

© 2017 Biplab Deka

INTERACTION MINING MOBILE APPS

BY

BIPLAB DEKA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Assistant Professor Ranjitha Kumar, Chair
Assistant Professor Lav Varshney
Professor Brian Bailey
Professor Andrew Singer

ABSTRACT

Millions of mobile apps are used by billions of users every day. Although the design of these apps play an important role in their adoption, the design process still remains complex and time intensive. At the same time, existing apps embody multiple solutions to numerous design problems faced by app developers. *How do we make this design knowledge embedded in existing apps accessible to designers? And how can it help simplify the app design process?*

This dissertation introduces interaction mining, a technique to capture the designs of mobile apps in a way that supports data-driven design applications. It presents systems that implement interaction mining for Android apps without requiring any access to their source code making it possible to design mine apps at an unprecedented scale. It presents Rico, the largest publicly available mobile app design repository to date. It discusses how such repositories created using interaction mining can be used to train models that enable applications such as keyword and example-based search interactions for mobile screens and user flows. It also presents zero-integration performance testing (ZIPT), a novel technique for testing app designs. It demonstrates how ZIPT can be used to help designers understand which examples to draw from in the early stages of the app design process and perform comparative testing at scale with low cost and effort in the later stages of the process.

ACKNOWLEDGMENTS

I would like to thank my advisor, Ranjitha Kumar, for her advice and guidance throughout the process of working on this dissertation. She challenged me to reach beyond low-hanging fruit and helped me develop a sense of what problems would be rewarding to solve. In fact, it was her enthusiasm for design that got me thinking about the problems that I explore in this dissertation. I would also like to thank my committee members Brian Bailey, Lav Varshney, and Andrew Singer for their valuable feedback and suggestions on how to improve and better present my work.

I have been fortunate to have great collaborators. Zifeng (Forrest) Huang, contributed significantly to all the work presented in this dissertation. It was his initial expertise about all things mobile that gave me the confidence to start tinkering with designs of mobile apps. Chad Frazen contributed significantly to Rico and single handedly developed the web interface for ZIPT. A big thank you to both of them. I would like to thank my collaborators and mentors at Google: Jeff Nichols, Yang Li, and Dan Afergan for their support and guidance throughout the Rico and ZIPT projects.

I am also grateful for other mentors I have had in graduate school. I would like to thank Rakesh Kumar for the things he taught me about research in the process of writing my first several papers in grad school, Jerry Talton for all the valuable feedback he has given me on my work, and Brian Lilly for his positive encouragement and for always being there when I needed advice on anything outside research.

I would also like to thank the several undergraduate and masters students in the data-driven design group who have worked with me and have contributed in ways big and small to my work. Thank you to Kedan Li, Jinda Han, Devin Ho, Sujay Khandekar, Abhishek Harish, Thomas Liu, Krishna Dusad, and Erik Luo.

I am also grateful for the support of my parents, Niru and Narayan, and

my brother Angshuman. Finally, I thank my wife Mary for her immense support and patience as I dealt with the ups and downs in my research over the last several years.

TABLE OF CONTENTS

LIST OF TABLES	viii
LIST OF FIGURES	ix
CHAPTER 1 INTRODUCTION	1
1.1 Challenges in the Current App Design Process	2
1.2 Designing with Data-Driven Tools	3
1.2.1 Food Logging in a Diabetes App	3
1.2.2 Playlist Interactions in a Music App	4
1.3 Contributions	5
1.4 Overview	7
1.5 Statement on Prior Publications	7
CHAPTER 2 BACKGROUND AND RELATED WORK	9
2.1 Mobile Apps	9
2.1.1 The Mobile App Design Process	9
2.1.2 Design Components of a Mobile App	10
2.1.3 The Structure of a Mobile App	11
2.2 Design Mining	12
2.3 Data Mining Mobile Apps	13
2.3.1 Mining Play Store Metadata	13
2.3.2 Mining App Packages	14
CHAPTER 3 INTERACTION MINING MOBILE APPS	16
3.1 Goals for Interaction Mining	16
3.2 ERICA: Interaction Mining for Android Apps	17
3.2.1 Design Choices	17
3.2.2 Implementation	19
3.2.3 Data Collection	23
3.3 Rico: Scaling Interaction Mining	23
3.3.1 Crowdsourced Exploration of Apps	25
3.3.2 Automated Exploration of Apps	28
3.3.3 Benefits of Hybrid App Approach	28
3.3.4 Data Collection	30
3.4 Discussion and Future Work	32

CHAPTER 4 REPRESENTATIONS AND MODELS OF MOBILE APP DESIGN	33
4.1 App Design Representations	33
4.1.1 Interaction Trace	33
4.1.2 Interaction Graph	35
4.2 The Rico App Design Dataset	37
4.2.1 Rico Dataset Contents	37
4.3 Identifying Unique UI Designs	38
4.4 Modeling UI Similarity	40
4.5 Discussion and Future Work	41
CHAPTER 5 DESIGN SEARCH	43
5.1 Finding App Design Examples in Practice	43
5.2 Flow Search	45
5.2.1 Identifying User Flows	45
5.2.2 Flow Search Interface	50
5.3 Example-Based UI Search	50
5.4 Discussion and Future Work	52
CHAPTER 6 ZERO-INTEGRATION PERFORMANCE TESTING	54
6.1 Background	54
6.2 Formative Study	55
6.3 The ZIPT Platform	56
6.3.1 Test Creation	58
6.3.2 Visualizations and Metrics	58
6.3.3 Implementation	60
6.4 Case Studies in Using ZIPT	61
6.4.1 Identifying Usability Issues	61
6.4.2 Analyzing Comparative Performance	64
6.5 Evaluation	72
6.5.1 Potential Uses Cases	73
6.5.2 Perceived Benefits	74
6.5.3 Generated Design Insights	74
6.6 Discussion and Future Work	75
CHAPTER 7 CONCLUSION	77
7.1 Future Directions	78
7.1.1 Scaling Up Interaction Mining	78
7.1.2 Interaction Mining in Other Interactive Domains	79
7.1.3 Unified Design Exploration Platform	79
7.1.4 Testbed for Studying Factors that Affect Usability	79
7.1.5 Additional Data-Driven Mobile App Design Applications	80
7.1.6 Assessing Data-Driven Design Tools	83

7.1.7 Non-Professional Use Cases	83
REFERENCES	84

LIST OF TABLES

3.1	The Android apps used in our evaluation of coverage benefits of hybrid exploration	29
4.1	Comparison of the Rico dataset with other app design datasets	36
6.1	Participant feedback about the overall usefulness of ZIPT and its features.	73
7.1	The five classes of design applications that the Rico dataset supports, correlated with the parts of the dataset intended to support them.	80

LIST OF FIGURES

3.1	ERICA’s architecture	20
3.2	ERICA’s data collection infrastructure	21
3.3	Coverage of human app exploration for three popular Android apps	24
3.4	Automated crawlers are often stymied by UIs that require complex interaction sequences, such as the three shown here.	25
3.5	Rico’s crowdsourcing web interface	27
3.6	The performance of our hybrid exploration system compared to human and automated exploration alone, measured across ten diverse Android apps	29
3.7	Summary statistics of the Rico dataset	31
4.1	Visual representation of an interaction trace	34
4.2	Visual representation of an interaction graph	35
4.3	Rico dataset contents	37
4.4	Example UI screens that have the same design but different content	39
4.5	Overall approach for example-based searching	40
4.6	Procedure for training an autoencoder to learn UI layout similarity	41
5.1	Example login flows in two online app design repositories	44
5.2	Overall approach for detecting user flow examples in user interaction traces	47
5.3	User flow examples detected using element-based detectors	48
5.4	User flow examples detected using layout-based detectors	49
5.5	Search interface for finding and visualizing example flows found in the ERICA dataset	51
5.6	Results from examples-based querying of the Rico dataset	53
6.1	An overview of ZIPT’s workflows	57
6.2	Usability issue with the search bar in a transit app	62
6.3	Usability issue in adding items to a list in a food logging app	63
6.4	Usability issue with the scoped searching in the Foursquare app	65

6.5	Usability issue with the visual search icon in Pinterest	66
6.6	User flows for the store locator features in the Macy's and Best Buy apps	67
6.7	Comparative evaluation of the store locator features in Macy's vs Best Buy	68
6.8	Comparative evaluation of adding events in two calendar- ing apps	70
6.9	Comparative evaluation of adding songs to a new playlist in Spotify vs YouTube Music	71
7.1	Common locations of seven popular mobile UI elements	81
7.2	Generating Android UI source code from captured UI data	82

CHAPTER 1

INTRODUCTION

In today’s crowded app marketplaces, good design can be a key differentiator for apps [1], impacting their adoption and ultimately their success. App design, however, remains a complex process comprising multiple design activities: researchers, designers, and developers must all work together to identify user needs, create user flows (user experience design), determine the proper layout of user interface (UI) elements (UI design), and define their visual (visual design) and interactive (interaction design) properties [2]. To create durable and engaging applications, app builders must consider hundreds of solutions from a vast space of design possibilities, prototype the most promising ones, and evaluate their effectiveness heuristically and through user testing.

To aid designers in navigating such complex design processes, several recent works have proposed leveraging data-driven design tools in domains such as web design, fashion design, 3D modeling and graphic design. Such tools can assist designers throughout the different stages of the design process – helping them search for examples for design inspiration [3, 4], understand successful patterns and trends [3, 5], generate new designs [6, 7], and evaluate design alternatives [8].

Such data-driven design tools, however, remain largely unavailable to mobile app designers. This is primarily because of the lack of suitable design datasets to power such tools. While open design repositories (such as Thingiverse [9] for 3D models) exist for some domains, no such large-scale design repository exists for mobile apps. Similarly, while designs for webpages can be programmatically mined at scale [3], mobile app designs remain siloed in closed-sourced apps, making them hard to access. Without scalable methods to capture and suitably expose the designs of existing apps, the design knowledge embodied in them remains unutilized – an unfortunate situation given the many challenges in the app design process such data could help

overcome.

This dissertation develops a technique, interaction mining, that allows capturing and exposing the designs contained in existing apps. It involves using people to understand and interact with UIs, and machines to capture design and interaction data seamlessly in the background. It demonstrates how interaction mining can be implemented at scale to mine design repositories using thousands of existing Android apps and how such repositories can support training machine-learning models that enable novel design interactions.

1.1 Challenges in the Current App Design Process

Early on in the design process, designers generate ideas on possible solutions to a design problem. They often look at relevant examples from existing apps. These examples serve as sources of inspiration and help understand the landscape of possible solutions [10, 11]. For instance, a designer working on a search interaction for a news app might want to broadly understand the different ways search is implemented in news apps or even other types of apps. However, there is no efficient way to find such examples at present. The designer would either depend on their memory and try to recollect search interactions that they have seen in the past or they would ask a colleague. However, there are only so many examples that a person can commit to memory and retrieve at will. Even if they ended up getting a few good example suggestions, they would have to manually download the apps, interact with them and capture some of their screens. Only then would they be able to bring these examples into the design process. This is time consuming and laborious.

Another complication that arises is that even if a designer is able to find relevant examples, there is no easy way to understand how these examples perform under different scenarios or how they compare with one another. Perhaps the designer sees a scoped search interaction [12] in a popular app and decides to use that but might not realize that such an interaction works well only if the search scope is very clearly communicated to a user (an example app that violates this principle is shown in Figure 6.4). Usability tests could help understand such nuances, but running them takes a lot of time and effort, making them unattractive for testing examples. Thus,

early on, decisions about which examples to draw from are made based on guesswork and intuition. This is unfortunate given that the decisions being made at this stage are generally large in scope and mistakes made in these stages are harder to rectify in later stages.

1.2 Designing with Data-Driven Tools

This dissertation demonstrates the possibility of a future where data-driven design tools help overcome challenges in the design process. It demonstrates two such tools that leverage the data that interaction mining produces. The first is a design search engine that designers can use to find examples. It indexes designs from a large number of existing apps and allows keyword-based and example-based queries over them. Designers can visualize the different design components of the returned results, such as user flows, UIs, and animations.

The second application is zero-integration performance testing (ZIPT), a novel approach to performance testing app designs. It enables designers to understand the performance characteristics of different examples that they might be considering. It also can be used for comparative testing of one’s apps against those of others with significantly lower cost and effort than existing testing methods. Below we present two example of how these tools can help designers learn from the designs of others.

1.2.1 Food Logging in a Diabetes App

Suppose Jane is prototyping a food logging flow for a diabetes management app and wants to understand the space of possible designs. First, Jane must find a set of relevant apps to analyze. Using the search engine, Jane searches for “diabetes” and finds a number of existing diabetes management apps such as *mySugr* and *Diabetes:M*. Since the search interface also allows her to run queries with semantic keywords that describe the tasks supported by an app, Jane also searches for “food logging” and finds food journaling apps such as *Lose It!* and *MyPlate* which also contain food logging flows. Thus designers can find apps in distal categories that contain relevant design features.

Based on her search results, Jane creates a comparison set containing both

diabetes management and food journaling apps to measure user performance on their food logging flows. She defines a scripted usability test, where she instructs crowd workers to log the last meal they ate using the app they are assigned from the comparison set. Additionally, she specifies a set of open-ended survey questions to elicit feedback around the task experience.

After defining the tests, Jane deploys them on ZIPT. It recruits crowd workers from Amazon Mechanical Turk and allows them to complete the tasks on the different apps. As the crowd workers perform their chosen tasks, the platform captures design and interaction data streams in the background, and processes them into a multi-modal representation of user traces.

Once the tests are complete, Jane can review aggregate metrics and visualizations over the collected user traces, as well as the collated qualitative feedback. The performance results reveal that the meal logging flows found in the food journaling apps have statistically higher completion rates than those in diabetes management apps. Jane uses the platform’s interactive flow visualization to quickly identify and inspect interaction traces where users could not finish the task. Jane finds that many of these users had a hard time initiating the food logging flow in the diabetes apps, and notices that these medically oriented apps have information dense UIs. In contrast, Jane observes that the meal logging UIs in the food journaling apps are more minimal, although the flows themselves involve more steps. Based on this data, Jane decides to start her design process with a food logging flow that has simple UIs modeled after the ones in the food journaling apps.

1.2.2 Playlist Interactions in a Music App

Suppose Susan is designing the functionality to add songs to a playlist in a music app. She searches for a “Add to playlist” flows in the search engine and reviews how users can do so in popular apps such as *YouTube Music* and *Spotify*. She notices that *Spotify* allows users to add a song to a new playlist in two ways. Users can “add to playlist” on a song and then specify that it is going to be part of a new playlist. Alternatively, they can “create new playlist” from a screen for managing playlists, and then navigate to a song and add it. *YouTube Music*, she notices, does not support the second use case.

She guesses that perhaps the second use case is not something that many users use. Although in the past she would have gone ahead with *YouTube Music's* simpler design, she decides to run a quick test on ZIPT to see if this assumption is correct. She scripts a test where she asks users to add two songs to a new playlist and collects data from 50 users for both *Spotify* and *YouTube Music*.

She discovers from the aggregate data that *YouTube Music* has a 80% completion rate versus 94% for that of *Spotify*. On inspecting the qualitative feedback, she discovers that several users on *YouTube Music* were confused by being unable to create empty playlists. One individual trace revealed that a crowd worker even used the help functionality of the app to learn how to create playlists. Based on this data, Susan decides to go the *Spotify* route and support both ways of adding songs to a new playlist.

1.3 Contributions

This dissertation advances knowledge of how to capture designs in an interactive domain and what applications such design data can enable. Specifically, it makes the following contributions:

- **Design mining for interactive domains.** Design mining in prior work has remained confined to non-interactive designs domains such as graphic designs, 3D models, or webpages (treated as static designs). This dissertation develops an approach, *interaction mining*, that brings design mining to an interactive domain – mobile apps – where designs change with user interactions. It demonstrates that by capturing and combining three types of data – a visual representation of the UI state, a structural representation of the UI state, and user interactions – it is possible to capture, analyze, and visualize the design of mobile apps.
- **An architecture for scalable interaction mining.** This dissertation presents a web-based architecture for implementing interaction mining that promotes scalability in two ways. First, it requires no modification to an app's source code, which makes it possible to mine any Android app. Second, using a web-based app usage model enables crowdsourcing human exploration of apps over the Internet, making it

possible to mine a large number of apps. Using this architecture, we collected and released a dataset that contains app designs from over 9700 Android apps, making it the largest dataset of its kind.

- **Hybrid app exploration for higher coverage.** Prior work has used either human or automated approaches for interacting with app UIs to explore their different states. This dissertation discusses the limitations of these two approaches and demonstrates that higher coverage of app states is possible by combining the two approaches.
- **Automated identification of semantic user flows.** Although user flow examples play an important role in the app design process, existing repositories of user flows do not scale because of manual curation. This dissertation develops a machine-learning-based approach that identifies user flow examples in interaction traces, and uses it to develop the largest user flow search engine that contains over 3000 examples of user flows in Android apps.
- **Example-based searching for mobile UIs.** This dissertation presents an unsupervised approach for training a deep-learning model to capture similarity between mobile UI layouts. This enables example-based search interactions over UI datasets.
- **Performance testing of examples.** This dissertation brings design testing earlier into the design process. The zero-integration performance testing approach presented here enables designers to run tests on examples of interest in the ideation phase. This is in contrast to the current practice of testing for the first time after having invested the time and effort in building prototypes.
- **Comparative testing with low cost and effort.** Zero-integration performance testing can also be used in the later stages of the design process to compare the performance of designs in one's apps to those in others. Used in this manner, it allows comparative testing at a significantly lower cost and effort compared to techniques that are currently available. For example, it costs 100× less than using a video-based remote usability testing platform such as <http://www.usertesting.com>.

1.4 Overview

This dissertation is divided into seven chapters, setting out the origins of design mining in the literature, motivating the development of platforms for interaction mining mobile apps, describing two applications that interaction mining enables, and sketching a future for how data-driven design tools enabled by interaction mining can transform the creative design process of mobile app design.

Chapter 2 provides an overview of the history of design mining in other domains, describes the technologies and components that constitute the design of a mobile app, presents the different use cases and techniques for mining mobile apps, and discusses the challenges in holistically capturing the *design* of a mobile app.

Chapter 3 introduces interaction mining as a technique for capturing the design of mobile apps and discusses its design goals. It presents the design choices and implementation details of two interaction mining systems, ERICA¹ and Rico, developed for mining Android apps.

Chapter 4 describes two representations of mobile app designs that are captured by our interaction mining systems and describes how these representations can be used to train models that enable real-time design interactions. It also presents the details of the Rico mobile app design dataset that we publicly released.

The next two chapters introduce applications built using interaction mining data. Chapter 5 presents search engines for mobile app designs that support keyword- and example-based search interactions. Chapter 6 presents the motivation and design rationale behind our system for zero-integration performance testing, ZIPT, and describes how it enables designers to learn from designs in the vast array of existing apps.

Finally, Chapter 7 sketches directions for future work.

1.5 Statement on Prior Publications

Materials presented in this dissertation have also appeared in three prior conference publications. The technique of interaction mining and the flow search

¹ERICA is an acronym for **E**nabling **R**ealtime **I**nteraction **C**apture for **A**ndroid apps.

interface was published at the ACM User Interface Software and Technology (UIST) Conference in 2016 [13]. The work on scaling interaction mining, the Rico dataset, and our approach to example-based searching over UIs was published at UIST 2017 [14]. ZIPT was also published at UIST 2017 [15].

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter provides a broad overview of the technologies related to the design and implementation of mobile apps. It also presents a survey of prior work in design mining, summarizes prior work in mining mobile apps, and discusses the challenges in capturing the design of mobile apps.

2.1 Mobile Apps

Mobile apps are self-contained applications that run on devices equipped with a mobile operating system such as iOS [16] or Android [17]. Consumers can generally download apps to their devices from online app stores for their platforms. In recent years, apps have seen increasing popularity with consumers. Millions of apps are available in app stores for iOS and Android with over 90 billion app downloads and nearly 900 billion hours of app usage in 2016 alone [18].

2.1.1 The Mobile App Design Process

Given the sheer number of apps available today, it is very common for multiple apps to exist that help users accomplish the same task. This has made it challenging for apps to stand out amongst other similar apps. Increasingly the user experience afforded by app has become a key criterion for their adoption and ultimately their success [1].

Designing compelling user experiences for mobile apps, however, still remains a complicated, time-consuming process. Popular apps are often designed by teams of multiple professionals including designers, user experience (UX) researchers, and engineers. The process generally starts with a design goal such as the addition of a new feature. From there on, app creation hap-

pens in four stages. In the first stage, designers generate ideas on potential solutions to the design problem. At this stage, they often look for examples of similar features in other apps for design inspiration. Often the generated ideas are combined, remixed, and iterated upon as well.

A few of the ideas generated in the ideation phase are prototyped. For mobile apps, this could involve low-fidelity prototypes such as paper prototypes or high-fidelity interactive, digital prototypes built using tools such as Invision [19], Origami [20], or Pixate [21]. To understand their effectiveness, these prototypes are tested using various methods such as heuristic evaluation [22], usability testing, or performance testing.

Once a design has been chosen based on the test results for prototypes, it enters the implementation stage where developers get involved. They help translate the design mock-ups to actual code and realize the vision of the designers for the app. Once implemented, the feature might go through another round of usability and performance testing to ensure that it is effective in an end-to-end usage scenario. This is important since many components of an app cannot be tested with a prototype. For example, although a particular search results page might seem to work as well as a prototype, on using it with real data a designer might discover that the screen space allocated for each result is not large enough. Or they might discover that the usability is poor, not because of a poor UI, but because the search backend returns results with unacceptable delays.

Once such implementation kinks are worked out and the feature has been sufficiently tested, it is released to end users. At this stage, app creators often collect usage metrics to monitor if the feature works for users as expected. If they have enough users, they also employ A/B testing techniques to test design variations and further refine the feature.

2.1.2 Design Components of a Mobile App

Throughout the design process outlined in the previous section, designers develop multiple components of an app's design including user flows, user interfaces (UIs), visual details, and interaction details [2]. Different designers usually specialize in designing one or a few of these components. First, for each task an app supports, UX designers perform user research and try to

understand the logical steps to accomplish the task. This is captured as a *user flow* – a sequence of UI screens and corresponding user actions that let users perform that task. An example user flow could be for *onboarding* a new user in a finance app by helping connect all of their bank accounts to the app.

Once the UX designer has laid out the paths that they intend users to take in an app, UI designers then design the individual UI screens of the app. For each UI, they determine the elements to be included and their overall layout. They are also responsible for ensuring that the UIs visually communicate the right paths and that a consistent design language is applied across all UIs.

Once UIs have been developed, visual designers (also referred to as graphic designers) decide the visual details of the UIs and their elements. This includes deciding things like color palettes and typography. They often also create the final pixel-perfect assets (such as icons) to be used in implementing the app.

Finally, motion designers create the dynamic interactions in an app. This could include designing how UI elements animate in response to user interactions or how entire screens transition from one to another. Common examples of animations include menus sliding out or spinner icons when a UI is waiting for data.

2.1.3 The Structure of a Mobile App

Most mobile apps are distributed as closed-sourced packages. This is true for the two most popular mobile app platforms: iOS and Android. The implementations developed in this dissertation focus on the Android platform because of the extensive use it has seen in the research community due to its open-source nature.

Android apps are written using languages such as Java, C++, and Kotlin [23]. The code is compiled using the Android SDK (software development kit) tools along with any data and resource files into an APK (Android package), which is an archive file. A single APK file contains all the contents of an Android app and is the file that users download from app marketplaces (such as the Google Play Store) and install on their devices.

Android apps consist of four main programmatic components: activities,

services, content providers, and broadcast receivers [23]. Activities are used to build app UIs, whereas services run in the background without any associated UIs. Content providers interface with app data sources (such as a database) and broadcast receivers are used to respond to system-wide broadcast messages (such as notifications). Of these components, users interact directly with only the activities. Apps typically have multiple activities with a few “main” activities that are ones in which an app can be started. An activity can start other activities as well. This process often involves passing asynchronous messages called *intents* which contain any necessary data.

An activity’s user interface is composed of UI elements or *views*. Views can respond to user inputs and are organized in a hierarchy called the *view hierarchy*. Developers can declare the layout of these user interfaces using human readable XML layout files that get included in the APK. In addition, the layouts can also be defined in the code, allowing for programmatically construction of UIs at runtime. The APK also includes important metadata about an app in the “manifest” file. It specifies all the resources in an app, including the name of its activities, and, in particular, defines its main activities. It is relatively straightforward to unpack an APK file and inspect the manifest as well as the XML layout files using open source tools [24]. As a result, this is what researchers generally do when they want to mine some of the static design data present in public Android apps.

2.2 Design Mining

Design mining, as proposed by Kumar et al. [3], refers to mining the design details of artifacts in a domain such that traditional data-mining and knowledge-discovery techniques can be readily used on that data. They demonstrated design mining for webpages using a platform, Webzeitgeist, that captured and combined the visual (screenshots) and structural representations (document object model or DOM trees [25]) of webpages. The resultant data enabled them to answer questions about design demographics, automate design curation, and build data-driven design tools, such as tools for example-based retrieval of designs.

This notion of using automated knowledge-discovery techniques on the mined data is what makes design mining different from traditional ways of

collecting and curating designs. Graphic designs, for instance, are collected and curated by designers in the form of images. While images are useful for visualizing the designs and even testing for their effectiveness with user studies [8], analyzing designs with automated techniques using just images is hard. As a result, for design webpage designs, Kumar et al. advocated capturing not just screenshots but also their DOM trees. Processing these two data streams together allowed Webzeitgeist to easily surface relevant design data (such as popular colors or layouts) to the data-driven design applications that it supported.

This dissertation extends this line of research to mobile apps. Design mining mobile apps, however, presents a new challenge. Unlike artifacts in other design domains, such as graphic designs, infographics, 3D models, or even webpages (as explored by Kumar et al.) that are static (do not change over time), app designs contain dynamic components such as user flows and animations. These dynamic components are central to the design of an app, but capturing them is hard since they only manifest themselves during runtime when a user interacts with an app. The technique of interaction mining that we present in the next chapter, answers the question: How can we capture dynamic designs such as those of mobile apps?

