Toward Improving Developer Tools

Implementation details are a key component of app accessibility. A range of automated and manual tools are available to help developers identify problems and guide repair, such as guidelines, runtime scanners, unit tests, and manually using assistive technologies. These tools play an important role in supporting accessibility practices. However, these tools have limitations. For example, automated techniques cannot robustly identify all accessibility failures, accessibility novices may lack the knowledge to perform manual testing or integrate accessibility-specific unit tests. Once an accessibility failure is identified, developers may still find understanding and repairing the failure challenging. Repairs can vary in technical complexity, robustness, and the resulting end-user experience. Most commercial tools provide limited scaffolding for repair; they primarily provide text-based explanations of discovered accessibility failures and point to written documentation

In this section, I explore developer tool designs that educate developers about accessibility and support them in effectively and efficiently improving the accessibility of their apps. I first describe high level goals and concepts that drove my designs. Next, I present my methods for design ideation and prototyping. I then use a set of my designs to talk through the complexities of accessibility repair and how novel designs can meet the design goals. Finally, I reflect on tensions between design goals and future research directions in this space.

6.1.1 Design Goals & Dimensions

In the following subsections, I define the three primary goals of my designs: *efficient*, *educational*, and, *effective*. I then introduce the concepts of *context* and the *runtime-technical implementation gap* that play a role throughout my designs.

6.1.1.1 Efficient, Educational, Effective

Efficient tools support developers in following accessibility practices without the perception or reality of adding substantial time and effort. Developer tools, generally, strive to streamline development practices. This efficiency in particularly necessary for accessibility practices. Accessibility can be neglected if the developer sees it as too resource intensive, especially under competing pressures. Efficient tools have an opportunity to integrate accessibility into workflows rather and counter the perception that accessibility is a separate add-on.

Technical and accessibility knowledge can impact whether a developer can identify and repair accessibility failures. *Educational* tools can improve both knowledge bases. Improving accessibility knowledge can prompt a developer to follow accessibility practices even without tool use and may help developers consider accessibility in other stages of their work. Developers who did not learn about accessibility during their core technical education may resist spending time separate from their work learning about accessibility. Tools that provide accessibility-specific technical education within the setting of the developer's app implementation can aid in the learning process.

Effective tools result in more accessible apps regardless of developer knowledge or engagement. Two components of effectiveness on which my designs focus are the accuracy and breadth of accessibility failure identification. Toward more effective repairs, I explore how tools can refine generic solutions to better fit specific implementations.

6.1.1.2 Context

The context surrounding an accessibility failure can impact a developer's ability to identify it and can inform the appropriate repair. Three layers of context I consider are *element context*, *screen context*, and *design context*. *Element context* provides information about the core element associated with an accessibility failure (e.g., element class name, interactivity, if it is focusable with assistive technology, visual text or images, accessibility label). *Screen*

context includes other elements on the screen and their association with the offending element. This may include visible labels or multiple buttons that provide the same action. *Design context* considers higher-level design intentions of the screen or app such as if it functions as a log-in screen, a gallery of images, or a list of interactive items.

Figure 31 illustrates different types of contexts for a single screen element. The app provides two seek bars to adjust the height and width of a rectangle on the screen. This example app is a slight modification of an official example project provided by Android to demonstrate using Card Views [133]. Let us assume the height seek bar is the element of interest. A view without context may present only a cropped screenshot of the element or other element identifier. Adding element context identifies the element as a seekbar that is interactive, implemented using the `android.widget.SeekBar` class, and does not have a `contentDescription`. Screen context can expand the view to include information about the nearby visual, text-based label element. Design context may identify that the function of the element is to modify a numerical value to visually change a shape on the screen. Existing tools primarily focus on individual elements (i.e., element context), not relationships between elements (i.e., screen and design context).

6.1.1.3 Connecting Runtime & Technical

Accessibility testing should be done throughout the app development cycle to address accessibility failures as soon as possible. Some accessibility failures can be identified in the static, un-compiled, technical implementation (i.e., source code). Other failures are only identifiable at runtime. Tools for implementation analysis (e.g., static analysis, guidelines) benefit from proximity to the implementation. Access to implementation details facilitates more direct repairs, for example, by guiding a developer to the exact line of code where an unlabeled element is defined. Runtime tools provide additional coverage of accessibility failures and more closely reflect the end-user experience. However, repairs provided by runtime analysis can be challenging to act upon as a developer must identify which

components of the implementation correspond to the runtime interface. In my designs, I explore combining the actionability of technical implementation assessment with the coverage of runtime analysis.

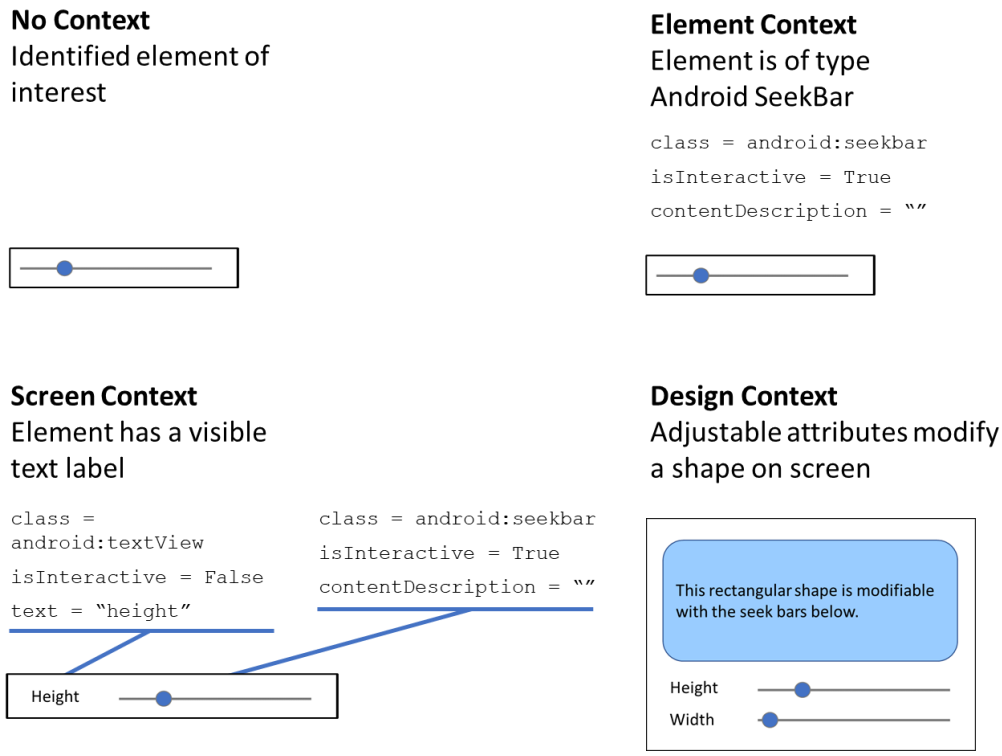


Figure 31: An app's screen element identified with different types of context: no context, element context, screen context, and design context.

6.1.2 Method

I used two methods to explore improving developer tools: design ideations and a tool prototype. My design ideations are unconstrained by technical limitations in existing tools and infrastructures. I then implemented a subset of designs in a plug-in for Android Studio [117]. This work is not intended to provide a technical solution artifact. Rather, the technical prototype allowed me to reflect more directly on how my designs can complement existing workflows. Technical challenges of implementation and my own experience interacting with

my prototype informed further iteration of my designs. I discuss my design ideas and prototype tool throughout my this section.

6.1.2.1 Android Studio Plug-in

I implemented a subset of designs in a prototype tool to explore integration in existing development workflows and technical challenges of bridging the implementation-runtime gap. My prototype focused on label-based and size-based accessibility failures. I built a two-part development tool composed of an Android Studio plug-in and an accompanying app that ran on a physical Android device to test runtime accessibility. The Android Studio plug-in integrates accessibility feedback with the technical implementation (i.e., source code). The assessment app provides runtime information to supplement the technical implementation interface. I chose to extend Android Studio because it is popular for app development, attempts to integrate static and runtime feedback, and has features aimed at a range of skill levels and expertise.

Key technical details of the implementation include encoding robust identification for elements, displaying accessibility information within the existing Android Studio interface, and leveraging runtime assistive technologies.

One challenge of repairing runtime-identified accessibility failures is finding the source code that corresponds to the runtime element. While mapping between runtime and implementation can be time-consuming in general, it is worsened by Android's lack of robust element identification. Toward addressing this challenge, my plug-in automatically generates a `resourceId` for every element declared in the XML layout file and maintains information about the source code location of that element. Consistent use of `resourceIds` gives the runtime testing app and the plug-in a shared element identification scheme. While my technical prototype adds this attribute to elements statically defined in the layout XML, I can envision extending the technique to also set `resourceIds` for dynamically declared elements.

The ability to map runtime elements to technical implementations allows tools to give more direct guidance or automatically change the source code.

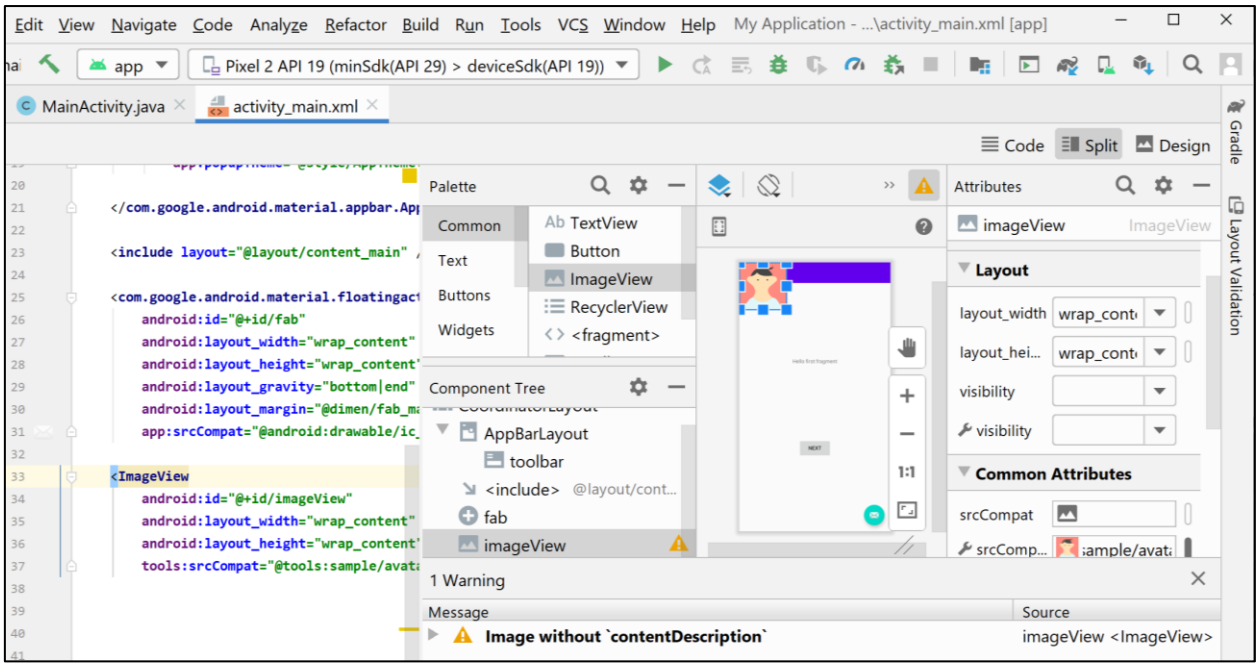


Figure 32: The existing Android Studio interface combines a source code editor, an interactive design interface for adding elements and altering their attributes, and results from the Android Lint tests.

The existing Android Studio interface (Figure 32) combines interactive design interfaces, text-based source code editors, and static analyses (e.g., Linters). These features support developers with a range of technical knowledge. Multiple interaction options support efficiently iterating through stages of design and implementation. My plug-in uses similar design features to integrate accessibility testing into development workflow. My plug-in integrates into the interface as a panel, like the source code editor and interactive design panel (Figure 34). Interacting with elements displayed in the accessibility panel focuses the relevant code in the source code editor and highlights the element in the interactive design display.

The runtime accessibility assessment app runs as an Accessibility Service on a physical device. It captures runtime accessibility information through the Accessibility API. Moreover, through prompting the developer to turn on other assistive technologies (e.g., TalkBack), the assessment app can simulate assistive technology interactions. For example, with the screen

reader turned on, the assessment app performs gestures to linearly navigate the app. It then captures the linear navigation order and details of what the screen reader announces. Runtime screen reader labels reflect interaction between an app's source code and assistive technologies. For example, a labeled favorite button may be announced as "Favorite Button, double click to activate" or "Favorite, checkbox, unchecked" depending on what the developer added as a label, what class was used to implement it, and how the screen reader was implemented to treat different types of elements. I used a physical device in my prototype because, at the time of my prototype, devices most robustly support interactions using assistive technologies and best reflect the end-user experience. As device simulations (e.g., emulators, Android Studio v4.0 Layout Inspector) improve, it is possible physical devices would no longer be needed to achieve my designs.

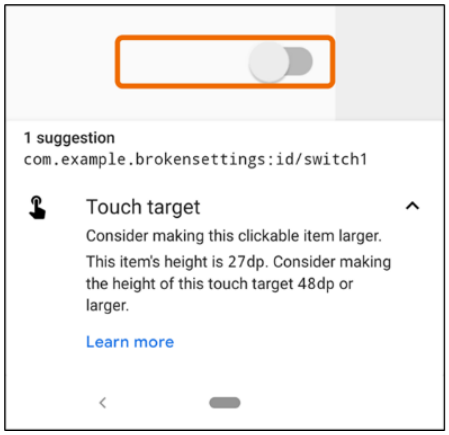
6.1.3 Design Explorations

My design exploration focused on dynamically-sized elements meeting minimum size requirements, supporting accessibility novices in identifying label-based inaccessibility, and improving repair recommendations by leveraging context.

6.1.3.1 Dynamic Sizing

An element's size is determined at runtime to support dynamic resizing and a range of device dimensions. General developer guidelines discourage using absolute values to size screen elements. Rather, element size is defined relative to parent container elements. For example, the size of an element inserted into a Linear Layout container can be defined using the `layout_width` and `layout_height` attributes with `wrap_content` or `match_parent` values. Relative sizing attributes make it challenging to statically test if an element will meet guideline minimum sizing once rendered. Runtime accessibility tests use absolute, runtime values to test element size. However, a developer may get confused when trying to repair runtime-identified undersized elements in source code that uses dynamic sizing attributes

(Figure 33). According to the Android accessibility documentation, one repair for undersized elements is to set the padding and minimum sizing attributes to absolute values that sum to at least 48dp for each dimension [2]. Moreover, in my large-scale analysis, many buttons that were big enough appeared to be visually large or use padding (Section 5.3.4), demonstrating that resizing seems to work in practice. My design prototype uses runtime analysis to identify undersized elements and forefronts the technical attributes that ensure minimum sizing while maintaining dynamic sizing functionality (Figure 34 and Figure 35).




```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">

   <Switch
        android:id="@+id/switch1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
     />
</LinearLayout
```

Figure 33: A runtime assessment tool identifies a switch is too narrow with a rendered height of 27dp, less than the recommended 48dp minimum. Within the source code, the switch's size is defined relative to its linear layout container, not in an absolute dp value.

Within the main Android Studio interface, my plug-in design displays each element's runtime size, highlighting sizes that do not meet minimum guidelines (Figure 34). To optimize efficiency, the "Set Minimum Size" button automatically sets the `minHeight` and `minWidth` attributes to ensure dynamically sized elements render at least 48dp x 48dp. However, this solution may alter the visual appearance of the element. A developer can open the resizing dialog box to explore additional options for resizing (Figure 35).

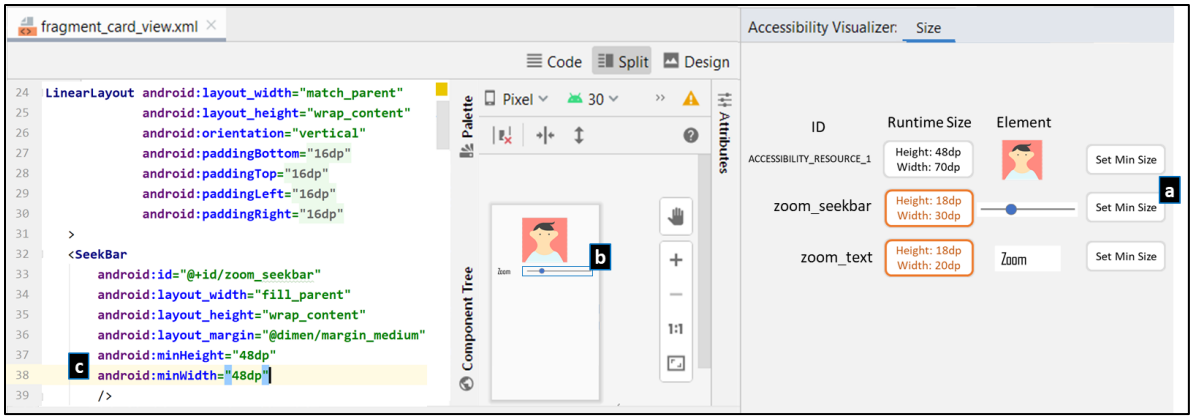


Figure 34: The accessibility testing panel in my plug-in uses runtime measures to identify undersized elements. Connecting technical and runtime, interacting with elements in the accessibility panel (a) highlights elements on the layout editor (b) and in the source code (c). The "Set Min Size" button (a) automatically performs the most efficient repair of undersized elements setting minHeight and minWidth attributes to 48dp (c). Clicking on the runtime size buttons brings a dialog with additional resizing choices.

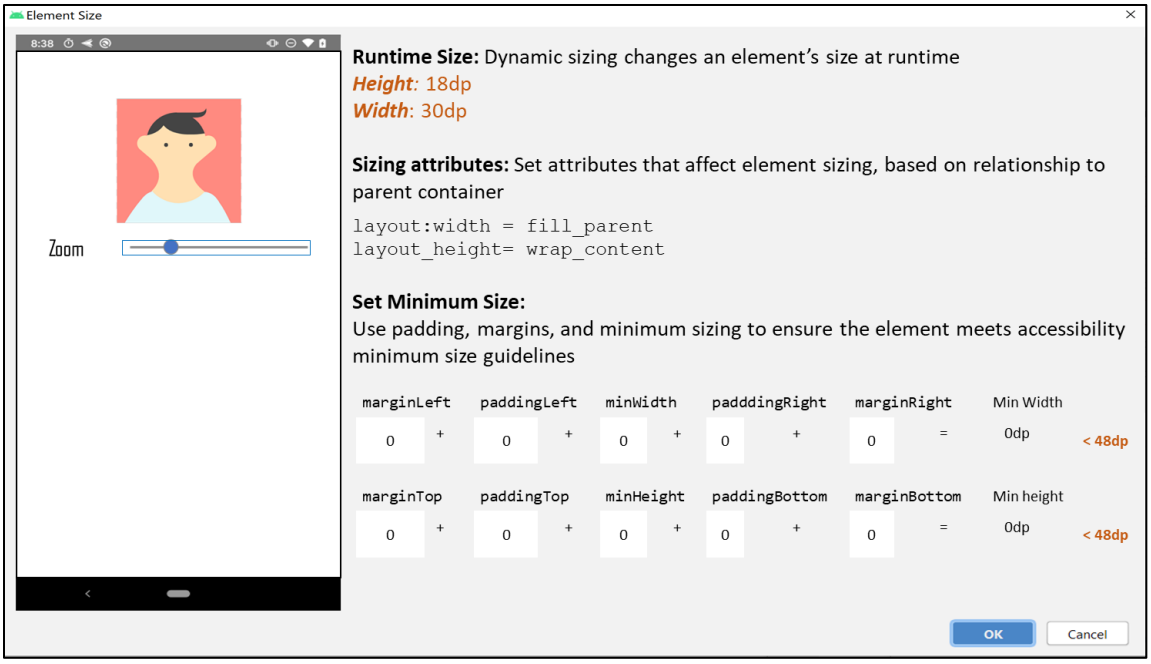


Figure 35: The resizing dialog scaffolds developers in using a range of attributes to ensure, at runtime, elements meet minimum size guidelines, dynamic sizing functionality, and desired visual presentation.

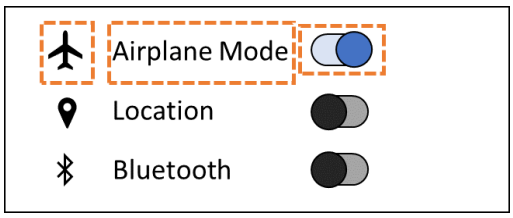
The dialog works toward efficiency even in this more involved repair by briefly explaining the element's dynamic size, displaying the existing attributes that contribute to its size, and creating an easily editable interface to set padding and minimum size attributes to meet minimum values. This dialog makes the suggestions in current written guidelines immediately actionable in the setting of a given app implementation. Moreover, it presents how these attributes work together to create a size rather than requiring a developer to identify the right combination of attributes. By scaffolding developer use of absolute and dynamic sizing, this design aims to help developers achieve accessibility, dynamic sizing, and visual design goals.

6.1.3.2 Leveraging Context to Improve Repairs

One opportunity to improve the effectiveness and efficiency of accessibility repairs is by leveraging element, screen, and design contexts to generate recommendations that are specific to an app's implementation. In this section, I step through an example showing the evolution of repair suggestions by considering additional context. Throughout the following examples, I note how tools (including my prototype) could support these context-aware repairs. The repairs I present were informed by documented accessibility practices, my experience as an intern developer working on accessibility-focused apps at Microsoft and Google, and by patterns seen in my large-scale analyses.

Solution 1 shows an app settings screen with images, text labels, and switch buttons for toggling Airplane Mode, Location, and Bluetooth. In this implementation, each image, text label, and switch are an independent element; they will be independently focused and announced by assistive technologies as described in the "Linear Screen Reader Announcements" in the figure. Accessibility failures in this implementation include the image and switch buttons missing labels, the switch button being smaller than the recommended size, and the redundant focusable targets. Redundant targets (i.e., the image, text, and switch focus independently but contribute to a single functionality) make traversing the screen more time consuming. While the text provides a visible label and a screen reader would

announce it before the switch in linear order, if someone uses the Explore By Touch technique with the screen reader, they can encounter multiple, indistinguishable "On, Switch" and "Off, Switch" announcements.



Linear Screen Reader Announcements

Unlabeled Image

Airplane Mode

On, Switch

Unlabeled Image

Location

Off, Switch …

Accessibility Failures

- Unlabeled Image
- Unlabeled Switch
- Switch too small
- Extraneous targets

Implementation:

- Image, Text, and Switch are independent elements.
- Switch alone is an interactive target

Solution 1: An app settings screen implemented with independent elements for the icon, text label, and switch for each functionality.

Generic repairs suggested by existing tools and guidelines include resizing the interactive switch element and labeling the switch and image. Solution 2 shows one implementation of these repairs, focused on the first setting option. The switch button and image are labeled with `contentDescriptions` and the switch is enlarged using the minimum sizing techniques discussed in the prior section. This implementation addresses the undersized toggle and unlabeled elements. However, the app still has redundant targets and enlarging the switch altered the visual design. Moreover, the developer must now maintain three separate labels,

the two `contentDescriptions` and the visual text label. If the functionality changes, future developers must ensure all labels are updated to match.

Linear Screen Reader Announcements

Airplane

Airplane Mode

On, Airplane Mode, Switch

Accessibility Failures

- Redundant labels on extraneous targets

Implementation

- Switch and Image have `contentDescription`
- Switch enlarged with minimum sizing

Solution 2: The app settings screen accessibility is improved by labeling the image and switch and enlarging the switch. The interface still has the accessibility failure of redundant targets an also requires additional maintenance by developers if the functionality changes.

Leveraging the relationship between the image, text, and switch—i.e., the *screen context*—creates an opportunity for more robust repairs, as demonstrated in the Solution 3. In our settings example, the `labelFor` element attribute can assign the text element as the label for the switch element. To remove the redundant targets from the end-user experience, the `importantForAccessibility` attribute of the non-interactive targets (i.e., the image and text) can be set to `False`; assistive technologies will not focus those elements and screen readers will not announce them. This solution also addresses the problem of maintaining multiple labels.

Existing tools and guidelines may recommend repairs based on screen context, such as `labelFor` relationships. However, due to challenges to automatically inferring element associations, these tools are limited to somewhat generic recommendations (i.e., not suggesting what element should be assigned as the label) and not able to automatically update source code. To support more direct support for screen-context-based repairs, my Android Studio tool design prompts the developer to provide information about relationships

between elements. In the advanced label dialog, my design presents the runtime label, provides a field for the developer to add a direct label (i.e., a `contentDescription`), and an option to select another element that provides a label. If a labeling element is selected, the tool automatically encodes the `labelFor` relationship between the elements. An extension to this design may prompt developers to confirm the associated label did not provide a different interactive functionality and mark the label as not important for accessibility focus.

Linear Screen Reader Announcements

On, Airplane Mode, Switch

Implementation

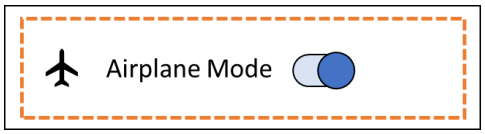
- Text provides accessibility label for Switch using `labelFor` attribute
- Image and Textview set as `notImportantForAccessibility`
- Switch is enlarged

Solution 3: The text label can be explicitly set as the switch label using the labelFor attribute. This reduces the maintenance overhead of redundant labels. Non-interactive elements that do not contribute new information can be set to be ignored by assistive technologies to remove extraneous targets.