2.3 Data Mining Mobile Apps

In this section we discuss prior works from multiple domains on extracting useful knowledge from mobile apps. These works primarily mine information from two sources: the app stores (such as the Google Play Store) and the mobile app packages (such as the Android APK files).

2.3.1 Mining Play Store Metadata

The Google Play Store contains a wealth of information about any app. An app's webpage includes metadata about the popularity of the app (such as the number of downloads), feedback from users in the form of 5-star ratings and text reviews, a category label, screenshots of a few UIs from the app, and often times a textual description of the features found in the app. As a result, researchers have mined data from the Google Play Store by crawling

its web pages for a variety of studies.

A common use of play store metadata is as a proxy for the quality of apps. An app’s quality is judged by users based on a variety of factors (such as animations [26], menus [27], and visual diversity and consistency between different screens [28, 29]) and hence cannot be as easily collected as say for graphic designs. Thus using Play Store metrics as a proxy for app quality becomes attractive. Many studies have used the number of downloads, average rating, or the number of ratings for an app as a proxy for the quality of an app [29, 30, 31]. Sometimes multiple metrics are also combined to overcome biases associated with using just one of them [30]. Another unique feature of app stores as compared to traditional software deployment mechanisms is the presence of the large amount of user feedback. Several studies have investigated the nature of this feedback [32, 33, 34, 35, 36]. Finally, app store metadata has also been used to study app development and release practices such as frequency of updates [37] or the dialogue between app creators and consumers [38, 39].

2.3.2 Mining App Packages

Several prior works have mined app packages (such as APK files for Android apps) for design data. They follow one of two approaches – static or dynamic. A static approach involves unpacking the package file using open-source tools [24] and inspecting static XML files that describe the layouts of different UIs. This approach has been used to study design pattern changes over time [40], identify frequently-used UI components [41], study the complexity of app UIs [30], and understand characteristics of highly rated apps [42]. The main limitation of this approach is that it can only capture UI information that has been specified declaratively and misses UIs that are specified programmatically at runtime. In addition, since this approach does not run an app, it cannot capture components of an app’s design that are dynamic, such as user flows or animations.

Dynamic approaches for mining mobile apps, conversely, capture data at runtime. For instance, Li et al. [43] used this approach to capture UI information of Android apps at runtime and leveraged the text data found in the UI layouts to enable text-based searching for Android components. Dy-

dynamic mining methods can be difficult to implement due to the challenges associated with driving an app’s graphical UI to reach new states [44], and hence have found limited usage in design mining apps. They have nonetheless been deployed for testing [45], detecting security vulnerabilities [46, 47], and modeling web applications [48]. The main limitation of these approaches, however, is that they fail to explore app UIs in a manner that mirrors human usage. This makes it difficult to extract meaningful user flows from the captured data — an important component of an app’s design. Interaction mining, as developed in this dissertation, overcomes these limitations by seamlessly incorporating humans into the app mining process, which allows them to drive the graphical UI and demonstrate meaningful flows.

CHAPTER 3

INTERACTION MINING MOBILE APPS

Data-driven tools can help designers in the complex app design process. Building such tools for mobile app design requires a way to capture the design of apps. This chapter introduces interaction mining for mobile apps. Interaction mining captures and analyzes both static (UI layouts, visual details) and dynamic (user flows, motion details) components of an application’s design. Mined at scale, the data produced by interaction mining enables tools that scaffold the app design process: finding examples for design inspiration, understanding successful patterns and trends, generating new designs, and evaluating alternatives. This chapter also presents the architecture and design details of systems for large-scale interaction mining of Android apps that enabled mining close to 10,000 apps. This large-scale mining enabled creating a design repository with 72,000 unique UIs and 3 million design elements.

3.1 Goals for Interaction Mining

The aim of interaction mining is to build a design repository and a set of data structures to support design tools that scaffold the entire mobile app design process. Accordingly, an interaction mining system must possess the following capabilities:

Capture of multiple design components. An app’s design consists of multiple components: how users move from one UI to another to complete tasks, how UIs are laid out, the visual details of the UIs and their elements, and how each UI responds to user interactions. Supporting data-driven design tools for app design requires capturing each of these components. This capture is challenging because it requires collecting, combining, and correlating multiple types of data, including both visual and structural representations

of UIs and their elements, as well as user interaction details.

Coverage of important UI states. In order for the design data produced by interaction mining to help designers understand the experiences an app affords, it must capture UI states that are frequently used by humans. Discovering and navigating to each of these states in an automated way is difficult, and requires an understanding of how to interact with different UIs. Additionally, states must be visited in a natural, semantic order to be able to capture meaningful user flows.

Scalability. The utility of interaction mining hinges on the scale of the corpus it can build. Larger repositories improve both diversity of results in example-based applications as well as the accuracy of machine-learning models trained on the mined data. Mining strategies that require access to an app’s source code or the cooperation of an app’s developer, therefore, are less useful than those that take a black-box approach to working with the millions of extant mobile apps.

3.2 ERICA: Interaction Mining for Android Apps

ERICA is a system that enables interaction mining for Android apps in a black-box manner. It combines the strengths of humans and machines, using humans to drive app interactions and computers to capture design and interaction data.

3.2.1 Design Choices

Dynamic Mining

Mobile design data can be mined in two ways: statically by unpacking app packages and dynamically by running an app. ERICA uses the dynamic approach since a static approach cannot capture dynamic design components such as user flows and motion details.

Human-Powered Exploration

To generate meaningful interactions for different UIs, ERICA leverages humans to interact with an app while it captures their interactions and the resultant UIs. This is in contrast to using an automated agent to interact with the UI elements programmatically (used for example in DynoDroid [45]). ERICA employs a human-powered approach instead of an automated one for several reasons.

First, human interactions are more likely to produce realistic user flows. The space of possible interaction sequences in apps is large, and only a small fraction of such sequences represent semantically-meaningful flows. It is challenging for automated methods to discern these semantically-meaningful flows. Humans, on the other hand, naturally use apps to perform meaningful tasks, and hence data mined during human usage often contain meaningful user flows.

Second, many apps require user input (e.g., usernames, passwords, queries, content, etc.) before meaningful interaction can occur. Automatically creating valid user data across a large number of apps is a challenging problem. In addition, many apps need to be populated with user-generated data before their UIs reflect the full set of possible interactions and features. For example, many screens in the *Evernote* app (useful for note-taking) have limited interactions until a user creates folders and some notes. With an automated approach, it is challenging to create such data across a large number of diverse apps.

Third, for captured UI data to be meaningful, a new UI state should only be captured after the UI has been completely updated in response to a user interaction. Automatically detecting the completion of UI updates is another challenging problem, since updates can happen asynchronously in response to external events (for example, when data from a remote server is received). Humans, however, are capable of detecting the completion of UI updates with relative ease.

Web-Based Interaction Interface

ERICA users interact with Android apps through a web-based interface, as opposed to directly accessing the devices on which the apps run. There

are two reasons for this design decision. First, having a web-based interface makes it possible to use apps remotely, making it easier to crowdsource data collection over the Internet. In addition, a browser-based architecture lowers the barrier for users to use the system. Users do not need to use a specific device or install any software on their devices. Second, the servers that power ERICA have much greater compute capability than a phone or tablet, allowing interaction data to be captured in the background without negatively affecting user experience.

Although users interact with apps through a web interface, ERICA still offers a near-native experience for users. ERICA is powered by a responsive web frontend which, when used in fullscreen mode on a mobile browser, mirrors the screen of the device on which the app is running (see Figure 3.2).

Physical Devices

ERICA runs Android apps on physical phones and tablets connected to the ERICA server. Physical devices offer more predictable UI performance and support a wider variety of features (such as OpenGL 2.0) across a larger number of apps, compared to the best Android emulators available today.

3.2.2 Implementation

ERICA comprises four components: an app crawler, a data collector, a post-processor, and a web API (Figure 3.1). The app crawler obtains APK files for Android apps from the Google Play Store. The data collector captures the state of an app’s UI as users interact with it. The post-processor combines the captured data streams into a single unified representation – an *interaction trace* (details of this representation are presented in Section 4.1.1). It also computes visual and textual features for UI elements and stores them in a database for subsequent querying. Client applications access ERICA’s data through a web API.

App Crawler

The app crawler uses a Google Play Store API (application programming interface) to download Android APK files for apps, along with play store

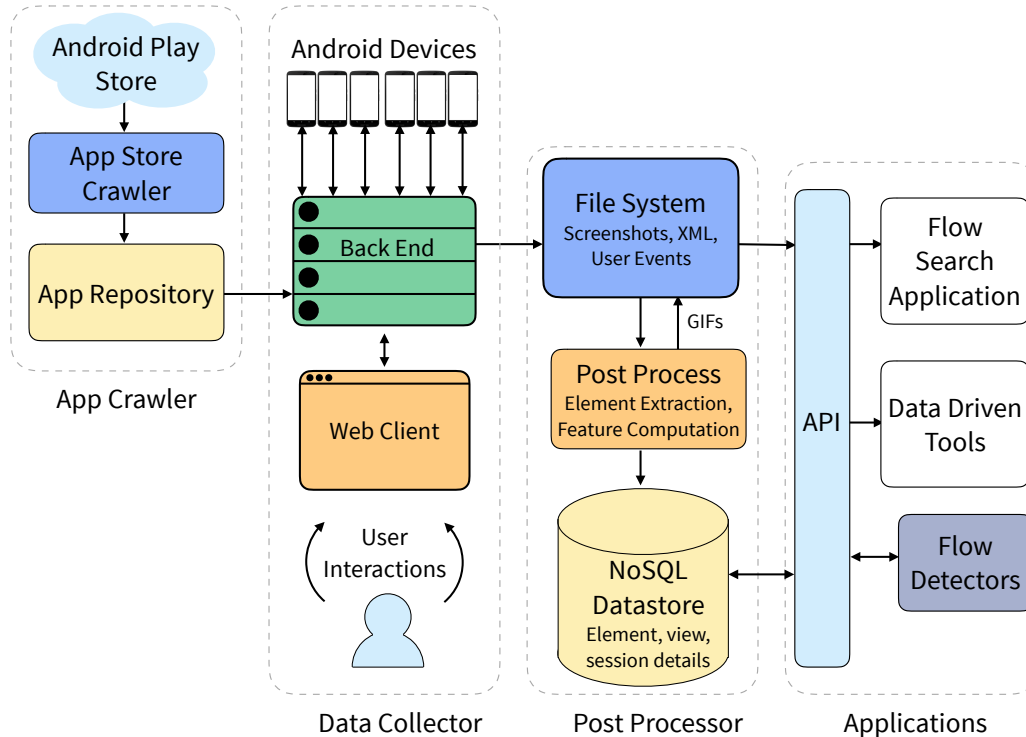


Figure 3.1: ERICA’s architecture. Apps from the Google Play Store are run on physical Android devices. Users interact with these apps through a web interface that allows the system to capture their interactions as well as the state of the app’s UI. Post-processing scripts compute structural properties of elements, and UIs and store them in a NoSQL datastore. Client applications access data through a web API.

metadata such as app rating, number of downloads, and app category.

Data Collector

The data collector allows users to interact with Android apps through a web interface and captures three kinds of data in realtime: screenshots (visual representation of UIs), view hierarchies (structural representation of UIs), and user interactions. The collector comprises three components: a set of Android devices, a backend server, and a web client (Figure 3.2). Each Android device is a physical phone connected to the backend server via USB, and each one runs a modified version of Android OS.

The backend manages the devices and installs Android apps on them. It sends the device’s UI as compressed JPEG images to the web client using a WebSocket connection. It also relays user inputs from the web client to the

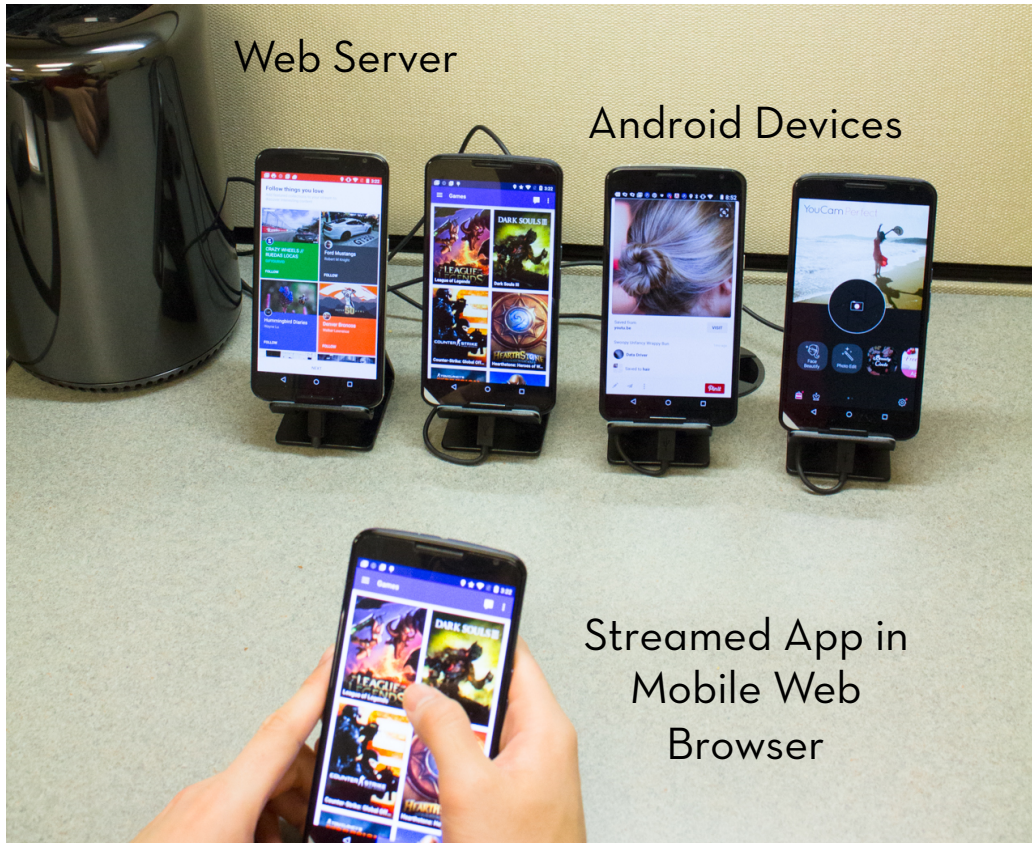


Figure 3.2: ERICA’s data collection infrastructure. Here ERICA’s web client is being used in a browser on a phone in fullscreen mode to interact with an app running on one of the other Android devices connected to the server (second from the left). ERICA collects information about the app in realtime while the user is interacting with it.

device. Finally, it saves the captured data to the local file system.

The web client displays images of the app’s UI in a browser and allows users to interact with it. It has a responsive layout and can be used on devices of varying form factors, including desktops and mobile devices. On a mobile device, with the browser in full-screen mode, it offers a near-native app experience (Figure 3.2 shows one of the device screens being mirrored in the user’s browser). Both the backend server and the web client are implemented on top of the `OpenSTF` framework [49]. The `minitouch` library allows the web client to support all common multitouch gestures when used on touchscreen devices, and two-finger pinch, rotate, and zoom on regular PCs [50].

Implementing the data collector required overcoming two technical challenges:

Capturing UI Data with Low Latency. Capturing UI data in an app without any perceptible user delay is non-trivial. To capture UI changes in response to user interactions, ERICA requests the structured representation of an app’s UI every time a user interacts with it. Android’s default `UiAutomator` service, which is commonly used to request view hierarchies from Android devices, is poorly suited for this purpose as it takes multiple seconds to return the data [51]. Instead, we modified the Android OS to implement a custom service that responds within about a hundred milliseconds. This service was built by augmenting the Android classes that define UIs and UI elements with custom methods that dump properties of their instances at runtime. The service returns a hierarchical representation of the UI in XML format and has properties (such as position and size) for all the elements on the screen. The service also allows capturing additional properties of UI elements that are not available through the `UiAutomator` service, such as font family and size information for text elements.

Capturing screenshots of a UI multiple times a second enables a smooth user experience when interacting with apps through the web frontend. It also enables high-fidelity viewing of the motion details of the app (animations and transitions) from the captured data. ERICA uses the `minicap` library [52] to enable the capture of more than ten screenshots per second, configured to trigger whenever pixels on the device screen change.

Inferring Gestures. When a user interacts with an app’s UI through the web client, the client captures user input events (such as `TouchUp`, `TouchMove`, and `TouchDown`) and their coordinates. These events are relayed to the Android device via the server. Android translates these low-level input events to gestures (such as clicks, scrolls, longtaps, etc.), and, if applicable, dispatches them to the appropriate element on the UI [53].

Accurately inferring high-level gestures from the low-level input events captured by the web client is a challenging task. Instead, ERICA captures gestures directly from the Android OS by modifying Android to output the type and target element of a gesture every time one is detected.

Post-Processor

The post-processor reconciles the three streams of captured data and combines them to form a unified representation called an *interaction trace* (Sec-

tion 4.1.1). It parses the structural representation of UI screens to identify individual UI elements, crops the elements from the screenshot, and saves them to the file system. Element properties (such as size, location, and contained text) are also saved to a NoSQL datastore for subsequent querying. Multiple screenshots are combined to form animated GIFs that show how a UI responds to different user interactions. A web API provides access to the data stored in the file system and in the NoSQL datastore.

3.2.3 Data Collection

We demonstrated the usefulness of ERICA by using it to collect user interaction traces from more than 1000 Android apps, and built a corpus by crawling the top 100 apps from 30 different categories on the Google Play Store. We recruited 28 participants for lab-based data collection through posters and email lists on campus, briefed them about the study, and gave them a short tutorial on ERICA’s web interface. Each participant was free to use each app totally unsupervised and without any time limit. To protect participants’ privacy, we instructed them to input fake personal data as necessary.

The participants completed 1186 sessions for 1150 apps. After rejecting sessions with less than 5 interactions, we were left with 1065 sessions for 1011 apps, consisting of more than 18,000 unique UI screens, 52,000 gestures, 500,000 interactive elements, and 6.7 million total elements. On average, sessions lasted 4 minutes (min 30 seconds, max 21 minutes) and had 51 interactions over 18 unique UI screens. Users employed 5 Android activities in each app on average, and performed 6.2 interactions in each activity (activities are components of Android apps that allow users to perform a specific task [54]).

3.3 Rico: Scaling Interaction Mining

ERICA was the first system to demonstrate the feasibility of interaction mining. It was, however, limited in two ways. First, it was designed for a lab-based setup where users were on the same network as the server. The effort required to recruit and manage users for lab-based crawling sessions

limited the scale at which we could operate it for app crawling. Second, ERICA achieved low coverage of an app’s UI states. One way to improve that is to combine multiple user traces. Figure 3.3 illustrates how coverage increased for three apps of varying complexity (measured by the number of Android activities found in their APK files) as multiple traces were combined. We observe that, as in heuristic evaluation, 5 to 8 users appear to provide optimal coverage [22]. For truly complex apps, coverage may remain low even after aggregating many user traces, since many UI states exist that humans do not visit during regular usage.

To enable interaction mining at a larger scale, we developed Rico, an interaction mining platform that extended the ERICA architecture in two ways. First, it was designed to work over the Internet, enabling us to hire paid workers from online crowd marketplaces for app crawling. Second, it combined human-powered and programmatic exploration for higher coverage. Humans rely on prior knowledge and contextual information to effortlessly interact with a diverse array of apps. Apps, however, can have hundreds of UI states, and human exploration clusters around common use cases. These factors result in human exploration achieving low coverage over UI states for many

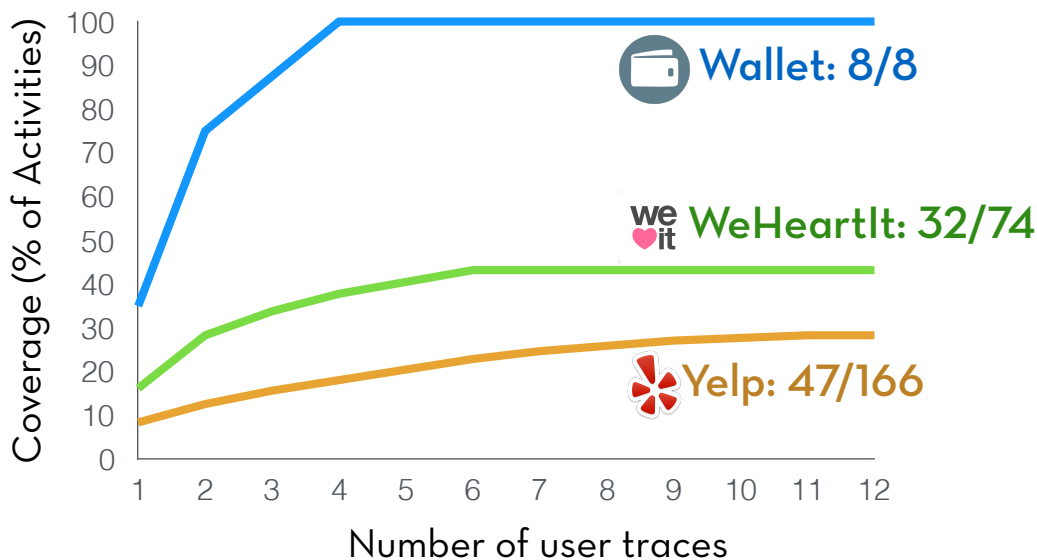


Figure 3.3: Coverage of activities in the app *Wallet* (top), *Weheartit* (middle), and *Yelp* (bottom) with respect to the number of users whose interaction traces were combined. Coverage does not increase significantly after 5 to 8 users since there is significant overlap between activities that different users visit.

apps [55, 13]. Automated agents, on the other hand, can be used to exhaustively process the interactive elements on a UI screen [47, 44]; however, they can be stymied by UIs that require complex interaction sequences or human inputs (Figure 3.4) [56].

Rico’s hybrid approach for design mining mobile apps combines the strengths of human-powered and programmatic exploration: leveraging humans to unlock app states that are hidden behind complex UIs, and using automated agents to exhaustively process the interactive elements on the uncovered screens to discover new states. The automated agents leverage a novel content-agnostic similarity heuristic (see Section 4.3 for details) to construct an *interaction graph* (Section 4.1.2) to efficiently explore the UI state space. Together, these approaches achieve higher coverage over an app’s UI states than what is possible using any one of them alone (Section 3.3.3).

3.3.1 Crowdsourced Exploration of Apps

For crowdsourced exploration, Rico uses a web-based architecture similar to that of ERICA. A crowd worker uses apps through a web application, which establishes a dedicated connection between the worker and a phone in our mobile device farm. The system loads an app on the phone, and starts continuously streaming images of the phone’s screen to the worker’s browser.

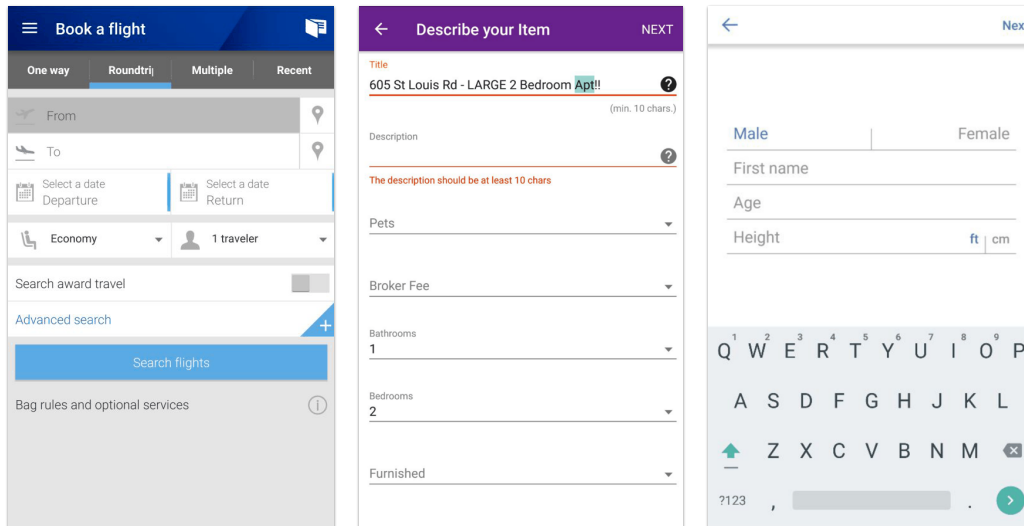


Figure 3.4: Automated crawlers are often stymied by UIs that require complex interaction sequences, such as the three shown here.

As the worker interacts with the screen on his browser, these interactions are sent back to the phone, which performs the interactions on the app.

Rico extends the ERICA architecture to enable large-scale crowdsourcing over the Internet. It adds a token-based authorization system that supports both short- and long-term engagement models. For micro-task-style crowdsourcing on platforms like Amazon Mechanical Turk, it generates URLs with tokens. When a worker clicks on a URL with a valid token, the system installs an app on a device and hands over control to the user for a limited time. To facilitate longer term engagements on platforms such as Upwork, Rico provides a separate interface through which workers can repeatedly request apps and use them. This interface is protected by a login wall, and each worker is provided separate login credentials. Compared to ERICA, we also lowered the bandwidth consumed by Rico by lowering the frame rate for screen updates and by using higher a compression ratio for the streamed JPEG images.

Rico’s view hierarchies capture the superclass names of all elements in the view hierarchy, something not present in the ERICA view hierarchies. Superclass names are useful since we discovered that developers frequently use custom derived class names such as `AppNameView` from `TextView`. Using superclass names, it is possible to correctly expose such an element as a text element to downstream applications.

We show the Rico web interface in Figure 3.5. On the left-hand side, crowdworkers can see the streamed phone screen. They can interact with it using their computer keyboard and mouse. The interface also supports relaunching the app they were supposed to use in case they exit the app by mistake. On the right-hand side, the interface provides usage instructions and features to overcome two operational challenges we faced in large-scale crowdsourcing. First, to minimize the capture of personally identifiable information, the interface provides a name, email address, location, and phone number for crowd workers to use when apps request such information. Second, the system is set up in a way that when apps send emails or text messages to the specified email addresses and phone numbers, crowd workers can access those messages through the web interface. This allows them to complete app verification steps with minimal effort.