The `labelFor` repair is an improvement over the initial implementation, but there are still limitations; the switch's resizing visually altered the design and developers must understand accessibility focus to determine how to set the `importantForAccessibility` attribute. The next level of repair presented in Solution 4 uses additional screen context and non-accessibility-specific implementation best practices to increase the switch's size, provide labels, and reduce redundancy.

While the `labelFor` relationship connects two elements, our example has three related elements: the image, text, and switch. These three elements can be combined into a single

target by placing them in a container element and using the `TouchDelegate` functionality to make the container handle interactions; clicking anywhere in the region of the image, text, and switch will activate the switch. To avoid overlapping clickable elements failures, only one element (i.e., the container element) should handle the interaction. Therefore, the interactive child elements (i.e., the switch) should have attributes defining interactivity (e.g., `isClickable`) set to `False`. The container announces a single label based on its contents. This implementation reduces redundant label maintenance, removes extraneous targets, and expands the clickable target size while minimally affecting the original visual design. Moreover, `TouchDelegate` and `isClickable` functions are more general than accessibility-specific attributes and therefore may be more approachable for developers.



Linear Screen Reader Announcements

On, Airplane Mode, Switch

Implementation

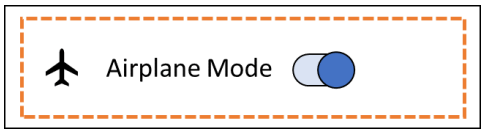
- Image, Text, and Switch are grouped in a container
- Container delegated to accept interaction and inherit label
- Switch is set as not interactive

Solution 4: The text, image, and switch element can be grouped in a larger container element that handles the interaction. This removed redudant targets and increases the clickable region size.

The `touchDelegate` solution is effective but has notable technical complexity. Using d*esign context* can simplify this repair further. The Switch element available in the official Android API supports implementing a switch functionality with image and text labels. Rather than grouping an `ImageView`, `TextView`, and `Switch` element (as in our example), a developer can implement a Switch and use the `text` and `drawable` attributes to label it (Solution 5). This implementation provides the most accessible experience: a large target, inherited labels, and

no redundant targets. Grouping labels and interactive elements (primarily through the class API) was a pattern I identified in radio button and checkboxes that were big enough, further supporting this technique is used and effective.

A tool that could infer the *design context* of an interface—in our example that the group of elements implemented a visually labeled switch button— could guide developers to use existing API elements to meet visual design and accessibility goals.



Linear Screen Reader Announcements

On, Airplane Mode, Switch

Implementation

- Switch element with `text` and `icon` attributes.

Solution 5: Using the text and icon attributes of a Switch element can create an accessible implemention of the settings page visual design with less overhead than using separate elements.

I envision tools that combine automatically detected accessibility failures and element context with annotations elicited from developers to construct additional screen context and design context. The tool could use these contexts to guide developers in more robust repairs or perform them automatically.

This section demonstrates how element, screen, and design context inform more effective treatments. My tools prompt human annotation to construct the context needed to recommend those repairs. I explore how annotation can support more robust testing in more depth in Section 6.2.

6.1.3.3 Label-Based Inaccessibility

In my tools, I explore presentation techniques to support developers identifying label-based accessibility failures. Figure 36 presents example apps I will use in this section. All the images in a music app (Figure 36, left) are missing labels, including key functions such as play and

rewind and informative images like the album art. In a graphing app (Figure 36, middle), the graphing modes are displayed as icon-based buttons with the duplicate, uninformative label "tool image." As discussed above, the settings app (Figure 36, right) has three unlabeled icons and three unlabeled switches that a screen reader may announce as "Switch, On."

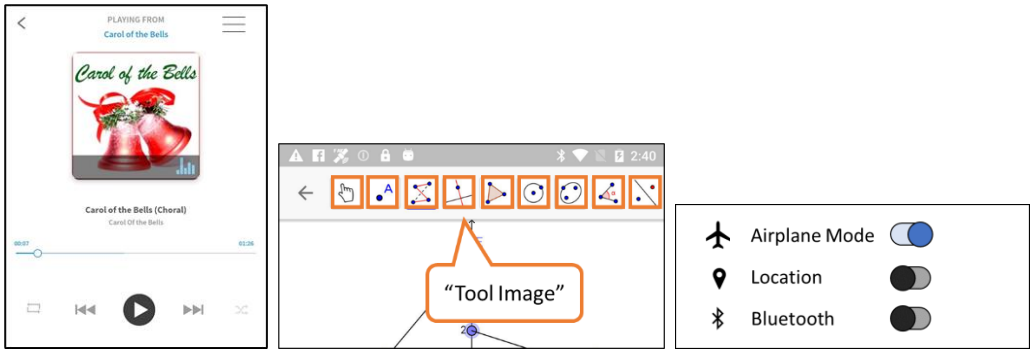


Figure 36: Three apps with label-based accessibility failures. (left) The Freegal music playing app; each of the image-based buttons is unlabeled including the play, rewind, and back buttons and the album art. (middle) A graphing app has duplicate labels; all of the graphing option buttons are labeled "tool image." (right) A settings app with missing labels on images and switches.

Current strategies for identifying label-based accessibility failures include using automated testing tools, using a screen reader, and testing with end-users who use assistive technologies (Figure 37). These practices help accessibility-novice developers understand why labeling matters and how their apps perform as an audio-based interface. However, there are limitations with these techniques. Text-based documentation may not adequately convey the role of labeling to developers who are unfamiliar with audio-based interactions. For example, in the graphing app the developer knew to add a label, "tool image," but the duplicate label failure suggests they may not fully understand the label replaces, not supplements visuals. Learning to use assistive technology is an important skill for performing accessibility assessments. However, a screen reader may overwhelm novice users. Without further scaffolding, a developer may abandon the learning or build inaccurate impressions that using a screen reader is an innately bad experience, similar to critiques of simulation activities [90].

Finally, testing with disabled people and accessibility experts is essential to creating accessible, usable apps. However, using those individuals to identify every accessibility failure, especially for well-understood practices such as labeling, is burdensome and wasteful considering the difficulty in connecting with such experts.

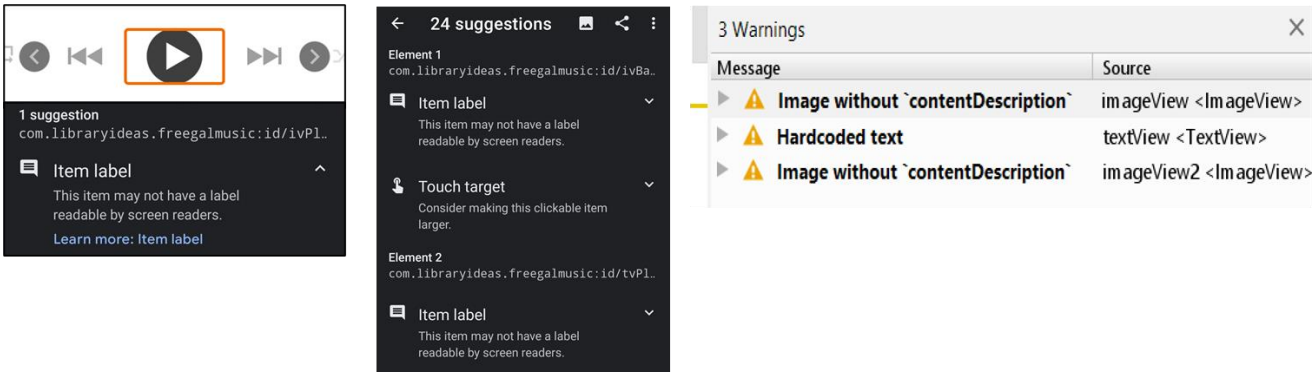


Figure 37: Examples of existing techniques for identifying label-based accessibility failures. A runtime scanner and Android Studio Lint tests identify a missing label failure and provide text-based description of the failures and point to documentation

My designs for identifying and repairing label-based accessibility barriers build on existing tools. I explore visualizations of label-based experiences, connecting runtime label announcements to implementation details, and lowering the barrier for a developer to experience audio-based interaction with their app.

To support developers that are familiar with visual and touch-based interfaces (e.g., smartphone touchscreens), I explore a label-only display design that presents the app screen in its runtime layout with all image-based elements replaced by their accessibility labels. Figure 38 shows this design applied to the music, graphing, and settings apps. This design attempts to balance a familiar visual layout for screen-reader-inexperienced developers while conveying the importance of descriptive labels.

One potential value of this design is its glanceability; a developer who prefers visual feedback can quickly overview their screen to see missing or insufficient labels. For example, a developer who added "tool image" as their icon labels may realize they cannot distinguish

functionality between nine identically labeled elements. For efficient repairs, the labels on the interface could be editable and update the appropriate attribute of the element (e.g., `contentDescription`). A more educational design may guide developers to update the attributes themselves or use the context-driven techniques explored in the prior section. This technique could complement other visualization techniques, such as those by Takagi *et al.* [85] who created a valued tool that visualized the time it took for a screen reader to access website content in linear order.

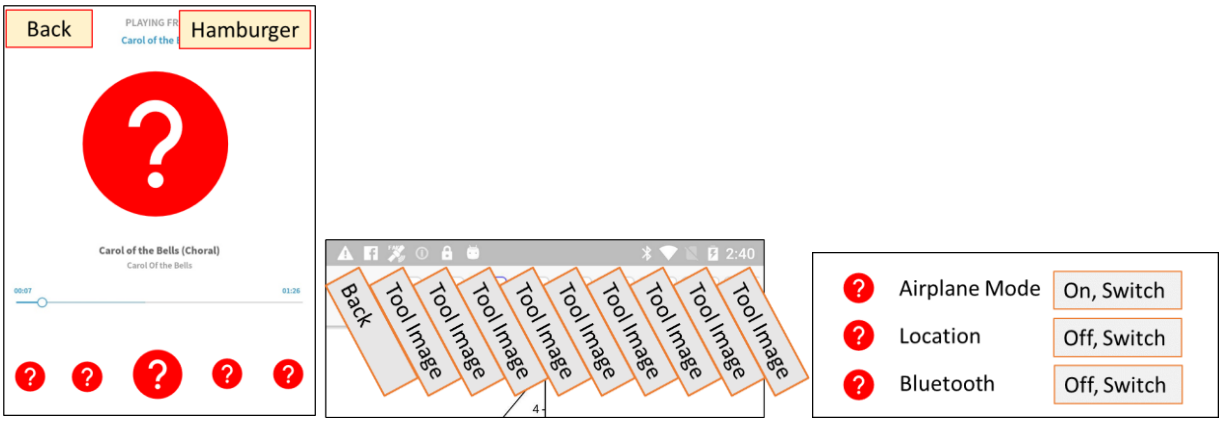


Figure 38: Label-only displays of app screens that replace all image-based elements with their screen reader labels. A large, red question mark indicates a missing label.

While the label-only design may be more approachable than documentation for some, developers may still not internalize the important role of labeling; they may imagine their users will never see a label-only interface and therefore not prioritize the repairs. Moreover, developers may not catch all label-based accessibility failures due to the label-only presentation preserving non-image, visual cues. For example, in the switch example from the settings screen, the developer may use the visually proximate text labels to determine the generic switch labels are adequate. To further educate developers on audio-based interfaces, I envision a tool to capture and replay screen reader traversals of an app. With automated traversal techniques, the developer does not immediately need to learn basic screen reader interactions to listen to an audio experience of the app.

The simulated screen reader experience could use multiple traversal techniques to capture different experiences including using linear navigation and explore by touch methods as well as methods that jump between headers, interactive elements, or links. These playbacks could convey audio-based experiences with the app that may educate about assistive technologies and highlight different accessibility barriers. For example, using explore by touch to traverse the right column of the switch settings screen would playback "On, Switch," "Off, Switch," "Off, Switch," presenting the accessibility barrier caused by not explicitly labeling switches. Salehnamadi *et al.* [77] recently published Latte, a tool with similar functionality that automatically traverses an app interface with assistive technologies to capture accessibility failures. While additional work is needed to reduce the potential negative effects of simulation, techniques such as mine and Latte could complement other efforts to educate developers on audio-based and assistive-technology experiences with their apps.

6.1.4 Tensions Between Design Goals

In the above sections, I discuss how my designs provide efficient, effective, and educational support for developers to address accessibility in their apps. I now reflect on tensions between these goals.

Efficiency can be especially appealing for developers who are not fully committed to learning and integrating accessibility practices in their workflow. When prioritizing efficiency, there are tensions between short-term and longer-term efficiency. Solutions that prioritize short-term efficiency may take longer to maintain in the future, as in the settings label example (Solution 3); multiple, redundant `contentDescriptions` are a short-term-efficient solution to missing labels but multiple labels must now be maintained. Extrinsic factors such as institutional support and company culture are likely needed for developers to prioritize long-term efficiency. However, tools that scaffold repairs can make long-term solutions more achievable.

Some accessibility failures can be repaired both efficiently and effectively. For example, in the label-only presentation of a screen (Figure 38), the visualization may make it efficient for vision-based developers to identify the problem. Moreover, by editing the labels directly on the visualization, they can effectively and efficiently label elements with `contentDescriptions`. However, in this case, education may be sidelined. The label-only visualization allows the developer to make this effective repair efficiently, but if they stop using the tool they may not know the core technical knowledge of labeling attributes.

Like efficiency, short-term and long-term tradeoffs exist with prioritizing education. Educating developers on accessibility practices can improve awareness, help introduce accessibility earlier into their workflow, and help developers take partial responsibility for accessibility rather than consider it the sole job of accessibility professionals. However, teaching developers while they work on production apps that will be released could reduce the accessibility of that app, at least in the short-term. An accessibility-novice developer presented with an audio-based playback of their app may begin to understand screen readers and the role of labels. However, that understanding will initially be limited; they may develop negative biases against how difficult audio-based interfaces are (and thereby underestimate the capacity of daily screen reader users) or may not support more advanced screen reader interaction techniques. Moreover, developers may overestimate their understanding of accessibility or put too much trust in the tool teaching them. This could problematically lead to them underutilizing professional testing and end-user input. Designs that seek to educate developers about accessibility while they are developing apps for release must be mindful of how the assistive technology experience is represented and should highlight limitations and uncertainties within the tool and the developer's understanding. Organizations that adopt such educational tools must ensure their infrastructure also supports input from accessibility experts and end-users; educating developers should add to and not replace these roles.

This work is not the first to encounter tensions in creating efficient, effective, and educational tools to support people with a range of knowledge backgrounds in building technology. My designs explore how these tensions manifest in the accessible app development process. While my designs were not tested with developers, they aim to inspire our thinking in ways tools can be improved.

6.2 Improving Testing Tools within Company Ecosystems

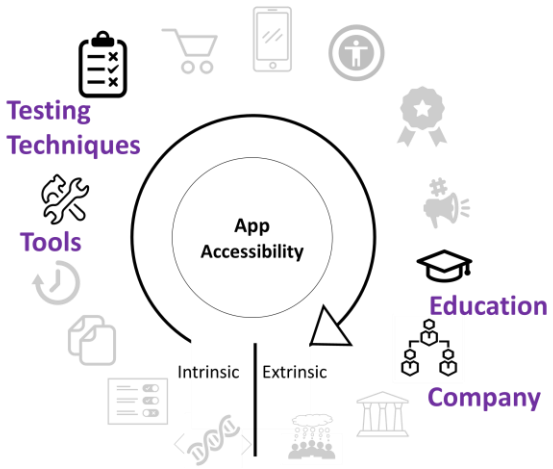


Figure 39: My epidemiology-inspired environmental factor model frames my work on testing tools and professional accessibility testing ecosystems. I consider factors of testing tools, testing practices/techniques, education and expertise of testers and developers, and the company structure where professional testing occurs.

Accessibility testing is essential for identifying accessibility failures before an app is released. Fitting within the environmental factors model (Figure 39), the testing stage of app development has many scopes of influence. First, we can consider how an individual's testing techniques and tools support identifying and repairing accessibility failures. We can then scope out to more extrinsic factors, like how accessibility testing fits into the larger company ecosystem including collaboration between teams, access to resources, and company culture. Within this space, I aim to (1) improve accessibility testing tools and (2) understanding the practices and challenges of professional accessibility testers in a large tech company environment. Toward these goals, I built a testing framework for annotating automated

results to increase accuracy and coverage. I then prototyped a testing tool and gathered feedback from professional accessibility testers. Finally, I began surveying professional accessibility testers and developers that worked with those testers on their current practices and challenges with communicating accessibility assessments between teams.

In the context of my thesis, this work contributes to identifying factors that impact app accessibility (T2) and informing the design of testing tools (T3). The main contributions of this work are:

- An extension to the open-source Android Accessibility Test Framework for Android [43] to support annotating automated test results using a question-and-answer design. My API additions were released in February 2020 for version v3.1 of the test framework.
- A proof-of-concept design for integrating annotation into automated, runtime accessibility testing tools. Components of my prototype design were released in the January 2021 v2.2 update of Google's Accessibility Scanner.
- Identifying professional accessibility tester practices that could inform future improvements to testing tools.
- A preliminary exploration of collaboration practices between professional testers and app developers.

In this section, I first describe the design and technical details of my annotation testing framework extension and testing tool prototype. I present method and participant details for my prototype user study with professional testers. I then present the user study results reflecting the current practices and challenges of professional testers and feedback on the prototyped tool. Next, I detail the method and preliminary results of the Tester-Developer Collaboration survey. I discuss testing tool improvements and factors in the larger testing ecosystem. Finally, I reflect on how my epidemiology-inspired framework helped motivate and situate accessibility testing effectiveness in the collaborative, company ecosystem.

6.2.1 Supporting Human Annotation of Automated Accessibility Testing Results

This work explores using human annotation to reduce inaccuracies and expand coverage in automated accessibility testing tools. Toward this goal, I (1) extended Google's open-source Accessibility Test Framework for Android [43] to support annotation and (2) prototyped annotation functionality into Google's Accessibility Scanner app [44]. Collaborating with Google's Accessibility Engineering team allowed me to have greater access to the tools and to create direct industry impact. In the following subsections, I present my designs for annotating a range of accessibility tests and design details of my prototyped extension to the Accessibility Scanner.

6.2.1.1 Annotation Designs

I used a question-answer design to elicit annotations from testers. The questions validated the results of automated testing and captured additional information where automation coverage fell short. To translate accessibility tests into the question-answer annotation design, I compiled a set of accessibility failures that had ambiguity in automated results (e.g., detecting foreground and background colors for contrast testing), were not covered by the current automated tests, or whose recommended repairs could be improved with additional information (similar to my developer tool design for eliciting context from Section 6.1). I drew the accessibility failures from the current Accessibility Test Framework for Android automated tests and accessibility guidelines.

I considered the following accessibility tests:

- Clear links: Assistive technologies can traverse a screen by jumping between links, similar to traversing by headings. Thus, link test should be stand-alone informative. For example, link text "here" is ambiguous whereas link text "click here to unsubscribe" is clear.

- Touch target size: Following Google guidelines, clickable elements must be at least 48dp x 48dp.
- Errors identified and described in text with suggestions: For example, a form field error should give a text description of the error and how to remediate rather than just visually highlighting the form field in red.
- Caption or transcribed videos and audio: Audio-based media should have a written transcript or captions.
- Custom elements: Assistive technologies may announce information about element type (e.g., "checkbox", "button"). Android API classes provide this information through their `classname`. Custom classes may not have types recognized by assistive technology and must be encoded to provide the most closely related recognized class [134].
- Meaningful labels: Accessibility labels should provide accurate and sufficient information about image-based elements.
- All important elements focusable: Elements should be appropriately exposed in the Accessibility API to be used by assistive technologies. This test is similar to avoiding my few TalkBack-focusable elements failure.
- Color contrast: Contrast ratios foreground and background must meet accessibility guideline minimums [135].
- Linear navigation order: Elements should be in a logical and efficient order for linear traversal with assistive technologies such as a switch or screen reader.
- Logical grouping: Some assistive technologies, such as Switch Access, increase navigation efficiency using a hierarchical, multi-stage navigation process. People can select between larger screen regions and then select a specific element within that region. Developers should group elements in a meaningful hierarchy.

For each accessibility test, I defined the purpose, target screen elements, and a flow of annotation questions. Each question included the prompt wording, information presented to the tester, and possible responses. As an example, Figure 40 shows the question-answer annotation design flow for the all important elements focusable test.

Purpose: Determine if *all important elements are focusable* with assistive technology.

Target Element: Screen regions that contain unfocusable elements.

Question 1:
Prompt: Are there any elements that are important but not highlighted?
Presentation: Screenshot with focusable elements highlighted.
Response: Yes, No, I don't know
(if Yes to Question 1, continue to Question 2)
Question 2:
Prompt: Select important elements on the screen that are not highlighted.
Presentation: Screenshot with focusable elements highlighted.
Response: Coordinates of screen taps to identify unfocusable elements.

Figure 40: A Question-Answer design flow ideation for all important elements focusable test which is currently not performed by the automated tests in the Accessibility Test Framework for Android.

6.2.1.2 Testing Tool Prototype

From the considered set of accessibility tests, I implemented two in the testing tool prototype: text color contrast and all important elements focusable with the TalkBack screen reader. At the time of the project, the Accessibility Scanner had automated tests for color contrast and did not test that all important elements are focusable. I chose these two tests to demonstrate annotation extending both the accuracy and coverage of the Scanner.

The Accessibility Scanner is built atop the Accessibility Test Framework for Android. To build my prototype extension to the Scanner, I first extended the Accessibility Test Framework for Android API to support the question-answer annotation functionality. My prototype extension to the Scanner was built using my extension to the framework API. My API update was released in version 3.1 of the test framework. The color contrast test annotation feature released in the Accessibility Scanner v2.2 in January 2021 was directly based on my prototype and API extension.

Color Contrast Testing

For every color contrast failure that was automatically detected, my tool presented the automated results and a prompt to validate or correct the colors (Figure 41). A tester could initiate the annotation using the "Q?" button on the results page. The first question asked, "Are these the correct foreground and background colors?" while presenting a screenshot of the target element and the automatically-detected colors as a color swatch and hex value. Possible answers were: "both correct," "text color incorrect," "background color incorrect," "text and background color incorrect," or "I don't know". If "both correct" or "I don't know" was selected, the tool kept the automated result and removed the "Q?" button from the results.

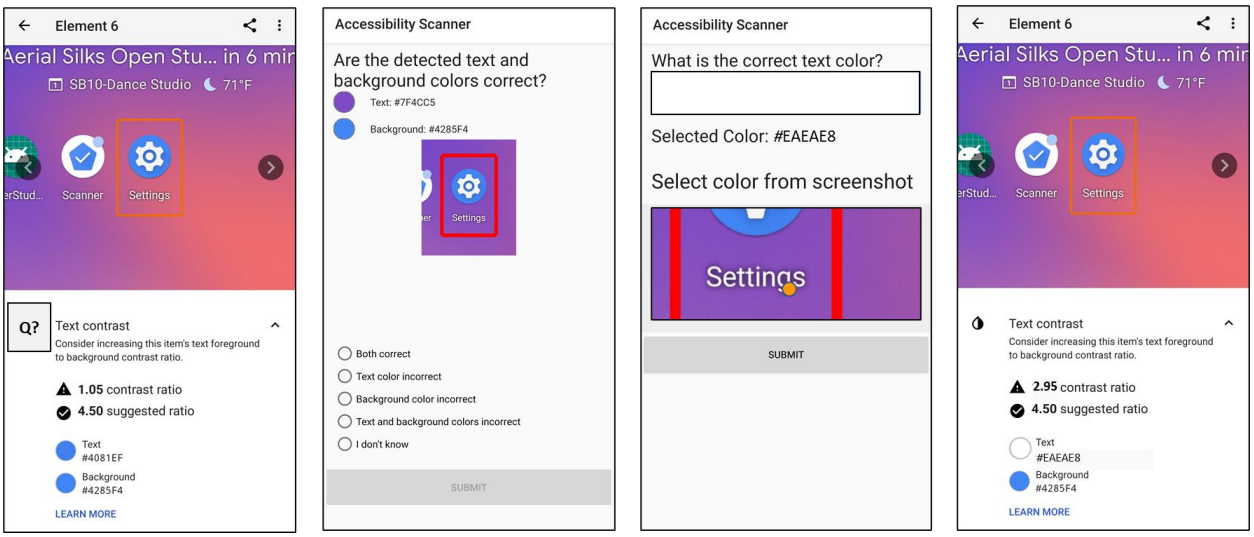


Figure 41: Annotation flow for color contrast testing. Left-to-right. (1) The interface displays the contrast between automatically selected colors. A Q? button indicates annotation prompts are available. (2) The first prompt asks the tester to indicate which, if any, automatically detected colors are incorrect. (3) If any colors are incorrect, the tester is prompted to pixel-select the correct color from a zoom-able screenshot. (4) The updated contrast test is presented in the main testing tool interface.

If either color was incorrect, the tool advanced to the question "What is the correct text/background color?" and presented an interface for zooming in on the element to select a pixel representing the correct color, similar to an eye-dropper color selection tool. The color contrast evaluation would rerun with each color update. The tool provided best-effort contrast

results from available colors after every step of annotation. For example, if a tester answered "both foreground and background colors are incorrect" but only updated the foreground color, the contrast would be calculated with the tester-provided foreground and automatically determined background.