Our use of crowdsourcing is different from prior work. Prior work has generally used crowdsourcing to collect labeled data for training models [57] or

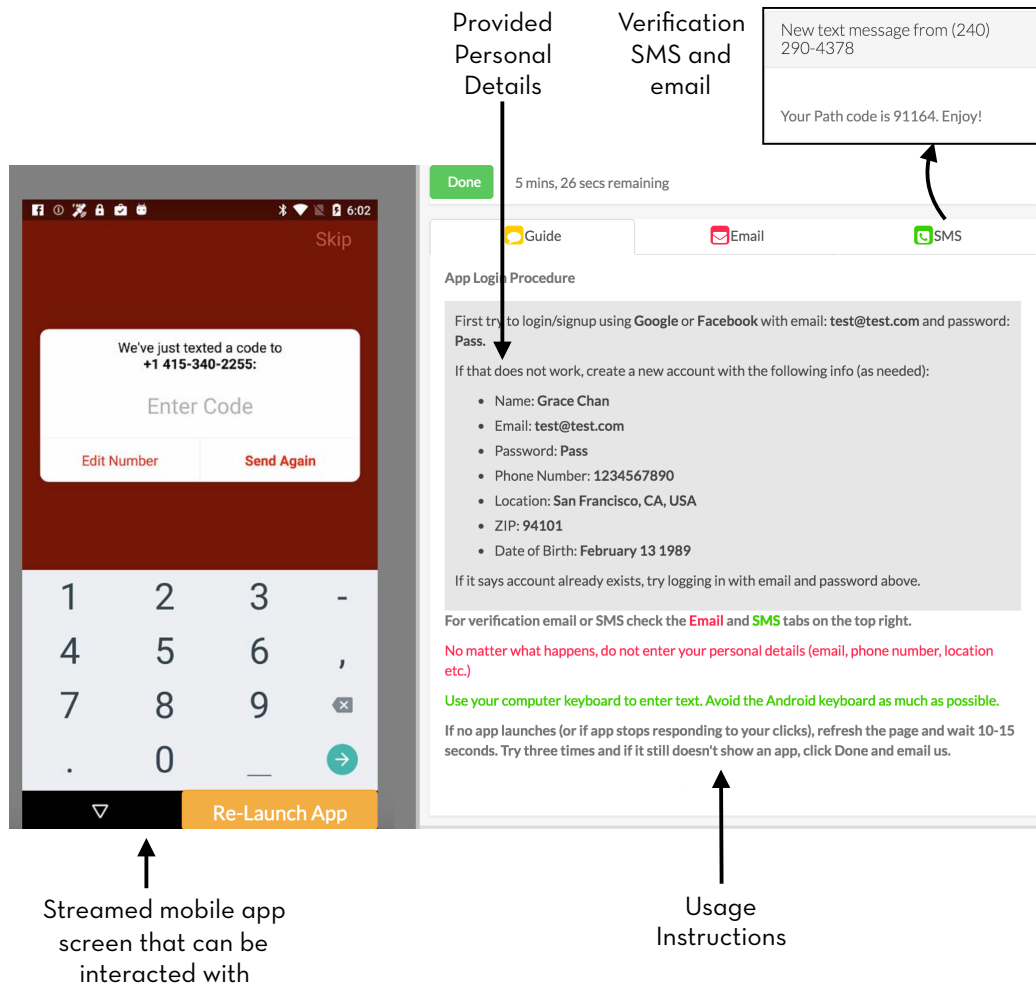


Figure 3.5: Rico's web interface. On the left, crowd workers can interact with the app screen using their keyboard and mouse. On the right, they are provided instructions and details such as the name, location, phone number, and email address to use in app. The interface also allows workers to access SMS (short message service) and email messages sent to the provided phone number and email to complete the app verification processes.

for performing evaluations through surveys or performance oriented tasks [8, 58]. In contrast, our work requires crowdworkers to use mobile apps without performing any labeling or annotations.

3.3.2 Automated Exploration of Apps

To move beyond the set of UI states uncovered by humans, Rico employs an automated exploration system. Existing automated crawlers hard code inputs for each app to unlock states hidden behind complex UIs [59, 56]. We achieve a similar result by leveraging the interaction data contained within the collected user traces: when the crawler encounters a interface requiring human input, it replays the interactions that a crowd worker performed on that screen to advance to the next UI state.












Similar to prior work [59, 56], the automated mining system uses a depth-first search strategy to crawl the state space of UIs in the app. For each unique UI, the crawler requests the view hierarchy to identify the set of interactive elements. The system programmatically interacts with these elements, creating an interaction graph that captures the unique UIs that have been visited as nodes, and the connections between interactive elements and their resultant screens as edges. This data structure also maintains a queue of unexplored interactions for each visited UI state. The system programmatically crawls an app until it hits a specified time budget or has exhaustively explored all interactions contained within the discovered UI states.

3.3.3 Benefits of Hybrid App Approach

To measure the coverage benefits of our hybrid exploration approach, we compare Rico’s crawling strategy to human and automated exploration alone. We selected 10 apps (Table 3.1) from the top 200 on the Google Play Store. Each app had an average rating higher than 4 stars (out of 5) and had been downloaded more than a million times. We recruited 5 participants for each app, and instructed them to use the app until they believed they had discovered all its features. We then ran the automated explorer on each app for three hours, after warming it up with the collected human traces.

Prior work [54, 55, 13] measured coverage using Android activities, a way of

Table 3.1: The Android apps used in our evaluation. Each had a rating higher than 4 stars (out of 5) and more than 1M downloads on the Google Play Store.

Name	Rating	Downloads	Description
 Polyvore	4.7	1 - 5M	Fashion social-network and marketplace
 Fabulous	4.6	1 - 5M	Goal-setting app
 Issuu	4.3	1 - 5M	Magazine browsing and collection
 Foursquare	4.1	10 - 50M	City guide and reviews
 Yelp	4.3	10 - 50M	Guide for local businesses
 Newsrepublic	4.4	10 - 50M	World news digest
 Etsy	4.3	10 - 50M	Homemade and Vintage goods marketplace
 Todoist	4.4	10 - 50M	To-do list and reminder
 WeHeartIt	4.5	10 - 50M	Photo-sharing social network
 Weather Channel	4.3	50 - 100M	Weather tracker
 Evernote	4.6	100 - 500M	Note-taking app for collaboration

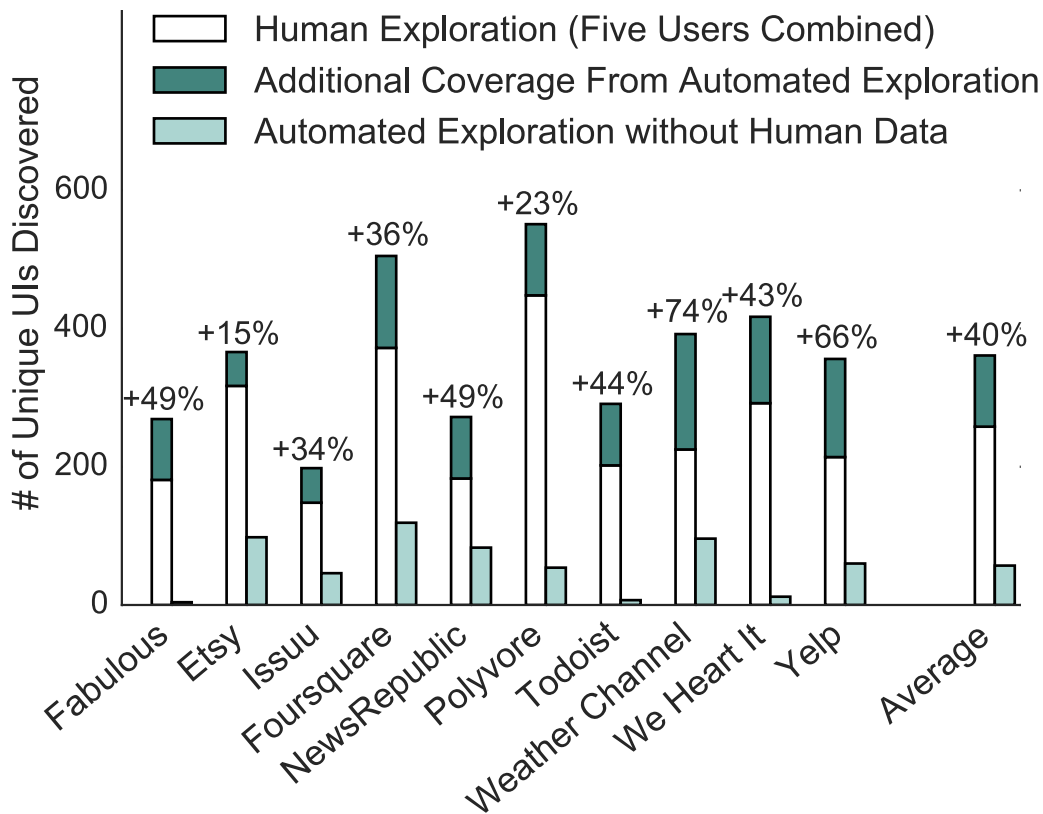


Figure 3.6: The performance of our hybrid exploration system compared to human and automated exploration alone, measured across ten diverse Android apps

organizing an Android app’s codebase that can comprise multiple UI screens. While activities are a useful way of statically analyzing an Android app, developers do not use them consistently: in practice, complex apps can have the same number of activities as simple apps. In contrast, we use a coverage measure that correlates with app complexity: computing coverage as the number of unique UIs discovered under the similarity heuristic (Section 4.3).

Figure 3.6 presents the coverage benefits of a hybrid system, combining human and automated exploration increases UI coverage by an average of 40% over human exploration alone. For each app, hybrid exploration reached Android activities that weren’t reached by human exploration. For example, on the Etsy app, the hybrid system discovered screens from 7 additional activities beyond the 18 discovered by human exploration.

We also evaluated the coverage of the automated system in isolation, without bootstrapping it with a human trace. The automated system achieved 26% lower coverage across the tested apps than Rico’s hybrid approach. This poor performance is largely attributable to the gated experiences that pure, automated approaches cannot handle. For instance, Todoist and WeHeartIt hide most of their features behind a login wall.

3.3.4 Data Collection

We used Rico to mine 9,772 Android apps spanning 27 categories. The resulting dataset comprises 10,811 user interaction traces and 72,219 unique UIs (Figure 3.7). We excluded from our crawl categories that primarily involve multimedia (such as video players and photo editors) as well productivity and personalization apps. Apps in the Rico dataset have an average rating of 4.1 stars, and data pertaining to 26 user interactions.

We crowdsourced user traces for each app by recruiting 13 workers (10 from the U.S. and 3 from the Philippines) on UpWork. We chose UpWork over other crowdsourcing platforms because it allows managers to directly communicate with workers, a capability that we used to resolve technical issues that arose during crawling. UpWork workers also preferred longer term work commitments, which was better for us than micro-task-based models since it minimized the time we had to spend training and overseeing workers until they became proficient in using our interface. We instructed workers

to use each app as it was intended, based on its play store description for no longer than 10 minutes.

In total, workers spent 2,450 hours using apps on the platform over five months, producing 10,811 user interaction traces. We paid \$19,200 in compensation, or approximately \$2 to crowdsource usage data for each app. To ensure high-quality traces, we visually inspected a subset of each user’s submissions. After collecting each user trace for an app, we ran the automated crawler on it for one hour. We discuss the contents of the Rico dataset and its intended applications in the next chapter.

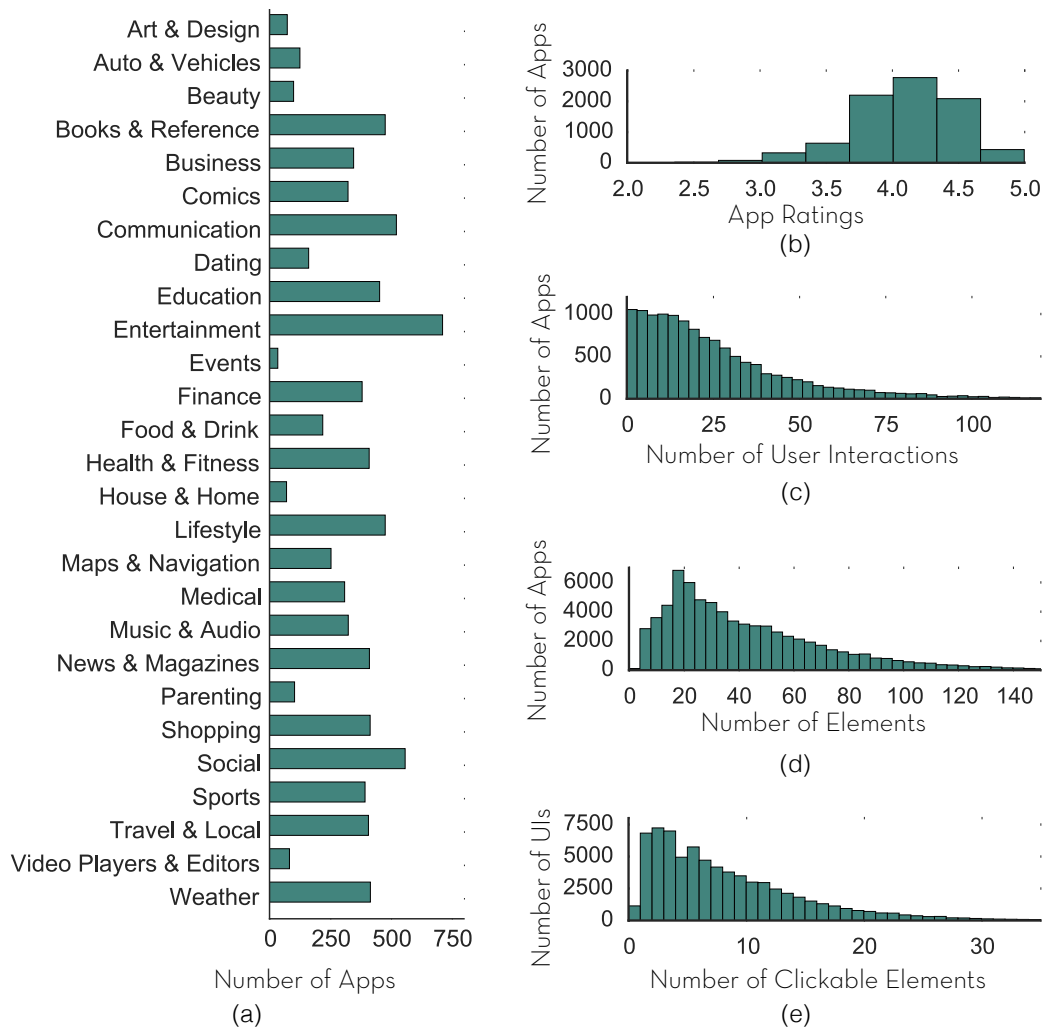


Figure 3.7: Summary statistics of the Rico Dataset: app distribution by (a) category, (b) average rating, and (c) number of mined interactions. The distribution of mined UIs by number by of (d) total elements and (e) interactive elements.

3.4 Discussion and Future Work

With systems like Rico, it is now possible to mine design data at scale from thousands of publicly available apps. We discuss how the mined data can be processed into useful representations of app designs in Chapter 4 and present two applications that use these representations in Chapter 5 and Chapter 6.

There are a number of avenues for future work on extending interaction mining systems. Interaction mining outside of a web-based paradigm could be explored. For instance, mining interaction data directly on end-user devices could have certain advantages, such as capturing more nuanced interaction patterns that are influenced by a user’s context and capturing the effects of different device sizes and configurations. On-device mining could also be more financially sustainable compared to using paid crowdworkers. This could enable mining app designs longitudinally over larger periods of time to build datasets that capture design changes over time.

Another promising direction of future work is towards lowering the need for human exploration. Perhaps more sophisticated automated models for app exploration that emulate human interaction could be used. Human interaction traces such as those captured by Rico would be an ideal dataset for training such models. Future work could also explore the benefits of a system like Rico outside of design mining, in areas like security and software testing where high app state coverage is also desirable [55, 47, 56].

CHAPTER 4

REPRESENTATIONS AND MODELS OF MOBILE APP DESIGN

Interaction mining systems such as ERICA and Rico enable capturing detailed design and interaction data for Android apps. In this chapter, we discuss how the captured data can be processed to expose the different components of an app’s design and to build representations that are useful for downstream data-driven applications. This chapter also illustrates how models of design similarity can be trained on this data to help find unique designs in the dataset and to enable example-based design search.

4.1 App Design Representations

The representations used to capture designs in a domain are closely tied to what capabilities are desired in the design tools. For example, capturing graphic designs as images is useful for designers to visualize designs and to gather feedback on them. However, they are not very useful for building design search and retrieval systems. Having access to structured representation of a design (such as a DOM tree for webpages) allows easier search and retrieval [3]. In this section, we describe two representations of mobile app designs that we have found to be a good fit for a wide range of data-driven design applications.

4.1.1 Interaction Trace

An interaction trace (shown in Figure 4.1) captures an app’s design as a sequence of app UI’s. For each UI, it contains a collection of screenshots (visual representation), a view hierarchy (structural representation), and the user interaction performed on that screen. The set of screenshots for a UI, taken together, captures how the UI transitioned into the next UI in response

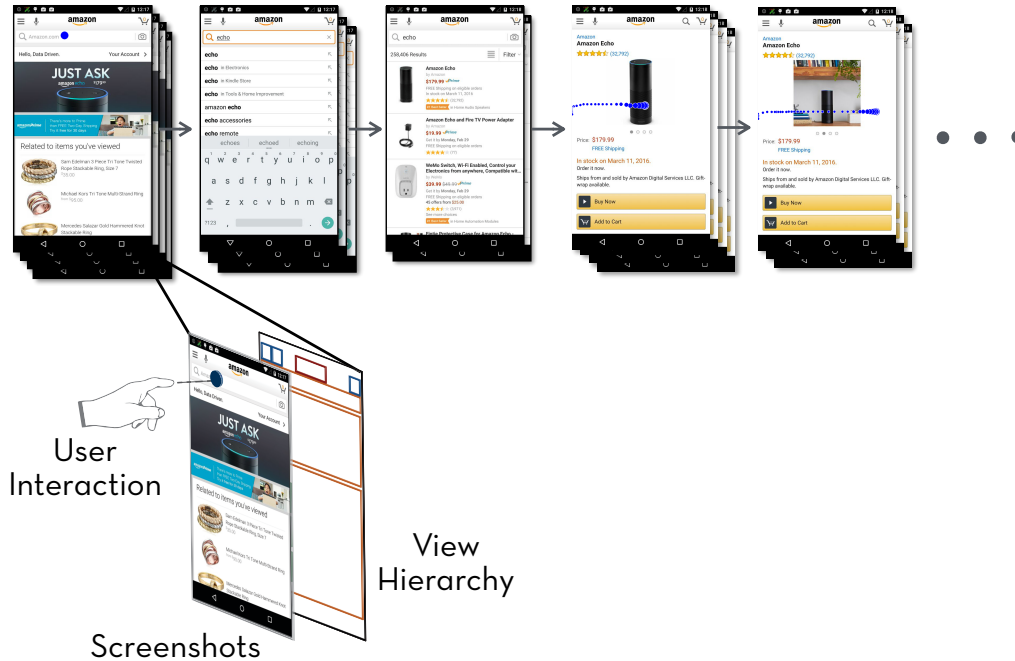


Figure 4.1: Visual representation of an interaction trace. It consists of a sequence of UIs. For each UI it contains a collection of screenshots, a view hierarchy, and the user interaction performed on that screen. The series of screenshots for a UI taken together show how the UI responded to the user interaction.

to the user interaction performed on it.

View hierarchies capture all of the elements comprising a UI, their properties, and relationships between them. For each element, it exposes the element’s visual properties (such as screen position, dimensionality, and visibility), textual properties (such as class name, id, and displayed text), structural properties (such as a list of its children in the hierarchy), and interactive properties (such as the ways a user can interact with it). Additionally, view hierarchies mined by Rico contain Android superclasses for all elements (e.g., `TextView`), which can help third-party applications reason about element types.

Combining the three types of data captured in an interaction trace enables multiple use cases. Combining the screenshots with the user interactions allows visualizing user behavior in an app. The blue dots in Figure 4.1, for instance, show where users tapped or scrolled on the different screens. View hierarchies expose element text and when combined with screenshots allow accurately cropping out individual elements and analyzing their visual details

(such as color) and shapes. Thus, it becomes possible to index and search over UI elements based on text, shape, or visual attributes. Having multiple screenshots during UI changes allows combining them to visualize motion details. And finally, if an interaction trace is mined during human usage (as is done by ERICA and Rico), it often captures meaningful user flows in the app. These flows can be identified using automated methods (see Section 5.2) by combining user interaction details with view hierarchies. Thus an interaction trace captures and helps index and search all four components of an app’s design: user flows, UIs, visual details and animations.

4.1.2 Interaction Graph

An interaction graph (shown in Figure 4.2) comprises of nodes that represent unique UI screens encountered in an app. Edges connect interactive UI

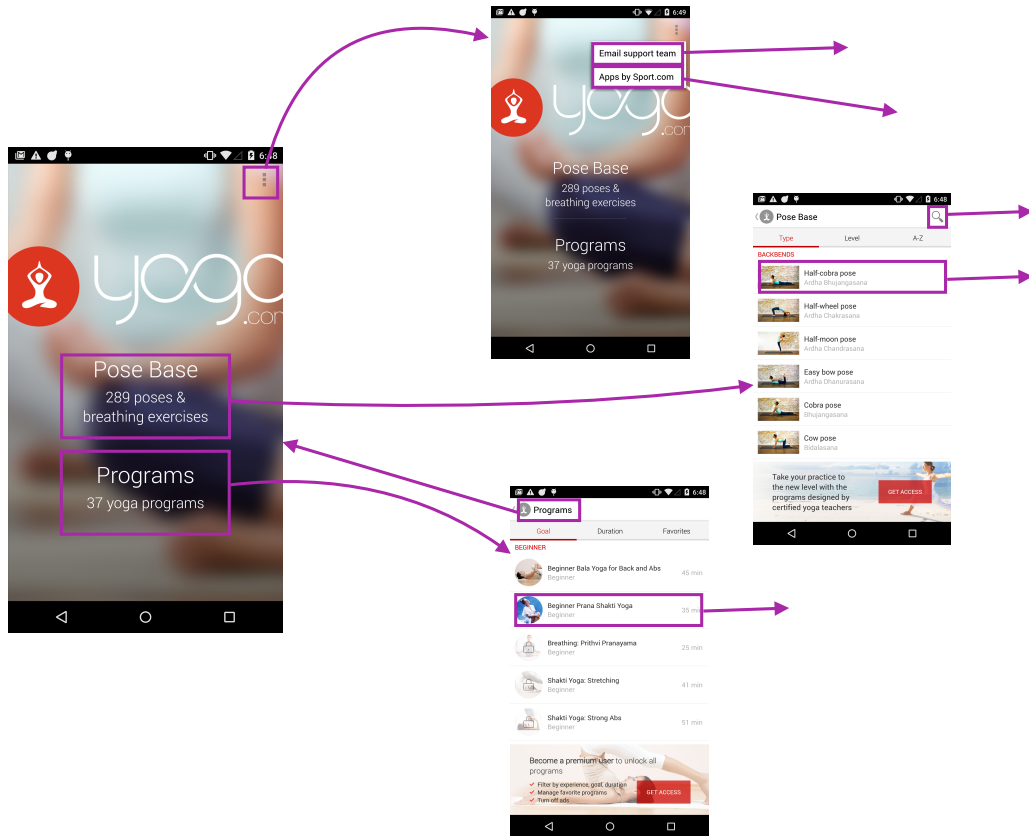


Figure 4.2: Visual representation of an interaction graph. Nodes in this graph represent unique UI screens. Edges connect interactive elements to UI screens that are reached by interacting with them.

elements to UIs that are reached by interacting with those elements.

Interaction graphs are well suited for maintaining state when performing automated exploration of an app (Section 3.3.2). Interaction graphs can also be created from interaction traces produced during human exploration to analyze the complexity of apps. In both cases, creation of such a graph requires a way to identify if two UI screens encountered by an automated crawler (or by a human) are the same. We use a content-agnostic similarity heuristic described in Section 4.3 to do that.

Existing Android app datasets expose different kinds of information: Google Play Store metadata (e.g., reviews, ratings) [60, 61], software engineering and security related information [62, 63], and design data [41, 40, 13]. The Rico dataset captures both design data and Google Play Store metadata.

Table 4.1 compares the Rico dataset with other popular datasets that expose app design information. Design datasets created by statically mining app packages, such as the ones used by Shirazi et al. [41] and Alharbi and Yeh [40] contain view hierarchies, but cannot capture data created at runtime such as screenshots or interaction details. The Rico and ERICA datasets were created by dynamically mining design and interaction data from apps at runtime. The Rico dataset is four times larger than the ERICA dataset, and presents a superset of its design information. The Rico dataset is also the only one of these datasets to be publicly released and is available for download at <http://interactionmining.org/rico>.

Table 4.1: Comparison of the Rico dataset with other app design datasets

	Year	# Apps	# UIs	Mining Technique	View Hierarchies	Screenshots	User Interactions
Shirazi et al.	2013	400	29K	Static	●	○	○
Alharbi et al.	2015	24K	-	Static	●	○	○
ERICA	2016	2.4K	18.6K	Dynamic	●	●	●
Rico	2017	9.7K	72.2K	Dynamic	●	●	●

4.2 The Rico App Design Dataset

4.2.1 Rico Dataset Contents

For each app, the Rico dataset contains Google Play Store metadata, a set of user interaction traces, and a list of all the unique, discovered UIs through crowdsourced and automated exploration (Figure 4.3). The play store meta-

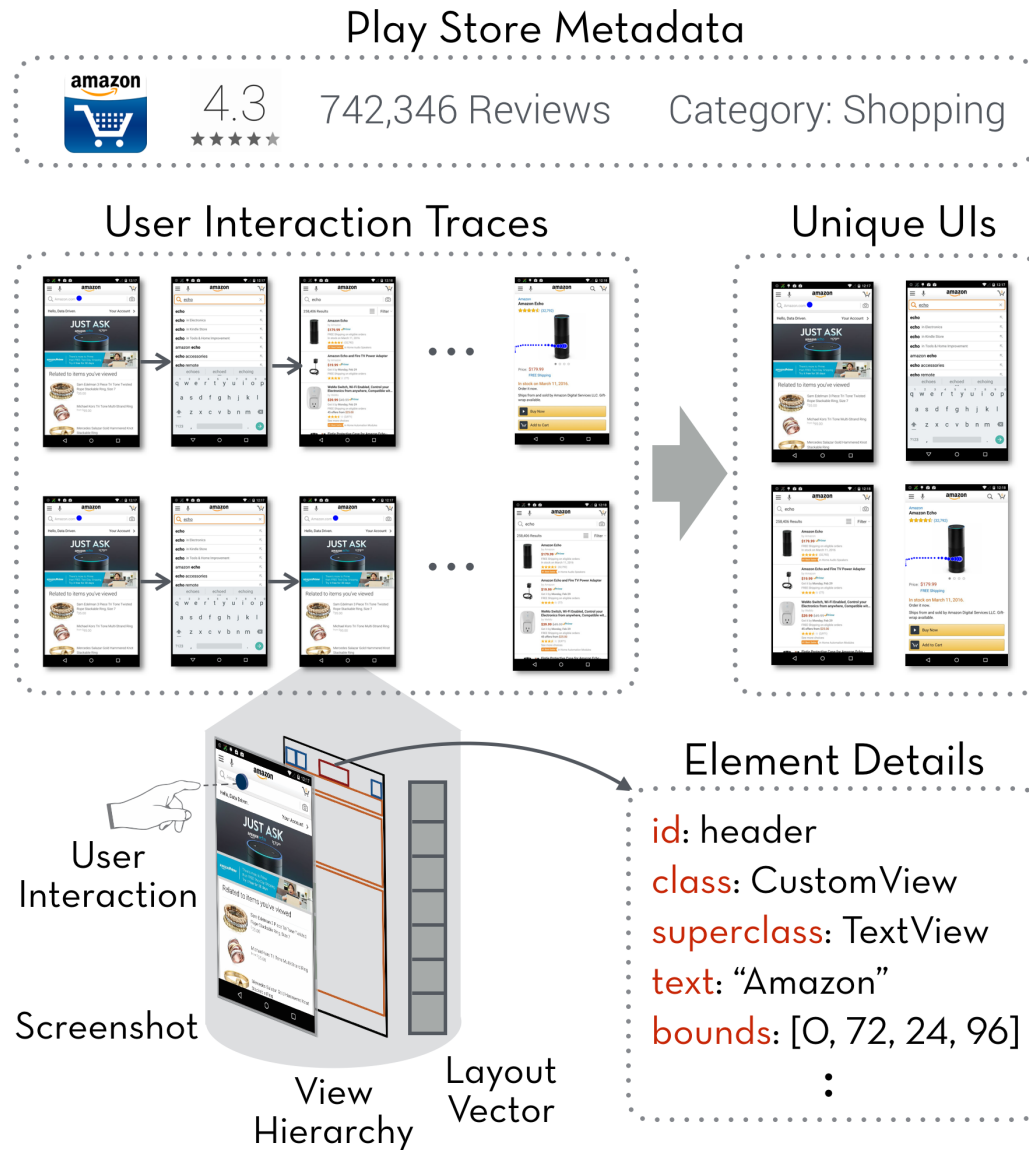


Figure 4.3: The Rico dataset contains Google Play Store metadata, a set of user interaction traces, and a list of all the unique UIs discovered during crawling.

data includes an app’s category, average rating, number of ratings, and number of downloads. Each user trace is composed of a sequence of UIs and user interactions that connect them. Each UI comprises a screenshot, an augmented view hierarchy, a set of explored user interactions, a set of animations capturing transition effects in response to user interaction, and a learned vector representation of the UI’s layout. Overall, the Rico dataset contains more than 3M elements, of which approximately 500k are interactive. On average, each UI comprises eight interactive elements.

The Rico dataset also contains an additional view of each app’s design data: while the ERICA dataset provides a collection of individual user interaction traces for an app, the Rico dataset additionally provides a list of the unique UIs (more than 72,000) discovered by aggregating over user interaction traces and merging UIs based on a similarity measure. This representation is useful for training machine-learning models over UIs that do not depend on the sequence in which they were seen. Lastly, each unique UI in the Rico dataset is annotated with a low-dimensional vector representation that encodes its layout (Section 4.3), which can be used to cluster and retrieve similar UIs from different apps.

The Rico dataset was built to support a variety of data-driven applications for mobile app design. Chapter 5 describes how it can be used to train models of design that enable novel design search interactions including example-based search. Section 7.1.5 discusses four other applications that it could support: mobile layout generation [64, 65], UI code generation [66, 67], user interaction modeling, and user perception prediction [68, 69, 70, 71].

4.3 Identifying Unique UI Designs

Database-backed applications can have thousands of views that represent the same semantic design concept and differ only in their content (Figure 4.4). Exposing the set of unique UIs in the Rico dataset required us to identify such UI screens that represent the same design in an interaction trace. We also needed a way to do that during automated exploration of apps (Section 3.3.2). After Rico’s crawler interacts with a UI element, it must determine whether the interaction led to a new UI state or one that is already captured in the interaction graph.

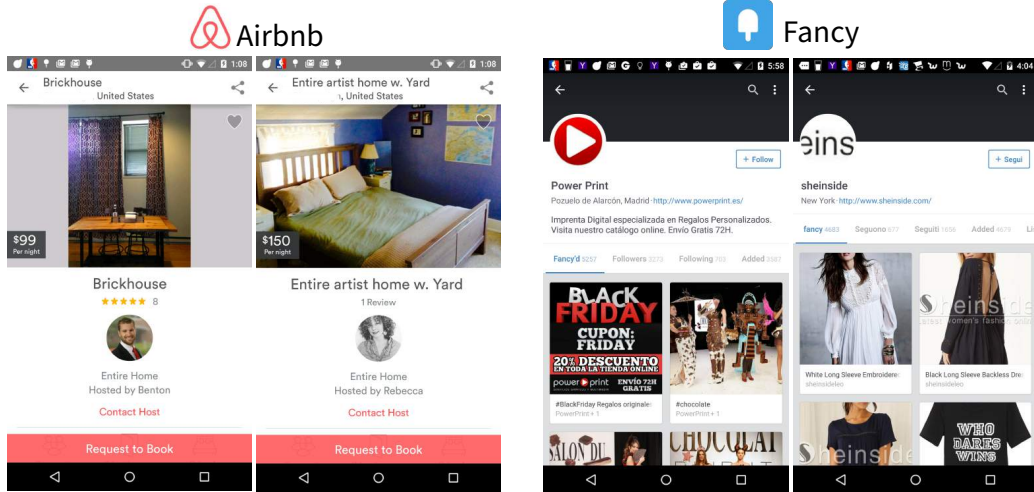


Figure 4.4: Pairs of UI screens from apps that are visually distinct but have the same design. Our content-agnostic similarity heuristic uses structural properties to identify these sorts of design collisions.

Identifying unique UIs screens for mobile apps, however, is challenging. This is because unlike web applications that usually have different URLs for each unique application screen, there is no such identifier that a mobile app exposes during usage (or mining). Although interaction mining captures the Android activity names [54] for each screen, they do not have a one-to-one correspondence with UI designs. Activities in practice can contain multiple UI screens that exhibit different designs.

Therefore, we developed a bespoke content-agnostic similarity heuristic to compare UIs. This similarity heuristic compares two UIs based on their visual and structural composition. If the screenshots of two given UIs differ by fewer than α pixels, they are treated as equivalent states. Otherwise, the crawler compares the set of element `resource-ids` present on each screen. If these sets differ by more than β elements, the two screens are treated as different states.

We evaluated the heuristic with different values of α and β on 1,044 pairs of UIs from 12 apps. We found that $\alpha = 99.8\%$ and $\beta = 1$ produces a false positive rate of 6% and a false negative rate of 3%. We use these parameter values for automated crawling, and computing the set of unique UIs for a given app.

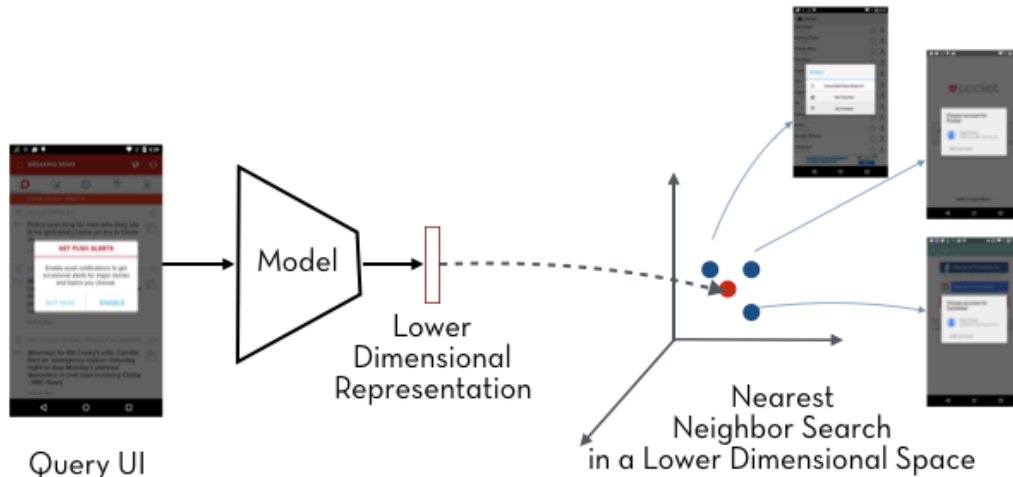


Figure 4.5: Overall approach for example-based searching for UIs. A query UI is first mapped to a lower-dimensional space in a way that similar UIs are mapped close to one another. Then nearest-neighbor searches are performed in this space to find similar UIs.

4.4 Modeling UI Similarity

To facilitate query-by-example searches over the dataset (Figure 4.5), the Rico dataset contains a vector representation for each UI that encodes its layout. This vector representation can be used to find structurally — and often semantically — similar UIs, supporting example-based search over the dataset (see Figure 5.6 for results of search queries). This vector is obtained through training an autoencoder to learn a lower-dimensional embedding for UI layouts.

An autoencoder is a neural network that involves two models — an encoder and a decoder — to support the unsupervised learning of lower-dimensional representations [72]. The encoder maps its input to a lower-dimensional vector, while the decoder maps this lower-dimensional vector back to the input’s dimensions. Both models are trained together with a loss function based on the differences between inputs and their reconstructions. Once an autoencoder is trained, the encoder portion is used to produce lower-dimensional representations of the input vectors.

To create training inputs for the autoencoder that embed layout information, we constructed a new image for each UI encoding the bounding box regions of all leaf elements in its view hierarchy, differentiating between text and non-text elements (Figure 4.6). Rico’s view hierarchies obviate the need

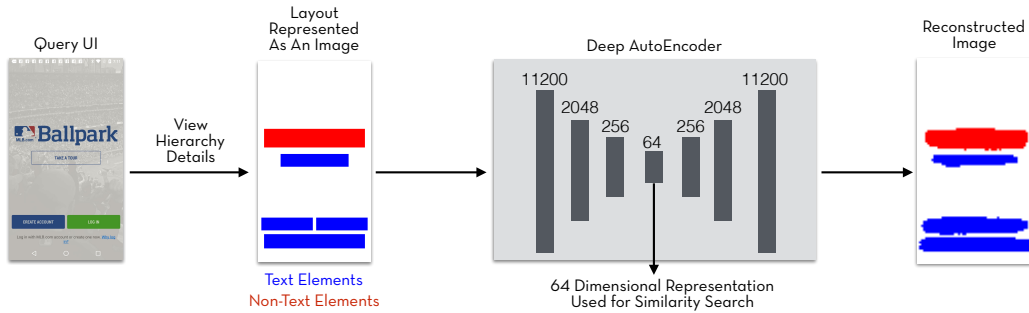


Figure 4.6: We train an autoencoder to learn a 64-dimensional representation for each UI in the repository and to encode structural information about its layout. This is accomplished by creating training images that encode the positions and sizes of elements in each UI, differentiating between text and non-text elements.

for noisy image processing or OCR techniques to create these inputs. In the future, if we can predict functional semantic labels for elements such as *search icon* or *login button*, we can train embeddings with even richer semantics.

The encoder has an input dimension of 11,200, and an output dimension of 64, and uses two hidden layers of dimension 2,048 and 256 with ReLU (Rectified Linear Unit) non-linearities [73]. The decoder has the reverse architecture. We trained the autoencoder with 90% of our data and used the rest as a validation set, and found that the validation loss stabilized after 900 epochs or approximately 5 hours on a Nvidia GTX 1060 GPU. Once the autoencoder was trained, we used the encoder to compute a 64-dimensional representation for each UI, which is also included in the Rico dataset.

We discuss an example-based UI search interface built using this learned layout vector in Section 5.3.

4.5 Discussion and Future Work

There are a number of opportunities to extend and improve the Rico dataset. New models could be trained to annotate the dataset with richer labels, like classifiers that describe the semantic function of elements and screens (e.g., search, login). Designers can perform keyword searches over these functional semantic classes to find relevant elements or screens in a user interaction trace. Similarly, researchers could crowdsource additional perceptual annotations (e.g., first impressions) over design components such as screenshots and animations, and use them to train newer types of perception-based pre-

dictive models.

Unlike static research datasets such as ImageNet [74], the Rico dataset will become outdated over time if new apps are not continually crawled and their entries updated in the database. Therefore, another important avenue for future work is to explore ways to make app mining more sustainable. One potential path to sustainability is to create a platform where designers can use apps and contribute their traces to the repository for the entire community's benefit.

CHAPTER 5

DESIGN SEARCH

Designers often draw upon prior work in the form of examples to solve new problems [75, 76, 77]. Examples of previous work serve as sources of inspiration, reduce barriers to entry, lower the cognitive burden of routine tasks, and help understand the landscape of possible solutions [10, 11]. In this chapter, we discuss how datasets created through interaction mining enable building search applications to support mobile app designers in their quest for examples in the design process.

5.1 Finding App Design Examples in Practice

In a series of formative interviews we conducted with app designers (details in Section 6.2), participants emphasized the importance of looking at examples in the design process. They mentioned doing so to keep up-to-date with popular design patterns, to find and reuse design patterns that work well for different use cases, and to conduct competitive analysis.

Several methods to find examples also emerged in these interviews. Designers mentioned depending on their own memory of designs seen in the past, asking colleagues for pointers, and searching online design forums and design repositories. Each of these current methods for finding examples, however, has its limitations. Depending on one’s memory and that of others does not scale. There are only so many examples that a designer can commit to memory and retrieve at will. While online design forums (such as Dribbble [78] and Behance [79]) may contain examples from real apps, it is hard to tell them apart from the large number of design prototypes that are submitted by designers to solicit feedback and showcase their skills.

Design repositories (such as uxarchive.com [80] or pptrns.com [81]) are a recent development in the design community. These repositories archive user

flows in popular apps, given their importance in the design of an app. These user flows are showcased as sequences of screenshots (Figure 5.1). These repositories, however, are manually curated by designers who must identify desirable flows, manually capture associated screenshots, upload them, and tag them with useful keywords. The effort required in doing so makes them difficult to scale to large numbers of apps [82]. In this dissertation, we develop a scalable approach to creating such app design repositories by automatically identifying user flows in interaction traces captured during app usage.

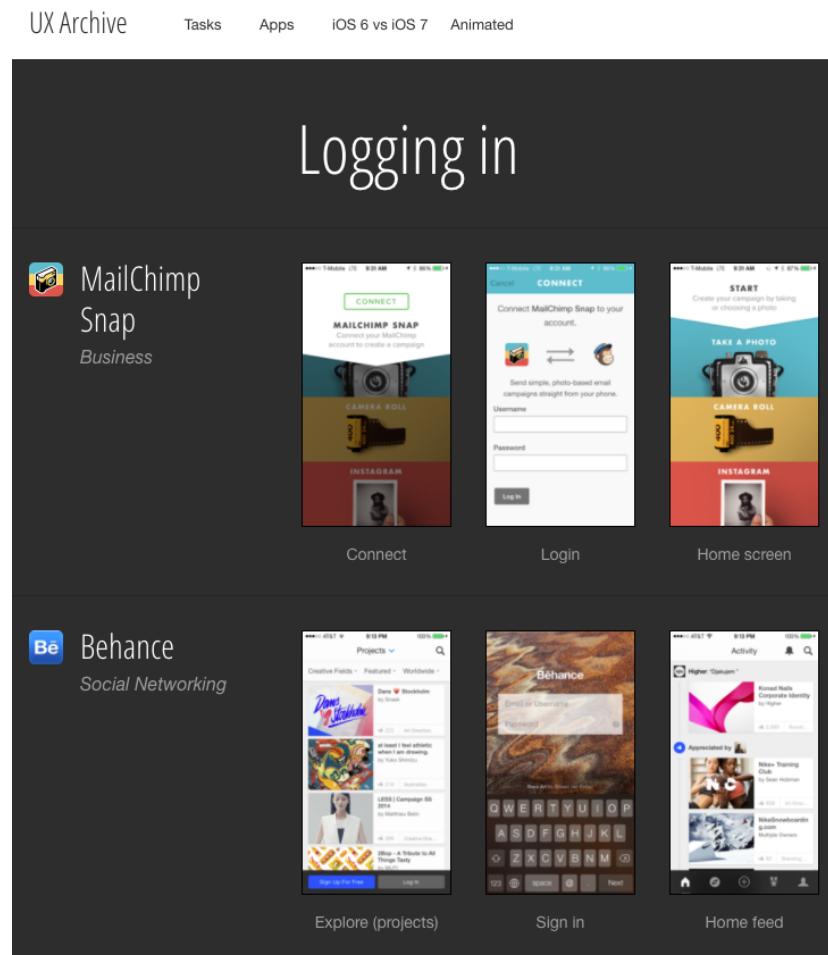


Figure 5.1: Design repositories such as uxarchive.com showcase user flows from popular apps using sequences of screenshots. Here we see examples of login flows from two apps.

5.2 Flow Search

Interaction mined datasets contain metadata that make it possible to support keyword-based searches out of the box. For instance, the view hierarchy contains programmer-defined annotations such as the activity name for each UI and the class name and id for all UI elements. Usually these annotations are semantically meaningful. A developer might call an activity a `loginActivity` or a `searchActivity`. Or they might tag an element with an id of `settingsBtn`. Thus, elements and UIs can be indexed based on these annotations and made searchable using keywords. Using this approach, we built a keyword-based search interface over the Rico dataset that is available online at <http://uisearch.interactionmining.org>

Indexing user flows in interaction traces, however, is not as straightforward. In the rest of this section we describe a machine-learning-based approach to automatically identify user flows by combining information contained in view hierarchies with user interaction information.

5.2.1 Identifying User Flows

We developed two methods to identify user flows in interaction traces: an element-based method and a layout-based one. These methods are based on two insights. First, apps are generally designed with visual, textual, or structural cues in their elements (or layouts) that help users identify important semantic tasks. For example, a magnifying glass icon usually indicates a search flow, and the words “sign in” usually indicate a login flow.

Second, a user must interact with an indicative element in order for it to be part of a corresponding flow. For example, if we know that a user clicked the magnifying glass icon on a particular UI screen, we expect that screen to be part of a search flow. Based on these insights, we combine the user interaction and UI data in an interaction trace to identify meaningful flows.

Element-Based Flow Detection

To identify elements indicative of common flows, we clustered the elements users interacted with in the ERICA dataset based on their visual and textual

features. We then built classifiers to automatically identify similar elements in new traces.

Clustering Interactive Elements. We clustered both text elements and icons. We identified text elements using the text string associated with them in the view hierarchy. Non-text elements smaller than 200×200 pixels and with aspect ratios between 0.5 and 2 were chosen as candidate icons.

To cluster icons, we converted them to grayscale and scaled the images to 50×50 pixels. We trained an autoencoder [72] with 2500 inputs and two hidden layers (with 500 and 200 neurons, respectively) to learn a 20-dimensional feature vector, and then performed k-means clustering in this space. We chose 100 clusters, since the mean-squared error plateaued at that number. Icons from some of the largest clusters are presented in Figure 5.2. We observed that many of the clusters correspond to icons with semantic meaning such as “sharing”, “searching”, or “liking”.

We computed the frequency of words and phrases across all text elements in our dataset. Some of the text elements with the highest frequency are presented in Figure 5.2. As with icons, these elements also contain semantic meaning related to flows.

Classifying Elements. Based on these clusters, we developed classifiers to automatically identify similar interactive elements. For icons, we trained neural network-based classifiers with an architecture similar to the autoencoder’s (2500 inputs, two hidden layers with 200 and 100 neurons). For text elements, we built simple binary keyword-identification classifiers for each flow. Some flows (such as “search”) have both a icon-based and a text-based classifier. Examples of flows detected in our dataset with these classifiers are shown in Figure 5.3.

Layout-Based Flow Detection

We can also detect flows by identifying screens with specific UI layouts (or Android components) on which specific gestures are performed. We developed detectors for three such patterns. Figure 5.4 shows examples of these flows in our dataset. Consuming flows were identified by searching for UIs with more than 2000 characters of text where users have also scrolled at least twice. Composing flows were detected by searching for UIs with a deployed keyboard and with text input boxes that covered more than 60% of the re-

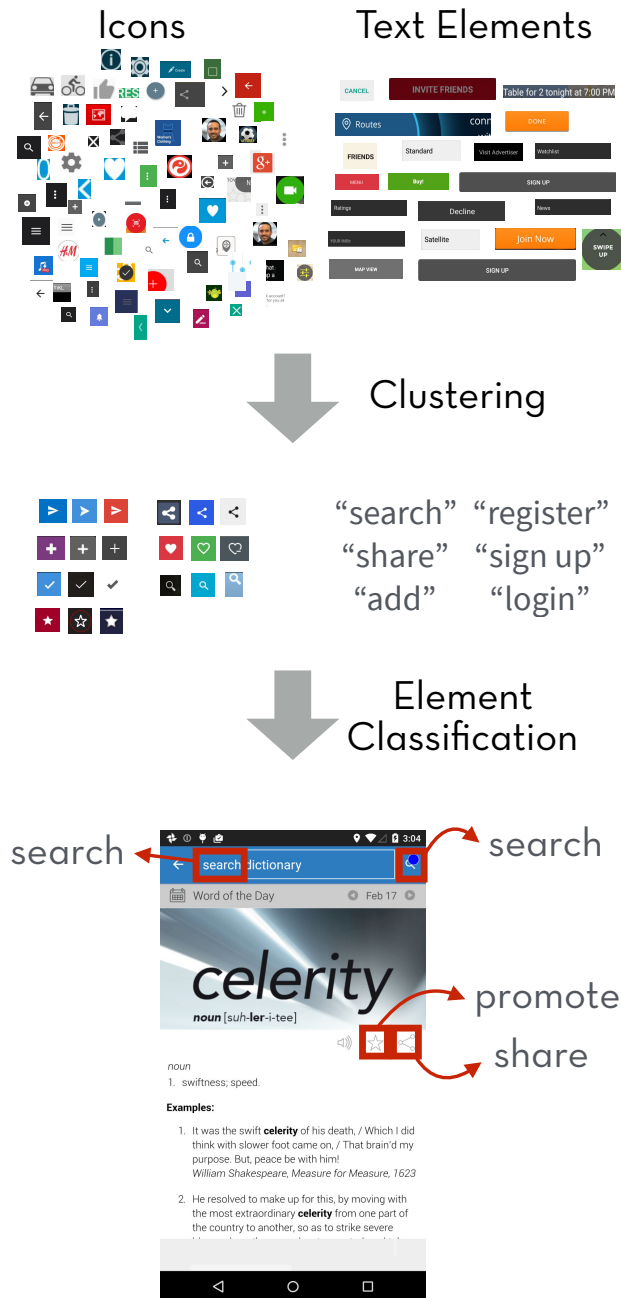


Figure 5.2: Unsupervised clustering of interactive elements produced clusters that helped identify the most common types of semantically-meaningful elements in the ERICA dataset. We used this information to build visual and text-based classifiers to detect these elements in UIs, and leverage them to identify semantically meaningful flows in user interaction traces.

maining screen space. Onboarding flows were found by searching for the Android ViewPager component (generally seen as a sequence of dots on the

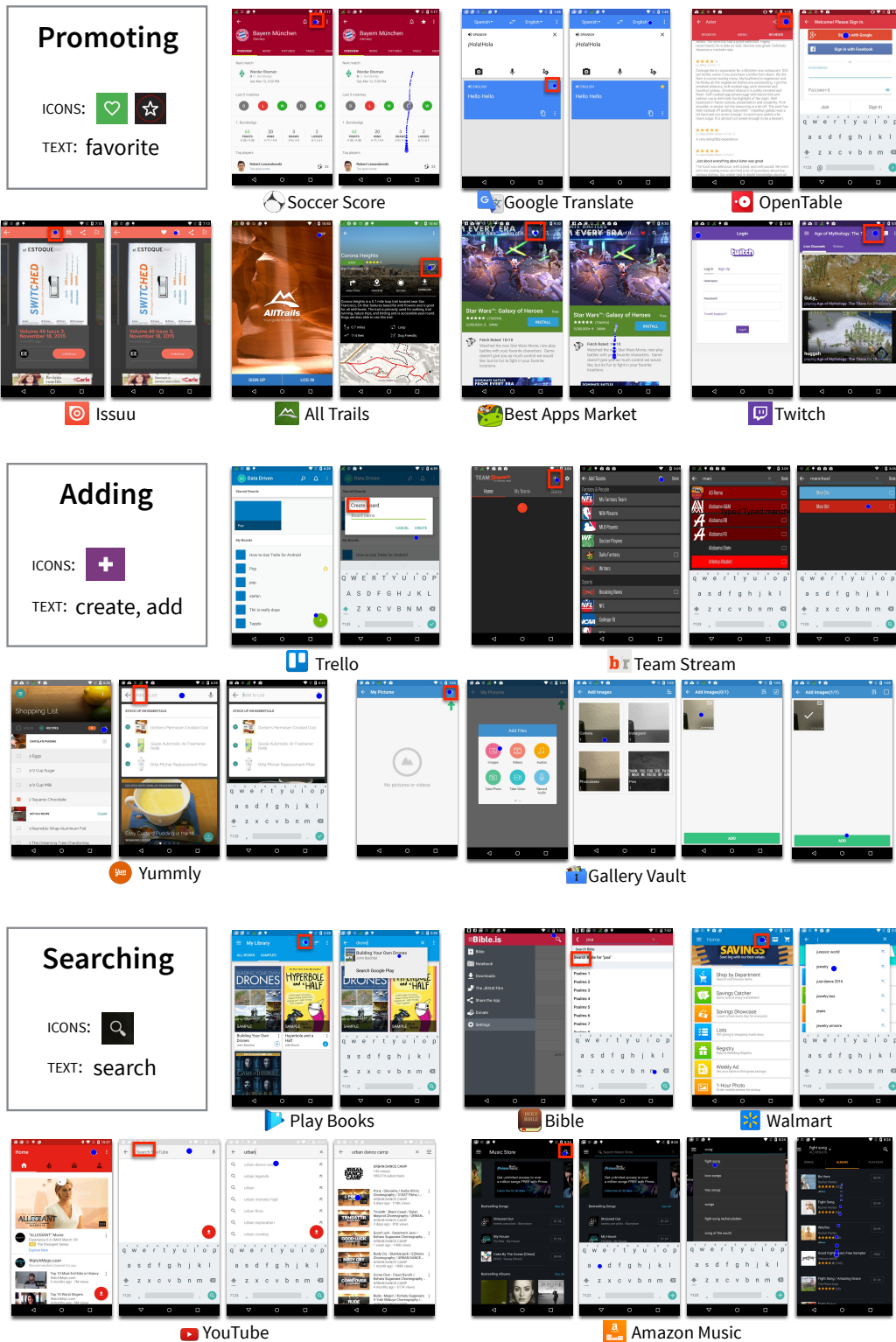

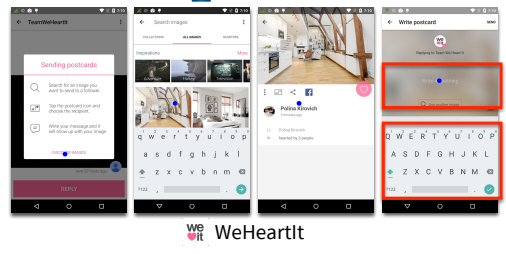
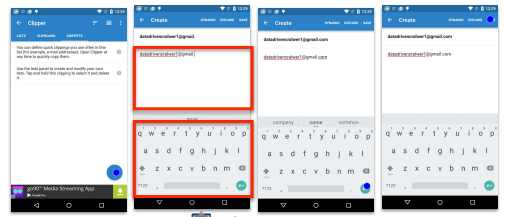
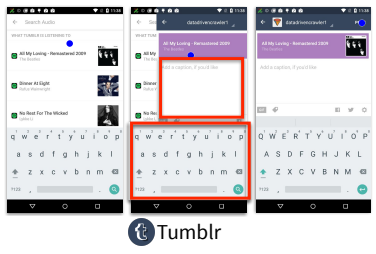


Figure 5.3: Examples of three different types of flows detected in our dataset by using element-based detectors. Visual markers and textual keywords that were used to identify each type of flow are also shown.