Focusable Elements

The Accessibility Scanner did not have an automated test for checking that all elements are focusable with assistive technology. In my prototyped extension, every screen generated an annotation prompt for focusable elements (Figure 42). Upon clicking the "Q?" button, the first annotation prompt presented the screenshot with every element identified as focusable outlined and the question "Does this screen contain any important items that are not outlined?" Possible responses were "yes," "no," and "I don't know." If the tester selected "yes," the next step prompted the tester to "Double tap on any important items that are not outlined" on the screenshot. Testers could also remove erroneous annotations by double tapping the area again.

Once the tester submitted the annotation, the tool generated an accessibility failure of type "Element not exposed" for each annotated screen region, identifying elements by the pixel coordinates of the tap location. As an alternative to the double tapping, I tried a drag-and-release interaction to draw boxes that better encapsulated different sized screen elements. However, the small touch screen made it challenging to create accurate boxes, especially on screens with dense interfaces. I therefore chose the double tap interaction for the prototype.

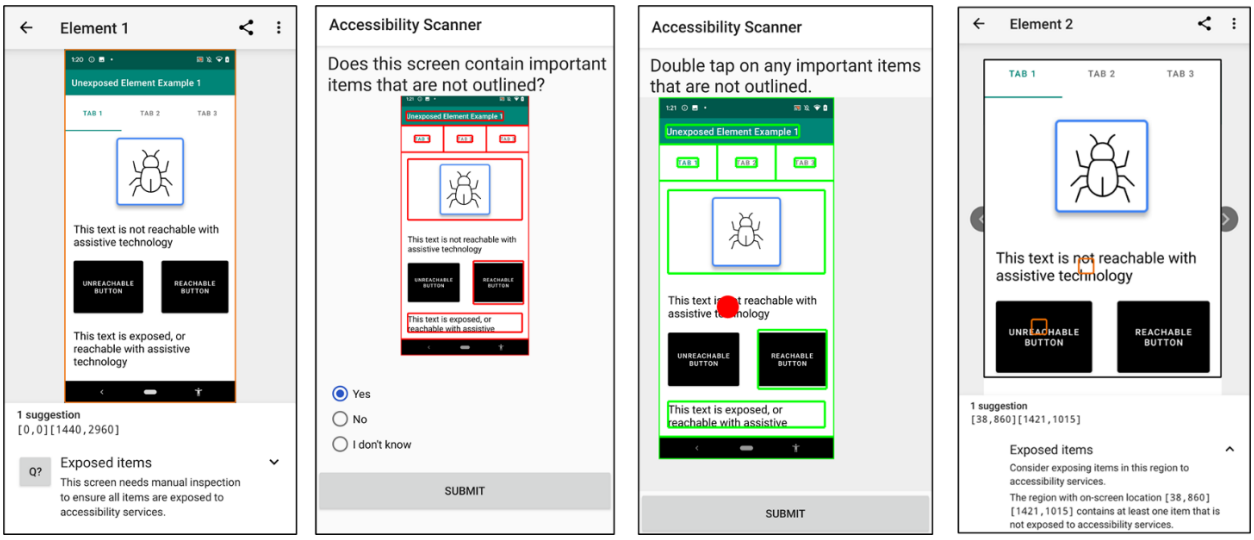


Figure 42: Annotation flow for focusable element testing. Left-to-right. (1) Since there is no corresponding automated test, a prompt is provided on all tested screens, as indicated by a "Q?" button. (2) The first prompt displays a screenshot outlining all elements identified to be focusable and asks if there are any screen elements that are important but not identified as focusable. (3) The tester is prompted to double-tap any screen area where an important but unfocusable element exists. A red dot indicates where a tester annotation. (4) An error is generated for each region selected by the tester. The element is identified by pixel coordinate.

6.2.2 Professional Tester User Study

I performed a user study with professional accessibility testers to understand their current practices and gather feedback on the annotation concept. To probe into the larger collaborative ecosystem of testing, I then began a survey of the professional testing team and developers that worked with them.

Participants completed a pre-study survey, a user study with the prototype tool, and a post-study survey reflecting on their experience. To ensure my methods reflected the language of the professional testers, I iterated the study design with an accessibility user experience researcher, the technical manager of an accessibility project, and managers from the accessibility testing team, including people who had previously held the professional tester role.

6.2.2.1 Participants

I recruited from a dedicated accessibility testing team at Google. Eight full-time accessibility testers with experience testing apps completed the study. Participants varied in years of professional accessibility testing experience: 3 participants had 1-3 years, 3 participants had 3-5 years, and 2 participants had 5-10 years of experience. All participants were sighted.

Testers who were blind or had vision impairments were not able to participate in the study due to accessibility failures in my testing tool prototype. This is a significant limitation of my work, as these testers provide a particular level of expertise from using assistive technology in their daily life that could be very useful for improving testing tools and workflows.

6.2.2.2 Pre-Study Survey

Before the prototype user study session, participants completed a survey on their current accessibility testing practices, including what the most time-consuming aspect of their accessibility testing workflow is, how often they test for different accessibility failures, what automated and manual techniques they use, and how those tools can be improved.

6.2.2.3 User Study

At the beginning of the study, participants were given an overview of the study session structure and purpose, informed their participation would not be shared with their manager or influence their job performance evaluations, and asked if they consented to the study, to being recorded, and to being quoted anonymously in publications. Sessions were video and audio recorded; video focused on the phone interactions. Audio was transcribed with an automated service and corrected by me.

The main task of the study was to test app screens for two accessibility failures: low contrast and ensuring all elements are focusable. Each tester confirmed they were familiar with both accessibility failures. I created 6 pairs of apps screens to test (12 total screens). Each pair was similar in the number of, types of, and complexity of accessibility failures.

Participants performed testing in two conditions: using Standard Testing techniques and using my Annotation Prototype tool. All participants did the standard testing round first. At the beginning of each condition, the participant tested a "warm-up" screen (not one of the 12 screens mentioned above) to ensure they understood the protocol and allowed them to fine-tune the tool settings (e.g., TalkBack speaking speed) without affecting task-completion-time measures.

For each pair of screens, one was assigned to the Standard Testing condition and one to the Annotation Prototype condition for each participant. The screen's condition assignment was balanced across participants so that each app screen was tested four times using each technique. Balancing screens across testing conditions mitigated variations in testing difficulty between apps to allow between participant performance comparisons. In each condition, the assigned screens were presented in a random order.

In both conditions, testers assessed the screens on a mobile phone equipped with standard Android accessibility services (e.g., TalkBack, Switch) and the existent Accessibility Scanner or the annotation prototype Scanner. If a tester would normally use a technique they did not have access to (e.g., a specific program on their desktop computers), I asked that they verbally announce they would use that technique and then complete the testing to the best of their ability using just the phone. A shared office space setup prevented us from performing the study at their workstation.

For the Annotation Prototype condition, testers were asked to use the annotation prototype version of the Accessibility Scanner. The annotation feature was first demonstrated to the testers and they were prompted to ask any questions. Participants then tested a sample app screen untimed until they felt comfortable with the tool. For the tasks, testers were asked to use my prototype annotation tool first, but to then use any other techniques needed to be confident in their results.

For each screen, we measured what accessibility failures the testers found and how long it took to complete the task. If testers announced they would use techniques they did not have access to during the study, we marked that as a successfully-found bug due to their expert knowledge.

Participants were asked to announce when they finished testing the screen for contrast and focusable elements. Task timing started when participants began interacting with the test screen and ended at the tester's announcement. To improve the accuracy of the timing measure, participants were instructed to focus on the task and reserve comments until after they completed the task. If the participants began a discussion in the middle of a task or started testing for failures outside of contrast and focusable elements, a researcher reminded them to save comments or limit their testing. The task timing was adjusted to account for any mid-task discussion that did occur.

Each study session took 1-2 hours. Participants were given opportunities to take breaks. At the end of the session, participants were thanked for their time, reassured their input was valuable and this was not a performance test, and reminded to complete the post-study survey. Six sessions happened in-person at the participants' workplace; two studies were performed remotely.

6.2.2.4 Post-Study Survey

I emailed a post-study survey to participants after the user study session. This survey asked questions about the perceived usefulness of the prototype tool, whether tests were easier or more difficult, and general thoughts around the tool.

6.2.3 Results: Current Tester Practices and Challenges

In the surveys and prototype user study, testers reflected on their current practices and challenges. Due to the scope of the prototype, they primarily discussed text color contrast and all elements focusable testing. In their standard practices, testers use a mix of automated

and manual tools including the Accessibility Scanner, online contrast testing tools, and navigating the app with assistive technologies. Two limitations of automated testing they mentioned were inaccuracies in results and a lack of coverage of all accessibility considerations. Manual testing supplemented the limitations of automated testing and better represented the end user experience. In pre-study survey responses, some testers noted manual screen reader testing and writing reports were two of the most time-consuming aspects of accessibility testing.

6.2.3.1 Color Contrast

Testers described three main stages of contrast testing: identifying foreground and background colors, calculating the contrast between the colors, and comparing the contrast to a guideline-informed threshold.

Many testers first use their professional intuition to "eyeball" the screen and identify potential color contrast problems. They then use tools to test that intuition and collect details for the report. Some testers use the Accessibility Scanner's automated contrast testing. If the automated test seems reasonable (i.e., fits within their intuition), appears to use appropriate colors, and identifies all instances of low contrast, some testers will use those results in their accessibility reports. However, testers stated the Scanner results can be inconclusive, untrustworthy, or not give feedback on all suspicious cases. In these cases, the testers use more extensive, manual strategies to test contrast.

To identify foreground and background colors, some testers search the source code for encoded colors or ask the development team. However, a lack of access to source code, the time and technical knowledge needed to search code, and hesitations around contacting developers made those techniques less appealing to some testers.

The most common technique for identifying colors was uploading screenshots from the mobile testing device to a desktop to use pixel-selection tool. They then put the identified color values

into a contrast calculating website. While common, uploading a screenshot was time consuming. Moreover, it was challenging to ensure the selected pixel represented the encoded color value. In designs implemented as single-color text on a solid background, a range of pixel colors may be rendered due to antialiasing, shadowing, or differences in device resolutions (Figure 43). Multi-colored text and complex backgrounds (e.g., text overlaying an image) further complicated color selection. The public Android accessibility guidelines for contrast do not mention complex cases such as text over images.

Figure 43: The full app screen shows complex contrast cases where white text overlays a multicolored image. Even in the case of two-color design such as the white text on the blue button, antialiasing and other rendering variation produce pixels with a range of hex values.

In cases where exact colors are not exposed to the accessibility service API, the Accessibility Scanner app selects best-guess colors. Potential color section techniques include using the most frequently occurring pixel colors. Contrastingly, testers visually identified the text and the background directly in contact with the text to color sample. In complex cases, such as text over images, testers mentioned selecting the background and text colors that appeared to have the lowest contrast. For example, in Figure 43 the "I have an account" link is white text across an image with shades of white and orange. Automated testing may determine white as the foreground and orange as the background due to the color's high frequency. A

professional tester may instead grab the background color from the light-colored area of the background image as it was directly behind white text and likely has the lowest contrast. In complex cases, testers may perform multiple contrast calculations using different color samples to ensure all text had sufficient contrast from its immediate background.

Guidelines for minimum contrast value are context dependent. Thresholds may be lower for bold text and higher for smaller font sizes. Since the testers often do not have access to the source code and the runtime accessibility service API did not explicitly provide text size, it could be challenging for testers to determine the appropriate threshold. To address this problem, one tester created a spreadsheet with text of varying sizes. He then pulled his font size reference on the test device alongside the app he was testing and visually compared the text sizes to make the best guess at the applicable threshold. Since this study, the Accessibility Service API v30 was released which provides text size.

6.2.3.2 Focusable Elements

Current automated testing does not provide feedback on whether screen elements are reachable with assistive technologies; testers rely on manual testing. Most testers discussed using the TalkBack as their assistive technology to check focus in Android. Some also used other assistive technologies and advanced features of TalkBack, such as traversal settings that highlight specific screen content (e.g., traversing only interactive elements, only links, or only headings). Most testers did not have a dedicated testing cycle for checking that all elements are focusable, but rather test for many accessibility failures as they navigate the app screen. Some testers noted that if they notice an un-focusable element, they may use additional methods to get more complete feedback, such as a different traversal settings or navigation strategies (e.g., linear navigation versus explore by touch).

6.2.4 Results: Prototype Feedback

The user study explored if annotation tools could support more efficient and accurate testing. I present qualitative and quantitative results from the user study and related surveys reflecting on the potential value, limitations, and opportunities for annotation tools.

Table 17: Time spent testing each screen using Standard Testing and the Annotation Prototype. Each A and B screen set represents one of the 6 pairs. Each screen was tested by four testers using Standard Testing and four testers using the Annotation Prototype. The average difference between the Standard Testing and Annotation Prototype conditions displays how much faster, on average, testing with the Annotation Prototype was. A negative, italicized number indicates Standard Testing was, on average, faster.

	Standard Methods (min:sec)			Annotation Prototype (min:sec)			S-A Average Diff (min:sec faster with A)
Screen	**Average**	**Min**	**Max**	**Average**	**Min**	**Max**	
1a	2:39	1:10	4:45	2:08	1:31	2:49	0:31
1b	2:11	0:58	4:45	2:27	1:54	3:22	*-0:09*
2a	2:26	1:22	4:24	2:29	2:04	3:14	*-0:03*
2b	1:40	1:01	3:00	2:28	0:49	5:40	-0:48
3a	2:27	1:31	3:30	1:32	0:45	2:28	0:55
3b	1:55	0:29	3:27	1:43	1:18	2:17	0:12
4a	1:48	1:29	2:19	0:54	0:41	1:17	0:54
4b	1:09	0:21	2:40	1:52	0:59	3:02	*-0:43*
5a	2:09	1:23	3:24	2:10	1:33	3:07	*-0:01*
5b	2:36	0:55	5:38	3:19	1:03	5:12	*-0:43*
6a	2:35	1:43	3:43	2:24	1:39	3:00	0:11
6b	4:59	1:49	11:00	4:15	3:01	5:51	0:44
							Avg: 0:06

Table 17 presents summary statistics on the time taken to test each screen using Standard Testing and the Annotation Prototype. I do not assess for statistically significant differences in testing times due the relatively small number of participants who tested each screen using each method and limitations in the time measurements, described below. However, the trends in testing times combined with qualitative feedback suggest annotation is a promising technique for testing tools. In the following sections, I detail participant feedback on the potential usefulness and limitations of annotation-based testing tools.

6.2.4.1 Limitations

Many factors confounded the time-on-task measure. First, due to a lack of familiarity with the prototype and usability problems in the design, some participants spent time trying to find and understand the new functionalities. Another confound was due to us asking the testers to only test apps for low text contrast and that all elements were focusable. This focus reflected the scope of my prototype and reduced the study time. However, the scope clashed with the more comprehensive testing workflow testers normally use. Some testers were hesitant to announce they completed the assessment or spent time detailing other accessibility failures they identified. The desire to be thorough was likely exacerbated due to the research team being perceived as influential over participant jobs despite being assured otherwise. Many testers also thought aloud and reflected on their process and the prototype during the task. Finally, the length of the study likely caused fatigue or boredom for some participants. The task times presented can therefore not be interpreted as the time testers actually take to perform their jobs. However, the timings combined with qualitative feedback give insight into the strengths, weaknesses, and opportunities of integrating annotation into testing tools.

In addition to affecting time-on-task, the relationship between the testers and researchers likely introduced biases throughout the study. As noted above, testers may have viewed the research team as having higher standing within the company. Testers also knew the research team were also developers on the Accessibility Scanner product. They may therefore be biased toward giving positive feedback about the Accessibility Scanner and my prototype extension. I tried to mitigate this effect by emphasizing this was an exploratory intern project and their honest feedback would help us improve the Accessibility Scanner.

Finally, some testers were wary of automated tools replacing their jobs. Some were therefore hesitant to associate with our work on improving automated tools. We worked to convey to the testers that we aim for this tool to support not replace their expertise. While not the

intention of our work, it is a valid concern that organizations may undervalue human experts as automated techniques improve.

It is important to keep these limitations in mind while interpreting the results in the following sections as they likely created bias both in favor of and against my prototype. Further, the potential impact of standard practices, social dynamics, and job security concerns on my results reenforces the importance of exploring the wider ecosystem of app accessibility testing.

6.2.4.2 Text Color Contrast Testing

Overall, the contrast test annotation feature was well-received and showed potential to improve the accuracy and efficiency of testing. Positive feedback included streamlining on-device testing as opposed to exporting to desktops. One tester also noted that the annotation prompts made them more aware of inaccuracies in automated tests. Opportunities for improvement included having a better color-selection interface and a desire to test colors beyond those flagged by the automated testing.

Although the time-on-task results (Table 17) do not show notable differences between Standard Testing and the Annotation Prototype, the study did not capture time that would have been spent on external techniques (e.g., uploading screenshots to the desktop, searching source code). One tester estimated the desktop upload and color testing process took them five minutes. During the user study, testers frequently stated they would use desktop tools to perform or confirm contrast tests in the Standard Methods case. Some testers noted they would still use their desktop tools in the Annotation Prototype condition despite pixel color selection being supported on device. Reasoning included not trusting the annotation tool and finding the on-device pixel selection (zooming and pixel selecting with a finger) challenging and inaccurate.

Even with the challenges of pixel selection, testers liked the idea of doing on-device pixel selection for contrast testing to streamline their process. In addition to correcting erroneous automated results, as supported by my prototype, testers wanted the ability to test any colors on the screen. This functionality would allow them to capture contrast failures not identified by the automated methods. This feedback suggests annotation could support more efficient testing, especially with a better color selection design.

6.2.4.3 Focusable Elements

Overall, testers did not find the focusable elements annotation helpful for efficiency or accuracy. As noted above, testers often manually traversed the app with TalkBack to test for element labels, for navigation order, and for an overall correct and usable experience. Elements not being exposed were identified during that process. Some testers noted that they would never fully replace manual testing with automated tools since manual techniques most closely reflect the end-user experience. One potential value in the annotation tool was in alerting testers to problems they should pay special attention to in manual testing. Most positive feedback on this functionality was in the potential to use the annotations in their reports. Some testers included marked up screenshots in their standard reporting practice. As with the color-selection workflow, annotating the screenshot on device rather than exporting to a desktop tool could be useful.

6.2.5 Method: Tester-Developer Collaboration Survey

During the prototype user study, testers highlighted that their collaboration with developers was key to ensuring accessibility failures were addressed. To better understand the dynamics between developers and accessibility testers, I surveyed the professional testing team and developers who worked with the testing team on their apps.

The survey had very few responses and is biased toward developers with accessibility knowledge. While these limitations must be considered when interpreting results, I include

the responses as they reveal the complexity of environmental factors that contribute to successful app accessibility evaluation and can inspire avenues for providing more support to app creation teams.

6.2.5.1 Participants

The survey was sent to the professional accessibility testing team from which my user study participants were recruited. For the developer perspective, the accessibility engineering team at Google sent the survey to their engineer contacts. To date, two developers and two testers have responded. I did not capture personally identifiable information in the collaboration survey so cannot report if the tester respondents were participants in my prototype study.

One tester had 1-2 years of experience working in accessibility, the other had 3-5 years of experience. The survey asked how many accessibility failures they have reported in their professional career at the company. Both testers had reported more than 10 low color contrast errors in mobile apps. One tester had reported 5-10 non-focusable elements failures and the other had reported over 10.

For the developers, one had been working on a mobile app development team for 3-5 years, the other for 6-10 years. Both developers reported "fully understanding color contrast requirements" (5/5 on Likert scale) and often or occasionally tested for color contrast in their work. Both developers had worked on more than 10 color contrast failures. One developer was very familiar (4/5 on Likert scale) and one was extremely familiar (5/5 on Likert scale) with the requirement to make all elements focusable with assistive technologies. They often or sometimes tested for this requirement in their work. Both developers had worked on 6-10 non-focusable element failures on mobile.

6.2.5.2 Survey

The survey asked a series of Likert-scale and open-ended questions on the tester or developer's experience in their field, knowledge with accessibility practices, experience with

accessibility testing, and tools used during that process. The survey also asked about collaboration between the testing and development teams including what information is most helpful to receive to test or repair an app, what information is least helpful, what are the biggest challenges, and how to improve that collaboration.

6.2.6 Results: Tester-Developer Collaboration

One main communication from testers to developers was the report of accessibility failures found during testing. Many testers named writing reports as one of the most time-consuming processes. Reports needed to be educational, actionable, and relevant to developers. An educational report ensured developers understood the failure and where the result came from. An actionable report provided sufficient information for developers to efficiently fix the failure. Relevant failures were within the scope of what the developer could change. Current practices and challenges in each of these components of reporting are described below. I draw insights from the user study and tester-developer collaboration survey.

6.2.6.1 Understanding

Differences in expertise between technical development and accessibility requirements created challenges in collaboration. As one developer noted:

> *"[Developers and testers] often use different terminology/vocabulary, both in terms of an accessibility defect and when referencing different components/UI within an application. This makes comprehension somewhat difficult when conveying information about a defect in written form." (Developer, Tester-Developer Survey)*

A tester similarly stated:

> *"We should bridge the technical knowledge gap between testers and developers ... For testers who work on UI elements and surface components of applications or sites, we should have HTML training and resources.*

> *For the app development teams, a lack of familiarity with a11y [accessibility] can hinder meaningful discussions with testers… development teams should also learn about the fine points of accessibility and what it all means." (Tester, Tester-Developer Survey)*

Both respondents raise the challenge of communicating across expertise gaps. The tester identified education as a potential intervention while the developer noted the role of communication medium (i.e., accessibility report contents).

Testers used a variety of details and media to convey the accessibility failure, including information about the testing device, pointers to documentation describing the problem, and screenshots. One developer preferred video to screenshots, noting:

> *"It was helpful to see [the tester's] interactions with a device/assistive technology to know precisely what they believe is problematic. A screenshot is somewhat helpful, but a video of their interactions with TalkBack, for example, is usually helpful for quickly understanding the perceived issue." (Developer, Tester-Developer Survey)*

On videos, one tester remarked that uploading videos took a relatively long time, highlighting a tension between tester efficiency and developer efficiency.

Another knowledge gap occurred around the information available to testers about the app they are testing. One developer commented it was challenging to work with testers who did not know the app very well. The developer desired testers to provide feedback on the expected output of a screen reader or suggest labels for elements. Testers voiced barriers to meeting that developer desire; testers sometimes lacked information such as what use cases to focus on or sufficient details on the intended functionality of an element or app. This lack of information could hinder the ability of testers to scope their testing and make detailed reports.

6.2.6.2 Actionable and Efficient

Once a developer understands the existence of an accessibility failure, they want information to make efficient repairs. The two developers who completed the survey listed colors detected and screen element identifiers as the most important information for fixing low contrast failures. Element identifiers and a description of what expected functionality is missing were the most important information for elements not being focusable failures.

Testers noted hex color values, calculated contrast value, and device used as important information to provide in low color contrast reports. Element identifiers was listed as important information for un-focusable elements failures. Testers listed documentation as an important component of all accessibility reports.

For color contrast testing, many testers emphasized providing the correct hex color. A few testers noted that if their identified hex codes deviated from the values developers found in the code, this could create extra work through a back-and-forth with developers or may reflect poorly on their job performance if the failure is ultimately dismissed by developers due to color inaccuracies. As a tester stated:

> *"Unless we know the hex value is the right value, we'll not be able to get complete confidence…That question actually comes up every time. So how do we interact with the product and the product team is going to get back to the saying that this is not the right [value]" (Tester, Prototype User Study).*

The need for color values created a tension in the testing workflow; as noted above, exact hex values are difficult to identify. A few testers were sometimes uncomfortable reporting a low contrast failure if they did not feel confident in their hex values. Another tester noted that inaccuracies were to be expected and reflected that uncertainty by reporting contrast calculations values to fewer decimal places.

One of the two testers who responded to the tester-developer collaboration survey stated the calculated contrast value was the most important information to provide whereas the exact colors used in the calculation were not important. This belief was driven by an effort to educate developers that contrast depends on the relationship between colors, not the colors themselves. As the tester responded to what information is most useful to get color contrast failures repaired:

> *"The ratio between the text and the background. This can help developers understand that the ratio, not color choice, is the important part of the color contrast guideline." (Tester, Tester-Developer Survey)*

Developers and testers both recognized the need for element identification. However, robustly identifying screen elements can be challenging in Android. Developers did not explicitly state what constitutes a useful identifier but noted pixel coordinates on a screenshot were the least useful. One technical identifier comes from the `resourceId` attribute of an element. However, elements are not guaranteed to have a `resourceId`. For elements that do, there are not robust methods for a tester to identify the `resourceId` at runtime. One tester in the survey noted using visible labels to identify an element on a screen. Other testers mentioned using annotated screenshots to identify elements, potentially referring to the least preferred method of the developers.

Documentation played multiple roles in tester reports. Some testers used documentation to validate their reports, for example, by linking the color contrast guidelines. Others referred to guidelines to support their repair suggestions. For example, a few testers noted during the user study that placing text over an image does not follow accessibility best practices. In addition to identifying the text over image was a contrast problem, they would suggest an improved design such as using an opaque background or outlining the text. One tester noted they would first check the documentation before making the recommendation and link that documentation in their report.