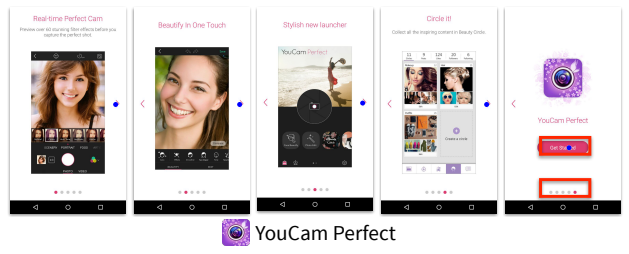
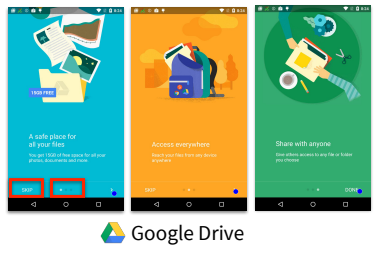
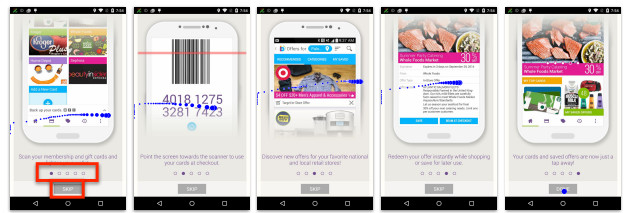
Composing

Keyboard Deployed + $\frac{123}{\text{Android EditText}}$ Occupies 60% of Remaining Screen

Onboarding

Android ViewPager + TEXT: Skip, Get Started



Consuming

2000 Characters in TextView + 2 Scroll Interactions

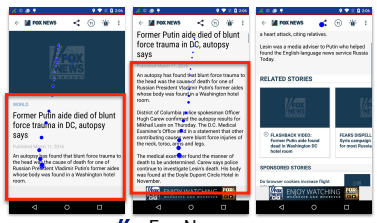
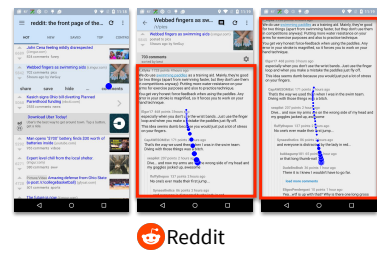


Figure 5.4: Examples of three different types of flows detected in our dataset by using layout or Android component-based features. Patterns that were used to identify each type of flow are also shown.

screen) along with the words “get started”, “skip”, or “tutorial”.

Overall, we used element- and layout-based flow detection methods to identify 23 flow types that are popular in existing flow repositories and occur frequently in our dataset. 6.5% of the elements users interacted with in the ERICA dataset were indicative of one of these flows. On average, each app contained three such flow examples.

5.2.2 Flow Search Interface

Our pipeline for automated user flow identification allowed us to build — in two months — a repository with flows from $7\times$ more apps than UX Archive, the most popular sxtant repository, which has been in operation for four years [80]. We developed a web interface for searching and visualizing the user flows found in our repository (Figure 5.5), and made it available online at <http://flowsearch.interactionmining.org>.

The interface offers two ways to search for flow examples. Users can select the name of a flow from a list, and see search results from apps showing the relevant UI screen with the indicative element highlighted in red. Users can also use text-based queries to search for flows. These free-form queries return results based on the text contained in UI elements as well as the elements’ `classname` and `id`, since developers frequently use descriptive names for these fields.

Clicking on a search result takes the user to the flow viewer interface. It shows the UI screen from the results page with the two previous and subsequent screens from the interaction trace, as well as the interactions the user performed on these screens. It does not explicitly show the beginning or end of an example flow, but rather lets the user go back and forth in the trace to view more UI screens as desired. In addition to viewing a flow as a sequence of screenshots, users can also view animated GIFs of the UIs responding to the shown user interactions by hovering over the screenshots.

5.3 Example-Based UI Search

Interfaces that support example based searching are useful for exploratory searches without clear goals. They are also used when keywords are hard to

Search Interface

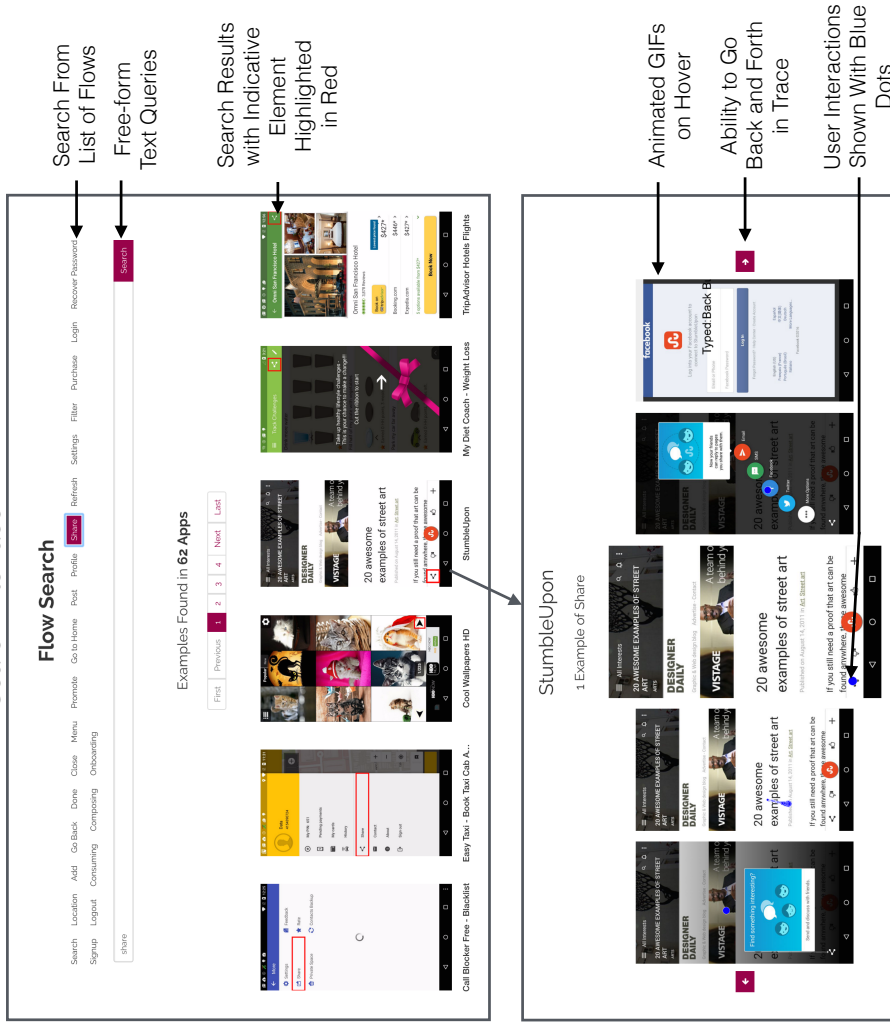


Figure 5.5: Search interface for finding and visualizing example flows found in the ERICA dataset. Users can search for one of the 23 pre-identified flows or perform free-form text-based queries. Each search result shows one UI screen from an app with the indicative element for the flow highlighted in red. Clicking on one of the results takes users to a flow viewer interface that shows the screen from the results page in the context of the nearby screens in the interaction trace. Hovering on an image brings up an animated GIF showing how the UI responded to the user interaction shown on it.

articulate. Such example-based search interfaces have been explored in prior work for webpage designs. *d.tour* is a design gallery that supports “show more/less like” queries implemented using neighborhood searches in feature spaces that describe a webpage’s use of space, color, text, and images [83]. *Webzeitgeist* supports direct queries with example webpages [3]. It retrieves similar webpages using nearest-neighbor searches in a learned space.

We implemented an example-based search system for UI designs using the layout vectors learned over the Rico dataset (Section 4.4). Figure 5.6 shows several example query UIs and their nearest neighbors in the learned 64-dimensional space. The results demonstrate that the learned model is able to capture common mobile and Android UI patterns such as lists, login screens, dialog screens, and image grids. Moreover, the diversity of the dataset allows the model to distinguish between layout nuances, such as lists composed of smaller and larger image thumbnails.

This search system is available online at <http://uisearch.interactionmining.org>.

5.4 Discussion and Future Work

This chapter demonstrates how interaction mining enables building app design search systems with lower effort and at a much larger scale than what was possible before. There are several important directions for improving app design search systems. Our work focused on layouts for example based-searching. Models that learn other dimensions of similarity (semantic similarity for instance) between UIs could be explored.

Our keyword- and example-based search interfaces were inspired by similar interfaces in other domains. The underlying assumption is that app designers would use these modalities in ways similar to other designers. Field studies could confirm this hypothesis or help uncover differences in search behavior that could inform such interfaces. Search interactions other than keyword and example-based ones could also be explored. For example, retrieving designs based on rough, low-fidelity sketches could be explored [84]. We used sequences of screenshots to show user flows (and GIFs to show animations) based on the designs of current popular app design repositories [80, 81]. Future work could explore better ways to present the design of apps in search

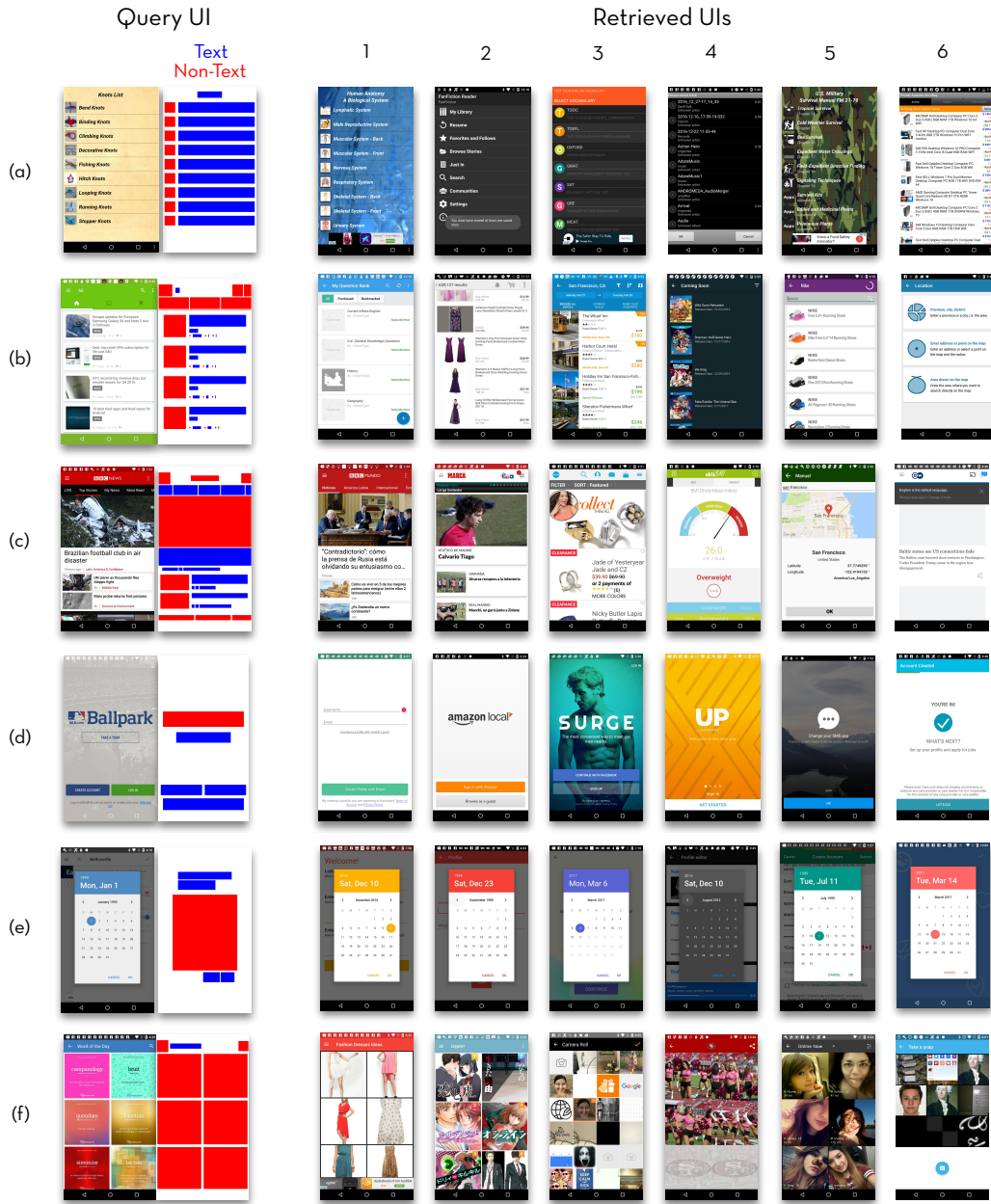


Figure 5.6: The top six results obtained from querying the repository for UIs with similar layouts to those shown on the left, via a nearest-neighbor search in the learned 64-dimensional autoencoder space. The returned results share a common layout and even distinguish between layout nuances such as lists composed of smaller and larger image thumbnails (a,b).

interfaces. In addition, it could explore modifications to the presentation format to improve the search engine’s effectiveness towards its larger goal of helping designers ideate better. For instance, perhaps artificially ensuring diversity of presented results will lead to better ideas being generated.

CHAPTER 6

ZERO-INTEGRATION PERFORMANCE TESTING

Mobile app designers use a range of techniques to evaluate their design ideas. This chapter presents, zero-integration performance testing (ZIPT), a new technique and an online platform for design evaluation that allows performance testing using crowdsourced interaction data. It demonstrates how ZIPT can be used in the early stages of the design process to understand which examples to draw from and in later stages to enable comparative testing with low cost and effort.

6.1 Background

Throughout the mobile app design process, designers seek to understand the artifacts they build and the experiences those artifacts allow users to have. During ideation, designers attempt to evaluate the relative merits of potential satisficing designs. Before design specifications are sent to the engineering team, designers endeavor to detect usability issues. Once an app is implemented, designers attempt to optimize user performance metrics and benchmark them against competitors.

To evaluate the performance of mobile designs, designers employ a number of testing techniques, which generally fall into one of three categories: A/B [85], usability [22], and analytics-driven testing [86]. While these techniques all provide value to designers and elevate the practice of design beyond guesswork and intuition, the costs associated with optimizing design performance can be steep.

A/B and analytics-driven testing require substantial engineering resources to build out alternative solutions and instrument them for testing. Manually aggregating unstructured usage data (e.g., video and audio streams) acquired during usability testing is costly and time consuming [87].

This chapter presents a design performance testing approach that makes it economically feasible to answer the fundamental question of app design: “given the space of possibilities, which solution performs best?” This approach is based on the key observation that, given the millions of mobile apps available today, it is likely that any UX problem a designer encounters has been tackled many ways by a number of existing apps. This approach leverages existing app implementations to enable comparative testing at scale. It also allows app designers to go beyond searching for examples [83, 13, 3] and helps them collect the data needed to decide which examples to draw from in their own work.

This approach is manifest in ZIPT, which allows designers to collect detailed design and interaction data over any Android app — including apps they do not own and did not build — and correlate this data with quantitative metrics and qualitative feedback. Analytics and A/B testing tools require access to an app’s source: at a minimum, a developer must instrument the app with tracking code to begin collecting data. ZIPT, in contrast, provides comparable functionality with zero integration: it requires no access to code, and can be deployed by any user over any app in the Android store. Unlike usability testing platforms, ZIPT automatically captures and aggregates structured interaction data from app usage: therefore, it can provide user performance data for third-party apps at a tiny fraction of the cost (30¢ a user versus \$50 on a usability testing platform like [usertesting.com](#)).

6.2 Formative Study

Prior to building ZIPT, we conducted interviews with five Google employees — two interaction designers, two UX researchers, and one UX Engineer — to understand how practitioners leverage examples in their current work, and to discuss how they might incorporate crowdsourced performance data on extant mobile designs. Echoing the literature, all participants referenced examples early in the design process, especially for competitive analyses. The interaction designers, in particular, reported frequently studying existing flows — logical sequences of screens for performing a task — in competitors’ apps and manually comparing them.

Although mere knowledge of the existence of a design example can be

useful, the practitioners we interviewed often expressed a desire to go further and understand how well an example performs: whether users can understand a design and use it as intended to accomplish their goals. When choosing examples to build upon, participants also considered how easy it would be to communicate the benefits of a particular design pattern to the rest of their team. All participants indicated that they do not turn to examples enough and wished to do so more frequently.

All participants expressed interest in crowdsourcing user performance data to evaluate existing design patterns. Participants offered different ways in which this data might be useful in their design process. P3 observed, “if getting such data was easy, I would ask team members to generate it for patterns they see in the wild before they bring it to me to build a prototype.” P2 said, “I would use such a process to quickly pinpoint the *what* (what works or does not work) and use other methods to find out the *why*.” P4 said, “I can think of two ways I would use such data. One is for finding unexpected behavior and the other is for storytelling.”

To conduct remote usability tests, participants wished to collect performance and interaction data over specific tasks in apps. They requested aggregate usability metrics such as completion rate, as well as the average time and number of interactions required to finish the specified task. To identify common usage patterns and unexpected behavior, designers also wished to examine the interaction paths users took to accomplish a task. At an app level, designers hypothesized that it would be useful to collect general user perceptions about usability and branding.

While participants were generally positive about crowdsourced usability testing, they expressed concerns around ensuring uniform device setup across workers: they were wary of trusting crowd workers to follow configuration steps such as software installation. Although participants indicated that they might use crowdsourcing to test their own apps, several expressed reservations about inadvertently exposing features that were not yet public.

6.3 The ZIPT Platform

The ZIPT platform comprises three stages (Figure 6.1). In the first stage, designers define the scripted usability tests they want to run over a set of

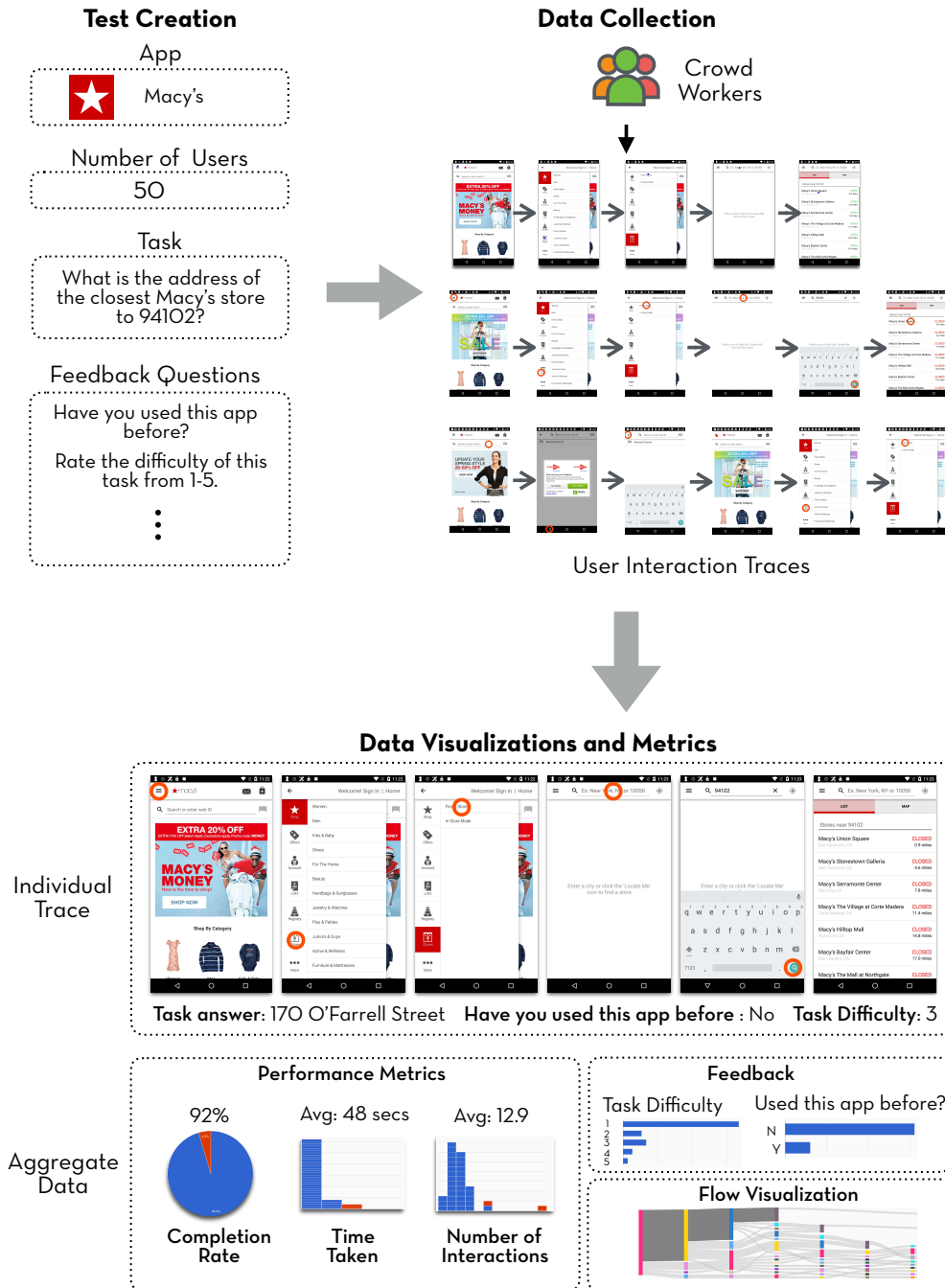


Figure 6.1: An overview of ZIPT: designers define usability tests (top left), which are deployed to crowd workers (top right). ZIPT computes and presents designers with aggregate performance metrics and visualizations based on the data collected from crowd workers (bottom).

apps. In the second stage, ZIPT deploys the user-defined tests to crowd workers. Finally, in the third stage, ZIPT computes and presents designers with aggregate performance metrics and visualizations based on the data collected from crowd workers.

6.3.1 Test Creation

ZIPT allows designers to define a usability test by uploading a set of Android APK files, specifying the number of traces to be collected, and providing a description of the task for crowd workers to perform. The description can be open ended (e.g., “browse for music”), or very specific (e.g., “create a playlist with three songs”). It can be written in plain text or in the Jade templating language if designers want to customize its presentation.

Designers can customize tasks in two additional ways. First, they can construct tasks with built-in verification, requiring a user to provide an answer to a question in addition to performing a task in the app (e.g., “What is the address of the store closest to the 94102 zip code?”). Phrasing tasks as questions can be useful for quickly identifying traces where users could not complete the task. Second, designers can provide a specific starting point for crowd workers within the app other than the home screen. This feature is useful for testing specific flows that a user would encounter deeper within an app. Designers specify custom starting points by demonstration. ZIPT runs the app within the browser, records interaction paths taken by designers, and replays those interactions before presenting the app to a crowd worker.

ZIPT requires each crowd worker to self-report whether or not they successfully completed the specified task. It also allows researchers to add other pre- and post-task questions, which can be used to collect demographic data for cohort analysis and qualitative feedback to understand usability and brand perception. ZIPT supports questions with Likert-scale, multiple choice, and free-form text answers.

6.3.2 Visualizations and Metrics

Once designers submit a test, ZIPT deploys it on Amazon Mechanical Turk. As crowd workers use the apps, the platform automatically captures design

and interaction data streams. After collecting the requisite number of user traces, ZIPT computes aggregate metrics and visualizations, and presents a results dashboard comprising four types of data:

Individual Traces. ZIPT allows designers to inspect each crowd worker’s interaction path through the app. User traces are presented as sequences of screenshots with user interaction annotations. Each trace also contains a user’s answers to task and feedback questions, if any were specified by the designer.

Performance Metrics. ZIPT computes and presents three performance metrics: completion rate, time on task, and number of interactions performed. The performance dashboard visualizes completion rates as pie charts, and the other two metrics as histograms. Researchers can interactively select different performance segments in these visualizations to view their associated traces.

Qualitative Feedback. The dashboard also collates crowd workers’ answers submitted in the pre- and post-task questionnaires. It uses bar charts for summarizing Likert-scale and multiple-choice answers. Like the performance visualizations, designers can select segments in these charts to inspect corresponding traces. For example, designers can quickly identify and inspect all traces where crowd workers reported high task difficulty to uncover potential usability bugs.

Flow Visualization. ZIPT presents interactive flow visualizations to help designers compare and contrast the different paths taken by users to accomplish tasks. These visualizations build on prior work on representing user flows in mobile apps with Sankey flow diagrams [88]. Color-coded nodes representing different screens in a user interaction trace are arranged sequentially along the horizontal axis. The nodes in the visualization are connected by bands whose thickness is directly proportional to the number of users who took the path defined by its node endpoints. These diagrams can allow designers to quickly understand aggregate interaction data, as well as inspect individual screens to identify usability issues. For example, designers can interact with the nodes to view screenshots and with the bands to see user interaction between two screens. Additionally, a designer can create a flow by demonstration in an app and use it to highlight paths of interest in the visualization. For example, by defining a *golden trace* for a task — the intended path a user should take to complete a task — and highlighting this

path in the diagram, a designer can quickly determine the screens that cause the most confusion and are most likely to prevent users from completing the task.

6.3.3 Implementation

To ensure that all crowd workers use apps under identical conditions without having to install any software on their devices, ZIPT uses a web-based streaming approach for collecting data from mobile apps. Apps run in a controlled environment on a mobile device farm, and are streamed to client devices through the browser. This streaming approach has been used in recent systems including ERICA for mining mobile app designs [13] and MobiPlay for implementing record and replay systems for app testing [89], and been piloted by Google for mainstream Android app usage [90].

We modeled ZIPT’s web-based app streaming implementation after ERICA [13]. Apps are run on a set of Android devices connected to a server that hosts the ZIPT web application. The web application continuously streams the phone screens as compressed JPEG images. When users interact with these images in their browser, the frontend web application captures these interactions and sends them to the phones via the server. When an app’s screen changes as a result of these interactions, users see the updated screens in their browser with a slight delay. ZIPT supports diverse interactions including tapping, scrolling, two finger pinch-and-zoom, and keyboard text entry.

In contrast to ERICA and MobiPlay, where app streaming occurs over a local network, ZIPT must have good performance over the public Internet. As a result, ZIPT uses a low frame rate (5 to 10 frames per second) and high compression ratio to minimize bandwidth requirements. To produce acceptable latencies, ZIPT only recruits crowd workers within the U.S. (where the server is located). In addition, ZIPT logs the incurred latency for each user interaction, and can filter out traces impeded by network performance.

Like ERICA [13], ZIPT logs screenshots, view hierarchies, and interactions produced during usage, and combines them to produce interaction traces. ZIPT computes completion rates based on user-reported data, and the other two metrics — time on task and number of user interactions — directly from the interaction traces. When examining individual traces, a designer can fix

any mislabeled traces, where a user incorrectly self-reported the completion of a task. ZIPT factors in network and server-to-phone communication latencies when computing time on task for a more accurate estimate. In the future, ZIPT can be extended with mechanisms for outlier detection [58] or data validation [91, 92, 93, 93, 94, 95] as necessary.

To construct flow visualizations, ZIPT automatically computes the set of screens that represent the same UI state and merges them together into a single node. To do so, ZIPT employs the content-agnostic similarity heuristic presented in Deka et al. [14], which compares screens based both on their structural and visual properties. Designers can also manually merge and split nodes to fine tune the visualization.

6.4 Case Studies in Using ZIPT

We used ZIPT to identify usability issues in — and compare the performance of — a number of popular apps. In this section, we present a series of case studies to demonstrate how the data collected and visualized by ZIPT can provide novel design insights for apps used by millions of people. For these case studies, we selected 10 popular Android apps that target a diverse set of day-to-day consumer activities. Via ZIPT, we crowdsourced performance data over tasks central to these apps, collecting between 15 to 50 user traces for each task. We paid Amazon Mechanical Turk workers 30¢ for each trace and used a server backed by three Android devices, which allowed us to collect approximately 15 traces per hour.

6.4.1 Identifying Usability Issues

ZIPT demonstrates that combining aggregate flow and individual trace visualizations allows designers to quickly identify potential usability problems in apps. Researchers can detect usability problems by examining both common and unusual user traces. For example, if many users drop off at a common node in the flow visualization, this may signify a usability problem in the critical path of the task. On the other hand, if most users successful follow the golden path but a small number are unable to complete the task, this may indicate a usability issue in a part of the app that users are not able

to easily recover from. We encountered both cases in the user traces we crowdsourced with ZIPT.

In one test, we asked crowd workers to find the next train's departure time from a specific station on *The Transit App*, which provides real-time information updates on public transportation systems. The easiest way to complete the task is to search for the station using the search bar at the top of the app's home screen. From the flow visualization, we observed that only 21% of users (6 out of 28) initially attempted to search for the answer (Figure 6.2). We hypothesize that more users would have tried the search bar if it was more visually prominent, many search bars have a box around them or a high contrast line underneath them.

In another test, we used ZIPT to ask crowd workers to “add a cookie to your food log” on *MyNetDiary*, an app used to track calories. From the resultant flow visualization, we noticed a large user drop-off on the data entry UI (Figure 6.3), and that 21% (5 out of 24) of users encountered an error on the next screen. We discovered that the error occurs when a unit (oz, cookie,

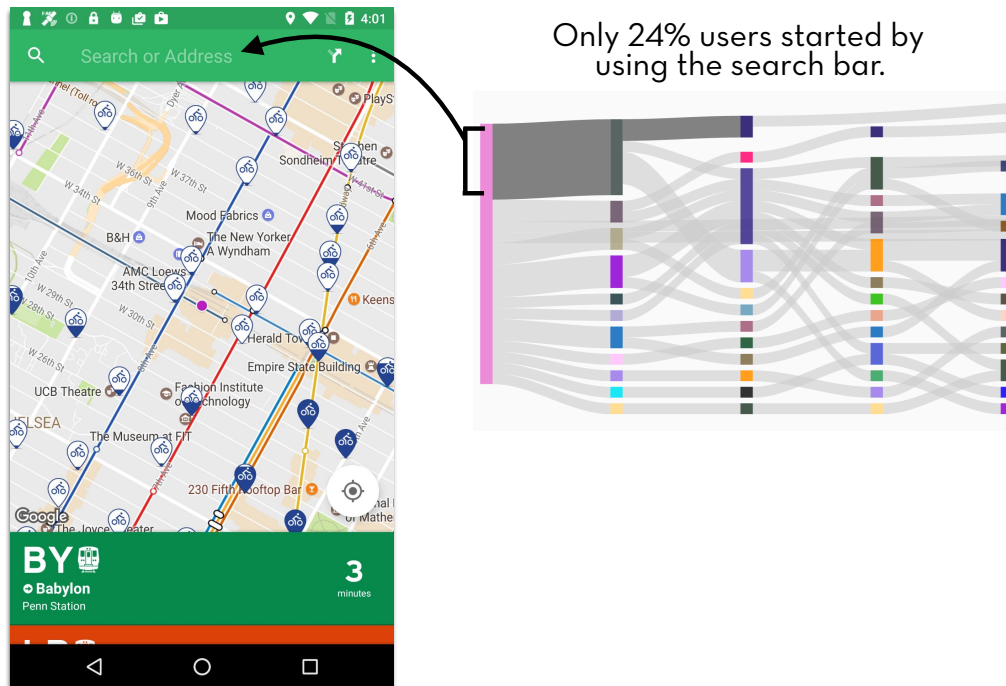


Figure 6.2: Although the easiest way to find the departure time of the next train from a station is to use the search bar, only 21% percent of *The Transit App* users did so initially.

gram) is not selected on the data entry screen. We hypothesize that the unit options are not easily recognizable as radio buttons, and positioning them to the side of the quantity field may have exacerbated the issue since users are accustomed to scanning forms vertically rather than horizontally [96].

By examining outliers — users who fail to complete a task after spending considerable time on it — ZIPT makes it possible to uncover usability issues that are not directly related to the specified task. For instance, we asked

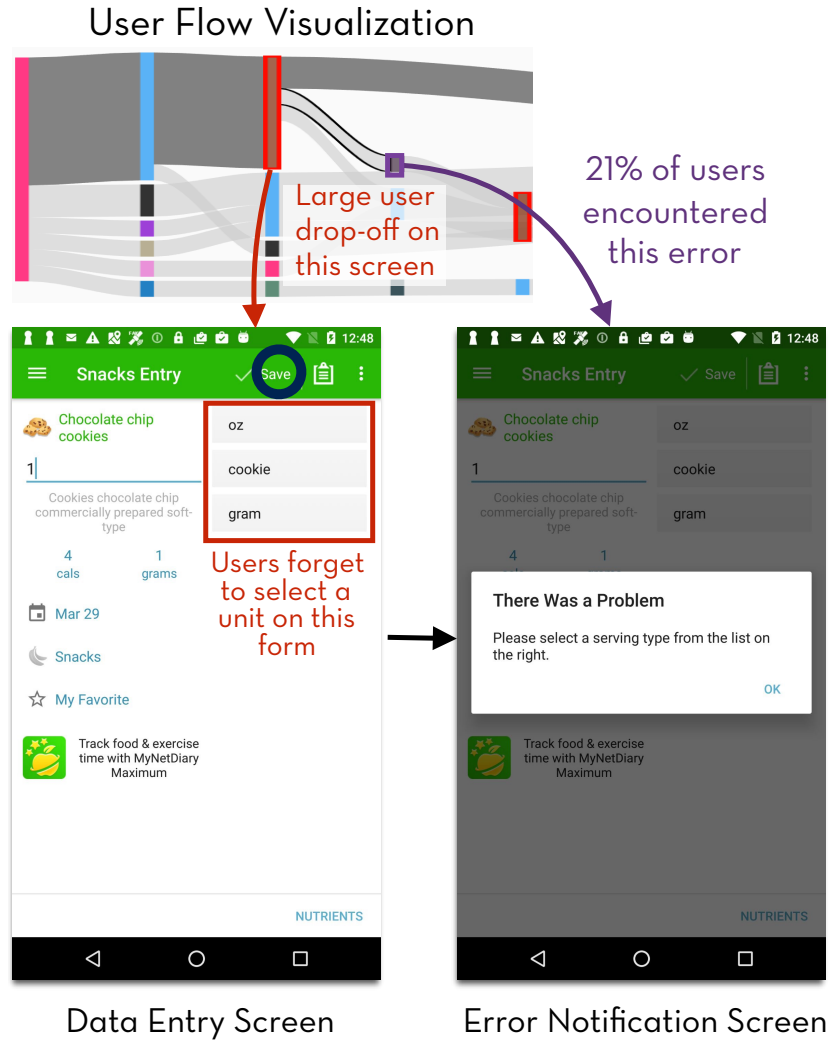


Figure 6.3: ZIPT helps designers discover usability issues in existing Android apps. The user flow visualization shown here summarizes the paths crowd workers took while trying to “add a cookie to their daily food log” in *MyNetDiary*, a calorie counter app. Using the flow visualization, designers were able to quickly identify a UI screen with a large user drop-off and its associated usability issues.

crowd workers to find a specific restaurant’s phone number on *Foursquare*, a social app for discovering places. Most crowd workers easily completed the task; however, we noticed that one user became stuck searching for the specified restaurant in the “lists” tab. *Foursquare* uses scoped search, a design pattern where search bars on different screens search over different types of entities. On *Foursquare*, the “search” tab supports search over places, while the “lists” tab supports search over user lists (Figure 6.4). Although scoped search is a popular design pattern, when the scope is not communicated clearly, it can lead to usability problems [12]. In this app, even though the scope changed between tabs, the placeholder text in the search box remained the same.

In another test, ZIPT helped uncover a usability issue on *Pinterest*, a social, visual inspiration app. *Pinterest* recently added a visual discovery feature that allows users to quickly search for similar images by clicking on a white circular icon on the bottom right corner of an image (Figure 6.5) [97]. We asked crowd workers to “pin an image to a board,” and then identified and inspected the trace of an outlier who had fixated on the visual discovery icon, tapping it several times on different images before abandoning the task. Commonly, an icon placed on an entity such as an image brings up a menu for additional actions that can be performed on it, we suspect that other users will have to “unlearn” their expected behavior on the *Pinterest* app.

6.4.2 Analyzing Comparative Performance

ZIPT affords designers — for the first time — the ability to perform comparative analysis at scale. During ideation, designers can leverage existing implementations to understand the relative performance of different design patterns. After implementation, ZIPT allows designers to benchmark against competitors’ apps. We demonstrate how ZIPT’s quantitative and qualitative performance data can be combined to conduct comparative performance analyses. Moreover, researchers can inspect flow visualizations and individual traces to better understand *why* these performance differences exist, uncovering usability issues, user expectations, and general trends in user behavior.

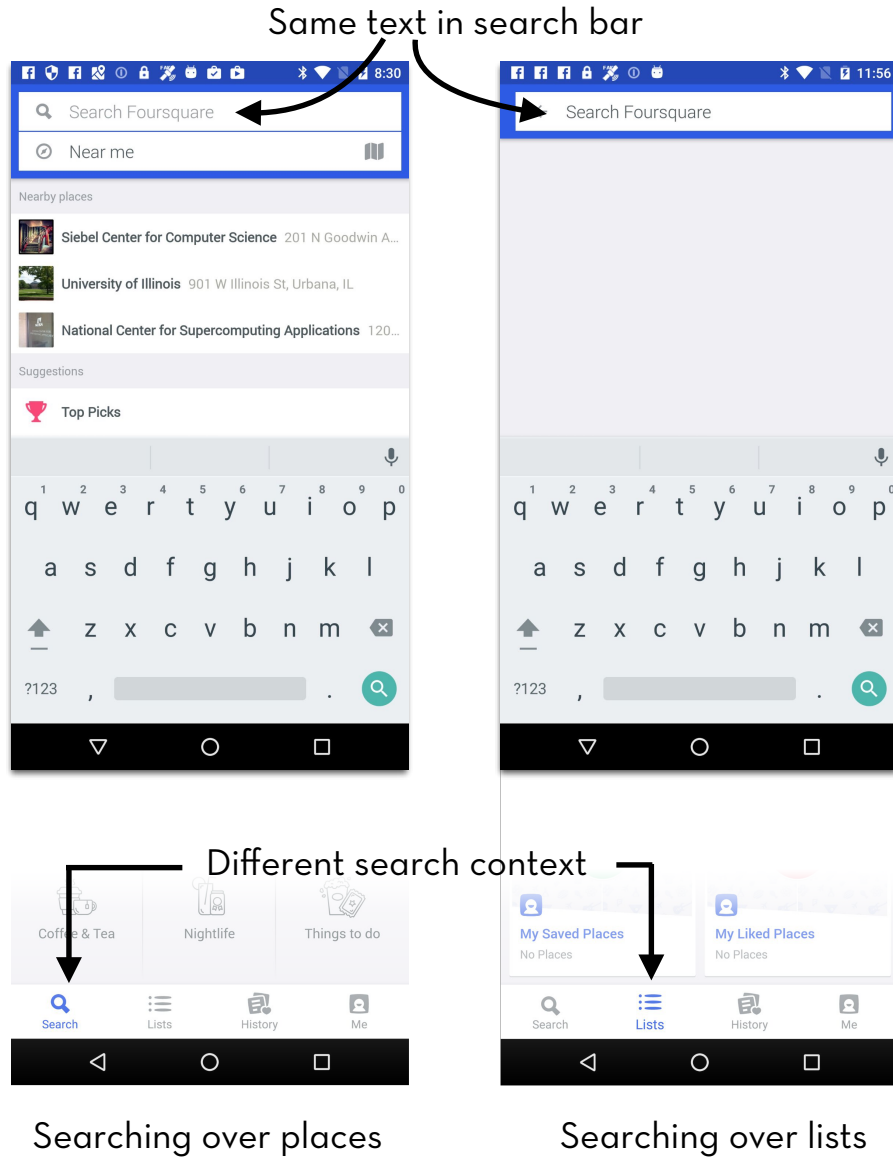


Figure 6.4: The *Foursquare* app presents search bars with different scope based on which tab is selected. However, the placeholder text does not change when tabs are switched to indicate a scope change, which can confuse users.

Finding Store Locations in Shopping Apps

Big-box retailer apps use different navigation patterns such as a *hamburger icons with flyout menus* and *tabbed navigation* to present their main capabilities (see Figure 6.6). To understand the relative performance of these patterns, we analyzed how users perform store location searches on two pop-

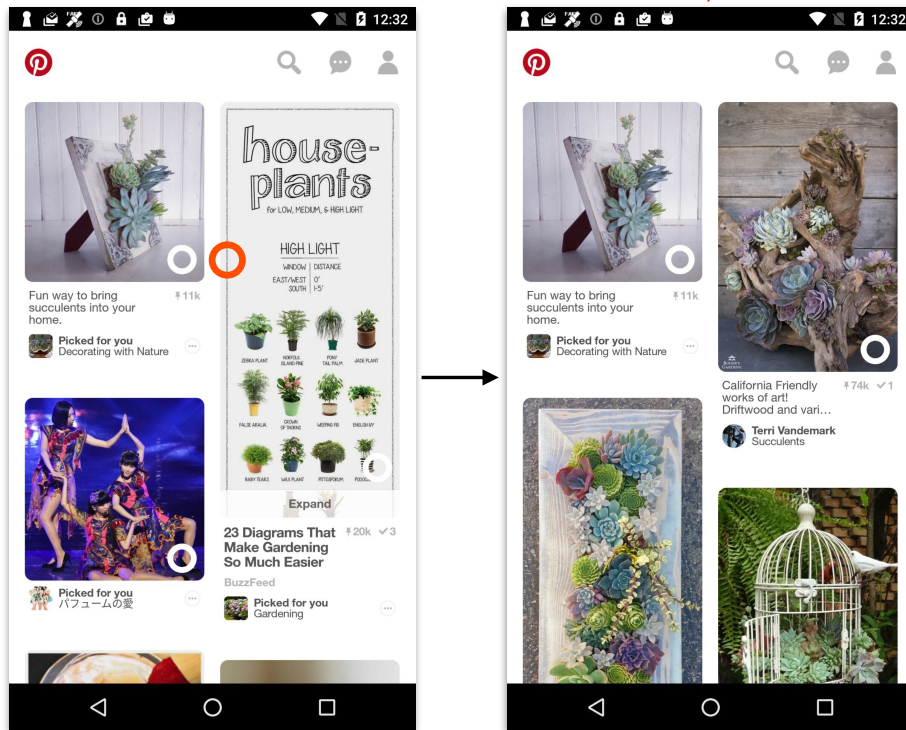
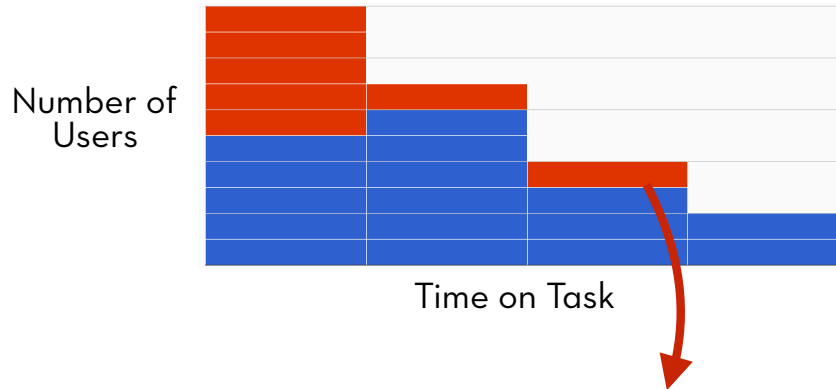
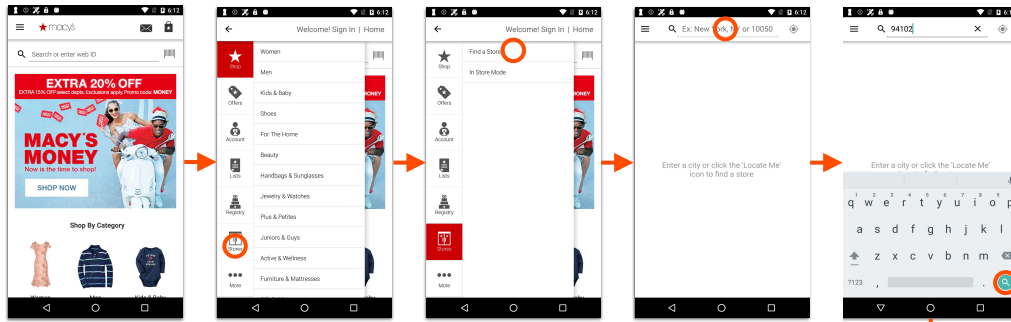


Figure 6.5: We discovered a potential usability issue on *Pinterest* by inspecting an outlier’s trace. After spending considerable time on the task, this user was unable to add an image to a board because the new visual search icon (white circle) behaved in an unexpected way.

ular retail apps — *Macy’s* and *Best Buy*. Finding a specific store location is an important feature of these apps, since it directly impacts revenue.

We used ZIPT to ask crowd workers “what is the address of the store to closest to the 94102 zip code?” We deployed the test for both the *Macy’s* and *Best Buy* apps and collected 31 and 35 traces, respectively. In the *Macy’s* app, there is only one golden path for answering the question: through a

Macy's allows one way to find a store by ZIP code



Best Buy allows two ways to complete the same task

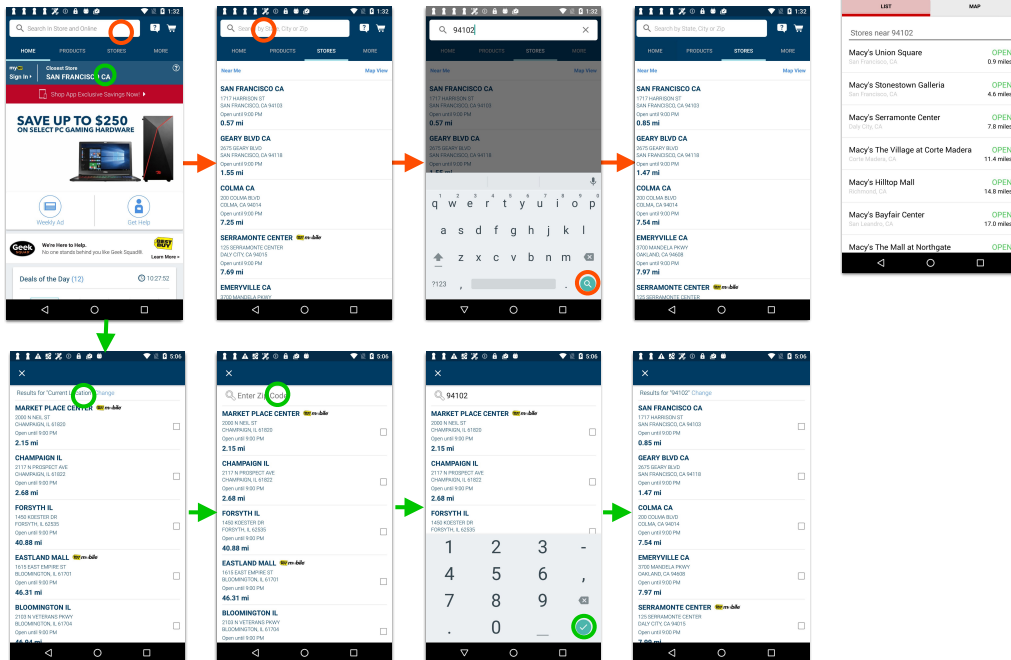
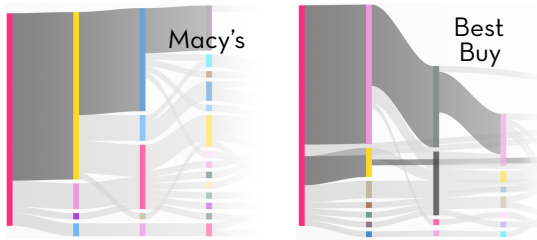


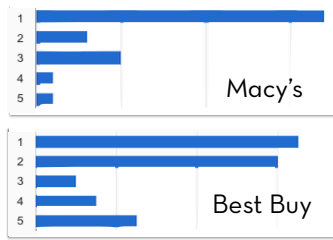
Figure 6.6: We asked crowd workers to find the address of a store location closest to a specified zip code on the *Macy's* and *Best Buy* apps. The *Macy's* app uses a hamburger icon with flyout menu navigation, whose sub-menu contains the store locator. The *Best Buy* app uses tabbed navigation and offers two ways to accomplish the task.

sub-menu in the flyout menu triggered by the hamburger icon (Figure 6.7). The *Best Buy* app, in contrast, offers two ways to perform this task. From the tabbed navigation, users can select the “Stores” tab and search for the specified ZIP code. Users can also click on the “Closest Store” location prominently displayed on the home screen and change the ZIP code to find the same information.