6.2.6.3 Relevant

Even in instances where developers understood and agreed that an accessibility failure was present, they could not always repair the failure. As articulated by a developer:

> *"The most common reason for being unable to repair a contrast issue is because the problem originates in UI not owned by the app, but rather a shared component or library, or an embedded UI from another source (like web content owned by another team)." (Developer, Tester-Developer Survey)*

A tester who filled out the survey acknowledged that not all accessibility failures could be repaired, but articulated challenges in prioritizing their own work due to a lack of information:

> *"Because I'm not on the team to which I'm submitting bugs and don't know details, many bugs are unfixable because of external dependencies, are obsolete, or something like that which makes the bug a waste of time." (Tester, Tester-Developer Survey)*

Filing irrelevant accessibility failure reports can waste the tester and developer time. Moreover, this may impact any job performance metrics that depend on successfully repairing failures. However, testers did not always have sufficient information about the app to identify elements in the app that should not be tested.

6.2.7 Reflecting on Testing Tools and Ecosystems

In this section, I discuss how expert insight can inform accessibility testing tool design and on the larger ecosystem of factors that impact tester-developer collaborations.

6.2.7.1 Improving Testing Tools

Based on feedback from my user study, I consider how professional testing tools can be further improved by supporting freeform testing, incorporating expert contrast color testing methods, and supporting contrast threshold selection.

Many current accessibility tools, including the Accessibility Scanner, prioritize reducing false positives; accessibility non-experts may abandon a tool that flags too many accessibility failures erroneously. In contrast, professional accessibility testers wanted to investigate any *potential* failure efficiently; they were more concerned with false negatives in automated tests. For example, during the user study, testers wanted to use the convenience of the prototype's contrast calculator and report generator to test any color on the screen. Slight modifications to the Scanner prototype could allow a more freeform testing. In the released update of the Scanner that supports color contrast annotation, the color selection interface is not limited to a cropped screenshot, as it was in the prototype. Although not an intended feature, this allows testers to perform more freeform contrast testing. In a discussion with the Accessibility Scanner developer team, they mentioned feedback on the released color contrast annotation functionality suggested experts liked that testing flexibility. A more intentional implementation (e.g., being able to get to the color selection interface without needing an automated contrast issue as the trigger) could be even more useful.

Prompting Scanner users to confirm or correct automated two-color contrast testing is a promising move toward improved accuracy. For the two-color contrast test annotation in my Scanner prototype, some professional testers noted it drew their attention to inaccuracies in automated results; an important detail as automated tools increase in use and influence. There is also an opportunity to use expert practices to improve the tool's complex contrast testing. For complex images, professional testers often tested the contrast of the text to its least-contrasting immediate background or did multiple contrast tests. In the automated pass, the tool could follow similar practices by identifying the foreground content (e.g., text) and compare all pixel colors that are in a certain vicinity of that foreground content. This algorithm may be more likely to capture problems in complex images.

Finally, while the Scanner does support setting custom contrast thresholds, there is an opportunity to support selection of those thresholds. For example, the tool could try to

determine text size or text attributes such as bold. If these attributes could not be automatically detected, the tool could integrate professional workarounds such as presenting text of a known size alongside the app to allow testers to make a best guess. This added support could help the testers in their more complete, nuanced evaluations. The tool-provided details could also help bridge the communication gap between testers and developers by providing additional technical details as justification to developers.

6.2.7.2 Testing Ecosystem

Many factors impact tester-developer collaboration on app accessibility. These range from intrinsic factors related to individual practices to extrinsic factors regarding the work environment. My research thus far focused on tester practices and barriers testers face trying to meet the needs of developers (i.e., producing understandable, actionable, efficient, and relevant accessibility reports). However, accessibility is a shared responsibility; these problems cannot be solved by testers alone.

In this discussion, I address three layers of factors beyond the tester that impact app accessibility: (1) individual developer practices, (2) social dynamics between teams, and (3) company-driven culture and incentives. Further study is needed to understand the magnitude of that influence and potential interventions. While this discussion is prompted by my research, exploring the full pipeline of accessibility reporting and the team's institutional structure was outside the scope of this work. My discussion does not reflect details of the Google teams I worked with.

Developers play a direct role in the app accessibility testing workflow. They may provide the app and supplementary information to testers and influence the expected content of accessibility reports. Developer submission practices contribute to challenges surfaced in my results. Element identification is one such example. The two developers I surveyed noted the importance of robust element identification that was not based on screenshots. Testing tools,

infrastructure, and tester education can only partly meet this expectation. Developers must understand the obstacles testers face (e.g., not having access to source code), collaborate on mutually achievable and useful identification schemes, and submit sufficient information to meet those schemes (e.g., an un-obfuscated app APK with `resourceIds` on all elements). Developers similarly affect a tester's knowledge of an app, what is in scope for developers to fix, and developer preferences for report details; all challenges surfaced in my results. Understanding how developers approach their role in accessibility and exploring techniques to scaffold submitting apps for accessibility testing could improve collaboration and testing outcomes.

Social and power dynamics between the teams affects collaboration and the distribution of personal responsibility for accessibility. For example, testers discussed hesitations of "bothering" or "overburdening" developers and said it was on them to develop technical competencies to communicate with developers. My discussions with developers did not surface similar hesitancies. Notably, in a response presented above, a developer stated the value of having videos was "to know precisely what [the testers] *believe* is problematic" (emphasis added). Identifying tester feedback on accessibility problems as a *belief* positions the developer as the primary judge of accessibility report validity. More work is needed to understand if developers worry about overburdening testers with accessibility questions, if they feel the need to individually increase their accessibility knowledge before engaging with testers, and what authority they give to accessibility feedback from professional testers. Understanding these social and power dynamics is important for a well-rounded approach to improving app accessibility in a company ecosystem.

Personal responsibility and collaboration dynamics are impacted by company structure including explicitly through incentives, resources, and performance metrics and implicitly through culture, work hierarchies, and expectations. Ideally, accessibility would be prioritized enough to empower testers to voice all accessibility failures and incentivize developers and

testers to work together to make the most accessible app. Building on the above example of tester reservations, some testers noted hesitancy in submitting reports if they were uncertain in the result or if they felt the report did not have adequate information for developers. This hesitation benefits developer efficiency: reports are optimized for low false positives and to provide accessibility and technical repair details. However, this type of reporting could disadvantage tester effectiveness or efficiency; some inaccessible app functionality may not be reported, or testers must spend time finding adequate documentation and writing reports. How organizational hierarchies and infrastructure prioritize certain roles and distribute responsibility can impact app accessibility.

6.2.8 Summary

A range of factors impact whether accessibility failures are identified and repaired in apps (i.e., the ecosystem from my epidemiology-inspired framework). Professional testers perform more robust and accurate accessibility testing than automated systems alone. Tools cannot and should not fully replace professional testers. However, semi-automated tools can support testing processes. Toward the goal of improving the accuracy and efficiency of professional accessibility testers, I prototyped an annotation functionality in Google's Accessibility Scanner. Based on feedback from professional testers, the annotation functionality seems promising.

The professional accessibility testing team also surfaced tester-developer collaboration as a more extrinsic factor that impacts the success of app accessibility testing. My epidemiology-inspired framework provides a structure to improve testing tools and situate that tool use into the larger ecosystem of app accessibility.

6.3 Third-Party Runtime Repair

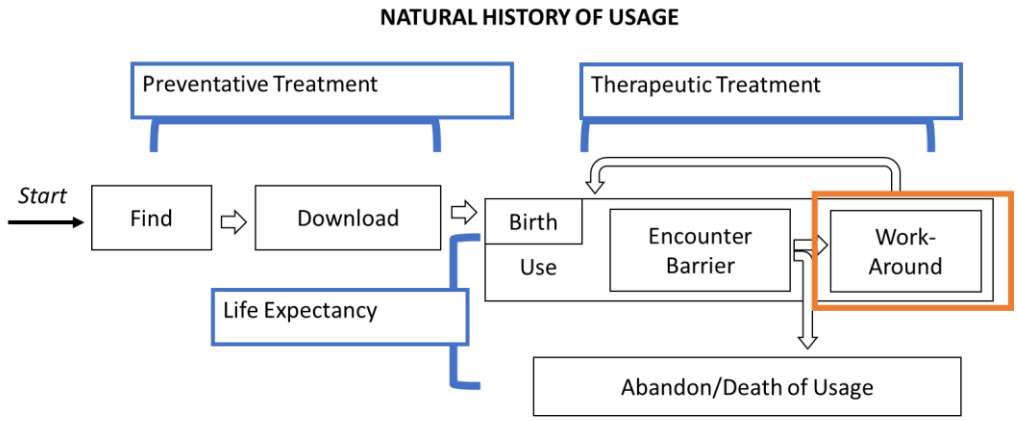


Figure 44: Third-party repair techniques, such as interaction proxies, can be a therapeutic treatment during the work-around cycle of an app's natural history of use.

Accessibility barriers are ideally prevented or repaired during app development. However, it can be necessary or desirable for third parties (i.e., individuals without access to an app's source code) to modify apps during their use. For example, barriers may not be fixed by developers in a timely manner due to competing priorities, slow update cycles, or because the app is not actively maintained. Even with engaged developers, third-party app modification can allow for more customized experiences. We implemented interaction proxies to explore techniques for and environmental factors impacting third-party repair. In the language of my framework, third-party repairs can provide therapeutic treatment during the work-around cycle of an app's natural history of use (Figure 44).

In this section, I will present an overview of the interaction proxies technique and the results of our user study. Further details of the implementation and design space can be found in our CHI publication [100]. For my dissertation, I will highlight the components of interaction proxies and feedback from the user study that highlights a range of factors that impact the potential use of third-party repairs.

6.3.1 Ecosystem-Driven Motivation

Although it occurred before I developed the epidemiology-inspired framework, the interaction proxies project was motivated by an ecosystem approach to app accessibility needs and tensions. Ideally, an app is released accessible and discovered failures are quickly updated in the app. However, factors throughout the ecosystem can impede that ideal reality. At the level of individual app repair, the app could be not actively maintained, accessibility could be deprioritized, or there could be no appropriate testing and feedback cycles to ensure repairs. More systematic changes from the platform or toolkits may further inhibit repairs or introduce new failures. Meanwhile, people have immediate access needs that cannot rely on an ideal world of natively accessible apps.

If someone encountered an accessibility failure, they have limited options to repair the app. They could try to contact developers through publicly available means of help centers, published emails, or reviews. Similarly, people may attempt to draw visibility and public support for improving accessibility problems through reviews or public media. Forums such as AppleVis [14] and Inclusive Android [121] provide discussion boards for app accessibility and sharing potential work arounds. However, these techniques can be opaque, time consuming, and have no guarantee of addressing accessibility barriers [14] and Inclusive Android [121] provide discussion boards for app accessibility and sharing potential work arounds. However, these techniques can be opaque, time consuming, and have no guarantee of addressing accessibility barriers.

Assistive technologies, at the time of interaction proxies in 2017, supported minimal end-user repair. For example, TalkBack allowed someone to provide a custom label to some elements. However, the element must be missing a label and have a `resourceId`. As mentioned above, the `resourceId` attribute is optional and often not specified for an element. Moreover, elements with uninformative labels cannot be updated, custom labels are challenging to share between users, and no other accessibility failures are end-user repairable by end users. Since

our work on interaction proxies, Apple has released a system to repair apps using AI to supplement accessibility metadata [30,99]. This system is a platform-level enhancement and does not enable third-party modifications.

In addition to fundamental accessibility repairs, an optimally accessible experience may vary between people. As explored in SUPPLE [39], people may benefit from different layouts, element sizing, and other visual and task design personalization. However, platform-level and app-level support for interface personalization is limited and mainly include features to adjust colors and slightly resize text.

We explored a third-party accessibility enhancement technique, interaction proxies, as a means to enable more people in the app ecosystem to modify apps to meet accessibility needs. Throughout the project, I refer to *repairs and enhancements*, highlighting the opportunity of third-party techniques to repair failures and create tailored experiences.

6.3.2 Interaction Proxies

Conceptually, an interaction proxy is a layer between an app's original interface and the manifest interface [32] that a person uses to perceive and manipulate the app. Manifested interaction can take many forms. For example, someone may perceive and manipulate an interface visually, aurally, or tactilely (e.g., Braille readout) using any number of devices (e.g., touchscreen, eye gaze, external keyboard). An interaction proxy can modify: (1) perception in an interaction, (2) manipulation in an interaction, or (3) both. The strategy is analogous to web proxies that modify a page between a server and a browser (including web proxies that modify pages to improve their accessibility [18,20,84]), but extended to address the design and implementation challenges of app accessibility.

With interaction proxies, a third-party developer can modify an app's interaction without the app's source code, without rooting the phone, without otherwise modifying or replicating the app's functionality, and while retaining the system capabilities (e.g., interaction proxies work

seamlessly with screen readers). Figure 45 illustrates insertion of an interaction proxy to implement repairs of missing labels in the Wells Fargo app. At a high level, labeled buttons are layered over the unlabeled original buttons. Interacting with the labeled buttons redirects the interaction to the original interface. When activated, the interaction proxy's modifications are integrated into the manifested interface (e.g., a screen reader will focus the modified button rather than the unlabeled one). The technical details of our interaction proxy implementation can be found in our publication [100].

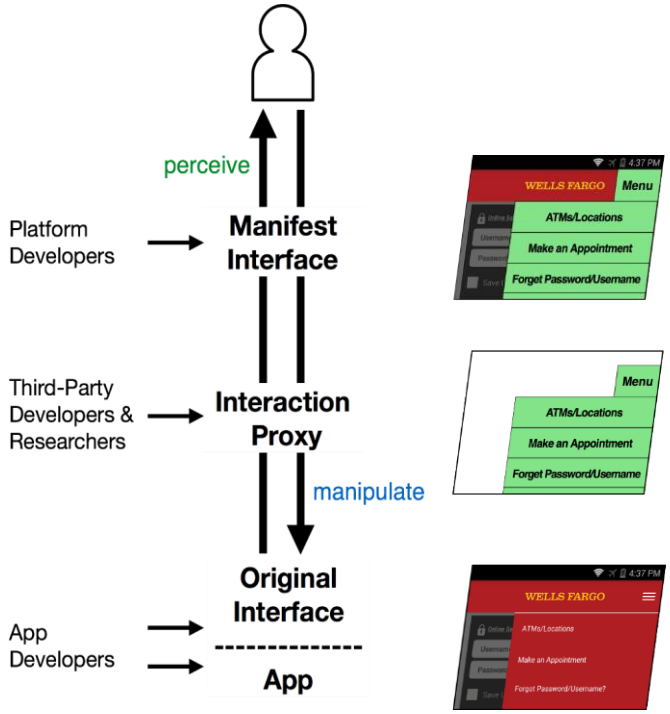


Figure 45: Interaction proxies are inserted between an app's original interface and the manifest interface a person uses to perceive and manipulate that app. This allows third-party developers and researchers to modify the interaction.

6.3.3 Interaction Remapping Space

To structure our proof-of-concept interaction proxy prototypes, we created a design space for repairing or enhancing app accessibility. As shown in Figure 45, an interaction proxy helps a person to bridge the gulf of evaluation and/or the gulf of execution [54]. We consider such

modifications in terms of how a proxy re-maps existing interaction into new interaction. Table 18 presents five re-mapping patterns, the interaction goal being achieved, and an example enhancement in that pattern. Although many potential enhancements fit cleanly into this design space, we note that some enhancements are better thought of as a combination of several interaction proxies, each of which re-maps an interaction according to this space. In our publication [100], we present additional proof-of-concept interaction proxies that demonstrate the design space.

Table 18: Design space of Interaction Re-mappings to improve app accessibility. Accessibility enhancements may be composed of multiple types of interaction re-mappings.

Re-mapping	Achieve	Example
Zero-to-One	Add a new interaction	Add accessibility rating to Google Play store
One-to-Zero	Remove an interaction	Stencils-based tutorial
One-to-One	Replace an existing interaction with another	Wrong label
One-to-Many	Replace an interaction with multiple interactions	2-stage click
Many-to-One	Replace multiple interactions with one interaction	Macro

6.3.4 User Study

We conducted two sets of interviews with blind and low-vision people who use screen readers. The first round of interviews identified accessibility barriers people encountered and the apps they wanted to use. The second interviews focused on the experience of using an interface with an interaction proxy, usefulness and potential of interaction proxies, and potential for adoption of such enhancements.

6.3.4.1 Preliminary Interview

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 proof-of-concept interaction

proxies we developed. Specifically, participants identified three common barriers that potentially could be addressed with interaction proxies: mislabeled elements, inaccessible functionality, and challenging navigation. Participants also identified three major app categories of interest: community engagement, productivity, and banking apps. Our proof-of-concept demonstrations and our second set of interviews focused on these needs and opportunities.

6.3.4.2 Proof-of-Concept Examples

Based on results from our preliminary interviews, we implemented three proof-of-concept enhancements to present to participants in our second user study.

Yelp

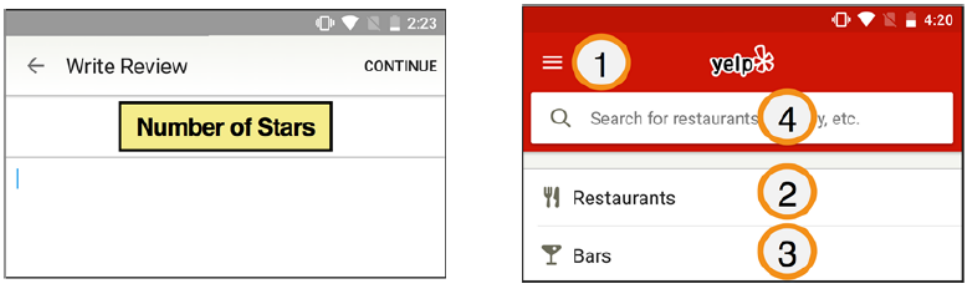


Figure 46: The Yelp app had unfocusable star buttons so reviews cannot be left using assistive technologies. On the search and results page, the search bar came very late in a linear navigation order, after the list of suggestions.

In the Yelp app for finding and rating businesses, the rating stars cannot be selected with an assistive technology and the search bar comes in an unintuitive linear navigation order (Figure 46). Using interaction proxies, we repaired the stars on the business rating page and moved the search bar earlier in the navigation order.

Toggl

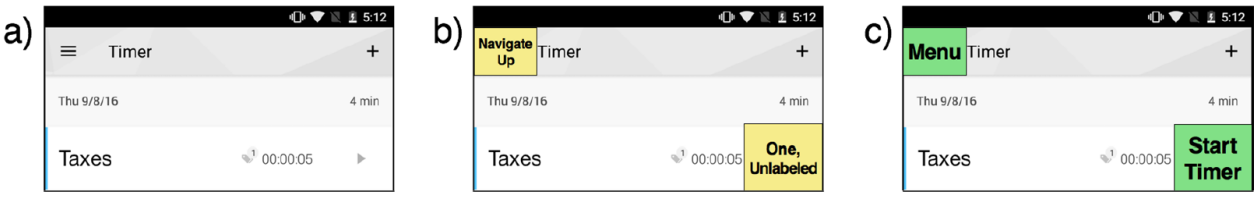


Figure 47: Interaction proxies fix missing labels and linear navigation in Toggl.

In the Toggl time tracking app, we repaired missing labels in elements associated with each existing timer (Figure 47). We also placed the "Start New Timer" button earlier in the linear navigation.

Wells Fargo

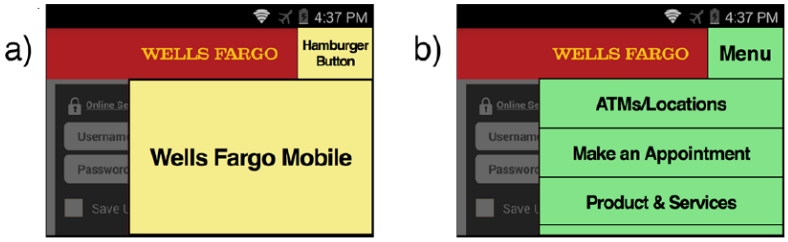


Figure 48: Interaction proxies repaired the mislabeled menu button and unfocusable menu items in the Wells Fargo app.

In the Wells Fargo mobile banking app, the menu icon is erroneously labeled "Hamburger Button" and the menu is unfocusable with assistive technology. As illustrated in Figure 48, we repaired the items in the dropdown menu to be accessible with a screen reader and repaired the label of the menu button.

6.3.4.3 Method

Our second set of interviews included six people who are blind or low-vision and use a screen reader (including two from the first interview). We asked each participant to use the TalkBack to interact with the three proof-of-concept apps presented above—Yelp, Toggl, and Wells Fargo—with and without interaction proxies applied. We used a Google Nexus 6P running Android 7.0. Participants discussed the interaction experience, the usefulness of interaction

proxies, and visions of how third-party enhancements could gain broad use. Interviews were transcribed and then analyzed using open coding.

Participants primarily used an iPhone, the more popular choice for people in the United States who use accessibility services [66]. 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.

6.3.4.4 Results

Participants discussed their interactions with the prototype and their thoughts on broad deployment.

Feedback on Prototype

Overall, participants thought our proof-of-concept prototype interaction proxies worked well for improving app accessibility. Participant responses to specific enhancements were more varied. For example, all participants agreed 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, there can still be usability barriers due to poor interface design choices.

All participants described how the types of enhancements we demonstrated could address inaccessibility they encounter in other apps and 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."

One goal of proxied interactions was seamlessness (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. 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 the responsiveness of interaction when the interaction proxy redirected screen reader focus. 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 enough to outweigh "oversensitivity." While this sentiment reflects the value of interaction proxies, it is important to evolve these methods to not accept providing sub-par experiences. Even when prompted, participants did not mention any other unusual or bothersome interactions.

Ecosystem of Deployment

In addition to getting prototype feedback, we talked with participants about how they envisioned third-party repairs being created and distributed. Participants reflected on their potential role as failure reporters, repair creators, and consumers.

All participants said they would be willing to submit apps that needed repair or enhancement if they thought the enhancement was likely to be created; reflecting a need in the community and a willingness to co-create solutions. This direct interaction between end users and third-party developers may also allow people to more directly define an accessible experience that works best for them as compared to giving feedback to app developers. Despite

willingness to contribute, some participants expressed concern over whether enough people would contribute enhancements.

An additional component of democratizing and personalizing repair is ensuring the repair infrastructure itself is accessible. Disabled people should not only be seen as passive receptors of repairs but capable of building those repairs if they have the interest. For example, P3 is a software developer and indicated he would be willing to create enhancements if good tools were available.

Reflecting on their role as consumers of enhancements, we asked participants about the potential process of acquiring third-party repairs, particularly around trust and security. While installing an app always carries security risks, accessibility services, such as interaction proxies, can be particularly risky as they require highly sensitive permissions to run on a device (e.g., capturing all interactions, including text and voice input and potentially screenshots). Participants indicated they would trust a third-party enhancement based on the reputation of the source (e.g., from Google or Apple marketplace), endorsements from known organization (e.g., National Federation of the Blind), and feedback from other blind users. The ability to access repairs thus depends on the environmental factors of where those repairs are located, shared experience of the community with the enhancements, and someone's risk tolerance. These factors create tension between enabling broader creation of accessibility enhancements while ensuring people have safe and reputable sources.

6.3.5 Summary

Interaction proxies explored the third-party repair and enhancement space of the app accessibility ecosystem. 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. P1 motivated a multi-faceted approach

by saying "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.

6.4 Chapter Summary

My work presents promising techniques for improving app accessibility throughout the app's development and use lifecycle. I explored tools for developers, professional testers, and third-party repair creators. Situating this research within my epidemiology-inspired framework, I noted where designs were inspired by large-scale analysis trends and discussed extrinsic factors that impact tool effectiveness. This work serves to inform efforts to improve app accessibility by providing novel designs for enhancing tools. Positive feedback from professional testers and people who use screen readers show promising directions for many of my tool concepts. Extrinsic factors surfaced during discussions included power dynamics, trust, and organizational structures. This work highlights the importance of considering such factors when working to improve app accessibility.

Chapter 7. DISCUSSION

My dissertation work extends our understanding of the state of app accessibility and explores ways to improve app accessibility throughout the app lifecycle. Situating this work within my epidemiology-inspired framework demonstrates the value of a population-scale, multi-factor approach to app accessibility. I will now discuss how my dissertation research demonstrates my thesis statement:

> *In mobile app accessibility, applying an epidemiology-inspired framework that emphasizes multi-factor and large-scale analyses can:*
>
> *(T1) reveal population-level trends of accessibility failures;*
>
> *(T2) aid in identifying a range of intrinsic to extrinsic factors that can impact app accessibility; and*
>
> *(T3) inform the design of tools for identifying and repairing accessibility failures.*

7.1 Population-Level Trends of Inaccessibility Diseases (T1)

Fitting within the epidemiology-inspired objective of determining the extent of disease in a population, I performed the first large-scale analysis of mobile app accessibility in Android. I started with a preliminary 100-app assessment and then scaled up to test approximately 10,000 free Android apps. In the language of my framework, the results of these analyses contribute to a census of mobile app accessibility health by measuring the prevalence of inaccessibility diseases (i.e., accessibility failures).

Within my analyses, I measured co-occurrence of failures, by-app prevalence, and by-class prevalence to give multiple views on severity, frequency, and spread of inaccessibility diseases. By-app prevalence gives insight into how widespread accessibility failures are. If accessibility failures are concentrated in a few apps, we may use individualized approaches

whereas common failures across the population may necessitate more systemic solutions. By-class analyses can help identify classes that are used across apps and have high likelihood of being inaccessible. These insights can help us understand where apps are most susceptible to inaccessibility and focus initial efforts to improve app accessibility on those common, failure-prone classes. Finally, co-occurrence analyses can provide insights on relationships between failures and highlight factors in the larger ecosystem.

My by-app analyses identified missing labels and undersized elements as particularly prevalent accessibility failures across apps. In testing approximately 10,000 apps, 23% had 90-100% of their `contentDescription`-dependent elements missing labels and 71% had more than 10% of their elements *too small in either* width or height. These high prevalence measures indicate people will encounter many of these failures across the app population.

Using knowledge-based and metric-based filtering, I further identified classes of elements that were highly used and had notable occurrences of missing labels. One such set of classes was image-based buttons—Clickable Images, Image Buttons, and Floating Action Buttons (FABs). Despite the functional and technical similarities of these classes, they differ in prevalence of missing labels. Approximately 53% of Image Buttons, 86% of clickable Images, and 93% of FABS were missing labels. These insights reinforced that missing labels are prominent across apps and identified highly susceptible classes.

Finally, I explored co-occurrence of accessibility failures in the 100-app assessment. Most apps have multiple accessibility failures. Cases where similar accessibility failures co-occur are of special interest. For example, missing labels and redundant labels are similar failures, but missing labels are more lethal. A missing label will result in a screen reader saying, "unlabeled button," versus a redundant label will cause it to say, "save button button." Occurrences of both types of failures within an app provoke the question of why some elements get poor labels while others get none. The fact that the co-occurrence was not an

isolated incident—at least 11% of the 100 apps tested had both errors—suggests underlying environmental factors.

My large-scale analyses contribute to our understanding of the state of app accessibility by measuring the prevalence of accessibility failures, assessing the spread of failures across apps, and identifying the susceptibility of highly used classes to failures. These metrics motivate the ongoing need to invest in reducing app inaccessibility. The population-scale assessment can also inform additional analyses and repairs, as detailed in the next sections.

7.2 Identifying Environmental Factors (T2)

I leveraged insights from my large-scale analyses, user studies, and existing literature to identify potentially impactful factors across the ecosystem. Factors ranged from more intrinsic (e.g., technical resources, documentation, design patterns) to more extrinsic (e.g., team social dynamics, end-user trust in repairs).

Directed by patterns in my large-scale analyses, I identified accessibility shortcomings in the more intrinsic factors of popular Android tools and frameworks. Guided by the high prevalence of missing labels in frequently used image-based button classes, I identified Android documentation, sample code, and Lint tests that did not follow or address best labeling practices. Questioning whether low quality, lesser-known apps are the main contributors of inaccessibility, I investigated the relationship between app ratings and accessibility, finding no strong trends. Finally, I observed design patterns associated with inaccessibility. One finding suggests third-party elements maintain their (in)accessibility across apps. Another result demonstrated that even among elements of the same class, the design use case of that element on a given screen could contribute to inaccessibility (e.g., a radio button used as a tab versus a page indicator).

I also explored more extrinsic factors (e.g., structural, organizational, social) that could impact inaccessibility. Accessibility is a collaboration between many stakeholders throughout the app lifecycles. In my work, I explored collaboration between professional testers and developers. I additionally highlighted the role of third-party developers and users with disabilities in the process of improving app accessibility. Power dynamics, communication infrastructure, trust, company culture, and education all emerged as factors in these relationships.

This research works toward the epidemiology-inspired objectives of *identifying risk factors and causations* and *evaluating existing treatments*. Accessibility shortcomings or strengths in tools that are used by many developers can impact accessibility across the app population. Tool and resource effectiveness are further impacted by the social, structural, and cultural context of their use. Work toward these objectives of identifying factors and evaluating existing resources (i.e., treatments) enriches our understanding of the state of app accessibility and informs improvements.

7.3 Inform Tool Design (T3)

I explored tools for supporting app accessibility at multiple stages within an app's natural history of development and use. My work aimed to improve tools for developers, professional testers, and third-party repair creators. Within the epidemiology-inspired models, these tools address the stages of development, testing, and workarounds during use. My approach and designs were informed by my large-scale, multi-factor analyses and by my user studies.

Toward making developer tools more efficient, effective, and educational, I designed new developer tool interfaces including my prototype Android Studio plugin. Two techniques I used were leveraging element, screen, and design context and increasing the connection between runtime and static analyses. In this dissertation, I presented a suite of designs centered around element labeling and resizing. I then discussed tensions between the design goals. As

an example of how my large-scale, multi-factor data analyses inspired my designs; my focus on labeling and element resizing was based on highly prevalent accessibility failures and my design for how to to support developers resizing elements using multiple techniques was based on design patterns in elements that were large enough due to grouping. This work highlights novel opportunities to better support developers in creating accessible apps.

Addressing the next phase of app development, I worked on tools for professional testers to improve the efficiency of their process and understand their collaboration with developer teams. My prototype extension to Google's Accessibility Scanner demonstrated that on-device annotation of automated accessibility test results is a promising technique for improving testing efficiency. Beyond improving testing tools, my work with professional accessibility testers and developers highlighted how knowledge, social dynamics, and a team's project scope impacted the effectiveness of accessibility testing. This project demonstrated testing tool improvements and began to highlight challenges and opportunities in the larger environment which testing tools are used within.

Finally, I presented my work on third-party techniques to support repairing and enhancing app accessibility post-release. Our proof-of-concept interaction proxies technique modified apps without accessing the source code, rooting the phone, or otherwise modifying the original app or assistive technologies. We further presented a design space of interaction re-mapping to help envision what app repairs and enhancements could look like. Feedback from people who are blind or had low vision and used screen readers showed excitement and interest in having third-party repairs. In their discussions, they also surfaced challenges with the third-party repair ecosystem. Topics included the structure needed to reliably produce third-party repairs in a timely fashion, the accessibility of the creation infrastructure, their trust in downloading third-party repairs, and the tension in investing time in a third-party repair versus focusing on getting the apps natively accessible.

These projects demonstrate how my epidemiology-inspired approach can (T3) inform designing tools to improve app accessibility throughout an app's lifecycle. Moreover, they expose how tools design can be informed by and affected by a spectrum of environmental factors.

7.4 Reflections

My dissertation work demonstrates the value of my epidemiology-inspired framework for understanding and improving app accessibility. I now reflect on the benefits and challenges in applying the framework in practice. I discuss the critique of having a framework that uses medical language while supporting the social model of disability. I then discuss challenges with using epidemiology-inspired jargon to communicate concrete research findings. Finally, I highlight my framework's value in prompting creative conversation, connecting with a range of researchers, and informing future research in app accessibility. I hope this reflection helps others leverage my epidemiology-inspired framework.

7.4.1 Medical Language in Accessibility Work

A major discussion we had when considering epidemiology as the systems science for our metaphor was the appropriateness of using medical models in an accessibility context. Medical systems, including epidemiology, have a complex history with disability that includes causing significant harm. The accessible computing community largely strives to approach research through the social model of disability. Oversimplified, the social model of disability [67] focuses on how the environment creates disabling situations (e.g., a lack of ramp creates access barriers for people who use wheelchairs). In this context, environment is broadly defined and includes physical architecture, technology, politics, and social structures. By focusing on the environment, the social model diverges from the medical model of disability, which aims to cure functional limitations and align people with normative values.

I intend for my epidemiology-inspired model to reflect the social model of disability; the onus of creating access lies in the apps not in someone adapting their interaction. The language of my framework identifies environmental and social factors to support and enforce creating accessible apps. In the presentation of my framework, I emphasize that the language of disease is being applied to the apps themselves, not to any of the people using them.

Yet these intentions and clarifications do not remove the history of epidemiology and medical language from my framework. This tension is true for the use of medical language as metaphors in other spaces as well. Lydia X. Z. Brown, a prominent disability advocate and scholar, provides a critique of applying epidemiology and disease language to systemic problems [103]:

> *"I'd encourage us all to reconsider framing racism and white supremacy as illnesses, diseases, or viruses - disability and disease are not metaphors! This kind of rhetoric erases disabled BIPOC, upholds ableism-racism, and is contrary to Disability Justice."*

Ultimately, I moved forward with publishing the epidemiology-inspired framework and have found it valuable, as described below. However, the tension between that value and the problematic components of using epidemiology language in an accessibility space is ongoing. Anyone using my epidemiology-inspired framework should therefore remain cognizant of the potential harms that may be perpetuated through medical language and forefront the focus on improving app accessibility to meet the needs, values, and goals of disabled people.

7.4.2 Using the Framework to Communicate Research

My framework provides a valuable perspective on app accessibility research. However, it is non-trivial to understand the framework and the nuance of accessibility it captures. In communicating research, the overhead of the framework can compete with clearly presenting results. I will now discuss the challenges I have encountered and techniques I have used in applying the framework in my research publications.

The main drawbacks of using epidemiology-inspired language to present research are the initial overhead of understanding the framework and the resulting difficulty in consuming the results independent of the framework. Reflecting the upfront cost of leading with the framework, one of my epidemiology-framework-centered-analysis publications spent nearly a page introducing the framework and defining terms. In addition to the logistic cost (e.g., using space in page-limited submissions), it also demands the reader invest focus, energy, and time in comprehending the framework.

Embedding epidemiology-inspired concepts throughout a publication means if someone does not invest in a basic level understanding of the framework, they may struggle to interpret the results. For example, in my framework-heavy publications [73,75], I refer to label-based accessibility failures as *determinants* of the *label-based inaccessibility button diseases*. I further articulate the value of the results through the epidemiology objectives, for example introducing results with "The analysis was structured using the epidemiology-inspired concepts of *determining the extent of a disease*, identifying potential *risk factors*, and *evaluating existing treatments."* In this example, using the framework to situate the large-scale analyses into the app ecosystem competes with communicating key contributions of the analysis: missing labels occur frequently across apps and popular developer resources have significant accessibility shortcomings.

Drawing from my experience, I advise others interested in applying my epidemiology-inspired framework to strongly consider the tradeoffs between using the framework language versus highlighting the direct take-aways of the research. Balancing between these two will depend on the audience, medium, and goals of any communication. For example, in my extended large-scale analysis publication [74], I prioritized sharing the results on how frequently failures occurred and patterns in the data. This is reflected in my language choice; I refer to "accessibility failures" rather than "*diseases"* and motivate the work in plain language such as "We now explore *why* and *under what conditions* some apps and classes of elements might

have been more susceptible to accessibility barriers than others" rather than epidemiology-inspired objectives. I then leveraged the value of the epidemiology framework in the discussion, where I use the framework concepts to highlight unique insights. I hope others can leverage the value from my framework while being mindful when using it to communicate research results.

7.4.3 Questions About Epidemiology Prompt Creative Insights on App Accessibility

The framework has provided substantial value in promoting discussions around app accessibility. The process of explaining, questioning, digesting, and reflecting on the epidemiology-inspired metaphor has instigated conversations, promoted creative thinking, and structured avenues for future work. In these situations, the framework's complexity and scope are assets. The framework has shaped my thinking and conversations. I hope others embrace the complexity of the framework to spark creative approaches to app accessibility and to explore the complex factors that impact app accessibility.

Compelling lines of discussion and thought emerge from the process of drawing parallels between app accessibility and epidemiology. Even people who do not know about app accessibility likely know general epidemiology concepts like contagiousness, virus spread, and public health initiatives like "employees must wash hands". This background can prompt simple but compelling questions.

For example, I once used malaria as an example when explaining the field of epidemiology. In response, someone asked "what's equivalent to a mosquito net?" This prompted us to discuss attributes of mosquito nets: a preventative treatment that is relatively well-known, cheap, easy to administer, easy to access, and highly effective. Moreover, multiple organizations have invested in public health initiatives to distribute and educate about mosquito netting. Applied to app accessibility, we thought about establishing barriers, metaphorical "netting", between carriers of accessibility failures and apps. One idea was

creating warnings for importing development libraries that have elements known to have or be associated with accessibility failures (e.g., the cross-platform libraries I identified in my analyses in Section 5.3.3). Pushing the metaphor further, we can think about what organizational efforts and "public health initiatives" around such app accessibility interventions would look like. This example demonstrates the creative discussions prompted by collaboratively building parallels between epidemiology and app accessibility. These discussions can enrich our understanding of the state of app accessibility and highlight novel opportunities for future research.

The framework additionally enriches later stage projects. For example, I think my ecosystem perspective was essential for capturing the social and company dynamics in the testing tools work (Section 6.2). That project began as a technical effort to improve automated tools accuracy. While the impact of team dynamics could have been identified without my framework, I believe creating and engaging with the framework throughout my career has primed me to identify those insights in my projects. Other researchers may similarly find that placing projects within my epidemiology-inspired framework can inform project direction, highlight limitations, and enrich discussions around how the work fits into the larger scope of app accessibility.

7.4.4 Connecting with a Broad Audience

The concepts in my epidemiology-inspired framework create points of connection with people working to improve accessibility at multiple levels within organizations. My framework complements ongoing discussions of ecosystem approaches to systemic change [56]. People advocating for accessibility in organizations often dedicate significant efforts to understanding and navigating multi-level structures and leverage a range of interventions from grassroots initiatives to top-level support. In conversations with some individuals in these roles, I think they connected the breadth and nuance of their roles to my epidemiology-inspired framework's multi-factor emphasis. I believe this connection enriched our discussions around

advancing development tools, challenges to putting valuable techniques into practice, and their approaches to institutional change.

Complementing the ability of the framework to enrich discussions with accessibility-focused people, there is also value in using the framework as a point of entry to accessibility research from outside the field. For example, I discussed my framework and accessibility work with a few epidemiologists. They both enjoyed the parallels and one suggested that the framework could be used to educate people outside of accessible computing about accessibility, as some epidemiology language and concepts may be more familiar to general audiences.

7.4.5 Summary

Overall, there are many benefits to applying my epidemiology-inspired framework. While it should be used mindfully when communicating research findings, I have found it invaluable for promoting novel approaches to app accessibility and connecting with a broad audience. I hope others can leverage these reflections to draw the most benefit from applying my epidemiology-inspired framework.

7.4.6 Researcher Position Statement

I provide this brief researcher position statement to capture how my personal and professional experience with accessibility, disability, and assistive technologies inform and bias my research. I do not personally use app assistive technologies. Continuing to build my understanding of accessibility through a user-centered, multi-faceted approach is essential. I learn about app accessibility through formal documentation (e.g., guidelines), publications in human-computer interaction research, industry professionals in accessibility, disabled activists, and my own user research. While my dissertation work emphasizes large-scale, data-driven analyses, our understanding of accessibility must be grounded in, informed by, and ultimately validated by the lived experience of people with disabilities and people who use assistive technologies.

7.5 Future Work

My epidemiology-inspired framework, large-scale and multi-factor analyses, and tool design exploration motivate multiple veins of future work. The epidemiology-inspired framework alone promotes thinking about how to "Break the Chain of Infection" (Figure 8), considering treatments throughout an app's lifecycle, and informing policy. In this section, I present potential future work in defining app accessibility, evolving the epidemiology-inspired framework, extending large-scale analyses, and considering multiple stakeholders in the ecosystem.

7.5.1 Defining Accessibility

An ongoing concept grounding my work is "what makes an app accessible?" My work thus far has focused on common, well-operationalized failures such as missing labels and undersized elements. While prior work, industry standards, and feedback from people who use assistive technology confirms the impact of these accessibility failures, addressing these failures represent a small step toward optimally accessible, usable, and enjoyable experiences for everyone. Future work can extend how we understand and facilitate accessible app experiences by (1) better capturing diverse accessibility needs, (2) grounding accessibility metrics in user-centered design, and (3) creating modifiable experiences adjust to individual preference, without creating a subpar or separate experience.

7.5.2 Evolving the Epidemiology-Framework

My epidemiology-inspired framework provided valuable structure to my work. After applying it to my work over the years, there are also opportunities to revise and extend it. A main ecosystem component of app accessibility that surfaced was accessibility as an indirect consequence of other pressures. For example, prior work in web accessibility cited the rise in CSS and search engine optimization enforced website structure that also benefitted

accessibility. Similarly, popular app frameworks or platforms that focus on uniform visual experiences, the general rise of voice assistants, and eyes-free platforms emerging in cars may also consequently increase app accessibility. In the current epidemiology-inspired framework, app accessibility is the central focus that factors impact (Figure 5). There is no language or structure that captures indirect accessibility benefits. Integrating that element could further scaffold understanding the impact of environmental factors and could identify other axes of leverage that could increase accessibility before institutions develop full accessibility buy-in.

Another exciting extension of the epidemiology-inspired framework is applying it to other domains both within and beyond accessibility. For example, in the ecosystem model, apps are centralized (Figure 5). That center circle can be changed to other technologies such as virtual reality, wearables, or voice assistants. This is particularly impactful as devices become more multi-modal and cross-platform, suggesting native apps may not be the only focus. Inaccessibility is one type of metaphorical *disease* to consider. It may also be worthwhile to apply an epidemiology-inspired approach to other attributes of technology. Security is one such attribute; it evolved from a deprioritized to a central attribute. Exploring the factors that contributed to that evolution could provide valuable insights. My epidemiology-inspired framework provides structure to do such explorations and comparisons.

7.5.3 Extending Longitudinal and Multi-Factor Analyses

My analyses provided a measure of the state of app accessibility and identified potential factors that influence accessibility at scale. Analyzing longitudinal data and data on what resources were used to build and maintain an app are two avenues for extending population-scale understanding of app accessibility.

My large-scale analysis assessed the state of app accessibility during a small moment in time during which the Rico dataset was collected. Assessing app accessibility over time could

provide insight into population scale trends, similar to the 14-year study performed on web accessibility. Population-scale, longitudinal analyses could reveal large system-wide impacts like platform updates, the evolution of popular visual design, and the evolution of cross-platform applications. Scoped longitudinal studies of an app or apps that come from a single developer or company over time could show the effectiveness of that company's accessibility practices (e.g., are accessibility failures introduced in major updates?). These trends could enrich our understanding of the evolution of app accessibility and influence of different factors over time.

In addition to capturing app data over time, increasing the type of information captured could contribute to understanding and improving app accessibility. Informed by trends in my large-scale analyses, I identified accessibility shortcomings of developer resources such as Android Lint tests and sample code. However, I could not determine how much of an impact those resources had on the inaccessibility trends observed at scale. Collecting and analyzing what resources are used when developing apps could provide more direct insight into the spread, influence, and effectiveness of different types of interventions. One example from my analyses was connecting classes from the Unity game engine to the few TalkBack-focusable failures (Section 5.3.3). Similar traces for more frameworks and higher-level resources (e.g., code repositories, guidelines, and testing tools) would provide valuable insight.

7.5.4 Understanding and Supporting Multi-Stakeholder Collaboration

Advancing app accessibility requires collaboration between multiple stakeholders in the app ecosystem. My research began to identify challenges in collaboration in two spaces: between developers and professional accessibility testers and between end-users and third-party repair systems. Future work could expand on understanding the stakeholders, existing collaborations, challenges, and opportunities in the larger ecosystem. This could include extending my work with professional testing teams, for example, by including designers, managers, marketing, developers as well. It could also scope even more extrinsically to

understanding what motivates company leadership, app marketplace leadership, or government legislators to provide support for and ensure progress on app accessibility.

Chapter 8. CONCLUSION

App inaccessibility is a wide-spread, multi-faceted problem that continues to impede equitable technology access for people with disabilities and people who use assistive technology. In this dissertation, I demonstrated the following thesis statement:

In mobile app accessibility, applying an epidemiology-inspired framework that emphasizes multi-factor and large-scale analyses can:

(T1) reveal population-level trends of accessibility failures;

(T2) aid in identifying a range of intrinsic to extrinsic factors that can impact app accessibility; and

(T3) inform the design of tools for identifying and repairing accessibility failures.

In Chapter 4, I introduced my epidemiology-inspired framework that emphasizes multi-factor and large-scale analyses. In Chapter 5, I presented assessments of populations of 100 and 10,000 Android apps (T1) revealing population-level trends of accessibility failures and (T2) identifying factors that impact those apps' accessibility such as developer use of game engines and design patterns. My work in Chapter 6 on developer, testing, and third-party repair tools (T3) informs the design of tools for identifying and repairing app accessibility failures and (T2) identifies additional factors that impact app accessibility such as collaboration, company structure, and trust that impact app accessibility. Collectively, this research demonstrates my thesis statement, improves our understanding of app accessibility, and can inform efforts to improve app accessibility.

Chapter 9. REFERENCES

1. Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. 2020. Accessibility Issues in Android Apps: State of Affairs, Sentiments, Andways Forward. *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE '20)*, IEEE Computer Society, 1323–1334.

2. "Android Open Source Project." Making Apps More Accessible | Android Developers. Retrieved April 16, 2018 from https://developer.android.com/guide/topics/ui/accessibility/apps.html.

3. Android Open Source Project. Improve Your Code with Lint. Retrieved April 16, 2018 from https://developer.android.com/studio/write/lint.html.

4. Android Open Source Project. Accessibility Developer Checklist. Retrieved May 4, 2016 from http://developer.android.com/guide/topics/ui/accessibility/checklist.html#requirements.

5. "Android Open Source Project." ImageView | Android Developers. Retrieved April 16, 2018 from https://developer.android.com/reference/android/widget/ImageView.html.

6. "Android Open Source Project." ImageButton | Android Developers. Retrieved April 16, 2018 from https://developer.android.com/reference/android/widget/ImageButton.html.

7. "Android Open Source Project." FloatingActionButton | Android Developers. Retrieved April 16, 2018 from https://developer.android.com/reference/android/support/design/widget/FloatingActionButton.html.

8. "Android Open Source Project." Accessibility - Usability - Material Design. Retrieved April 16, 2018 from https://material.io/guidelines/usability/accessibility.html#accessibility-writing.

9. "Android Open Source project." Quality Guidelines | Android Developers. Retrieved April 16, 2018 from https://developer.android.com/develop/quality-guidelines/index.html.

10. "Android Open Source Project." FloatingActionButtonBasic | Android Developers. Retrieved April 16, 2018 from https://developer.android.com/samples/FloatingActionButtonBasic/res/layout/fab_layout.html.

11. "Android Open Source Project." BasicAccessibility | Android Developers. Retrieved April 16, 2018 from https://developer.android.com/samples/BasicAccessibility/res/layout/sample_main.html.

12. Delvani Antônio Mateus, Carlos Alberto Silva, Marcelo Medeiros Eler, André Pimenta Freire, and André Pimenta. 2020. Accessibility of Mobile Applications: Evaluation by Users with Visual Impairment and by Automated Tools. *Proceedings of the Brazilian Symposium on Human Factors in Computing Systems (IHC '20)*, ACM, 1–10.

13. Apple Inc. 2012. Accessibility Programming Guide for iOS. Retrieved from https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/iPhoneAccessibility/Introduction/Introduction.html.

14. AppleVis. AppleVis: Empowering Blind and Low-Vision Users of Apple Projects and Related Applications. Retrieved May 5, 2016 from http://www.applevis.com/.

15. Tingting Bi, Xin Xia, David Lo, John Grundy, Thomas Zimmermann, and Denae Ford. 2021. Accessibility in Software Practice: A Practitioner's Perspective. *arXiv* 1: 1.

16. Jeffrey P. Bigham. 2014. Making the Web Easier to See with Opportunistic Accessibility Improvement. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '14)*, ACM Press, 117–122.

17. Jeffrey P Bigham, Chandrika Jayant, Hanjie Ji, et al. 2010. VizWiz: Nearly Real-time Answers to Visual Questions. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '10)*, ACM Press.

18. Jeffrey P. Bigham, Ryan S. Kaminsky, Richard E. Ladner, Oscar M. Danielsson, and Gordon L. Hempton. 2006. WebInSight: Making Web Image Accessible. *Proceedings of the ACM SIGACCESS conference on Computers and accessibility (ASSETS '06)*, ACM Press, 181.

19. Jeffrey P. Bigham and Richard E. Ladner. 2007. Accessmonkey: a Collaborative Scripting Framework for Web Users and Developers. *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility (W4A 2007)*, ACM Press, 25.

20. Jeffrey P. Bigham, Craig M. Prince, and Richard E. Ladner. 2008. WebAnywhere: A Screen Reader On-the-Go. *Proceedings of the International Cross-Disciplinary Workshop on Web Accessibility (W4A 2008)*, ACM Press, 73.

21. Matt Bishop. 2003. What is Computer Security? *IEEE Security & Privacy Magazine* 1, 1: 67–69.

22. Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. 2011. Falling Asleep with Angry Birds, Facebook and Kindle: A Large Scale Study on Mobile Application Usage. *Proc. MobileHCI 2011)*, ACM Press, 47–56.

23. Erin L. Brady, Yu Zhong, Meredith Ringel Morris, and Jeffrey P. Bigham. 2013. Investigating the Appropriateness of Social Network Question Asking as a Resource for Blind Users. *Proceedings of the conference on Computer supported cooperative work (CSCW 2013)*, ACM Press, 1225.

24. Erin Brady, Meredith Ringel Morris, and Jeffrey P. Bigham. 2015. Gauging Receptiveness to Social Microvolunteering. *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems - CHI '15*, ACM Press, 1055–1064.

25. Urie Bronfenbrenner. 1979. *The Ecology of Human Development: Experiments by Nature and Design*. Harvard University Press.

26. "Business Wire." 2017. Wells Fargo Launches Enterprise Accessibility Program Office. Retrieved April 16, 2018 from https://www.businesswire.com/news/home/20171130005201/en/Wells-Fargo-Launches-Enterprise-Accessibility-Program-Office.

27. Lucas Pedroso Carvalho, Bruno Piovesan Melchiori Peruzza, Flávia Santos, Lucas Pereira Ferreira, and André Pimenta Freire. 2016. Accessible Smart Cities?: Inspecting

the Accessibility of Brazilian Municipalities' Mobile Applications. *Proceedings of the 15th Brazilian Symposium on Human Factors in Computer Systems - IHC '16*, ACM Press.