In both apps, majority of users took a first step that matched that of the designer



The task was harder on Best Buy than on Macy's



Inspecting the cause of an incorrect answer highlighted a potential issue with the search functionality in the Best Buy app

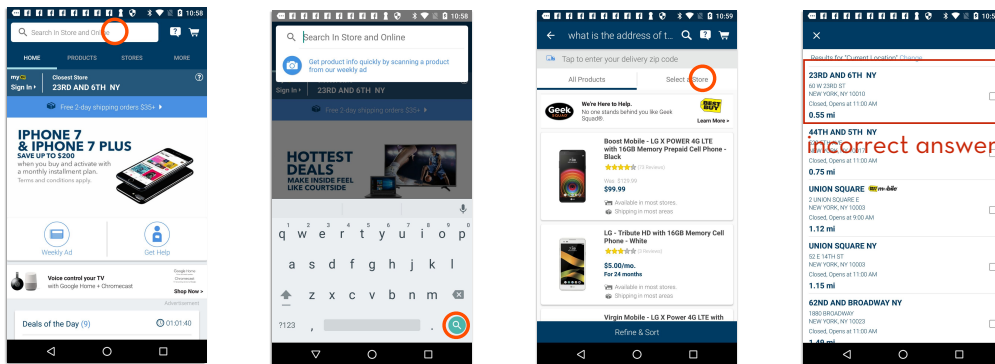


Figure 6.7: On average, users reported that the store location finding task to be more difficult to perform on the *Best Buy* app than on *Macy's*. Perhaps the hamburger icon is a more familiar design idiom than tabbed navigation with scoped search.

When we examined the data collected by ZIPT, we noticed that the task completion rate on the *Macy's* app (95.7%) was significantly higher than on *Best Buy* (82.5%). Moreover, on average, users reported that the task was harder to perform on the *Best Buy* app than on *Macy's*. By examining the user traces where crowdworkers submitted incorrect answers to the task question, we noticed one possible cause of confusion: users did not understand the *Best Buy* search bar's scope, which changes depending on the selected tab. Although *Best Buy* does change the search bar's placeholder text, some users still tried searching for stores on the "Home" tab, which returned zero results. Interestingly, 77% (24 out of 31) of *Macy's* app users took the correct first step of opening the flyout menu and following the design scent to find the store locator. Perhaps sticking to a familiar design idiom — the

hamburger icon — is more effective than having custom navigation.

Adding Events in Calendar Apps

Creating and adding events to a calendar is one of the most common tasks performed by users on scheduling apps. These apps generally provide action buttons on the home screen to support this essential feature. We picked two calendar apps — *Simple Calendar* and *DigiCal* — that offer slightly different action buttons. To study their relative performance, we created and deployed a ZIPT test for both apps, asking crowd workers to “add a new event to a calendar.” We collected 24 traces for *Simple Calendar*, and 18 traces for *DigiCal*.

Simple Calendar uses a distinct *floating action button* (FAB) that appears on the bottom right of the main screen (Figure 6.8). FABs are a popular design pattern and are part of the Android material design library [98]. They are recommended for highlighting promoted actions in an app and are frequently used to create new items (e.g., when composing a new email). *DigiCal*, in contrast, uses a less distinct + icon in the top action bar. Both apps allow users to complete the task by directly interacting with a time slot on the calendar, which requires two interactions to reach the screen where event details can be entered. By interacting with the action button, users can reach this screen in one step.

In both apps, we observed that most users attempted the task by directly manipulating the time slots on the calendar. This behavior is easy to spot in flow visualizations (Figure 6.8). Even though the FAB is high contrast and a popular pattern, only 17% (4 out of 24) of users interacted with it in *Simple Calendar*. The differences in action buttons had little effect on behavior, perhaps because users have a strong habit of manipulating calendar events directly.

Creating Playlists in Music Apps

Most music service apps support the creation of playlists for curating collections of songs. We studied the relative user performance of playlist creation flows in two popular music apps: *Spotify* and *YouTube Music*. We deployed a task on ZIPT asking crowd workers to “add two songs to a new playlist

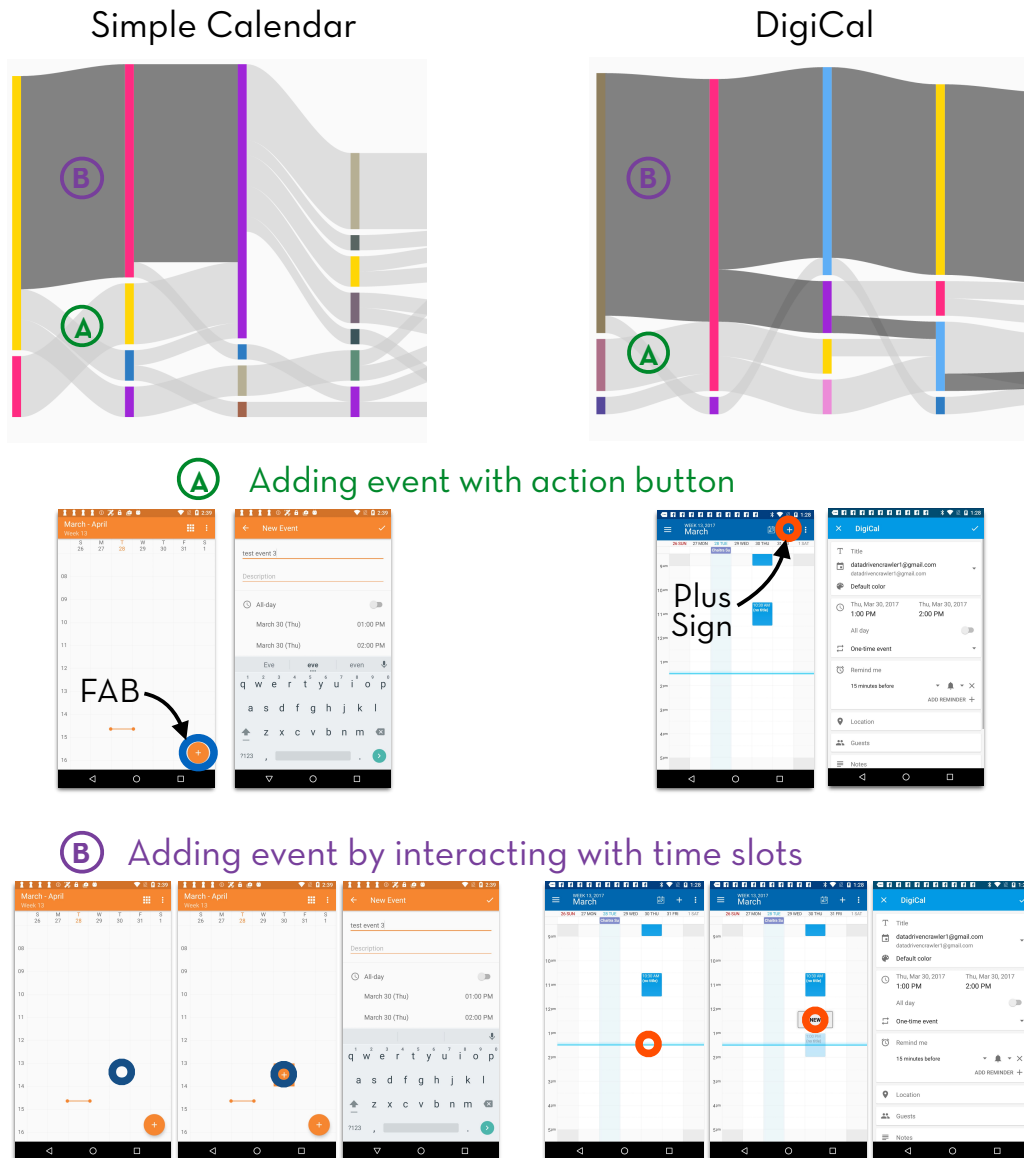
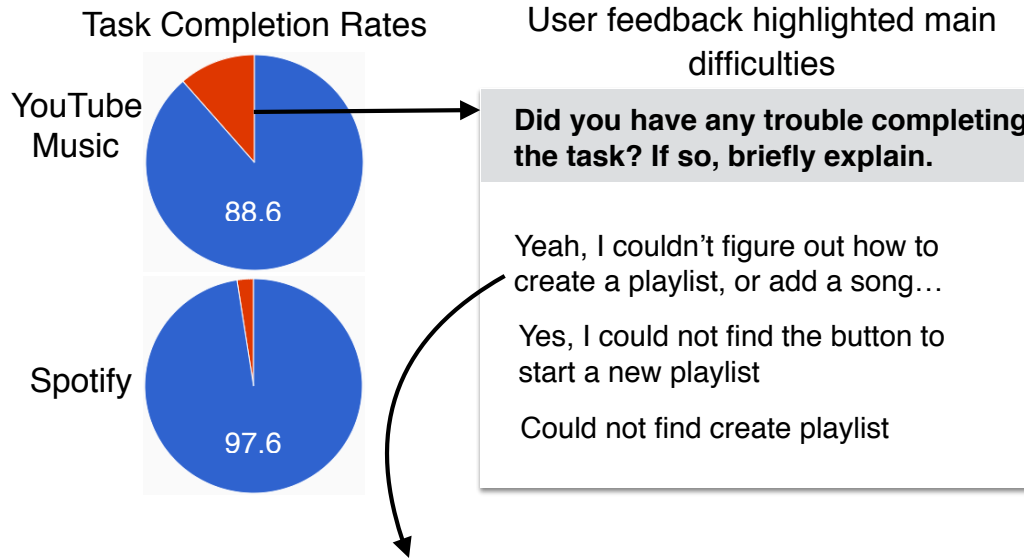


Figure 6.8: Both *Simple Calendar* and *DigiCal* provide two ways for users to add an event to their calendar: directly interacting with slots on the calendar or through an action button. Although the former strategy is slower than the latter, a larger percentage of users in both apps started the task using direct time slot manipulation.

that you create,” and collected 41 user traces for *Spotify* and 31 for *YouTube Music*.

We observed a higher task completion rate on *Spotify* than *YouTube Music* (92.9% versus 81.5% with $p = 0.17$) (Figure 6.9). Worker feedback on the task indicated that users were confused by being unable to create empty



A user turned to YouTube Music's help menu

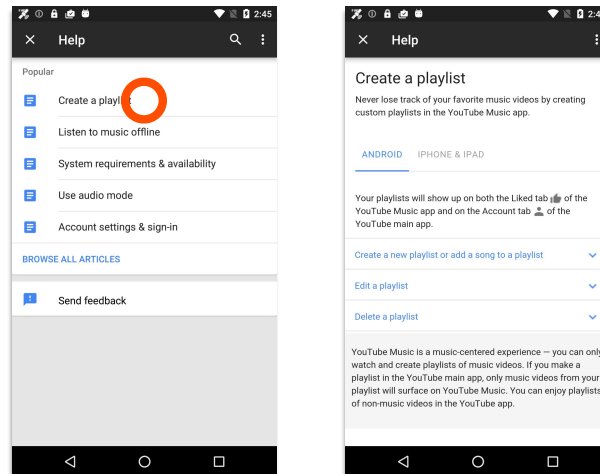


Figure 6.9: Feedback from the crowd explained why the completion rate of “adding two songs to a new playlist” was lower for *YouTube Music* than for *Spotify*. *YouTube Music* did not support creation of empty playlists, a capability expected by users.

playlists on *YouTube Music*. One individual trace revealed that a crowd worker even used the help functionality to learn how to create playlists in *YouTube Music*.

By inspecting the apps and the collected traces, we discovered that *Spotify* allows users to add a song to a new playlist in two ways. Users can “add to playlist” on a song and then specify that it is going to be part of a new playlist. Alternatively, they can “create new playlist” from a screen for man-

aging playlists, and then navigate to a song and add it. *YouTube Music* does not support the creation of empty playlists, which seems to be an expected capability.

6.5 Evaluation

We conducted user interviews to gauge the usefulness of ZIPT and to understand how the system might be integrated into the design process. We recruited four Google employees for the study: three of them were participants in our formative study (P3, P4, and P5), and the fourth (P6) was another UX researcher. The interviews lasted one hour and participants were compensated \$25.

Each interview comprised a system overview (30 minutes), system walkthrough (10 minutes), and discussion (20 minutes). During the system overview, we introduced the goals of the system and then demonstrated its features using examples from the case studies described in the previous sections. During the system walkthrough, we let each participant use the system to explore the data collected for the *MyNetDiary* calorie counter app. We then started a discussion about how each participant might incorporate the ZIPT system and data into their workflow.

After the interview, we sent participants a survey with 11 questions addressing the overall usefulness of the system and data presentation. The questions elicited responses on a 7-point Likert-scale from strongly disagree (1) to strongly agree (7). Three of the four participants completed the survey. We summarize the questions and their responses in Table 6.1.

Table 6.1: Participant feedback about the overall usefulness of ZIPT and its features.

	Question	Avg (S.D.)
Q1	The data produced by this system would be helpful in the app design process.	6.0 (0.0)
Q2	This system would help me understand what works well for design patterns that I see in existing apps.	6.0 (1.0)
Q3	This system would help me find unexpected user behavior.	6.0 (1.0)
Q4	This system would help me find usability bugs.	6.3 (1.2)
Q5	The data produced by this system would help me communicate the merits and demerits of a design pattern.	6.3 (0.6)
Q6	The metrics overview was helpful.	5.7 (1.2)
Q7	The flow visualization was helpful.	6.3 (1.2)
Q8	The user feedback was helpful.	6.0 (1.0)
Q9	The user interface was easy to use and learn.	6.0 (0.0)
Q10	I believe that the app MyNetDiary has a usability issue.	7.0 (0.0)
Q11	I would gather performance data for existing apps more often with this system.	6.7 (0.6)

6.5.1 Potential Uses Cases

Overall, participants reported that ZIPT would be helpful in their design process (Q1 to Q5). From the discussions, it was clear that participants valued the potential comparative use cases. P3 would use ZIPT to understand “if reusable components in different apps share similar usability issues.” P5 was interested in using ZIPT to benchmark his own apps against competitors’: “I think it fits nicely with how designers think about problems...We think a lot about critical user journeys...This is a way for us to focus on those journeys and benchmark them against similar apps.” He added, “If adding a record for food takes 10 seconds on my competitor’s app, that would be my target...If I have a set of core tasks that can be matched up with those in other apps, then I can understand how well I am doing.”

Participants found the aggregate data views were useful (Q6 to Q8) for both identifying user problems, and for understanding why they occur. P3 commented, “I really like that you can state your hypothesis about what the golden paths are and then you check to see if you were correct...and the ability to drill down into the flows where it didn’t happen and see in the traces where the derailing was, is very powerful.” Similarly, during the

demonstration of the feedback view, P4 remarked, “When you first showed me the task creation interface, I was dismissive of the open text fields, but when you are trying to come up with a theory of why things are going wrong, just skimming those things can be very useful.”

6.5.2 Perceived Benefits

Participants reported that they would use a system like ZIPT to collect performance data for existing apps more often (Q11). P4 commented that “Even small improvements that reduce the cost of user research have a big impact on people’s willingness to do research.” Participants appreciated that ZIPT reduced the monetary and engineering cost of testing and user research. P5 remarked that “It is way cheaper than usertesting.com.” He added, “The benefit (compared to building prototypes) is that you don’t have to create parts of the app that do not relate to the features that you want to test...like creating logins and realistic content...You can save time by using existing applications since all those things are already there.”

Participants realized that the structured data collection made the testing platform flexible and general. P6 stated, “You could do a UI evaluation or an interaction evaluation or a component evaluation. Each of those could benefit from an approach like this, especially once you have all the data, you can ask arbitrary questions of that data.” P5 viewed the ability to produce a “flow visualization using the view hierarchy” as ZIPT’s unique advantage. Participants also felt that the structured representation of data was complete. While viewing an individual trace, P4 remarked, “It seems there is very little that you lose from having an over-the-shoulder video camera.”

6.5.3 Generated Design Insights

After using ZIPT for 10 minutes to explore the user data for *MyNetDiary*, all participants reported that it had at least one usability issue (Q10, $\mu = 7.0$). P3 and P6 noticed the same issue we identified: the visual design of the data entry screen (Figure 6.3) did not clearly communicate that “something else was needed” (P6).

Participants identified additional usability issues as well. P4 and P5 men-

tioned that it was an “anti-pattern” that the error message was a global pop-up and not attached to the offending elements. After inspecting multiple traces, P3 noted, “There are a number of people here who expect that the flow should go to the bottom right when they are done instead of the save button in the top right.” Participants also spontaneously generated solutions to address some of these issues, including preselecting one of the options (P6, P3), restructuring the data entry as a two step process (P5), changing the component used for selection (P5), and modifying the layout of the screen to be diagonal (P3).

Participants felt comfortable generalizing from specific examples to broader design insights. P3 explained, “I am not a trained interaction designer, but even with the experience I have, I can look at the one problem case and I can infer from that what the general problem is.” Similarly, when we mentioned the case of users trying to create empty playlists in YouTube Music, P4 generalized that “People like to think of containers as things.” He named two specific products where he had seen similar user behavior in the past and added, “You want to be able to have an empty thing to put things in...Its how we think of stuff. You want to support the object-first use case where you start the collection from an object, but that’s not the first way people think of when you ask them to create the thing.”

6.6 Discussion and Future Work

The case studies described in this chapter illustrate that ZIPT can provide both the detailed data of usability testing and the scale of analytics-driven and A/B testing. ZIPT allows designers to combine quantitative and qualitative data to identify problems, quantify their magnitude, and gain insight into their causes. In the future, extending ZIPT to support more sophisticated statistical techniques could allow designers to specify a maximum budget and desired confidence interval for key performance indicators, instead of requiring them to mandate the number of user traces to collect.

More broadly, we envision ZIPT becoming part of an integrated mobile design exploration platform, where designers can search for examples using tools like the flow search system from Deka et al. [13], and then invoke ZIPT to understand their performance through a series of inexpensive, targeted

experiments. Observing how designers use ZIPT under realistic conditions would also help improve its task specification and data-analysis interfaces.

Although it was not designed to support this use case, ZIPT could also be an interesting platform for evaluating one's own apps. Initially, many app development teams do not have the resources to conduct usability testing, or the users to spend on A/B testing [99]. ZIPT could afford these teams the ability to iterate on their app and get low-cost, low-overhead usability feedback. In the words of one of our participants, "If it was just me, and it was my startup, and this was my application, after seeing this data, I would at least change the position of the save button and see if that makes the flow any better. I would be quite eager to do a couple of iterations until I get something that is as frictionless as possible."

CHAPTER 7

CONCLUSION

This dissertation demonstrates the possibility of mining designs for an interactive domain. It focuses on mobile app designs and demonstrates the value of being able to capture their designs at scale by developing novel data-driven design applications to assist designers in the app design process. In particular, it develops the following:

- Interaction mining for mobile apps: a technique to capture the multiple components of an apps design including both static and dynamic components
- ERICA and Rico: systems that implement interaction mining for Android apps using a black-box approach that make it possible to mine tens of thousands of apps
- Interaction trace: a representation of mobile app designs suitable for supporting data-driven design applications
- The Rico dataset: a publicly released dataset of designs from over 9700 Android apps with over 1200 interaction traces
- A content-agnostic similarity heuristic for detecting mobile UIs that represent the same design
- A deep-learning-based technique for capturing a lower dimensional representation of UI layouts useful for example-based searching
- A pipeline for automated identification and tagging of semantic user flows in interaction traces that allowed us to build the largest searchable repository of app designs
- Zero-integration performance testing: an approach for testing mobile apps that enables understanding performance characteristics of app

design examples for the first time and provides a low-cost, low-effort method for comparative testing at scale

7.1 Future Directions

7.1.1 Scaling Up Interaction Mining

There are benefits to be had from scaling up interaction mining beyond what we demonstrate in this dissertation . Larger datasets in general enable training more sophisticated models. For example, recent breakthroughs in supervised [74, 57] as well as unsupervised learning [100, 101, 102], have come from training on datasets with millions of examples. For applications based on design examples (such as example search), larger design datasets could provide a more diverse set of examples to draw from.

One potential way to scale up interaction mining would be to work with commercial app streaming providers. Google recently launched Android app streaming [103] as a way to let users use an app without the need to download it on their device. Multiple startups are also exploring the potential applications of such a usage model [104, 105]. If this app usage model becomes mainstream, a web-based approach like the one presented in this dissertation could enable interaction mining of mobile apps at a truly massive scale.

Another approach, could be to move beyond the web-based, paid crowd-worker approach used in this dissertation . Perhaps techniques for interaction mining on end-user devices could be developed. This could allow researchers to capture user behavior in context (such as when walking) and allow understanding how app usage is affected by different usage scenarios. Another potential benefit could be that the cost of design data acquisition could be lower than what it is for crowdsourcing. This might make it feasible to collect design data longitudinally and create a dataset that captures design change over time. However, for on-device mining to be adopted by a large number of users, suitable incentives such as those offered by market research firms [87] or consumer software companies [24] would have to be devised. Adequate privacy preserving measures will also have to be designed.

7.1.2 Interaction Mining in Other Interactive Domains

Interaction mining is the first technique to demonstrate the possibility of design mining for artifacts whose designs are dynamic and change depending on user interactions. Although this dissertation focused on using interaction mining for mobile apps, the general approach of combining a visual and a structural representation of states with data about user interactions and how states change based on these interactions is applicable to other interactive domains as well. For instance, prior work in analyzing designs on the web have treated web pages as static artifacts [3, 83, 68]. However, the vast majority of websites today are dynamic – ranging from websites that have a few Javascript enabled interactions to single-page web applications with highly interactive frontends [106, 107]. Interaction mining could be implemented for such dynamic webpages and web applications by capturing the DOM (document object model) trees [25, 3] as a structural representation of their UIs. It would also be interesting to see how far this approach can go in capturing the designs of VR, AR or mixed-reality interactions that are increasing in popularity.

7.1.3 Unified Design Exploration Platform

Another direction for our work is towards an online app exploration platform that unifies design search and testing. It would allow designers to run inexpensive, targeted tests on examples they find and make the test results available for everyone else to search over. Test results could also be aggregated to uncover design patterns that are effective across apps and help build theories of what makes them effective. Such a platform could democratize the design knowledge that remains embedded in individual apps today, and make it available for everyone to learn from.

7.1.4 Testbed for Studying Factors that Affect Usability

The web-based architecture for interacting with apps presented in this dissertation could be used to artificially alter the app usage experience and to allow researchers to conduct large-scale studies on different factors that affect usability and user performance of mobile apps. For example, the effect of

system delays on the usability and user engagement could be studied by introducing artificial delays in streaming the app screens to the web frontend [108]. Effects of color blindness [109, 110] could be studied by dynamically changing the streamed images. The effect of fat fingers [111] or limited dexterity could be simulated by suitably modifying the frontend web interface.

7.1.5 Additional Data-Driven Mobile App Design Applications

Interaction mining can be used to enable data-driven design applications beyond the ones developed in this dissertation. Below we discuss four such applications that the current dataset can support. Table 7.1 summarizes the different dataset components that support each application.

UI Layout Generation

Design datasets are also useful for training generative models of design. Prior work has trained generative models for arranging design elements and defining their attributes in domains such as graphic design [112], 3D modeling [6], and the web [6]. Generation of UI layouts, in particular, has been extensively studied using constraint-based systems [113], model-based systems [114, 115, 116, 117, 118], energy-based models [112], human-performance models [65], and Bayesian grammars [6].

Table 7.1: The five classes of design applications that the Rico dataset supports, correlated with the parts of the dataset intended to support them.

	Design Search	Mobile Layout Generation	UI Code Generation	User Interaction Modeling	User Perception Prediction
UI Screenshots	●	●	●	●	●
View Hierarchies	●	●	●	●	○
User Interactions	●	○	○	●	○
Animations	●	○	○	○	●
UI Similarity Annotations	●	○	○	○	○
App Store Metadata	●	●	○	○	●

Researchers can use the Rico dataset to train such models of mobile UI layouts. The Android view hierarchy contains all the elements comprising a UI screen, their attributes (e.g., position, dimensions), and the structural relationships between them. By combining screenshots and view hierarchies, researchers can compute visual features such as the color contrast between nested elements. Additionally, play store metadata can be leveraged to create specialized training sets. For example, app ratings and download metrics can be used as proxies for design quality, so that models can be optimized to emulate “good” layouts. Models can also be trained to take into account app category or UI and element semantics since layouts vary based on those factors (Figure 7.1).

UI Code Generation

Once a mobile app is designed, implementing its interfaces and interactions in code can be a time-consuming process. As a result, prior work has developed methods to automatically generate code from UI designs. REMAUI (Reverse Engineering Mobile Application User Interfaces), for instance, used computer vision and OCR to first segment an input image and then used heuristics for code generation [66]. pix2code used neural network-based models to directly generate Android code from UI images [67].

Design systems can leverage the Rico dataset to reverse engineer UIs (Figure 7.2). Since the Rico dataset contains both the screenshots and the Android components (in the view hierarchies) of a UI screen, it is an ideal dataset to refine heuristics (such as those used in REMAUI) or train models

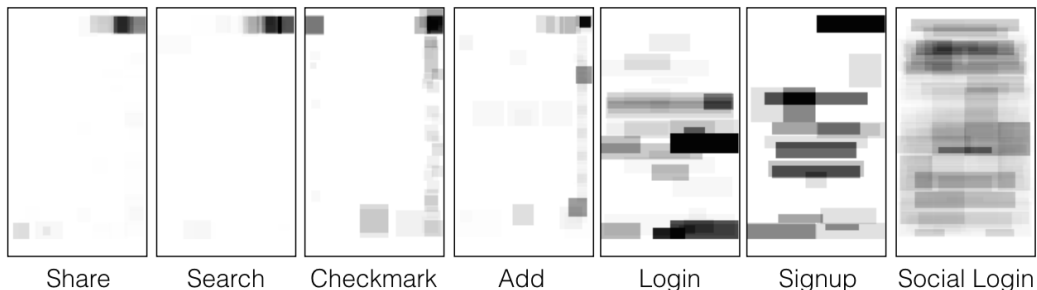


Figure 7.1: Different semantic elements are laid out differently in mobile UIs. Here we show heatmaps of seven popular types of elements in the ERICA dataset.