28. Parmit K. Chilana, Amy J. Ko, Jacob O. Wobbrock, Tovi Grossman, and George Fitzmaurice. 2011. Post-Deployment Usability: a Study of Current Practices. *Proceedings of the 2011 conference on Human factors in computing systems - CHI '11*, ACM Press, 2243–2246.

29. Raphael Clegg-Vinell, Christopher Bailey, and Voula Gkatzidou. 2014. Investigating the Appropriateness and Relevance of Mobile Web Accessibility Guidelines. *Proc. W4A 2014*, ACM Press, 1–4.

30. Devin Coldeway. 2020. iPhones Can Now Automatically Recognize and Label Buttons and UI Features for Blind Users | TechCrunch. *TechCrunch*. Retrieved May 14, 2021 from https://techcrunch.com/2020/12/03/iphones-can-now-automatically-recognize-and-label-buttons-and-ui-features-for-blind-users/.

31. Association for Professionals in Infection Control. 2017. Break the Chain of Infection. *Infection Protection and You: Healthcare Professionals*.

32. Alan Cooper. 1995. About Face: The Essentials of User Interface Design. .

33. Michael Cooper, Peter Korn, Andi Snow-Weaver, Gregg Vanderheiden, Loïc Martínez Normand, and Mike Pluke. 2013. *Guidance on Applying WCAG 2.0 to Non-Web Information and Communications Technologies (WCAG2ICT)*. .

34. "Data Driven Design Group." Rico: A Mobile App Dataset of Building Data-Driven Design Applications. Retrieved April 16, 2018 from http://interactionmining.org/rico.

35. Biplab Deka, Zifeng Huang, Chad Franzen, et al. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology - UIST '17*, ACM Press, 845–854.

36. Biplab Deka, Zifeng Huang, and Ranjitha Kumar. 2016. ERICA: Interaction Mining Mobile Apps. *Proc. UIST 2016*: 767–776.

37. Frank Deruyter, Michael L Jones, and John T Morris. 2018. Mobile Health Apps and Needs of People with Disabilities: A National Survey . *Journal on Technology & Persons with Disabilities* 6.

38. Trinh Minh Tri Do, Jan Blom, Daniel Gatica-Perez, et al. 2011. Smartphone Usage in the Wild: a Large-Scale Analysis of Applications and Context. *Proceedings of the Conference on Multimodal Interfaces (ICMI '11)*: 353–360.

39. 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.

40. Santiago Gil, Alexander Kott, and Albert-László Barabási. 2014. A Genetic Epidemiology Approach to Cyber-Security. *Scientific Reports* 4: 5659.

41. Michael Gilbert and Shabi Kashani. Designing A11y with Material Design. *Google I/O 2021*. Retrieved June 14, 2021 from https://events.google.com/io/session/fe65b6d1-0c41-4434-9924-1c2a2a1b67c0?lng=en.

42. April Glaser. When Things Go Wrong for Blind Users on Facebook, They Go Really Wrong. Retrieved October 30, 2020 from https://slate.com/technology/2019/11/facebook-blind-users-no-accessibility.html.

43. Google. 2015. Accessibility Test Framework for Android. Retrieved February 24, 2018 from https://github.com/google/Accessibility-Test-Framework-for-Android.

44. Google. 2016. Accessibility Scanner. Retrieved May 5, 2016 from https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor.

45. Google. Android Accessibility Developer Guidelines. Retrieved August 27, 2015 from https://developer.android.com/guide/topics/ui/accessibility.

46. Google Inc. Accessibility Test Framework for Android. Retrieved from https://github.com/google/Accessibility-Test-Framework-for-Android.

47. Google Inc. Add a Floating Action Button | Android Developers. Retrieved June 26, 2018 from https://developer.android.com/guide/topics/ui/floating-action-button.

48. "Google Open Source Project." Develop Apps | Android Developers. Retrieved April 16, 2018 from https://developer.android.com/develop/index.html.

49. Leon Gordis. 2004. *Epidemiology*. Saunders, Philadelphia, Pa.

50. Christina T. Haleas. 2019. Don't Ask Me What To Do, Just Let Me Sue You: Why We Need Clear Guidelines for Website Accessibility Under Title III of the Americans with Disabilities Act. *University of Illinois Journal of Law, Technology, & Policy* 2019 No. 2: 465–488.

51. Vicki L. Hanson and John T. Richards. 2013. Progress on Website Accessibility? *ACM Transactions on the Web* 7, 1: 1–30.

52. Yun Huang, Brian Dobreski, Bijay Bhaskar Deo, et al. 2015. CAN: Composable Accessibility Infrastructure via Data-Driven Crowdsourcing. *Proceedings of the Web for All Conference (W4A 2015)*, ACM Press, 1–10.

53. Amy Hurst, Jennifer Mankoff, Anind K. Dey, and Scott E. Hudson. 2007. Dirty desktops. *Proceedings of the 20th annual ACM symposium on User interface software and technology - UIST '07*, ACM Press, 183.

54. Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. 2009. Direct Manipulation Interfaces. *Human–Computer Interaction* 1, 4: 311–338.

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. *Extended Abstracts on Human Factors in Computing Systems (CHI EA '11)*, ACM Press, 1867–1872.

56. Annie Jean-Baptiste. 2020. *Building for Everyone: Expand Your Market with Design Practices From Google's Product Inclusion Team*. John Wiley & Sons, Inc., New Jersey.

57. 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. *Proceedings of the 10th international ACM SIGACCESS conference on Computers and accessibility - Assets '08*, ACM Press, 73.

58. 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. *Proceeding of the eleventh international ACM SIGACCESS conference on Computers and accessibility - ASSETS '09*, ACM Press, 115.

59. 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. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '11)*, ACM Press.

60. Shaun K. Kane, Jessie A. Shulman, Timothy J. Shockley, and Richard E. Ladner. 2007. A Web Accessibility Report Card for Top International University Web Sites. *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A) - W4A '07*, ACM Press, 148.

61. Walter Lasecki, Christopher Miller, Adam Sadilek, et al. 2012. Real-Time Captioning by Groups of Non-Experts. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2012)*, ACM Press, 23.

62. Dale F. Leipper. 1961. Oceanography—A Definition for Academic Use. *Transactions, American Geophysical Union* 42, 4: 429.

63. Jennifer Mankoff, Holly Fait, and Tu Tran. 2005. Is Your Web Page Accessible? A Comparative Study of Methods for Assessing Web Page Accessibility for the Blind. *Proc. CHI 2005*, ACM Press, 41–50.

64. Lauren R. Milne, Cynthia L. Bennett, and Richard E. Ladner. 2014. The Accessibility of Mobile Health Sensors for Blind Users. *International Technology and Persons with Disabilities Conference Scientific/Research Proceedings (CSUN 2014)*, California State University, Northridge., 166–175.

65. Israel J. Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E. Hassan. 2014. A Large-Scale Empirical Study on Software Reuse in Mobile Apps. *IEEE Software* 31, 2: 78–86.

66. J. Morris and J. Mueller. 2014. Blind and Deaf Consumer Preferences for Android and iOS Smartphones. In *Inclusive Designing*. Springer International Publishing, Cham, 69–79.

67. Mike Oliver. 1990. The Individual and Social Models of Disability. *Joint Workshop of the Living Options Group and the Research Unit of the Royal College of Physicians*, Vol. 23.

68. Débora Maria Barroso Paiva, André Pimenta Freire, and Renata Pontin de Mattos Fortes. 2021. Accessibility and Software Engineering Processes: A Systematic Literature Review. *Journal of Systems and Software* 171: 110819.

69. Kyudong Park, Taedong Goh, Hyo-Jeong So, Hyo-Jeong Association for Computing Machinery., HCI Society of Korea, and Hanbit Media (Firm). 2014. Toward Accessible Mobile Application Design: Developing Mobile Application Accessibility Guidelines for People with Visual Impairment. *Proceedings of HCI Korea (HCIK '15)*, Hanbit Media, Inc., 478.

70. John T. Richards, Kyle Montague, and Vicki L. Hanson. 2012. Web Accessibility as a Side Effect. *Proc. ASSETS 2012*, ACM Press, 79.

71. André Rodrigues. 2015. Breaking Barriers with Assistive Macros. *Proceedings of the ACM SIGACCESS Conference on Computers and Accessibility (ASSETS 2008)*, ACM, 351–352.

72. André Rodrigues and Tiago Guerreiro. 2014. SWAT: Mobile System-Wide Assistive Technologies. *Proceedings of the BCS Human Computer Interaction Conference (HCI '14)* , BCS Learning and Development Ltd., 341–346.

73. Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. 2017. Epidemiology as a Framework for Large-Scale Mobile Application Accessibility Assessment. *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility - ASSETS '17*, ACM Press, 2–11.

74. Anne Spencer Ross, Xiaoyi Zhang, James Fogarty, and Jacob O. Wobbrock. 2020. An Epidemiology-Inspired Large-Scale Analysis of Android App Accessibility. *ACM Transactions on Accessible Computing (TACCESS)*. 13, 1, Article 4 (April 2020), 36 pages. DOI:https://doi.org/10.1145/3348797

75. Anne Spencer Ross, Xiaoyi Zhang, Jacob O. Wobbrock, and James Fogarty. 2018. Examining Image-Based Button Labeling for Accessibility in Android Apps Through Large-Scale Analysis. *ACM SIGACCESS Conference on Computers and Accessibility (ASSETS 2018)*.

76. Alireza Sahami Shirazi, Niels Henze, Albrecht Schmidt, Robin Goldberg, Benjamin Schmidt, and Hansjörg Schmauder. 2013. Insights Into Layout Patterns of Mobile User Interfaces by an Automatic Analysis of Android Apps. *Proc. EICS 2013*, ACM Press, 275–284.

77. Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek. 2021. Latte: Use-Case and A. ssistive-Service Driven Automated Accessibility Testing Framework for Android. *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '21)*, ACM, 1–11.

78. Daisuke Sato, Hironobu Takagi, Masatomo Kobayashi, Shinya Kawanaka, and Chieko Asakawa. 2010. Exploratory Analysis of Collaborative Web Accessibility Improvement. *ACM Transactions on Accessible Computing (TACCESS)* 3, 2: 1–30.

79. Leandro Coelho Serra, Lucas Pedroso Carvalho, Lucas Pereira Ferreira, Jorge Belimar Silva Vaz, and André Pimenta Freire. 2015. Accessibility Evaluation of E-Government Mobile Applications in Brazil. *Procedia Computer Science* 67: 348–357.

80. Clauirton Siebra, Tatiana Gouveia, Jefte Macedo, et al. 2015. Usability Requirements for Mobile Accessibility. *Proceedings of the Conference on Mobile and Ubiquitous Multimedia (MUM '15)*, ACM Press, 384–389.

81. Camila Silva, Marcelo Medeiros Eler, and Gordon Fraser. 2018. A Survey on the Tool Support for the Automatic Evaluation of Mobile Accessibility. *Proceedings of the International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion (DSAI '18)*, Association for Computing Machinery, 286–293.

82. Henrique Neves da Silva, Andre Takeshi Endo, Marcelo Medeiros Eler, Silvia Regina Vergilio, and Vinicius H.S. Durelli. 2020. On the Relation between Code Elements and Accessibility Issues in Android Apps. *Proceedings of the Brazilian Symposium on Systematic and Automated Software Testing (SAST '20)*, Association for Computing Machinery, 40–49.

83. "Starbucks Newsroom." 2015. Global Accessibility Awareness Day: Starbucks Celebrates Digital Inclusion. Retrieved April 16, 2018 from https://news.starbucks.com/news/digital-accessibility-in-starbucks-stores?hootPostID=0df1827b8efbc8223734e48ae2b64f43.

84. H. Takagi and C. Asakawa. 2000. Transcoding Proxy for Nonvisual Web Access. *ACM Conference on Assistive Technologies (ASSETS '00)*, Association for Computing Machinery (ACM), 164–171.

85. Hironobu Takagi, Chieko Asakawa, Kentarou Fukuda, and Junji Maeda. 2003. Accessibility Designer: Visualizing Usability for the Blind. *ACM SIGACCESS Conference on Accessibility and Computing (ASSETS '04)*, Association for Computing Machinery (ACM), 177–184.

86. Hironobu Takagi, Shinya Kawanaka, Masatomo Kobayashi, Takashi Itoh, and Chieko Asakawa. 2008. Social Accessibility: Achieving Accessibility through Collaborative Metadata Authoring. *Proceedings of the ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '08)*, ACM Press, 193.

87. Hironobu Takagi, Shinya Kawanaka, Masatomo Kobayashi, Daisuke Sato, and Chieko Asakawa. 2009. Collaborative Web Accessibility Improvement: Challenges and Possibilities. *Proceedings of the ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '09)*, ACM Press, 195–202.

88. Desney S. Tan, Brian Meyers, and Mary Czerwinski. 2004. WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. *Extended Abstracts of the Conference on Human Factors and Computing Systems (CHI '04)*, ACM Press, 1525.

89. Arthur Tansley. 1987. What is Ecology? *Biological Journal of the Linnean Society* 32, 1: 5–16.

90. Garreth W. Tigwell. 2021. Nuanced Perspectives Toward Disability Simulations from Digital Designers, Blind, Low Vision, and Color Blind People. *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '21)*, ACM, 1–15.

91. Christopher Vendome, Diana Solano, Santiago Linan, and Mario Linares-Vasquez. 2019. Can Everyone use my app? An Empirical Study on Accessibility in Android Apps. *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, Institute of Electrical and Electronics Engineers Inc., 41–52.

92. Markel Vigo and Giorgio Brajnik. 2011. Automatic Web Accessibility Metrics: Where We Are and Where We Can Go. *Interacting with Computers* 23, 2: 137–155.

93. Richard B Warnecke, April Oh, Nancy Breen, et al. 2008. Approaching Health Disparities From a Population Perspective: the National Institutes of Health Centers for Population Health and Health Disparities. *American Journal of Public Health* 98, 9: 1608–15.

94. Jacob O. Wobbrock, Shaun K. Kane, Krzysztof Z. Gajos, Susumu Harada, and Jon Froehlich. 2011. Ability-Based Design. *ACM Transactions on Accessible Computing* 3, 3: 1–27.

95. Jacob O. Wobbrock and Julie A. Kientz. 2016. Research Contributions in Human-Computer Interaction. *Interactions* 23, 3: 38–44.

96. Aileen Worden, Nef Walker, Krishna Bharat, and Scott Hudson. 1997. Making computers easier for older adults to use. *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '97*, ACM Press, 266–271.

97. Shunguo Yan and P. G. Ramachandran. 2019. The Current Status of Accessibility in Mobile Apps. *ACM Transactions on Accessible Computing* 12, 1: 1–31.

98. Daihua X Yu, Bambang Parmanto, Brad E Dicianno, and Gede Pramana. 2015. Accessibility of mHealth Self-Care Apps for Individuals with Spina Bifida. *Perspectives in Health Information Management (AHIMA), American Health Information Management Association* 12, Spring.

99. Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, et al. 2021. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '21)*, ACM, 1–15.

100. 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. *Proc. CHI 2017*, ACM Press, 6024–6037.

101. Xiaoyi Zhang, Tracy Tran, Yuqian Sun, et al. 2018. Interactiles: 3D Printed Tactile Interfaces to Enhance Mobile Touchscreen Accessibility. *Proceedings of the ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '18)*, ACM.

102. 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. *Proceedings of the Web for All Conference (W4A 2015)*, ACM Press, 1–10.

103. 2021. Lydia X. Z. Brown on Twitter. Retrieved May 15, 2021 from https://twitter.com/autistichoya/status/1375920092471566336.

104. Apple Registers Trademark for "There's an App for That" | WIRED. Retrieved May 14, 2021 from https://www.wired.com/2010/10/app-for-that/.

105. Web Accessibility Laws & Policies | Web Accessibility Initiative (WAI) | W3C. Retrieved February 23, 2020 from https://www.w3.org/WAI/policies/.

106. BECU will restore accessibility for blind customers to its website and mobile banking app | The Seattle Times. Retrieved March 3, 2020 from https://www.seattletimes.com/business/becu-will-restore-accessibility-for-blind-customers-to-its-website-and-mobile-banking-app/.

107. Blind Facebook users are frustrated with screen-reader bugs. Former employees say Facebook isn't doing enough. Retrieved March 3, 2020 from https://slate.com/technology/2019/11/facebook-blind-users-no-accessibility.html.

108. Blind Californians and Advocates Sue Greyhound to Make Website and Mobile App Accessible - TRE LEGAL PRACTICE. Retrieved March 3, 2020 from https://www.trelegal.com/posts/blind-californians-and-advocates-sue-greyhound-to-make-website-and-mobile-app-accessible/.

109. Shrink, Obfuscate, and Optimize Your App | Android Developers. Retrieved April 28, 2019 from https://developer.android.com/studio/build/shrink-code.html.

110. MATE | Mobile Accessibility Testing for Android. Retrieved May 14, 2021 from https://android-mate.github.io/.

111. Apple iOS And Google Android Smartphone Market Share Flattening: IDC - Forbes. Retrieved March 27, 2016 from http://www.forbes.com/sites/dougolenick/2015/05/27/apple-ios-and-google-android-smartphone-market-share-flattening-idc/#4da3b7e82d4e.

112. Action Blocks - Apps on Google Play. Retrieved May 15, 2021 from https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.maui.actionblocks&hl=en_US&gl=US.

113. Use Live Caption - Android Accessibility Help. Retrieved May 14, 2021 from https://support.google.com/accessibility/android/answer/9350862?hl=en.

114. Microsoft Soundscape - Microsoft Research. Retrieved May 14, 2021 from https://www.microsoft.com/en-us/research/product/soundscape/.

115. Seeing AI App from Microsoft. Retrieved May 14, 2021 from https://www.microsoft.com/en-us/ai/seeing-ai.

116. Lookout by Google - Apps on Google Play. Retrieved May 14, 2021 from https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.reveal&hl=en_US&gl=US.

117. Android Studio. .

118. Floating Action Button Usage Guidelines. .

119. Material Design. Retrieved from https://material.io/guidelines/.

120. Apple Accessibility Scanner. Retrieved from https://developer.apple.com/library/content/documentation/Accsssibility/Conceptual/AccessibilityMacOSX/OSXAXTestingApps.html.

121. Inclusive Android . Retrieved May 14, 2021 from https://www.inclusiveandroid.com/.

122. Github - Google TalkBack, google/talkback. Retrieved May 16, 2021 from https://github.com/google/talkback.

123. Support Different Pixel Densities | Android Developers. Retrieved May 14, 2021 from https://developer.android.com/training/multiscreen/screendensities.html#TaskUseDP.

124. AppCompatImageView | Android Developers. Retrieved April 29, 2019 from https://developer.android.com/reference/android/support/v7/widget/AppCompatImageView.

125. Buttons - Material Design. Retrieved May 14, 2021 from https://material.io/components/buttons/android#using-buttons.

126. Content labels - Android Accessibility Help. Retrieved May 16, 2021 from https://support.google.com/accessibility/android/answer/7158690?hl=en.

127. googlearchive/android-BasicAccessibility: Deprecated: Retrieved May 16, 2021 from https://github.com/googlearchive/android-BasicAccessibility#readme.

128. Unity. Retrieved April 29, 2019 from https://unity.com/.

129. Apache Cordova. Retrieved April 29, 2019 from https://cordova.apache.org/.

130. Adobe - Adobe AIR. Retrieved April 29, 2019 from https://get.adobe.com/air/.

131. The Crosswalk Project. Retrieved May 14, 2021 from https://github.com/crosswalk-project.

132. Cocos2d-x - World's #1 Open-Source Game Development Platform. Retrieved April 29, 2019 from https://cocos2d-x.org/.

133. Views Widgets Samples: CardView. Retrieved May 14, 2021 from https://github.com/android/views-widgets-samples/tree/main/CardView.

134. Unsupported item type - Android Accessibility Help. Retrieved May 16, 2021 from https://support.google.com/accessibility/android/answer/7661305.

135. Color contrast - Android Accessibility Help. Retrieved May 16, 2021 from https://support.google.com/accessibility/android/answer/7158390?hl=en.

APPENDIX A: TALKBACK-FOCUSABLE CODE

Pseudo-code for determining if an element is TalkBack-focusable. On the left is Android's TalkBack implementation from version 6.0, obtained on February 14, 2018. On the right is my translation of the TalkBack code to work with the Rico dataset.

TalkBack 2/14/2018	***TalkBack-focusable* on the Rico dataset**

`boolean isAccessibilityFocusable(element):`
`  if element == null:`
`  return false`
`  if !element.isVisible():`
`  return false`
`  if element.isClickable() or isLongClickable():`
`  return true`
`  if element.isFocusable():`
`  return true`
`  if element.hasWebContent()and SupportsAction(ACTION_FOCUS):`
`  return true`
 	
`  if element.SupportsAnyAction(`
`  ACTION_FOCUS,`
`  ACTION_NEXT_HTML,`
`  ACTION_PREVIOUS_HTML_ELEMENT):`
`  return true`
`  if element.isTopLevelScrollItem()`
`  and element.isSpeaking():`
`  return true`
	`boolean isTalkBackFocusable(element):`
	`  // null elements do not appear in dataset`
	`  if element.Visibility == "visible":`
	`  return false`
	`  if element.Clickable == true or element.LongClickable == true:`
	`  return true`
	`  if element.Focusable == true:`
	`  return true`
	`  // following ATF,`
	`  // we ignore WebViews entirely`
	
	`  // no information on supported actions`
	
	`  // no information on scroll item`
	`  if element.isSpeaking():`
	`  return true`

`boolean isSpeaking():`
`  if element.hasText():`
`  return true`
`  if element.hasContentDescription():`
`  return true`
`  if element.isCheckable():`
`  return true`
 	
`  if element.hasNonActionable SpeakingChildren():`
`  return true`
	`boolean isSpeaking():`
	`  if element.Text != "":`
	`  return true`
	`  if element.ContentDescription != "":`
	`  return true`
	`  // no information on checkable`
	`  // but few classes of elements`
	`  // have this property`
	`  if element.hasNonActionable SpeakingChildren():`
	`  return true`

`boolean hasNonActionableSpeakingChildren():`
`  for child in element.Children():`
`  if !child.isVisible():`
`  return false`
`  if isAccessibilityFocusable(child):`
`  return false`
`  if child.isSpeaking():`
`  return true`
`  return false`
	`boolean hasNonActionableSpeakingChildren():`
	`  for child in element.Children:`
	`  if element.Visibility == "visible":`
	`  return false`
	`  if isAccessibilityFocusable(child):`
	`  return false`
	`  if child.isSpeaking():`
	`  return true`
	`  return false`

APPENDIX B: MODERATE USE CLASSES

This appendix presents accessibility failure data for the moderate use classes (i.e., in the top 1% of class use) tested for each failure. Each table provides the class name, number of apps with tested elements from that class, number of tested elements of that class in the dataset, number of elements with the accessibility failure, percent of elements with the accessibility failure, and proportion of elements with the accessibility failure.

Appendix B List of Tables

Table B.1: Missing labels in all relevant elements in moderate use classes

class	elements	apps with class	elements missing label	proportion missing label
android.widget.ImageButton	247015	4038	163008	0.66
android.widget.ImageView	635228	2976	580975	0.91
android.support.v7.widget.AppCompatImageView	157827	1845	133346	0.84
android.support.v7.view.menu.ActionMenuItemView	49529	1842	2537	0.05
android.support.v7.widget.AppCompatImageButton	108094	1725	73175	0.68
android.widget.EditText	35503	1276	28687	0.81
android.widget.LinearLayout	34332	1099	8062	0.23
android.widget.Button	97395	958	89058	0.91
android.support.v7.widget.ActionMenuPresenter$OverflowMenuButton	12642	884	121	0.01
com.android.internal.view.menu.ActionMenuItemView	23734	782	1649	0.07
android.view.View	31660	674	30828	0.97
android.widget.ListView	2718	671	2305	0.85
android.widget.ActionMenuPresenter$OverflowMenuButton	9780	647	17	0.00
android.support.design.widget.FloatingActionButton	6983	590	6484	0.93
android.widget.RelativeLayout	53765	545	5152	0.10
com.google.maps.api.android.lib6.gmm6.api.z	3885	492	0	0.00
android.widget.SeekBar	7414	489	7382	1.00
android.support.v7.widget.AppCompatButton	21075	422	18602	0.88
android.support.v7.internal.view.menu.ActionMenuItemView	10923	417	721	0.07
android.support.v7.widget.AppCompatCheckBox	53154	384	43670	0.82
android.widget.CheckBox	43428	375	43053	0.99
android.support.v7.widget.RecyclerView	1281	323	1149	0.90
android.widget.ToggleButton	341684	271	341150	1.00
android.support.v7.widget.SwitchCompat	8559	256	8480	0.99
android.widget.FrameLayout	8958	251	4863	0.54
org.apache.cordova.engine.SystemWebView	4963	238	0	0.00
android.support.v7.widget.AppCompatTextView	4612	224	2040	0.44

class	elements	apps with class	elements missing label	proportion missing label
android.widget.TextView	30003	221	7584	0.25
android.support.v7.widget.AppCompatSeekBar	1496	192	1496	1.00
android.widget.Switch	4044	183	3976	0.98
com.mopub.mobileads.HtmlBannerWebView	1757	143	0	0.00
android.widget.VideoView	733	133	728	0.99
com.facebook.ads.internal.view.a	1207	123	0	0.00
maps.D.p	1097	119	0	0.00
android.support.v7.widget.ActionMenuPresenter$d	1873	107	0	0.00
android.support.v7.widget.n	3181	107	1729	0.61
android.support.v7.widget.o	7693	100	5102	0.66
android.view.SurfaceView	2784	100	2770	0.99
android.widget.GridView	384	99	370	0.96
android.widget.NumberPicker	2367	96	2367	1.00
android.widget.ScrollView	676	94	654	0.97
android.support.v7.widget.AppCompatRadioButton	4760	91	4658	0.98

Table B.2: Missing labels in contentDescription-dependent elements in moderate use classes

class	apps with class	elements	elements missing labels	proportion missing label
android.widget.LinearLayout	6233	208614	2304	0.01
android.widget.TextView	5597	186161	1925	0.01
android.widget.ScrollView	5299	43934	612	0.01
android.widget.ListView	4189	41446	2293	0.06
android.widget.ImageButton	4038	126675	66458	0.52
android.widget.Button	3916	130195	34140	0.26
android.widget.RelativeLayout	3698	119104	2447	0.02
android.widget.ImageView	2976	121816	100474	0.82
android.support.v7.widget.AppCompatTextView	2962	111047	1342	0.01
android.support.v7.widget.AppCompatButton	2807	71534	7768	0.11
android.webkit.WebView	2480	0	0	NA
com.google.android.gms.ads.internal.webview.o	2344	54244	0	0
com.android.internal.widget.DialogTitle	2280	7704	0	0
android.support.v4.widget.DrawerLayout	2012	32160	7	0
android.support.v7.widget.AppCompatImageView	1845	58811	47807	0.81
android.support.v7.view.menu.ActionMenuItemView	1842	45379	1992	0.04
android.support.v7.widget.AppCompatImageButton	1725	62826	32402	0.52
android.widget.FrameLayout	1721	39422	1671	0.04
android.support.v7.widget.RecyclerView	1665	22169	1147	0.05
android.support.v4.view.ViewPager	1646	12432	278	0.02
android.widget.EditText	1617	26967	11199	0.42
android.support.v7.widget.AppCompatEditText	1350	0	0	NA
android.widget.ListPopupWindow$DropDownListView	924	2777	0	0
com.google.android.gms.ads.internal.webview.m	896	23279	0	0
android.support.v7.widget.ActionMenuPresenter$OverflowMenuButton	884	12318	121	0.01
com.android.internal.view.menu.ActionMenuItemView	782	20128	1173	0.06
android.widget.CheckBox	711	10636	6907	0.65

class	apps with class	elements	elements missing labels	proportion missing label
com.android.internal.app.AlertController$RecycleListView	692	1856	0	0
android.view.View	674	9912	9618	0.97
android.widget.GridView	657	4635	368	0.08
android.widget.ActionMenuPresenter$OverflowMenuButton	647	9780	17	0.00
android.support.v7.widget.CardView	611	20151	221	0.01
android.support.v7.widget.AppCompatCheckBox	602	11692	7498	0.64
android.support.design.widget.TabLayout$TabView	602	23398	326	0.01
android.support.design.widget.FloatingActionButton	590	6389	5932	0.93
android.support.v7.widget.AppCompatSpinner	555	6156	97	0.02
android.widget.Spinner	553	6478	207	0.03
android.support.v7.widget.DialogTitle	505	1541	0	0.00
com.google.maps.api.android.lib6.gmm6.api.z	492	3859	0	0.00
android.widget.SeekBar	489	5920	5888	0.99
android.support.v7.widget.SwitchCompat	453	4345	2518	0.58
android.widget.ToggleButton	436	12847	5052	0.39
android.support.v7.internal.view.menu.ActionMenuItemView	417	10133	671	0.07
android.widget.RadioButton	402	11315	3024	0.27
android.support.design.internal.NavigationMenuItemView	402	16475	51	0
android.widget.ExpandableListView	398	2636	138	0.05
android.widget.HorizontalScrollView	392	3091	150	0.05
android.support.v7.widget.ListPopupWindow$DropDownListView	368	855	1	0
android.support.v7.widget.AppCompatRadioButton	358	7940	1910	0.24
android.support.design.internal.NavigationMenuView	318	1404	5	0
android.support.v4.widget.NestedScrollView	294	2945	10	0
android.support.v7.widget.MenuPopupWindow$MenuDropDownListView	286	644	0	0
org.apache.cordova.engine.SystemWebView	238	4945	0	0
android.widget.Switch	231	1987	1580	0.80
android.support.v7.widget.AppCompatSeekBar	192	1446	1446	1
android.widget.NumberPicker$CustomEditText	188	0	0	NA

class	apps with class	elements	elements missing labels	proportion missing label
android.support.v7.widget.SearchView$SearchAutoComplete	184	0	0	NA
com.mopub.mobileads.HtmlBannerWebView	143	1701	0	0
android.support.v7.widget.i	137	2834	186	0.07
com.android.internal.widget.ScrollingTabContainerView$TabView	136	4792	42	0.01
android.support.v7.widget.AppCompatAutoCompleteTextView	133	0	0	NA
android.widget.VideoView	133	705	700	0.99
com.facebook.login.widget.LoginButton	133	309	4	0.01
android.widget.AutoCompleteTextView	124	0	0	NA
android.support.v7.widget.n	123	2098	913	0.44
com.facebook.ads.internal.view.a	123	1199	0	0
com.afollestad.materialdialogs.internal.MDButton	120	760	0	0
maps.D.p	119	1081	0	0
android.support.design.widget.Snackbar$SnackbarLayout	110	330	7	0.02
android.widget.TableLayout	109	966	0	0
android.support.v7.widget.o	107	5422	3638	0.67
android.support.v7.widget.ActionMenuPresenter$d	107	1859	0	0
android.support.v7.widget.ab	106	5272	81	0.02
android.view.SurfaceView	100	2784	2770	0.99
android.widget.SearchView$SearchAutoComplete	97	0	0	NA
com.jeremyfeinstein.slidingmenu.lib.CustomViewAbove	97	1626	18	0.01
android.widget.NumberPicker	96	2367	2367	1
android.support.design.widget.TabLayout$g	91	4416	127	0.03
com.startapp.android.publish.banner.bannerstandard.BannerStandard$1	91	1081	0	0
android.support.v7.widget.Toolbar	89	864	10	0.01
android.support.v7.widget.q	87	3850	2651	0.69
com.android.internal.policy.PhoneWindow$DecorView	86	952	0	0
com.github.clans.fab.FloatingActionButton	80	985	932	0.95
com.adobe.air.AIRWindowSurfaceView	74	1928	1928	1
android.widget.TabHost	73	784	0	0

class	apps with class	elements	elements missing labels	proportion missing label
com.mopub.mraid.MraidBridge$MraidWebView	72	412	0	0
com.google.android.gms.ads.internal.overlay.ai	70	196	196	1
android.support.design.widget.CheckableImageButton	70	469	464	0.99
android.support.design.widget.TextInputEditText	68	0	0	NA
com.amazon.device.ads.AdContainer	68	771	0	0
com.amazon.device.ads.AdLayout	68	728	0	0
android.support.v7.internal.widget.DialogTitle	67	148	0	0
org.apache.cordova.CordovaWebView	66	1575	0	0
android.support.v7.widget.p	64	1679	796	0.47
com.facebook.drawee.view.SimpleDraweeView	61	1757	1428	0.81
com.melnykov.fab.FloatingActionButton	60	571	540	0.95
android.support.v7.widget.aa	59	1942	1	0
android.support.v7.widget.m	57	1365	887	0.65
com.getbase.floatingactionbutton.FloatingActionsMenu$1	56	815	810	0.99
com.ksmobile.launcher.theme.base.view.HighlightTextView	56	492	0	0
ztu	56	369	0	0
org.xwalk.core.internal.XWalkContentView$XWalkContentViewApi23	54	1307	1307	1
android.support.v7.widget.h	54	1333	84	0.06
android.support.v7.widget.d$d	53	849	26	0.03
android.support.v7.widget.z	52	1811	413	0.23
com.mobeta.android.dslv.DragSortListView	50	318	19	0.06
android.widget.YearPickerView	50	547	0	0
android.support.v7.widget.AppCompatCheckedTextView	49	993	167	0.17
org.appcelerator.titanium.view.TiCompositeLayout	47	17016	8120	0.48
ti.modules.titanium.ui.widget.TiUILabel$1	47	7647	161	0.02
android.support.v7.widget.y	47	1871	13	0.01
android.support.v7.widget.an	46	913	122	0.13
com.wsi.android.framework.app.ui.widget.HorizontalScrollView	46	558	0	0
de.hdodenhof.circleimageview.CircleImageView	45	622	618	0.99

class	apps with class	elements	elements missing labels	proportion missing label
android.widget.TableRow	44	1576	11	0.01
com.mikepenz.materialdrawer.view.ScrimInsetsFrameLayout	44	1061	0	0
android.support.v7.widget.w	42	377	220	0.58
com.vervewireless.advert.internal.AdWebView	42	854	0	0
tj	41	93	19	0.20
se.emilsjolander.stickylistheaders.WrapperViewList	41	244	13	0.05
org.xwalk.core.internal.XWalkContentView	40	874	874	1
com.getbase.floatingactionbutton.FloatingActionButton	40	427	410	0.96
android.support.v7.widget.AppCompatRatingBar	39	255	255	1
android.support.v7.widget.x	39	1102	21	0.02
android.support.v7.widget.r	38	2189	971	0.44
com.facebook.react.views.view.ReactViewGroup	38	5328	173	0.03
android.support.percent.PercentRelativeLayout	38	836	11	0.01
com.facebook.ads.internal.f.a	38	440	0	0
com.github.paolorotolo.appintro.AppIntroViewPager	38	183	0	0
com.wsi.android.weather.ui.widget.BounceListView	38	104	0	0
com.makeramen.roundedimageview.RoundedImageView	37	616	594	0.96
android.support.v7.widget.u	37	672	413	0.61
com.facebook.widget.LoginButton	37	171	7	0.04
com.facebook.ads.internal.h.a	37	338	0	0
yei	37	318	0	0
com.android.volley.toolbox.NetworkImageView	36	741	538	0.73
ti.modules.titanium.ui.widget.TiUIButton$1	36	1747	1009	0.58
com.github.ksoichiro.android.observablescrollview.ObservableListView	36	419	4	0.01
ti.modules.titanium.ui.widget.TiUIScrollView$TiScrollViewLayout	36	367	2	0.01
com.actionbarsherlock.internal.widget.IcsListPopupWindow $DropDownListView	36	79	0	0

Table B.3: Elements too small either in moderate use classes

class	apps with class	elements	elements too small either	proportion elements too small either
android.widget.LinearLayout	4342	150631	40636	0.27
android.widget.ListView	4185	41355	346	0.01
android.widget.ImageButton	4030	124605	49631	0.40
android.widget.Button	3912	129054	49321	0.38
android.widget.RelativeLayout	2989	97608	27843	0.29
android.support.v7.widget.AppCompatButton	2805	70476	17989	0.26
android.widget.ImageView	2798	115667	60199	0.52
android.webkit.WebView	2478	27546	712	0.03
android.widget.TextView	2368	70225	46860	0.67
com.google.android.gms.ads.internal.webview.o	2344	54244	633	0.01
android.support.v7.widget.AppCompatTextView	2051	61659	43421	0.70
android.support.v7.view.menu.ActionMenuItemView	1842	45379	1636	0.04
android.support.v7.widget.AppCompatImageView	1749	56828	34595	0.61
android.support.v7.widget.AppCompatImageButton	1709	61624	24266	0.39
android.widget.EditText	1617	26967	19604	0.73
android.widget.FrameLayout	1439	31417	7194	0.23
android.support.v7.widget.AppCompatEditText	1350	23070	17747	0.77
android.widget.ListPopupWindow$DropDownListView	924	2777	21	0.01
com.google.android.gms.ads.internal.webview.m	896	23279	403	0.02
android.support.v7.widget.ActionMenuPresenter $OverflowMenuButton	884	12318	12029	0.98
com.android.internal.view.menu.ActionMenuItemView	782	20128	823	0.04
com.android.internal.app.AlertController$RecycleListView	692	1856	3	0.00
android.widget.CheckBox	686	10176	8358	0.82
android.widget.GridView	655	4599	140	0.03
android.widget.ActionMenuPresenter$OverflowMenuButton	647	9780	1831	0.19
android.view.View	638	9104	2617	0.29
android.support.design.widget.TabLayout$TabView	602	23357	3191	0.14

class	apps with class	elements	elements too small either	proportion elements too small either
android.support.design.widget.FloatingActionButton	585	6262	738	0.12
android.support.v7.widget.CardView	571	18928	1608	0.08
android.support.v7.widget.AppCompatCheckBox	559	9979	9336	0.94
android.support.v7.widget.AppCompatSpinner	553	6087	4301	0.71
android.widget.Spinner	553	6475	3145	0.49
com.google.maps.api.android.lib6.gmm6.api.z	492	3859	14	0.00
android.support.v7.widget.SwitchCompat	450	4329	3210	0.74
android.widget.ToggleButton	430	12688	6637	0.52
android.support.v7.internal.view.menu.ActionMenuItemView	417	10133	239	0.02
android.support.design.internal.NavigationMenuItemView	402	16475	952	0.06
android.widget.ExpandableListView	397	2635	153	0.06
android.widget.RadioButton	393	11124	6897	0.62
android.support.v7.widget.ListPopupWindow $DropDownListView	368	855	0	0.00
android.support.v7.widget.AppCompatRadioButton	326	7499	6142	0.82
android.support.v7.widget.MenuPopupWindow $MenuDropDownListView	286	644	0	0.00
org.apache.cordova.engine.SystemWebView	238	4945	9	0.00
android.widget.Switch	224	1948	1800	0.92
android.widget.NumberPicker$CustomEditText	188	2620	2619	1.00
android.support.v7.widget.SearchView$SearchAutoComplete	184	852	852	1.00
com.mopub.mobileads.HtmlBannerWebView	143	1701	7	0.00
android.support.v7.widget.i	136	2824	677	0.24
com.android.internal.widget.ScrollingTabContainerView $TabView	136	4792	229	0.05
android.support.v7.widget.AppCompatAutoCompleteTextView	133	1266	1012	0.80
com.facebook.login.widget.LoginButton	132	306	253	0.83
android.widget.AutoCompleteTextView	124	1054	810	0.77
com.facebook.ads.internal.view.a	123	1199	7	0.01
android.support.v7.widget.n	121	2735	1207	0.44

class	apps with class	elements	elements too small either	proportion elements too small either
com.afollestad.materialdialogs.internal.MDButton	120	760	0	0.00
maps.D.p	119	1081	1	0.00
android.support.design.widget.Snackbar$SnackbarLayout	110	330	91	0.28
android.support.v7.widget.ActionMenuPresenter$d	107	1859	1752	0.94
android.support.v7.widget.o	104	5346	2307	0.43
android.widget.SearchView$SearchAutoComplete	97	555	555	1.00
android.support.design.widget.TabLayout$g	91	4416	605	0.14
com.startapp.android.publish.banner.bannerstandard.BannerStandard$1	91	1081	11	0.01
com.github.clans.fab.FloatingActionButton	80	985	1	0.00
android.support.v7.widget.q	79	3795	2094	0.55
android.support.v7.widget.RecyclerView	79	661	14	0.02
com.adobe.air.AIRWindowSurfaceView	74	1928	0	0.00
android.support.v7.widget.Toolbar	73	750	17	0.02
com.mopub.mraid.MraidBridge$MraidWebView	72	412	1	0.00
android.support.design.widget.CheckableImageButton	70	469	464	0.99
com.google.android.gms.ads.internal.overlay.ai	70	196	0	0.00
android.support.design.widget.TextInputEditText	68	1418	1242	0.88
android.support.v7.widget.ab	67	2379	1449	0.61
org.apache.cordova.CordovaWebView	66	1575	1	0.00
android.support.v7.widget.p	62	1657	1265	0.76
com.facebook.drawee.view.SimpleDraweeView	61	1735	904	0.52
com.melnykov.fab.FloatingActionButton	59	562	64	0.11
android.support.v7.widget.m	56	1360	1066	0.78
com.getbase.floatingactionbutton.FloatingActionsMenu$1	56	815	18	0.02
com.ksmobile.launcher.theme.base.view.HighlightTextView	56	492	1	0.00
ztu	56	369	365	0.99
android.widget.ScrollView	55	249	0	0.00
android.support.v7.widget.h	54	1343	255	0.19

class	apps with class	elements	elements too small either	proportion elements too small either
org.xwalk.core.internal.XWalkContentView $XWalkContentViewApi23	54	1307	0	0.00
android.support.v7.widget.d$d	53	849	848	1.00
android.widget.YearPickerView	50	547	0	0.00
com.mobeta.android.dslv.DragSortListView	50	318	4	0.01
android.support.v7.widget.AppCompatCheckedTextView	48	975	739	0.76
org.appcelerator.titanium.view.TiCompositeLayout	47	17016	11898	0.70
ti.modules.titanium.ui.widget.TiUILabel$1	47	7647	7107	0.93
android.support.v7.widget.an	45	910	204	0.22
de.hdodenhof.circleimageview.CircleImageView	45	622	227	0.36
com.mikepenz.materialdrawer.view.ScrimInsetsFrameLayout	44	1061	0	0.00
android.support.v7.widget.y	43	1278	996	0.78
android.support.v7.widget.z	43	1371	789	0.58
android.widget.TableRow	43	1486	1066	0.72
com.vervewireless.advert.internal.AdWebView	42	854	0	0.00
se.emilsjolander.stickylistheaders.WrapperViewList	41	244	0	0.00
tj	41	93	26	0.28
com.getbase.floatingactionbutton.FloatingActionButton	40	427	2	0.00
org.xwalk.core.internal.XWalkContentView	40	874	0	0.00
android.support.v7.widget.x	39	779	492	0.63
com.facebook.ads.internal.f.a	38	440	1	0.00
com.wsi.android.weather.ui.widget.BounceListView	38	104	0	0.00
android.support.v7.widget.r	37	2207	1742	0.79
com.facebook.ads.internal.h.a	37	338	0	0.00

Table B.4: Elements too small both in moderate use classes

class	apps with class	elements	elements too small both	proportion elements too small both
android.widget.LinearLayout	4342	150631	1898	0.01
android.widget.ListView	4185	41355	0	0.00
android.widget.ImageButton	4030	124605	30424	0.24
android.widget.Button	3912	129054	16060	0.12
android.widget.RelativeLayout	2989	97608	3076	0.03
android.support.v7.widget.AppCompatButton	2805	70476	3593	0.05
android.widget.ImageView	2798	115667	43719	0.38
android.webkit.WebView	2478	27546	298	0.01
android.widget.TextView	2368	70225	5362	0.08
com.google.android.gms.ads.internal.webview.o	2344	54244	29	0.00
android.support.v7.widget.AppCompatTextView	2051	61659	4475	0.07
android.support.v7.view.menu.ActionMenuItemView	1842	45379	73	0.00
android.support.v7.widget.AppCompatImageView	1749	56828	24763	0.44
android.support.v7.widget.AppCompatImageButton	1709	61624	16194	0.26
android.widget.EditText	1617	26967	391	0.01
android.widget.FrameLayout	1439	31417	1104	0.04
android.support.v7.widget.AppCompatEditText	1350	23070	348	0.02
android.widget.ListPopupWindow$DropDownListView	924	2777	0	0.00
com.google.android.gms.ads.internal.webview.m	896	23279	4	0.00
android.support.v7.widget.ActionMenuPresenter $OverflowMenuButton	884	12318	293	0.02
com.android.internal.view.menu.ActionMenuItemView	782	20128	254	0.01
com.android.internal.app.AlertController$RecycleListView	692	1856	0	0.00
android.widget.CheckBox	686	10176	4763	0.47
android.widget.GridView	655	4599	0	0.00
android.widget.ActionMenuPresenter$OverflowMenuButton	647	9780	20	0.00
android.view.View	638	9104	1171	0.13
android.support.design.widget.TabLayout$TabView	602	23357	151	0.01

class	apps with class	elements	elements too small both	proportion elements too small both
android.support.design.widget.FloatingActionButton	585	6262	732	0.12
android.support.v7.widget.CardView	571	18928	21	0.00
android.support.v7.widget.AppCompatCheckBox	559	9979	4647	0.47
android.support.v7.widget.AppCompatSpinner	553	6087	8	0.00
android.widget.Spinner	553	6475	362	0.06
com.google.maps.api.android.lib6.gmm6.api.z	492	3859	0	0.00
android.support.v7.widget.SwitchCompat	450	4329	1648	0.38
android.widget.ToggleButton	430	12688	4236	0.33
android.support.v7.internal.view.menu.ActionMenuItemView	417	10133	42	0.00
android.support.design.internal.NavigationMenuItemView	402	16475	7	0.00
android.widget.ExpandableListView	397	2635	0	0.00
android.widget.RadioButton	393	11124	2353	0.21
android.support.v7.widget.ListPopupWindow$DropDownListView	368	855	0	0.00
android.support.v7.widget.AppCompatRadioButton	326	7499	944	0.13
android.support.v7.widget.MenuPopupWindow $MenuDropDownListView	286	644	0	0.00
org.apache.cordova.engine.SystemWebView	238	4945	0	0.00
android.widget.Switch	224	1948	790	0.41
android.widget.NumberPicker$CustomEditText	188	2620	5	0.00
android.support.v7.widget.SearchView$SearchAutoComplete	184	852	0	0.00
com.mopub.mobileads.HtmlBannerWebView	143	1701	1	0.00
android.support.v7.widget.i	136	2824	106	0.04
com.android.internal.widget.ScrollingTabContainerView$TabVie w	136	4792	0	0.00
android.support.v7.widget.AppCompatAutoCompleteTextView	133	1266	0	0.00
com.facebook.login.widget.LoginButton	132	306	1	0.00
android.widget.AutoCompleteTextView	124	1054	1	0.00
com.facebook.ads.internal.view.a	123	1199	0	0.00
android.support.v7.widget.n	121	2735	331	0.12

class	apps with class	elements	elements too small both	proportion elements too small both
com.afollestad.materialdialogs.internal.MDButton	120	760	0	0.00
maps.D.p	119	1081	0	0.00
android.support.design.widget.Snackbar$SnackbarLayout	110	330	0	0.00
android.support.v7.widget.ActionMenuPresenter$d	107	1859	26	0.01
android.support.v7.widget.o	104	5346	1252	0.23
android.widget.SearchView$SearchAutoComplete	97	555	1	0.00
android.support.design.widget.TabLayout$g	91	4416	38	0.01
com.startapp.android.publish.banner.bannerstandard.BannerStandard$1	91	1081	0	0.00
com.github.clans.fab.FloatingActionButton	80	985	0	0.00
android.support.v7.widget.q	79	3795	1351	0.36
android.support.v7.widget.RecyclerView	79	661	0	0.00
com.adobe.air.AIRWindowSurfaceView	74	1928	0	0.00
android.support.v7.widget.Toolbar	73	750	0	0.00
com.mopub.mraid.MraidBridge$MraidWebView	72	412	0	0.00
android.support.design.widget.CheckableImageButton	70	469	423	0.90
com.google.android.gms.ads.internal.overlay.ai	70	196	0	0.00
android.support.design.widget.TextInputEditText	68	1418	0	0.00
android.support.v7.widget.ab	67	2379	267	0.11
org.apache.cordova.CordovaWebView	66	1575	0	0.00
android.support.v7.widget.p	62	1657	394	0.24
com.facebook.drawee.view.SimpleDraweeView	61	1735	801	0.46
com.melnykov.fab.FloatingActionButton	59	562	62	0.11
android.support.v7.widget.m	56	1360	379	0.28
com.getbase.floatingactionbutton.FloatingActionsMenu$1	56	815	18	0.02
com.ksmobile.launcher.theme.base.view.HighlightTextView	56	492	0	0.00
ztu	56	369	99	0.27
android.widget.ScrollView	55	249	0	0.00
android.support.v7.widget.h	54	1343	134	0.10

class	apps with class	elements	elements too small both	proportion elements too small both
org.xwalk.core.internal.XWalkContentView $XWalkContentViewApi23	54	1307	0	0.00
android.support.v7.widget.d$d	53	849	22	0.03
android.widget.YearPickerView	50	547	0	0.00
com.mobeta.android.dslv.DragSortListView	50	318	0	0.00
android.support.v7.widget.AppCompatCheckedTextView	48	975	233	0.24
org.appcelerator.titanium.view.TiCompositeLayout	47	17016	1564	0.09
ti.modules.titanium.ui.widget.TiUILabel$1	47	7647	2078	0.27
android.support.v7.widget.an	45	910	8	0.01
de.hdodenhof.circleimageview.CircleImageView	45	622	201	0.32
com.mikepenz.materialdrawer.view.ScrimInsetsFrameLayout	44	1061	0	0.00
android.support.v7.widget.y	43	1278	14	0.01
android.support.v7.widget.z	43	1371	211	0.15
android.widget.TableRow	43	1486	0	0.00
com.vervewireless.advert.internal.AdWebView	42	854	0	0.00
se.emilsjolander.stickylistheaders.WrapperViewList	41	244	0	0.00
tj	41	93	1	0.01
com.getbase.floatingactionbutton.FloatingActionButton	40	427	2	0.00
org.xwalk.core.internal.XWalkContentView	40	874	0	0.00
android.support.v7.widget.x	39	779	29	0.04
com.facebook.ads.internal.f.a	38	440	0	0.00
com.wsi.android.weather.ui.widget.BounceListView	38	104	0	0.00
android.support.v7.widget.r	37	2207	652	0.30
com.facebook.ads.internal.h.a	37	338	0	0.00