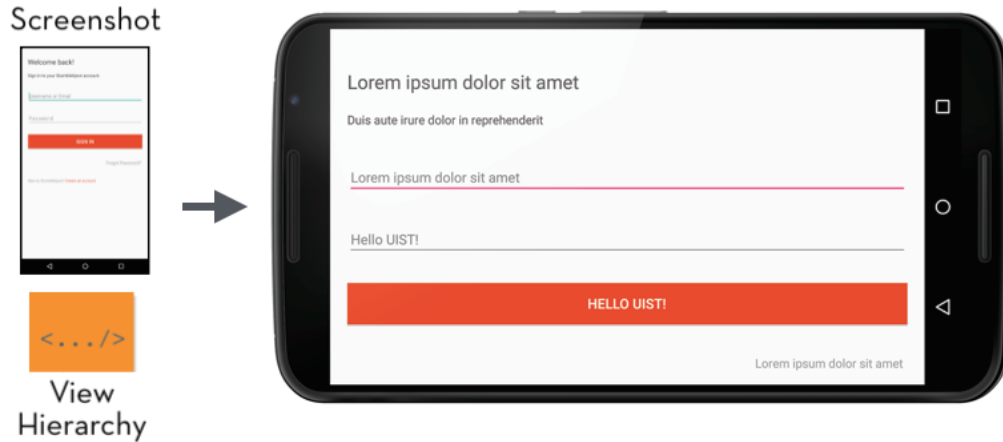


Figure 7.2: The representations of UIs captured by interaction mining can be used to generate Android source code that recreates the UI. We reverse engineered a UI from the app *StumbleUpon* and generated the Android source code, which is then rendered on an emulator with a different form factor.

used for code generation (such as the ones used in *pix2code*).

User Interaction Modeling

A key component of a mobile app’s design is the interactivity of its various UIs and elements [13]. Modeling how users interact with different UIs can support better automated testing for apps as well as app optimizations that pre-fetch data by predicting a user’s next action.

Rico captures user interaction data while an app is being. Each user trace for an app contains every user interaction event annotated with its type (such as “tap” or “scroll”) and the UI element that reported it. By finding the same UI element in the corresponding view hierarchy, models can learn from a richer set of features based on the element’s properties and metadata. Instead of predicting that a user will click on an element in the top-right corner of the screen, models can predict that a user will “check out”.

User Perception Prediction

Models of user perception help designers get early feedback on their designs. Prior work has explored models for predicting users’ first impressions of web pages [68, 119], mobile app screens [70, 120], and mobile app icons [121].

Other research has focused on predicting longer-term perception based on animations [26], menus [27], and visual diversity and consistency between different screens [28, 29].

To build perceptual models of mobile design using the Rico dataset, systems can compute features over UI screenshots and animations, and correlate them with play store metrics. For example, researchers could examine correlations between an app’s color palette and its average rating. In the future, researchers could use these screenshots and animations to crowdsource additional perceptual annotations.

7.1.6 Assessing Data-Driven Design Tools

Although design mining has enabled the creation of data-driven design tools that have the potential to assist design professionals in their work, their actual usage has not been studied. Longitudinal studies with design practitioners could help understand the impact such tools have on the creative process. Such research could study how effective these tools are in practice, how designers modify their practices to use such tools, adverse impacts such tools could have (such as lowering diversity of generated designs), and identify directions future tools should pursue.

7.1.7 Non-Professional Use Cases

The tools developed in this dissertation could also be useful outside of traditional app design teams that they were built to support. For instance, many apps are released by lone developers with no design experience and a limited budget. A system like ZIPT could help such developers (or novice designers) learn the essentials of design most useful for the app they are working on. Such learning benefits could also extend to instructional settings. Instructors of visual and interaction design classes can use search systems to find relevant examples to illustrate concepts they are teaching. In addition, they can illustrate with data which patterns work and under what conditions. Future work could study these use cases that were not explored in this dissertation.

REFERENCES

- [1] M. Gualtieri, “Best Practices In User Experience (UX) Design,” 2009. [Online]. Available: <https://www.forrester.com/report/Best+Practices+In+User+Experience+UX+Design/-/E-RES54101>
- [2] “UI, UX: Who Does What? A Designer’s Guide to the Tech Industry,” 2014, <http://www.fastcodesign.com/3032719/ui-ux-who-does-what-a-designers-guide-to-the-tech-industry>.
- [3] R. Kumar, A. Satyanarayan, C. Torres, M. Lim, S. Ahmad, S. R. Klemmer, and J. O. Talton, “Webzeitgeist: Design mining the web,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2470654.2466420> pp. 3083–3092.
- [4] C. Eckert and M. Stacey, “Sources of inspiration: A language of design,” *Design Studies*, vol. 21, no. 5, pp. 523–538, 2000.
- [5] R. He and J. McAuley, “Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering,” in *Proceedings of the 25th International Conference on World Wide Web*, 2016, pp. 507–517.
- [6] J. Talton, L. Yang, R. Kumar, M. Lim, N. Goodman, and R. Měch, “Learning design patterns with Bayesian grammar induction,” in *Proceedings of the 25th annual ACM symposium on User interface software and technology*. ACM, 2012.
- [7] M. E. Yumer, P. Asente, R. Mech, and L. B. Kara, “Procedural modeling using autoencoder networks,” in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, 2015, pp. 109–118.
- [8] A. Xu, S.-W. Huang, and B. Bailey, “Voyant: Generating structured feedback on visual designs using a crowd of non-experts,” in *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, 2014. [Online]. Available: <http://doi.acm.org/10.1145/2531602.2531604> pp. 1433–1444.

- [9] “Thingiverse,” 2017, <https://www.thingiverse.com>.
- [10] D. Gentner, S. Brem, R. W. Ferguson, A. B. Markman, B. B. Levi-dow, P. Wolff, and K. D. Forbus, “Analogical reasoning and conceptual change: A case study of Johannes Kepler,” *The Journal of the Learning Sciences*, vol. 6, no. 1, pp. 3–40, 1997.
- [11] C. Eckert, M. Stacey, and C. Earl, “References to past designs,” *Studying Designers*, vol. 5, pp. 3–21, 2005.
- [12] “20 Best Practices for Mobile App Search,” 2017, <https://www.raywenderlich.com/153260/20-best-practices-for-mobile-app-search>.
- [13] B. Deka, Z. Huang, and R. Kumar, “ERICA: Interaction mining mobile apps,” in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, 2016. [Online]. Available: <http://doi.acm.org/10.1145/2984511.2984581> pp. 767–776.
- [14] B. Deka, Z. Huang, C. Franzen, J. Hibschan, D. Afergan, Y. Li, J. Nichols, and R. Kumar, “Rico: A mobile app dataset for building data-driven design applications,” in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, 2017. [Online]. Available: <http://doi.acm.org/10.1145/3126594.3126651> pp. 845–854.
- [15] B. Deka, Z. Huang, C. Franzen, J. Nichols, Y. Li, and R. Kumar, “ZIPT: Zero-integration performance testing of mobile app designs,” in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, 2017. [Online]. Available: <http://doi.acm.org/10.1145/3126594.3126647> pp. 727–736.
- [16] “iOS - Apple,” <https://www.apple.com/ios>.
- [17] “Android,” <https://www.android.com/>.
- [18] “App Annie 2016 Retrospective Mobiles Continued Momentum,” <https://www.appannie.com/en/insights/market-data/app-annie-2016-retrospective/>.
- [19] “Invision,” <https://www.invisionapp.com/>.
- [20] “Origami,” <https://origami.design/>.
- [21] “Pixate,” <http://www.pixate.com/>.
- [22] J. Nielsen, “Finding usability problems through heuristic evaluation,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1992. [Online]. Available: <http://doi.acm.org/10.1145/142750.142834> pp. 373–380.

- [23] “Application Fundamentals,” <https://developer.android.com/guide/components/fundamentals.html>.
- [24] “Apktool: A tool for reverse engineering Android APK files,” <https://github.com/iBotPeaches/Apktool>.
- [25] “Document Object Model (DOM),” <https://www.w3.org/DOM/>.
- [26] N. Tractinsky, O. Inbar, O. Tsimhoni, and T. Seder, “Slow down, you move too fast: Examining animation aesthetics to promote eco-driving,” in *Proceedings of the 3rd International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, 2011. [Online]. Available: <http://doi.acm.org/10.1145/2381416.2381447> pp. 193–202.
- [27] S. Leuthold, P. Schmutz, J. A. Bargas-Avila, A. N. Tuch, and K. Opwis, “Vertical versus dynamic menus on the world wide web: Eye tracking study measuring the influence of menu design and task complexity on user performance and subjective preference,” *Computers in Human Behavior*, vol. 27, no. 1, pp. 459–472, 2011.
- [28] T. van der Geest and N. Loorbach, “Testing the visual consistency of web sites,” *Technical Communication*, vol. 52, no. 1, pp. 27–36, 2005.
- [29] A. Miniukovich and A. De Angeli, “Visual diversity and user interface quality,” in *Proceedings of the 2015 British HCI Conference*, 2015. [Online]. Available: <http://doi.acm.org/10.1145/2783446.2783580> pp. 101–109.
- [30] S. E. S. Taba, I. Keivanloo, Y. Zou, J. Ng, and T. Ng, “An exploratory study on the relation between user interface complexity and the perceived quality,” in *Web Engineering*. Springer, 2014, pp. 370–379.
- [31] I. M. Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. Hassan, “On the relationship between the number of ad libraries in an Android app and its rating,” *IEEE Software*, no. 1, pp. 1–1.
- [32] E. Guzman and W. Maalej, “How do users like this feature? a fine grained sentiment analysis of app reviews,” in *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*. IEEE, 2014, pp. 153–162.
- [33] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall, “What would users change in my app? summarizing app reviews for recommending software changes,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 499–510.

- [34] H. Khalid, “On identifying user complaints of iOS apps,” in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 1474–1476.
- [35] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, “What do mobile app users complain about?” *IEEE Software*, vol. 32, no. 3, pp. 70–77, 2015.
- [36] W. Maalej and H. Nabil, “Bug report, feature request, or simply praise? on automatically classifying app reviews,” in *Proceedings of the 23rd International Conference on Requirements Engineering*. IEEE, 2015, pp. 116–125.
- [37] S. McIlroy, N. Ali, and A. E. Hassan, “Fresh Apps: An empirical study of frequently-updated mobile apps in the Google Play Store,” *Empirical Software Engineering*, vol. 21, no. 3, pp. 1346–1370, Jun 2016. [Online]. Available: <https://doi.org/10.1007/s10664-015-9388-2>
- [38] S. McIlroy, W. Shang, N. Ali, and A. Hassan, “Is it worth responding to reviews? a case study of the top free apps in the Google Play Store,” *IEEE Software*, no. 1, pp. 1–1.
- [39] S. Hassan, C. Tantithamthavorn, C.-P. Bezemer, and A. E. Hassan, “Studying the dialogue between users and developers of free apps in the Google Play Store,” *Empirical Software Engineering*, 2017. [Online]. Available: <https://doi.org/10.1007/s10664-017-9538-9>
- [40] K. Alharbi and T. Yeh, “Collect, decompile, extract, stats, and diff: Mining design pattern changes in Android apps,” in *Proc. MobileHCI*, 2015.
- [41] A. Sahami Shirazi, N. Henze, A. Schmidt, R. Goldberg, B. Schmidt, and H. Schmauder, “Insights into layout patterns of mobile user interfaces by an automatic analysis of Android apps,” in *Proceedings of the Engineering Interactive Computing Systems Conference*, 2013.
- [42] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, “What are the characteristics of high-rated apps? a case study on free Android applications,” in *Proceedings of the International Conference on Software Maintenance and Evolution*, 2015.
- [43] K. Li, Z. Xu, and X. Chen, “A platform for searching UI component of Android application,” in *Proceedings of the International Conference on Digital Heritage*, 2014.

- [44] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna, “Challenges for dynamic analysis of iOS applications,” in *Proceedings of the 2011 IFIP WG 11.4 International Conference on Open Problems in Network Security*, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27585-2_6 pp. 65–77.
- [45] A. Machiry, R. Tahiliani, and M. Naik, “Dynodroid: An input generation system for Android apps,” in *Proceedings of the Symposium on the Foundations of Software Engineering*, 2013.
- [46] V. Rastogi, Y. Chen, and W. Enck, “AppsPlayground: Automatic security analysis of smartphone applications,” in *Proceedings of the Conference on Data and Applications Security and Privacy*, 2013.
- [47] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall, “Brahmastra: Driving apps to test the security of third-party components,” in *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2671225.2671290> pp. 1021–1036.
- [48] A. Mesbah, A. van Deursen, and S. Lenseslink, “Crawling Ajax-based web applications through dynamic analysis of user interface state changes,” *ACM Transactions on the Web*, vol. 6, no. 1, pp. 3:1–3:30, Mar. 2012.
- [49] “OpenSTF: Smartphone Test Farm,” 2016, <https://openstf.io/>.
- [50] “Minitouch: Minimal multitouch event producer for Android,” 2016, <https://github.com/openstf/minitouch>.
- [51] “Android UI Automator Framework,” 2016, <http://developer.android.com/tools/help/uiautomator/index.html>.
- [52] “Minicap: Stream Real-time Screen Capture Data Out of Android Devices,” 2016, <https://github.com/openstf/minicap>.
- [53] “Detecting common gestures,” 2016, <https://developer.android.com/training/gestures/detector.html>.
- [54] “Android Activities,” 2016, <https://developer.android.com/guide/components/activities.html>.
- [55] T. Azim and I. Neamtiu, “Targeted and depth-first exploration for systematic testing of Android apps,” in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2509136.2509549> pp. 641–660.

- [56] S. Amini, “Analyzing mobile app privacy using computation and crowdsourcing,” Ph.D. thesis, Carnegie Mellon University, 2014.
- [57] S. Bell and K. Bala, “Learning visual similarity for product design with convolutional neural networks,” *ACM Transactions on Graphics*, 2015.
- [58] S. Komarov, K. Reinecke, and K. Z. Gajos, “Crowdsourcing performance evaluations of user interfaces,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2470654.2470684> pp. 207–216.
- [59] K. Lee, J. Flinn, T. Giuli, B. Noble, and C. Peplin, “AMC: Verifying user interface properties for vehicular applications,” in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2462456.2464459> pp. 1–12.
- [60] “Database of Android apps on kaggle,” 2016, <https://www.kaggle.com/orgesleka/android-apps>.
- [61] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, and N. Sadeh, “Why People Hate Your App: Making Sense of User Feedback in a Mobile App Store,” in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2487575.2488202> pp. 1276–1284.
- [62] N. Viennot, E. Garcia, and J. Nieh, “A measurement study of google play,” in *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1. ACM, 2014, pp. 221–233.
- [63] M. Frank, B. Dong, A. P. Felt, and D. Song, “Mining permission request patterns from Android and Facebook applications,” in *Proceedings of the International Conference on Data Mining*, 2012.
- [64] P. O’Donovan, A. Agarwala, and A. Hertzmann, “DesignScape: Design with interactive layout suggestions,” in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, 2015. [Online]. Available: <http://doi.acm.org/10.1145/2702123.2702149> pp. 1221–1224.
- [65] K. Todi, D. Weir, and A. Oulasvirta, “Sketchplore: Sketch and explore with a layout optimiser,” in *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*, 2016. [Online]. Available: <http://doi.acm.org/10.1145/2901790.2901817> pp. 543–555.

- [66] T. A. Nguyen and C. Csallner, “Reverse engineering mobile application user interfaces with REMAUI,” in *Proceedings of the International Conference on Automated Software Engineering*, 2015.
- [67] T. Beltramelli, “pix2code: Generating code from a graphical user interface screenshot,” *Computing Research Repository*, vol. abs/1705.07962, 2017. [Online]. Available: <http://arxiv.org/abs/1705.07962>
- [68] K. Reinecke, T. Yeh, L. Miratrix, R. Mardiko, Y. Zhao, J. Liu, and K. Z. Gajos, “Predicting users’ first impressions of website aesthetics with a quantification of perceived visual complexity and colorfulness,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2470654.2481281> pp. 2049–2058.
- [69] L. Harrison, K. Reinecke, and R. Chang, “Infographic aesthetics: Designing for the first impression,” in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, 2015. [Online]. Available: <http://doi.acm.org/10.1145/2702123.2702545> pp. 1187–1190.
- [70] A. Miniukovich and A. De Angeli, “Computation of interface aesthetics,” in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, 2015. [Online]. Available: <http://doi.acm.org/10.1145/2702123.2702575> pp. 1163–1172.
- [71] Z. Bylinskii, N. W. Kim, P. O’Donovan, S. Alsheikh, S. Madan, H. Pfister, F. Durand, B. Russell, and A. Hertzmann, “Learning visual importance for graphic designs and data visualizations,” in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, 2017. [Online]. Available: <http://doi.acm.org/10.1145/3126594.3126653> pp. 57–69.
- [72] Y. Bengio, “Learning deep architectures for AI,” *Foundations and Trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, Jan. 2009. [Online]. Available: <http://dx.doi.org/10.1561/22000000006>
- [73] V. Nair and G. E. Hinton, “Rectified linear units improve restricted Boltzmann machines,” in *Proceedings of the International Conference on Machine Learning*, 2010, pp. 807–814.
- [74] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, 2012. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2999134.2999257> pp. 1097–1105.

- [75] J. L. Kolodner, “An introduction to case-based reasoning,” *Artificial Intelligence Review*, vol. 6, no. 1, pp. 3–34, 1992.
- [76] S. B. Paletz, K. H. Kim, C. D. Schunn, I. Tollinger, and A. Vera, “Reuse and recycle: The development of adaptive expertise, routine expertise, and novelty in a large research team,” *Applied Cognitive Psychology*, vol. 27, no. 4, pp. 415–428, 2013.
- [77] T. A. Dutton and L. L. Willenbrock, *The Design Studio: An Exploration of its Traditions and Potential*. Taylor & Francis, 1989.
- [78] “Dribbble - Show and tell for designers,” <https://dribbble.com/>.
- [79] “Behance - Showcase and discover creative work,” <https://www.behance.net/>.
- [80] “UX Archive,” 2016, <http://www.uxarchive.com/>.
- [81] “Pptrns,” 2016, <http://www.pptrns.com/>.
- [82] “Review: UX Archive.” 2013, <https://uxmag.com/articles/review-ux-archive>.
- [83] D. Ritchie, A. A. Kejriwal, and S. R. Klemmer, “D.tour: Style-based exploration of design example galleries,” in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, 2011. [Online]. Available: <http://doi.acm.org/10.1145/2047196.2047216> pp. 165–174.
- [84] Y. Hashimoto and T. Igarashi, “Retrieving web page layouts using sketches to support example-based web design.” in *2nd Eurographics Workshop on Sketch-Based Interfaces and Modeling*, 2005.
- [85] R. King, E. Churchill, and C. Tan, *Designing with Data*. O’Reilly Media, 2017.
- [86] J. Cardello, “Usability Metrics,” 2013, <https://www.nngroup.com/articles/usability-metrics/>.
- [87] J. Nielsen, “Usability Metrics,” 2001, <https://www.nngroup.com/articles/usability-metrics/>.
- [88] C. Grossauer, C. Holzmann, D. Steiner, and A. Guetz, “Interaction visualization and analysis in automation industry,” in *Proceedings of the 14th International Conference on Mobile and Ubiquitous Multimedia*, 2015. [Online]. Available: <http://doi.acm.org/10.1145/2836041.2841218> pp. 407–411.

- [89] Z. Qin, Y. Tang, E. Novak, and Q. Li, “Mobiplay: A remote execution based record-and-replay tool for mobile applications,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. New York, NY, USA: ACM, 2016. [Online]. Available: <http://doi.acm.org/10.1145/2884781.2884854> pp. 571–582.
- [90] “Introducing Android Instant Apps,” <http://android-developers.blogspot.com/2016/05/android-instant-apps-evolving-apps.html>.
- [91] S. Chang, P. Dai, L. Hong, C. Sheng, T. Zhang, and E. H. Chi, “AppGrouper: Knowledge-based interactive clustering tool for app search results,” in *Proceedings of the 21st International Conference on Intelligent User Interfaces*, 2016. [Online]. Available: <http://doi.acm.org/10.1145/2856767.2856783> pp. 348–358.
- [92] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich, “Soylent: A word processor with a crowd inside,” in *Proceedings of the 23rd Annual ACM Symposium on User Interface Software and Technology*, 2010. [Online]. Available: <http://doi.acm.org/10.1145/1866029.1866078> pp. 313–322.
- [93] A. J. Quinn and B. B. Bederson, “Human computation: A survey and taxonomy of a growing field,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2011. [Online]. Available: <http://doi.acm.org/10.1145/1978942.1979148> pp. 1403–1412.
- [94] A. Kittur, J. V. Nickerson, M. Bernstein, E. Gerber, A. Shaw, J. Zimmerman, M. Lease, and J. Horton, “The future of crowd work,” in *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2441776.2441923> pp. 1301–1318.
- [95] S. Dow, A. Kulkarni, S. Klemmer, and B. Hartmann, “Shepherding the crowd yields better work,” in *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, 2012. [Online]. Available: <http://doi.acm.org/10.1145/2145204.2145355> pp. 1013–1022.
- [96] “Web form design guidelines: An eyetracking study,” 2009, https://www.cxpathners.co.uk/our-thinking/web_forms_design_guidelines_an_eyetracking_study/.
- [97] “Search outside the box with new Pinterest visual discovery tools,” 2017, <https://blog.pinterest.com/en/search-outside-box-new-pinterest-visual-discovery-tools>.
- [98] “Material design for Android,” 2017, <https://developer.android.com/design/material/>.

- [99] S. Iitsuka and Y. Matsuo, “Website optimization problem and its solutions,” in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015. [Online]. Available: <http://doi.acm.org/10.1145/2783258.2783351> pp. 447–456.
- [100] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent Dirichlet allocation,” *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, Mar. 2003. [Online]. Available: <http://dl.acm.org/citation.cfm?id=944919.944937>
- [101] C. Doersch, S. Singh, A. Gupta, J. Sivic, and A. A. Efros, “What makes Paris look like Paris?” *ACM Transactions on Graphics*, vol. 31, no. 4, pp. 101:1–101:9, July 2012. [Online]. Available: <http://doi.acm.org/10.1145/2185520.2185597>
- [102] Q. V. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng, “Building high-level features using large scale unsupervised learning,” in *Proceedings of the 29th International Conference on Machine Learning*, 2012. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3042573.3042641> pp. 507–514.
- [103] “New Ways to Find (and Stream) App Content in Google Search,” 2015, <https://search.googleblog.com/2015/11/new-ways-to-find-and-stream-app-content.html>.
- [104] “App.io: App.io streams your mobile apps from the cloud to any device.” <http://www.app.io>.
- [105] “Appetize.io: Stream iOS & Android Native Apps in the Browser.” <http://www.appetize.io>.
- [106] M. S. Mikowski and J. C. Powell, *Single page web applications*. Manning Publications Co., 2013.
- [107] A. Mesbah and A. Van Deursen, “Migrating multi-page web applications to single-page Ajax interfaces,” in *Proceedings of the European Conference on Software Maintenance and Reengineering*. IEEE, 2007, pp. 181–190.
- [108] B. Taylor, A. K. Dey, D. Siewiorek, and A. Smailagic, “Using crowd sourcing to measure the effects of system response delays on user engagement,” in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016. [Online]. Available: <http://doi.acm.org/10.1145/2858036.2858572> pp. 4413–4422.

- [109] R. MacAlpine and D. R. Flatla, “Real-time mobile personalized simulations of impaired colour vision,” in *Proceedings of the 18th International ACM SIGACCESS Conference on Computers and Accessibility*, 2016. [Online]. Available: <http://doi.acm.org/10.1145/2982142.2982170> pp. 181–189.
- [110] K. Reinecke, D. R. Flatla, and C. Brooks, “Enabling designers to foresee which colors users cannot see,” in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016. [Online]. Available: <http://doi.acm.org/10.1145/2858036.2858077> pp. 2693–2704.
- [111] K. A. Siek, Y. Rogers, and K. H. Connelly, “Fat finger worries: How older and younger users physically interact with PDAs,” in *Proceedings of the 2005 IFIP TC13 International Conference on Human-Computer Interaction*, 2005. [Online]. Available: http://dx.doi.org/10.1007/11555261_24 pp. 267–280.
- [112] P. O’Donovan, A. Agarwala, and A. Hertzmann, “Learning layouts for single-page graphic designs,” *IEEE Transactions on Visualization and Computer Graphics*, 2014.
- [113] C. Jacobs, W. Li, E. Schrier, D. Barger, and D. Salesin, “Adaptive grid-based document layout,” *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 838–847, July 2003. [Online]. Available: <http://doi.acm.org/10.1145/882262.882353>
- [114] K. Z. Gajos, D. S. Weld, and J. O. Wobbrock, “Automatically generating personalized user interfaces with Supple,” *Artificial Intelligence*, vol. 174, no. 12-13, pp. 910–950, Aug. 2010. [Online]. Available: <http://dx.doi.org/10.1016/j.artint.2010.05.005>
- [115] D. R. Olsen Jr, “A programming language basis for user interface,” in *ACM SIGCHI Bulletin*, vol. 20, no. SI. ACM, 1989, pp. 171–176.
- [116] A. R. Puerta, “A model-based interface development environment,” *IEEE Software*, vol. 14, no. 4, pp. 40–47, 1997.
- [117] P. Sukaviriya, J. D. Foley, and T. Griffith, “A second generation user interface design environment: The model and the runtime architecture,” in *Proceedings of the INTERACT ’93 and CHI ’93 Conference on Human Factors in Computing Systems*, 1993. [Online]. Available: <http://doi.acm.org/10.1145/169059.169299> pp. 375–382.

- [118] P. Szekely, P. Luo, and R. Neches, “Beyond interface builders: Model-based interface tools,” in *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, 1993. [Online]. Available: <http://doi.acm.org/10.1145/169059.169305> pp. 383–390.
- [119] J. Koch and A. Oulasvirta, “Computational layout perception using Gestalt laws,” in *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, 2016. [Online]. Available: <http://doi.acm.org/10.1145/2851581.2892537> pp. 1423–1429.
- [120] A. Miniukovich and A. De Angeli, “Visual impressions of mobile app interfaces,” in *Proceedings of the 8th Nordic Conference on Human-Computer Interaction: Fun, Fast, Foundational*, 2014. [Online]. Available: <http://doi.acm.org/10.1145/2639189.2641219> pp. 31–40.
- [121] A. Miniukovich and A. De Angeli, “Pick me!: Getting noticed on Google Play,” in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016. [Online]. Available: <http://doi.acm.org/10.1145/2858036.2858552> pp. 4622–4633.