Table B.5: Elements too narrow only in moderate use classes

class	apps with class	elements	elements too narrow only	proportion elements too narrow only
android.widget.LinearLayout	4342	150631	2259	0.01
android.widget.ListView	4185	41355	92	0.00
android.widget.ImageButton	4030	124605	3083	0.02
android.widget.Button	3912	129054	3239	0.03
android.widget.RelativeLayout	2989	97608	2770	0.03
android.support.v7.widget.AppCompatButton	2805	70476	560	0.01
android.widget.ImageView	2798	115667	3496	0.03
android.webkit.WebView	2478	27546	34	0.00
android.widget.TextView	2368	70225	843	0.01
com.google.android.gms.ads.internal.webview.o	2344	54244	78	0.00
android.support.v7.widget.AppCompatTextView	2051	61659	541	0.01
android.support.v7.view.menu.ActionMenuItemView	1842	45379	857	0.02
android.support.v7.widget.AppCompatImageView	1749	56828	3402	0.06
android.support.v7.widget.AppCompatImageButton	1709	61624	1562	0.03
android.widget.EditText	1617	26967	304	0.01
android.widget.FrameLayout	1439	31417	279	0.01
android.support.v7.widget.AppCompatEditText	1350	23070	99	0.00
android.widget.ListPopupWindow$DropDownListView	924	2777	13	0.00
com.google.android.gms.ads.internal.webview.m	896	23279	24	0.00
android.support.v7.widget.ActionMenuPresenter $OverflowMenuButton	884	12318	11649	0.95
com.android.internal.view.menu.ActionMenuItemView	782	20128	422	0.02
com.android.internal.app.AlertController$RecycleListView	692	1856	0	0.00
android.widget.CheckBox	686	10176	781	0.08
android.widget.GridView	655	4599	4	0.00

class	apps with class	elements	elements too narrow only	proportion elements too narrow only
android.widget.ActionMenuPresenter$OverflowMenuButton	647	9780	1773	0.18
android.view.View	638	9104	396	0.04
android.support.design.widget.TabLayout$TabView	602	23357	669	0.03
android.support.design.widget.FloatingActionButton	585	6262	0	0.00
android.support.v7.widget.CardView	571	18928	55	0.00
android.support.v7.widget.AppCompatCheckBox	559	9979	608	0.06
android.support.v7.widget.AppCompatSpinner	553	6087	0	0.00
android.widget.Spinner	553	6475	29	0.00
com.google.maps.api.android.lib6.gmm6.api.z	492	3859	4	0.00
android.support.v7.widget.SwitchCompat	450	4329	154	0.04
android.widget.ToggleButton	430	12688	38	0.00
android.support.v7.internal.view.menu.ActionMenuItemView	417	10133	111	0.01
android.support.design.internal.NavigationMenuItemView	402	16475	63	0.00
android.widget.ExpandableListView	397	2635	18	0.01
android.widget.RadioButton	393	11124	36	0.00
android.support.v7.widget.ListPopupWindow$DropDownListVie w	368	855	0	0.00
android.support.v7.widget.AppCompatRadioButton	326	7499	162	0.02
android.support.v7.widget.MenuPopupWindow $MenuDropDownListView	286	644	0	0.00
org.apache.cordova.engine.SystemWebView	238	4945	8	0.00
android.widget.Switch	224	1948	26	0.01
android.widget.NumberPicker$CustomEditText	188	2620	0	0.00
android.support.v7.widget.SearchView$SearchAutoComplete	184	852	0	0.00
com.mopub.mobileads.HtmlBannerWebView	143	1701	0	0.00
android.support.v7.widget.i	136	2824	42	0.01
com.android.internal.widget.ScrollingTabContainerView$TabVie w	136	4792	147	0.03
android.support.v7.widget.AppCompatAutoCompleteTextView	133	1266	0	0.00

class	apps with class	elements	elements too narrow only	proportion elements too narrow only
com.facebook.login.widget.LoginButton	132	306	2	0.01
android.widget.AutoCompleteTextView	124	1054	0	0.00
com.facebook.ads.internal.view.a	123	1199	0	0.00
android.support.v7.widget.n	121	2735	115	0.04
com.afollestad.materialdialogs.internal.MDButton	120	760	0	0.00
maps.D.p	119	1081	0	0.00
android.support.design.widget.Snackbar$SnackbarLayout	110	330	0	0.00
android.support.v7.widget.ActionMenuPresenter$d	107	1859	1726	0.93
android.support.v7.widget.o	104	5346	323	0.06
android.widget.SearchView$SearchAutoComplete	97	555	0	0.00
android.support.design.widget.TabLayout$g	91	4416	220	0.05
com.startapp.android.publish.banner.bannerstandard.BannerStandard$1	91	1081	0	0.00
com.github.clans.fab.FloatingActionButton	80	985	0	0.00
android.support.v7.widget.q	79	3795	265	0.07
android.support.v7.widget.RecyclerView	79	661	14	0.02
com.adobe.air.AIRWindowSurfaceView	74	1928	0	0.00
android.support.v7.widget.Toolbar	73	750	0	0.00
com.mopub.mraid.MraidBridge$MraidWebView	72	412	0	0.00
android.support.design.widget.CheckableImageButton	70	469	2	0.00
com.google.android.gms.ads.internal.overlay.ai	70	196	0	0.00
android.support.design.widget.TextInputEditText	68	1418	0	0.00
android.support.v7.widget.ab	67	2379	13	0.01
org.apache.cordova.CordovaWebView	66	1575	0	0.00
android.support.v7.widget.p	62	1657	537	0.32
com.facebook.drawee.view.SimpleDraweeView	61	1735	4	0.00
com.melnykov.fab.FloatingActionButton	59	562	0	0.00
android.support.v7.widget.m	56	1360	219	0.16

class	apps with class	elements	elements too narrow only	proportion elements too narrow only
com.getbase.floatingactionbutton.FloatingActionsMenu$1	56	815	0	0.00
com.ksmobile.launcher.theme.base.view.HighlightTextView	56	492	0	0.00
ztu	56	369	0	0.00
android.widget.ScrollView	55	249	0	0.00
android.support.v7.widget.h	54	1343	4	0.00
org.xwalk.core.internal.XWalkContentView $XWalkContentViewApi23	54	1307	0	0.00
android.support.v7.widget.d$d	53	849	821	0.97
android.widget.YearPickerView	50	547	0	0.00
com.mobeta.android.dslv.DragSortListView	50	318	0	0.00
android.support.v7.widget.AppCompatCheckedTextView	48	975	6	0.01
org.appcelerator.titanium.view.TiCompositeLayout	47	17016	98	0.01
ti.modules.titanium.ui.widget.TiUILabel$1	47	7647	13	0.00
android.support.v7.widget.an	45	910	6	0.01
de.hdodenhof.circleimageview.CircleImageView	45	622	3	0.00
com.mikepenz.materialdrawer.view.ScrimInsetsFrameLayout	44	1061	0	0.00
android.support.v7.widget.y	43	1278	0	0.00
android.support.v7.widget.z	43	1371	15	0.01
android.widget.TableRow	43	1486	0	0.00
com.vervewireless.advert.internal.AdWebView	42	854	0	0.00
se.emilsjolander.stickylistheaders.WrapperViewList	41	244	0	0.00
tj	41	93	0	0.00
com.getbase.floatingactionbutton.FloatingActionButton	40	427	0	0.00
org.xwalk.core.internal.XWalkContentView	40	874	0	0.00
android.support.v7.widget.x	39	779	18	0.02
com.facebook.ads.internal.f.a	38	440	0	0.00
com.wsi.android.weather.ui.widget.BounceListView	38	104	0	0.00
android.support.v7.widget.r	37	2207	11	0.00

class	apps with class	elements	elements too narrow only	proportion elements too narrow only
com.facebook.ads.internal.h.a	37	338	0	0.00
com.facebook.widget.LoginButton	37	171	0	0.00
com.makeramen.roundedimageview.RoundedImageView	37	616	19	0.03
yei	37	318	0	0.00
android.support.v7.widget.aa	36	972	3	0.00
com.actionbarsherlock.internal.widget.IcsListPopupWindow $DropDownListView	36	79	0	0.00
com.github.ksoichiro.android.observablescrollview. ObservableListView	36	419	0	0.00
ti.modules.titanium.ui.widget.TiUIButton$1	36	1576	2	0.00
ti.modules.titanium.ui.widget.TiUIScrollView$TiScrollViewLayout	36	367	0	0.00
com.facebook.internal.WebDialog$3	35	126	0	0.00
android.support.design.internal.NavigationMenuView	34	120	0	0.00
android.support.v7.internal.widget.TintImageView	33	219	0	0.00
com.viewpagerindicator.TabPageIndicator$TabView	33	1212	4	0.00
android.support.v7.widget.u	32	658	0	0.00
com.doapps.android.mln.categoryviewer.articlestream. ArticleStreamItemView	32	392	0	0.00
com.google.android.gms.maps.MapView	32	260	0	0.00
android.support.v7.widget.t	31	424	0	0.00
com.android.volley.toolbox.NetworkImageView	31	624	18	0.03
org.nexage.sourcekit.mraid.MRAIDView$3	31	239	0	0.00
android.support.percent.PercentRelativeLayout	30	784	0	0.00
android.support.v7.widget.e	30	565	336	0.59
android.support.v7.widget.j	30	440	164	0.37
android.support.v7.widget.w	29	312	0	0.00
android.support.design.widget.TabLayout$f	28	1247	13	0.01
android.support.v7.widget.l	28	514	0	0.00
com.android.internal.view.menu.IconMenuItemView	28	388	0	0.00

class	apps with class	elements	elements too narrow only	proportion elements too narrow only
com.facebook.ads.internal.i.c	28	195	0	0.00
com.handmark.pulltorefresh.library.PullToRefreshListView $InternalListViewSDK9	28	225	3	0.01
com.rengwuxian.materialedittext.MaterialEditText	28	673	0	0.00
com.yinzcam.common.android.ui.IconLayoutButton	28	292	0	0.00
org.nexage.sourcekit.mraid.rtb.ReportButton	28	155	0	0.00
android.support.v7.widget.ak	27	574	10	0.02
in.srain.cube.views.GridViewWithHeaderAndFooter	27	354	0	0.00
android.support.v7.widget.ba	26	1384	47	0.03
android.widget.CheckedTextView	26	518	0	0.00
com.cyrilmottier.android.listviewtipsandtricks.widget. DontPressWithParentButton	25	3720	0	0.00

Table B.6: Elements Too short only in moderate use classes

class	apps with class	elements	elements too short only	proportion elements too short only
android.widget.LinearLayout	4342	150631	36479	0.24
android.widget.ListView	4185	41355	254	0.01
android.widget.ImageButton	4030	124605	16124	0.13
android.widget.Button	3912	129054	30022	0.23
android.widget.RelativeLayout	2989	97608	21997	0.23
android.support.v7.widget.AppCompatButton	2805	70476	13836	0.20
android.widget.ImageView	2798	115667	12984	0.11
android.webkit.WebView	2478	27546	380	0.01
android.widget.TextView	2368	70225	40655	0.58
com.google.android.gms.ads.internal.webview.o	2344	54244	526	0.01
android.support.v7.widget.AppCompatTextView	2051	61659	38405	0.62
android.support.v7.view.menu.ActionMenuItemView	1842	45379	706	0.02
android.support.v7.widget.AppCompatImageView	1749	56828	6430	0.11
android.support.v7.widget.AppCompatImageButton	1709	61624	6510	0.11
android.widget.EditText	1617	26967	18909	0.70
android.widget.FrameLayout	1439	31417	5811	0.18
android.support.v7.widget.AppCompatEditText	1350	23070	17300	0.75
android.widget.ListPopupWindow$DropDownListView	924	2777	8	0.00
com.google.android.gms.ads.internal.webview.m	896	23279	375	0.02
android.support.v7.widget.ActionMenuPresenter $OverflowMenuButton	884	12318	87	0.01
com.android.internal.view.menu.ActionMenuItemView	782	20128	147	0.01
com.android.internal.app.AlertController$RecycleListView	692	1856	3	0.00
android.widget.CheckBox	686	10176	2814	0.28
android.widget.GridView	655	4599	136	0.03
android.widget.ActionMenuPresenter$OverflowMenuButton	647	9780	38	0.00
android.view.View	638	9104	1050	0.12
android.support.design.widget.TabLayout$TabView	602	23357	2371	0.10

class	apps with class	elements	elements too short only	proportion elements too short only
android.support.design.widget.FloatingActionButton	585	6262	6	0.00
android.support.v7.widget.CardView	571	18928	1532	0.08
android.support.v7.widget.AppCompatCheckBox	559	9979	4081	0.41
android.support.v7.widget.AppCompatSpinner	553	6087	4293	0.71
android.widget.Spinner	553	6475	2754	0.43
com.google.maps.api.android.lib6.gmm6.api.z	492	3859	10	0.00
android.support.v7.widget.SwitchCompat	450	4329	1408	0.33
android.widget.ToggleButton	430	12688	2363	0.19
android.support.v7.internal.view.menu.ActionMenuItemView	417	10133	86	0.01
android.support.design.internal.NavigationMenuItemView	402	16475	882	0.05
android.widget.ExpandableListView	397	2635	135	0.05
android.widget.RadioButton	393	11124	4508	0.41
android.support.v7.widget.ListPopupWindow$DropDownListView	368	855	0	0.00
android.support.v7.widget.AppCompatRadioButton	326	7499	5036	0.67
android.support.v7.widget.MenuPopupWindow $MenuDropDownListView	286	644	0	0.00
org.apache.cordova.engine.SystemWebView	238	4945	1	0.00
android.widget.Switch	224	1948	984	0.51
android.widget.NumberPicker$CustomEditText	188	2620	2614	1.00
android.support.v7.widget.SearchView$SearchAutoComplete	184	852	852	1
com.mopub.mobileads.HtmlBannerWebView	143	1701	6	0.00
android.support.v7.widget.i	136	2824	529	0.19
com.android.internal.widget.ScrollingTabContainerView$TabVie w	136	4792	82	0.02
android.support.v7.widget.AppCompatAutoCompleteTextView	133	1266	1012	0.80
com.facebook.login.widget.LoginButton	132	306	250	0.82
android.widget.AutoCompleteTextView	124	1054	809	0.77
com.facebook.ads.internal.view.a	123	1199	7	0.01
android.support.v7.widget.n	121	2735	761	0.28

class	apps with class	elements	elements too short only	proportion elements too short only
com.afollestad.materialdialogs.internal.MDButton	120	760	0	0
maps.D.p	119	1081	1	0.00
android.support.design.widget.Snackbar$SnackbarLayout	110	330	91	0.28
android.support.v7.widget.ActionMenuPresenter$d	107	1859	0	0
android.support.v7.widget.o	104	5346	732	0.14
android.widget.SearchView$SearchAutoComplete	97	555	554	1.00
android.support.design.widget.TabLayout$g	91	4416	347	0.08
com.startapp.android.publish.banner.bannerstandard.BannerStandard$1	91	1081	11	0.01
com.github.clans.fab.FloatingActionButton	80	985	1	0.00
android.support.v7.widget.q	79	3795	478	0.13
android.support.v7.widget.RecyclerView	79	661	0	0
com.adobe.air.AIRWindowSurfaceView	74	1928	0	0
android.support.v7.widget.Toolbar	73	750	17	0.02
com.mopub.mraid.MraidBridge$MraidWebView	72	412	1	0.00
android.support.design.widget.CheckableImageButton	70	469	39	0.08
com.google.android.gms.ads.internal.overlay.ai	70	196	0	0
android.support.design.widget.TextInputEditText	68	1418	1242	0.88
android.support.v7.widget.ab	67	2379	1169	0.49
org.apache.cordova.CordovaWebView	66	1575	1	0.00
android.support.v7.widget.p	62	1657	334	0.20
com.facebook.drawee.view.SimpleDraweeView	61	1735	99	0.06
com.melnykov.fab.FloatingActionButton	59	562	2	0.00
android.support.v7.widget.m	56	1360	468	0.34
com.getbase.floatingactionbutton.FloatingActionsMenu$1	56	815	0	0
com.ksmobile.launcher.theme.base.view.HighlightTextView	56	492	1	0.00
ztu	56	369	266	0.72
android.widget.ScrollView	55	249	0	0
android.support.v7.widget.h	54	1343	117	0.09

class	apps with class	elements	elements too short only	proportion elements too short only
org.xwalk.core.internal.XWalkContentView $XWalkContentViewApi23	54	1307	0	0
android.support.v7.widget.d$d	53	849	5	0.01
android.widget.YearPickerView	50	547	0	0
com.mobeta.android.dslv.DragSortListView	50	318	4	0.01
android.support.v7.widget.AppCompatCheckedTextView	48	975	500	0.51
org.appcelerator.titanium.view.TiCompositeLayout	47	17016	10236	0.60
ti.modules.titanium.ui.widget.TiUILabel$1	47	7647	5016	0.66
android.support.v7.widget.an	45	910	190	0.21
de.hdodenhof.circleimageview.CircleImageView	45	622	23	0.04
com.mikepenz.materialdrawer.view.ScrimInsetsFrameLayout	44	1061	0	0
android.support.v7.widget.y	43	1278	982	0.76
android.support.v7.widget.z	43	1371	563	0.41
android.widget.TableRow	43	1486	1066	0.72
com.vervewireless.advert.internal.AdWebView	42	854	0	0
se.emilsjolander.stickylistheaders.WrapperViewList	41	244	0	0
tj	41	93	25	0.27
com.getbase.floatingactionbutton.FloatingActionButton	40	427	0	0
org.xwalk.core.internal.XWalkContentView	40	874	0	0
android.support.v7.widget.x	39	779	445	0.57
com.facebook.ads.internal.f.a	38	440	1	0.00
com.wsi.android.weather.ui.widget.BounceListView	38	104	0	0
android.support.v7.widget.r	37	2207	1079	0.49
com.facebook.ads.internal.h.a	37	338	0	0

APPENDIX C: ACCESSIBILITY FAILURE TESTS FROM ACCESSIBILITY TEST FRAMEWORK FOR ANDROID

Many of my accessibility failure tests were based on the Google-released Accessibility Test Framework for Android Checks, as referenced in the table below. I used version 2.1, accessed February 25, 2018 from their open-source GitHub repository found at https://github.com/google/Accessibility-Test-Framework-for-Android.

Accessibility Barrier Test	Accessibility Test Framework for Android Check
Few TalkBack-Focusable Elements	N/A
Missing Label	`SpeakableTextPresentViewCheck`
Duplicate Label	`DuplicateSpekableTextViewHierarchyCheck on element.Clickable == True`
Uninformative Label	N/A
Editable TextView with `contentDescription`	`EditableContentDescViewCheck`
Fully Overlapping Clickable Elements	`DuplicteClickableBoundsViewCheck`
Size-Based Inaccessibility	`TouchTargetSizeViewCheck`

APPENDIX D: FEW TALKBACK-FOCUSABLE ELEMENTS

TOOL CLASS FREQUENCIES

This appendix presents data tables for game engine and cross-platform tool classes associated with the few TalkBack-Focusable elements failure.

Appendix D List of Tables

Table D.1: Use and prevalence of the few TalkBack focusable elements failures for a set of game engine and cross-platform tools.

Tool	Unity	Cocos2d-x	Adobe Air	Crosswalk	Apache Cordova
# elements total	1848	3790	2676	2181	149
# total apps using class	63	99	102	94	8
# elements in few-TalkBack focusable	1848	1983	1655	1332	134
% elements in few-TalkBack focusable	100%	52%	62%	61%	90%
# elements in zero TalkBack focusable	1770	1983	748	0	0
% elements in zero TalkBack-focusable	96%	52%	28%	0%	0%
# elements in (0,1] TalkBack-focusable	78	0	907	1332	134
% elements in (0,1] TalkBack-focusable	4%	0%	34%	61%	90%

Table D.2: Cross-app use data for a set classes associated from game engine and cross-platform tools.

class	tool	# elements total	# total apps using class
com.unity3d.player.UnityPlayer	Unity	1733	59
com.unity3d.player.UnityPlayer$24	Unity	115	4
org.cocos2dx.lib.Cocos2dxGLSurfaceView	Cocos2d-x	1940	53
org.cocos2dx.lib.Cocos2dxEditText	Cocos2d-x	1229	33
org.cocos2dx.lib.Cocos2dxEditBox	Cocos2d-x	621	13
com.adobe.air.AIRWindowSurfaceView	Adobe Air	2676	102
org.xwalk.core.internal.XWalkContentView $XWalkContentViewApi23	Crosswalk	1307	54
org.xwalk.core.internal.XWalkContentView	Crosswalk	874	40
com.ansca.corona.CoronaEditText	Apache Cordova	149	8

Table D.3: Prevalence of the overall few TalkBack focusable element failure for a set classes from game engine and cross-platform tools.

class	# elements in *few-TalkBack focusable*	% elements in *few-TalkBack focusable*	# apps in *few-TalkBack focusable* using class	% apps in *few-TalkBack focusable* using class
com.unity3d.player.UnityPlayer	1733	100%	59	100%
com.unity3d.player. UnityPlayer$24	115	100%	4	100%
org.cocos2dx.lib. Cocos2dxGLSurfaceView	1025	53%	29	55%
org.cocos2dx.lib. Cocos2dxEditText	606	49%	18	55%
org.cocos2dx.lib. Cocos2dxEditBox	352	57%	6	46%
com.adobe.air. AIRWindowSurfaceView	1655	62%	65	64%
org.xwalk.core.internal. XWalkContentView $XWalkContentViewApi23	693	53.0%	34	63%
org.xwalk.core.internal. XWalkContentView	639	73.1%	31	78%
com.ansca.corona. CoronaEditText	134	89.9%	6	75%

Table D.4: Prevalence of the zero TalkBack focusable element failure for a set classes from game engine and cross-platform tools.

class	# element in zero TalkBack-focusable	% elements in zero TalkBack-focusable	# apps in zero TalkBack-focusable using class	% apps in zero TalkBack-focusable using class
com.unity3d.player.UnityPlayer	1733	100%	59	100%
com.unity3d.player.UnityPlayer$24	37	32%	3	75%
org.cocos2dx.lib.Cocos2dxGLSurfaceView	1025	53%	29	55%
org.cocos2dx.lib.Cocos2dxEditText	606	49%	18	55%
org.cocos2dx.lib.Cocos2dxEditBox	352	57%	6	46%
com.adobe.air.AIRWindowSurfaceView	748	28%	28	28%
org.xwalk.core.internal.XWalkContentView$XWalkContentViewApi23	0	0%	0	0%
org.xwalk.core.internal.XWalkContentView	0	0%	0	0%
com.ansca.corona.CoronaEditText	0	0%	0	0%

Table D.5: Prevalence of the (0,1] TalkBack-focusable element failure for a set classes from game engine and cross-platform tools.

Class	# elements in (0,1] TalkBack-focusable	% elements in (0,1] TalkBack-focusable	# apps in (0,1] TalkBack-focusable using class	% apps in (0,1] TalkBack-focusable using class
com.unity3d.player.UnityPlayer	0	0%	0	0%
com.unity3d.player. UnityPlayer$24	78	68%	1	25%
org.cocos2dx.lib. Cocos2dxGLSurfaceView	0	0%	0	0%
org.cocos2dx.lib. Cocos2dxEditText	0	0%	0	0%
org.cocos2dx.lib. Cocos2dxEditBox	0	0%	0	0%
com.adobe.air. AIRWindowSurfaceView	907	34%	37	36%
org.xwalk.core.internal. XWalkContentView $XWalkContentViewApi23	693	53%	34	63%
org.xwalk.core.internal. XWalkContentView	639	73%	31	78%
com.ansca.corona. CoronaEditText	134	90%	6	75%

APPENDIX E: GOOGLE+ AND FACEBOOK LOGIN LABELS

Table E.1: All 4 labels from the 687 Google+ yei and ztu class elements.

Labels for Google+ `yei` **and** `ztu` **class elements**
Touch to add your plus one to the existing
Touch to plus one.
17k
42k

Table E.2: All 28 labels from the 309 Facebook Login elements from the class

`com.facebook.login.widget.LoginButton`

Labels for Facebook Login `com.facebook.login.widget.LoginButton` **class elements**
Log in with Facebook
Connect with Facebook
<U+062A><U+0633><U+062C><U+064A><U+0644><U+0627><U+0644><U+062F><U+062E><U+0648><U+0644><U+0639><U+0628><U+0631><U+0641><U+0627><U+064A><U+0633><U+0628><U+0648><U+0643>
Login with Facebook
Sign up with Facebook
Log out
Conectar
Ingresar con Facebook
Sign In with Facebook
Enter with Facebook
SIGN IN WITH FACEBOOK
<missing label>
Continue with Facebook
CONTINUE WITH FACEBOOK
Facebook
<U+D398><U+C774><U+C2A4><U+BD81><U+C73C><U+B85C> <U+B85C><U+ADF8><U+C778>
Register with Facebook
Log In via Facebook
Sign Up with Facebook
Sign in with Facebook
Connect using Facebook
Get from facebook
Facebook<U+C73C><U+B85C> <U+B85C><U+ADF8><U+C778>
Đang nh<U+1EAD>p b<U+1EB1>ng Facebook
Facebook profili il<U+0259> qeydiyyatdan keç
Facebook profilind<U+0259>n daxil ol
LOG IN WITH FACEBOOK
SIGN UP WITH FACEBOOK